A Verified Code Generator for a Subset of Gypsy

William D. Young

Technical Report 33 October 1988

Computational Logic Inc.
1717 W. 6th St. Suite 290
Austin, Texas 78703
(512) 322-9951

This research was supported in part by the U.S. Government. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc. or the U.S. Government.

Acknowledgements

This work was sponsored in part at Computational Logic, Inc. by the Defense Advanced Research Projects Agency, ARPA Orders 6082 and 9151, and at the University of Texas at Austin by the Defense Advanced Research Projects Agency, ARPA Order 5246, issued by the Space and Naval Warfare Systems Command under Contract N00039-85-K-0085, and by the National Science Foundation (Grants DCR-8122039 and DCR-8202943.

Abstract

This report describes the specification and mechanical proof of a code generator for a subset of Gypsy 2.05 called Micro-Gypsy. Micro-Gypsy is a high-level language containing many of the Gypsy control structures, simple data types and arrays, and predefined and user-defined procedure definitions including recursive procedure definitions. The language is formally specified by a recognizer and interpreter written as functions in the Boyer-Moore logic. The target language for the Micro-Gypsy code generator is the Piton high-level assembly language verified by J Moore to be correctly implemented on the FM8502 hardware. The semantics of Piton is specified by another interpreter written in the logic. A Boyer-Moore function maps a Micro-Gypsy state containing program and data structures into an initial Piton state. We prove that an arbitrary legal Micro-Gypsy program is correctly implemented in Piton. By this we mean that execution of the resulting Piton state is semantically equivalent (under a mapping function we exhibit) to the result of executing the initial Micro-Gypsy state with the Micro-Gypsy interpreter. An interesting fact of our implementation is that we pass procedure parameters efficiently by reference even though we use a *value-result* semantic definition. We are not aware of any previous mechanical proof that passing parameters by reference correctly implements value-result semantics. In this report we define Micro-Gypsy and Piton, describe the translation process, and explain the correctness theorem proved.

Chapter 1
INTRODUCTION

1.1 Motivation

In this thesis we discuss the implementation and proof of a code generator for a subset of Gypsy 2.05 [GoodAkersSmith 86].[1] There were two principle motivations for undertaking this project. The first was a desire to push the limits of program verification and automatic theorem proving and thereby provide a rigorous machine checked proof of a code generator for an existing programming language. The second motivation was the desire to participate in the construction of a collection of tools for building extremely reliable software systems. The Micro-Gypsy code generator target language has an assembler which is verified to be correctly implemented on a piece of verified hardware.

We may refer to the software module described in this thesis as "the Micro-Gypsy compiler" or simply as "the compiler." In truth, it is a verified *code generator* which is described. Several of the tasks typically associated with compilation--lexical analysis, parsing, optimization, etc.--are not considered in this research. We briefly discuss in 9.2.2 extensions of this research to accommodate some of these tasks.

1.2 The Value of Verified Compilation

A compiler provides the ability to program/specify/design in a notation which is more elegant, abstract, or expressive than that which is native to a given piece of hardware. The concomitant gains are real only if the compilation process provides a *correct* translation from the source language to the target language--that is, one which generates semantically equivalent target language code for any given source language program. A correct translation is not necessarily an *adequate* one. There may be additional constraints, such as efficiency, which render some correct programs unacceptable. We are concerned in this thesis primarily with verifying the correctness of compilation. Optimization and related issues dealing with adequacy are discussed briefly in 9.2.2.

Care invested in guaranteeing the correctness of a compiler has multiple benefits.[2]

1. Compilers tend to be used extensively. Thus, errors in a compiler tend to have more significant impact than errors in less frequently used programs.
2. Errors in the compiler are often difficult to distinguish from errors in the source program, and thus may render the task of debugging prohibitively difficult. The programmer should not be

[1] Subsequent references to Gypsy should be understood to refer to Gypsy 2.05 unless some other dialect is indicated.

[2] Polak [Polak 81] provides an extensive discussion of these points.

called upon to debug the compiler.

3. Care invested in coding and verifying any source program is largely wasted if the compilation process produces a target program which is semantically divergent.

The benefits of verified compilation can be gained from verifying every translation instance. This, however, is not cost effective if the translator is to be used more than a very few times.

A (somewhat over-)simplified view of compiling is that it involves three components--parsing, code generation, and optimization. Our work is less ambitious in scope than some previous work--Polak [Polak 81], for example--which dealt with the entire compilation process rather than just the code generation phase. We feel that code generation is clearly the heart of compiling and the most interesting aspect from the point of view of verification. We discuss briefly in Section 9.1 the possible extension of our work to include parsing and optimization.

Our goal is a code generator which merits an extremely high degree of confidence by virtue of having a rigorous machine-checked proof. Moreover, we desire that the source language should have a value independent of the current project so that there is reason to believe that the language was not tailored explicitly to make the compilation trivial. It should be the case that the target language is actually implementable, preferably in a verifiable fashion, so that the benefits derived from the verification of the code generator are not vitiated by an incorrect implementation below. We feel that much previous work in compiler verification has failed to be as complete or rigorous as possible because the work has been either purely theoretical, resulting in no machine checkable proofs, has dealt with source or target languages tailored to make the semantic gap bridged by the compilation process quite narrow, or has been premised on a large number of assumptions of questionable validity.

1.3 Vertically Verified Systems

Apart from the general benefits to be gained from verifying any compiler, we had a special motivation for our specific project. This was the desire to participate in a larger research program aimed at providing a verified link between the world of high level language programs and program verification and that of machine language programs running on actual hardware. The Micro-Gypsy code generator will be an integral component of a larger software system designed to allow verified high level language programs to be translated into semantically equivalent machine language programs running on verified hardware. This larger view constrained both the choice of the source language and the target language and further distinguishes our efforts from previous research.

The desire to reliably translate *verified* high level language programs required choosing a verifiable source language. Gypsy 2.05 was an obvious choice. It is an integrated programming and specification language which has been the language of choice for many secure system development efforts [Smith 81, Keeton-Williams 82, Siebert 84, Good 84, Boebert 85]. Processing of the language and constructing proofs of Gypsy programs is carried out within a mechanized verification environment [GoodDivitoSmith 88]. The language has an elegant semantics which is restrictive enough to facilitate compilation. Moreover, the semantics has been largely formalized [Cohen 86].

Choice of the target language was dictated by the desire to fit the code generator into a "stack" of verified language processing components. The Piton [Moore 88] assembly language, developed by J Moore at Computational Logic, Inc., is a high-level assembly language with a verified implementation on the FM8502 microprocessor. This is an extension of the microprocessor design verified to the gate level by Warren Hunt [Hunt 85].

Building the Micro-Gypsy code generator on top of Piton gives us the ability to verify certain Gypsy 2.05 programs using the Gypsy Verification Environment, compile these programs with the Micro-Gypsy code generator described in this dissertation into Piton assembly language code, and assemble this using the verified Piton assembler into machine code for a verified microprocessor.[3] The resulting machine programs will have an associated degree of assurance of correctness far exceeding anything which can be gained by existing software engineering techniques.

The ability to "stack" these components to obtain a true *vertically verified computing system* is assured only if the interfaces coincide. For the compiler, this implies that the source language must be a genuine subset of Gypsy 2.05. Moreover, to preserve the semantics of Gypsy programs in the compilation process requires that the semantics assumed by the compiler proof must coincide with the semantics of Gypsy assumed in the program proofs in the Gypsy Verification Environment. On the lower end, to accomplish the goal of targeting a verified assembly language, it is necessary that the code generated be legal Piton code. Thus, the project is constrained at both "ends." There is very little freedom in manipulating either the source or target languages to make the compilation process easier.[4]

1.4 The Boyer-Moore-Kaufmann Proof Checker

The implementation and specification of the Micro-Gypsy compiler were written in the Boyer-Moore logic [BoyerMoore 88]. Proofs were carried out using the Boyer-Moore proof checker enhanced with an interactive interface by Matt Kaufmann [Kaufmann 88]. A description of the logic and the proof checker is contained in Appendix A. The choice of this proof environment (rather than, say, the Gypsy Verification Environment) was dictated by the desire to fit the Micro-Gypsy work into the stack of verified components as well as by the Boyer-Moore system's reputation for soundness.

1.5 Plan of the Dissertation

This dissertation describes the construction, specification, and mechanical verification of a piece of software. As software it is not remarkable in any way. The code generator certainly has less functionality and probably less elegance than the product of many an undergraduate compiler class. Its only remarkable feature is our claim that it generates code in a way which provably preserves semantic equivalence.

To defend this claim adequately we must explain what we mean by the correctness of a code generator and describe the software, its specification, and proof. Since the work was carried out in a completely formal notation and the proof entirely mechanically checked, the ultimate "witness" to the work is the script of events characterizing the compiler, its specification, and proof. This script is provided in several pieces. The language definitions and the definitions associated with the code generator are provided as lists of events (in alphabetical order) within the chapters which describe those components. The remaining events, including the majority of the lemmas involved in the proof are listed in Appendix B. The script of events represented by these lists is the real product of this work. The exposition in the following chapters can be regarded as a guide to reading and understanding that script.

---

[3]Several links are still missing in this chain. One involves the disparity between Gypsy 2.05 syntax and Micro-Gypsy syntax accepted by our translator. This issue is explained in Chapter 3. Another is the issue of the "stackability" of Micro-Gypsy on Piton. This is discussed in Section 8.3.

[4]As will be clear later, we did select a rather restricted subset of Gypsy to compile. We also had a small amount of input into the design of Piton; seven Piton instructions were added to accommodate the Micro-Gypsy project.

In Chapter 2 we give a broad overview of what we mean when we say that we have proved the correctness of a code generator. Chapters 3 and 4 describe Micro-Gypsy and Piton, the source and target languages, respectively, of the code generator. Chapter 5 explains the translation from execution environments for Micro-Gypsy into execution environments for Piton. In Chapter 6 we present our formal statement of the correctness theorem for our code generator; Chapter 7 outlines the proof of the correctness theorem. In Chapter 8 we illustrate the usefulness of our work by showing how to construct and verify a Micro-Gypsy program. Finally, Chapter 9 gives our conclusions and discusses the relation of this work to other research.

Chapter 2
INTERPRETER EQUIVALENCE

Much of the intellectual expenditure in the history of computing has been directed to the definition of new formal notations, assigning meanings to these notations, and to the problems of translating from one formal notation to another. Every translation must address the same fundamental issue--how to assure that the meaning of the source language is preserved by the translation process. Stated another way, how can we know that source language programs are *implemented* correctly in the target language. Our task in this dissertation is to provide a rigorous formal proof that in one case--for a particular translation process, source language and target language--the implementation is correct. In this chapter, we show the overall structure of this demonstration and the form of the theorem which asserts the correctness of our translation process.

2.1 Characterizing Semantics Operationally

Any programming language L defines an abstract space E_L of points which we will refer to as the *execution environments* for that language. Execution environments correspond to programs which can be written in the language along with the data upon which they operate and any other structures relevant to program execution. A representation of an execution environment for Lisp, for example, might contain a collection of Lisp function definitions, an alist associating variable names with values, and an S-expression to evaluate.

One way to describe the meaning of a statement or program written in a programming language is to provide an *operational* semantics in the form of an *interpreter* for the language. An interpreter for a language is merely a function $Int_L\colon E_L \rightarrow E_L$. The interpreter may be defined in terms of a *single-stepper* function $Step_L\colon E_L \rightarrow E_L$, where *Int* simply composes calls to $Step_L$ for some fixed number of iterations or until some halting condition is satisfied. Informally, an interpreter characterizes the meaning of a program by describing its effect upon its execution environment.

For a typical language, there is no guarantee that *Int* is a total function. Programs, whether intentionally or unintentionally, are often designed to be non-terminating. We can force the interpreter definition to be total by adding an additional argument which is the maximum number of steps we will allow a program to run. Using this artifice, we can recursively define an interpreter in the Boyer-Moore logic[5] as follows:

[5]Appendix A describes the Boyer-Moore logic and its implementation.

```
DEFINITION.
(INT1 STATE N)
   =
(IF (ZEROP N)
    STATE
    (INT1 (STEP1 STATE) (SUB1 N)))
```

Here `STEP1` is the single-stepper function describing the effect of running the interpreter for a single state transition. It is the particulars of `STEP1` which distinguish one such interpreter from that for other programming languages. An important task of `STEP1` is determining from the current state, the action to be performed. This may involve extracting an instruction from a program component of the state. This, for example, is the form of the interpreter for our target language Piton described in Chapter 4.

Quite often, the interpreter is such that some components of the execution environment, the program for example, are invariant. For these interpreters we may separate some static components of the execution environment from the dynamic components and make them separate arguments to the interpreter function to emphasize the distinction. One useful class contains interpreters of the form $Int: P \times S \rightarrow S$, where P denotes the collection of legal programs in the language and S the remainders of the execution environments. This is the abstract form of the interpreter for our source language Micro-Gypsy, for example, described in Chapter 3. The general form is

```
DEFINITION.
(INT2 PROG STATE N)
   =
(IF (ZEROP N)
    STATE
    (INT2 PROG (STEP2 PROG STATE) (SUB1 N))).
```

This form tends to emphasize the role of the program and make clear that the program is not subject to modification.

For an interpreter in either form, disallowing modification of the program removes the difficult task (inherent in Bevier's work [Bevier 87]) of proving that the interpreter does not modify the program being executed. Fixing the program defines a particular interpreter function $Int_p: S \; X \; N \rightarrow S$ which can be viewed as the semantics of the program **P**.

```
DEFINITION.
(INT2_P STATE N)
   =
(IF (ZEROP N)
    STATE
    (INT2_P (STEP2_P STATE) (SUB1 N)))
```

2.2 Interpreter Equivalence Theorems

Suppose that we have two languages *SL* and *TL* defined by interpreters, and wish to define a translation from *SL* to *TL*. "Translating," as we will use that term, means more than code generation; it means mapping entire execution environments. Part of this process is clearly code generation, but there is also the translation of data, and the functions often associated with *loading*. The goal of our translation is a target language execution environment suitable for interpretation. In this section we characterize the relationship between the source language and target language execution environments which holds in cases where we are willing to assert that the translation is correct.[6]

---

[6]This is an instance of the classic problem of program verification; a program--in this case the translator--is only "correct" with respect to a specification. We must define formally in our specification a relationship which accords with our intuitive and informal notions of what it means for a translation to be correct.

Our translator *Map-Down* has signature *Map-Down:* $E_{SL} \rightarrow E_{TL}$. Suppose that our source language execution environments are of the form $<P, S>$ where we distinguish the program component from the remainder of the state. We say that the translation is correct if,

$$\begin{array}{l} \forall P\ \forall S, \\ \quad <P, S> \in E_{SL} \\ \quad \rightarrow \\ \quad \textit{Map-Down}\ (Int_{SL}\ (<P, S>)) = Int_{TL}\ (\textit{Map-Down}\ (<P, S>))\ . \end{array} \tag{1}$$

We also say that *Map-Down* $(<P, S>)$ *implements* $<P, S>$.

(1) is a formalization of the classic commutative diagram shown in figure 2-1. We can obtain the results

$$\begin{tikzcd} \textbf{SL-initial} \arrow[r, dashed, "Int_{SL}"] \arrow[d, "\textbf{Map-Down}"] & \textbf{SL-final} \arrow[d, "\textbf{Map-Down}"] \\ \textbf{TL-initial} \arrow[r, dashed, "Int_{TL}"'] & \textbf{TL-final} \end{tikzcd}$$

Figure 2-1: Interpreter Equivalence Diagram 1

of stepping forward from execution environment $<P, S>$ in either of two ways. We can run the source language interpreter and map the result down to the target language domain. Alternatively, we can map down to an appropriate target language environment and step the result forward using the target language interpreter. We call (1) an *interpreter equivalence* theorem.

Figure 2-1 gives one possible form for an interpreter equivalence theorem. An alternative approach is to define an *abstraction* function *Map-Up:* $E_{TL} \rightarrow E_{SL}$, which can be thought of as an inverse mapping to *Map-Down*.[7] We call the translation correct if

$$\begin{array}{l} \forall P\ \forall S, \\ \quad <P, S> \in E_{SL} \\ \quad \rightarrow \\ \quad Int_{SL}\ (<P, S>) = \textit{Map-Up}\ (Int_{TL}\ (\textit{Map-Down}\ (<P, S>))). \end{array} \tag{2}$$

This gives us the commutative diagram in figure 2-2. Notice that if we can show that

$$\forall P\ \forall S,\ \textit{Map-Up}\ (\textit{Map-Down}\ (<P, S>)) = <P, S> \tag{3}$$

then (2) is an immediate consequence of (1).

If we add the clock parameters to our formalization, we must consider the correspondence between the number of "ticks" required for an execution at the *SL* level and that at the *TL* level. They need not be the same. Extending the interpreters to add the additional clock argument yields yet another version of our interpreter equivalence theorem,

---

[7] If *Map-Down* is not injective, it may not be possible to define *Map-Up* without reference to the high-level state. For example, the *Map-Down* function may map all high-level data values into bit-vectors. We cannot map these values back into a high level context without some type information for the inverse mapping. This is the case for the Piton proof [Moore 88] as well as for our code generator proof. This issue is discussed further below.

$$\begin{tikzcd}[column sep=huge, row sep=huge] \text{SL-initial} \arrow[r, "Int_{SL}"] \arrow[d, "\text{Map-Down}"'] & \text{SL-final} \\ \text{TL-initial} \arrow[r, "Int_{TL}"'] & \text{TL-final} \arrow[u, "\text{Map-Up}"'] \end{tikzcd}$$

Figure 2-2: Interpreter Equivalence Diagram 2

$$\begin{aligned} &\forall P\ \forall S\ \forall N\ \exists K, \\ &\quad <P, S> \in E_{SL} \\ &\rightarrow \\ &\quad Int_{SL}\ (<P, S>, N) = Map\text{-}Up\ (Int_{TL}\ (Map\text{-}Down\ (<P, S>)), K). \end{aligned} \tag{4}$$

The existential quantification makes this formalization unsuitable for a quantifier-free formal logic such as the Boyer-Moore logic. However we can eliminate the existential quantifier by providing a "witness function" *clock* which computes K explicitly. This yields, finally,

$$\begin{aligned} &\forall P\ \forall S\ \forall N, \\ &\quad <P, S> \in E_{SL} \\ &\rightarrow \\ &\quad Int_{SL}\ (<P, S>, N) \\ &\quad\quad = \\ &\quad Map\text{-}Up\ (Int_{TL}\ (Map\text{-}Down\ (<P, S>)), Clock\ (<P, S>, N)). \end{aligned} \tag{5}$$

(5) clearly implies (4) by existential generalization.

It is this final form of the interpreter equivalence theorem which we use in our proof of the code generator. We claim that such a theorem is a natural formalization of the remark that a translation process is correct. Consider, for example, any compiler from a high-level language into machine language. To assert the correctness of the compiler is to claim that it can be used in conjunction with the loader to generate (map-down to) a core image which can be run on the target machine. The result of this execution is a core image from which can be extracted (mapped up) values representing the values of the variables in the high level language. These values should be precisely what would be derived by interpreting the source program in the machine defined by the semantics of the high-level language.

An interpreter equivalence theorem may be more subtle than it at first appears. Keep in mind that we are formalizing the *intuitive* notion of the correctness of a translator. Some care is required to assure that in our reliance on the formalism, the intuition is not lost. It must be the case, for instance, that the instances of *Map-Down* and *Map-Up* are *reasonable* in a sense which is difficult to formalize. There are at least three ways in which one could imagine formalizing a correct interpreter equivalence theorem but one which defeats the spirit of the commuting diagram.

1. Suppose the *Map-Up* function can somehow gain access to the initial high-level state and merely apply function Int_{SL}, bypassing Int_{TL} entirely. We could guard against this subterfuge by insisting that the *Map-Up* function have no access to the initial high-level state. However, this may be impossible if the *Map-Down* function is not injective and hence not invertible.[8] Also, *Map-Down* could somehow encode the initial high-level state and store it in the low-

[8] We need type information, for instance, to properly interpret a core dump. We return to this issue in our discussion of our *Map-Up* function in Chapter 6.

level state.

2. Another inappropriate scenario would be for the *Map-Down* to apply Int_{SL} to obtain the final high-level state. It could then encode that result directly in the initial lower-level state. *Map-Up* then merely extracts the information, following a dummy computation with Int_{TL}.
3. Finally, the theorem would be trivially satisfied if each of the key functions was essentially a no-op, causing the diagram to degenerate to a single point.

These scenarios suggest that the interpreters and mapping functions must really have semantic content appropriate to their respective intuitive roles. Formalizing this notion of "semantic content" is a difficult research area outside the scope of this dissertation. Our approach is simply to present our formal definitions and argue informally that they have such semantic content. The reader is advised to scrutinize the formal definitions carefully.

Most of the remainder of this dissertation describes the construction and proof of an interpreter equivalence theorem. We describe interpreters for our source language Micro-Gypsy and target language Piton and mapping functions between Micro-Gypsy and Piton execution environments. From the construction it will be clear that our translation implements Micro-Gypsy programs as Piton programs in the spirit of figure 2-2. The actual interpreter equivalence theorem we prove is stated in Chapter 6 following the definitions of the two languages and our *Map-Down* function.

Chapter 3
MICRO-GYPSY

3.1 Gypsy and Micro-Gypsy

The source language for our code generator is an abstract syntax form of a subset (with certain caveats explained below) of Gypsy 2.05 [GoodAkersSmith 86] called Micro-Gypsy. Gypsy is a combined programming and specification language descended from Pascal. It includes dynamic types such as sequences and sets, permits procedural and data abstraction, and supports concurrency. The specification component of Gypsy contains the full first order predicate calculus and the ability to define recursive functions. Programs can be specified with Hoare style annotations, algebraically, or axiomatically. Proof rules exist for each component of the language. The Gypsy Verification Environment (GVE) is a collection of software tools which allow Gypsy programs to be developed, specified, and proved. Previously, Gypsy programs could also be compiled and run on certain DEC machines, though this capability is currently unavailable.

Micro-Gypsy contains a significant portion of the executable component of Gypsy including the simple data types and arrays, most of the Gypsy flow of control operations, and predefined and user-defined procedures including recursive procedures. It does not include Gypsy's dynamic data types, data abstraction, or concurrency. Our intent in defining the Micro-Gypsy subset of Gypsy was a language sufficient for coding simple verifiable applications of the variety for which Gypsy has been used. We claim that the subset is adequate to validate our overall approach to proving the correctness of a code generator but needs to be extended in a variety of ways to make it a useful programming tool. It is our belief that the subset can be extended straightforwardly to include other features of Gypsy. This is discussed further in Section 9.2.2.

Many features of Micro-Gypsy, the range of elements of Micro-Gypsy's integer type, for example, was influenced by our interest in compilation. We chose a subset of those features of Gypsy suitable for compilation. Some selection was necessary because Gypsy permits arbitrarily sized data structures, unbounded quantification, and other features which do not lend themselves to compilation, but also to make our project manageable.

Gypsy programs may be annotated using the specification component of the language and verified in the GVE. The Micro-Gypsy Lisp-like syntax used in this report is not acceptable to the GVE. However, this abstract syntax could easily be generated from official Gypsy syntax by a preprocessor.[9] Thus, Micro-

[9] Such a preprocessor was written for an earlier version of Micro-Gypsy by Ann Siebert. There is no such preprocessor for the current version of Micro-Gypsy, however. We will sometime refer to the Micro-Gypsy preprocessor but the reader should realize that this is a convenient fiction rather than an existing piece of software.

Gypsy programs could be annotated and verified using all of the mechanisms available for verifying programs in Gypsy 2.05. The verified program would be preprocessed into a form acceptable to our verified compiler and then compiled to a semantically equivalent Piton program. An alternative approach to proving Micro-Gypsy programs is to verify them directly in the Boyer-Moore logic using the semantics defined by the Micro-Gypsy interpreter. Both approaches are discussed and illustrated in Chapter 8.

Since annotations are regarded as comments at compile time,[10] the specification component of the language is not relevant to compilation, though it is relevant to the provability of Micro-Gypsy programs. We assume that the entire specification component of Gypsy can be retained for specifying and proving programs in the subset with the assumption that the Gypsy parser can be restricted to accept Micro-Gypsy programs annotated with Gypsy specifications. Constructing such a parser has not been considered in the course of this research.

Micro-Gypsy as it is characterized in this document is not strictly a subset of Gypsy 2.05 because of the abstract syntax we consider is different from the official Micro-Gypsy syntax and because our proof does not require that we enforce certain restrictions which Gypsy imposes. We envision compilation of Micro-Gypsy programs as a two-step process. User supplied programs are translated by the preprocessor into the abstract syntax form expected by our functions. This preprocessor rejects any programs which do not meet the stringent syntactic restrictions which define the "official" Micro-Gypsy and which guarantee that it is strictly a subset of Gypsy 2.05. Programs, for example, which use as identifiers Gypsy keywords such as **`IF`** and **`BEGIN`** are rejected; procedures which contain assignments to **`CONST`** parameters are rejected.

The second step in the compilation process, the step with which this dissertation is concerned, assumes that programs have passed this first filter. Nothing in this second step prohibits using **`IF`** or **`BEGIN`** as identifier names; no distinction is made between **`VAR`** and **`CONST`** parameters. It would not be difficult to enforce these additional restrictions, but they are not necessary for the proof we are undertaking. Thus our functions will correctly translate all Micro-Gypsy programs which could be produced by the preprocessor. They will "correctly" translate many programs as well which could not be the output of the preprocessor on any legal Micro-Gypsy program.

Micro-Gypsy is characterized by a recognizer and an interpreter. The recognizer ensures that the language satisfies the minimal set of syntactic constraints required for our proof; the interpreter provides an operational semantics for the language. The formal definition of Micro-Gypsy for the purposes of this dissertation is embodied in a collection of function definitions written in the Boyer-Moore logic and defining the recognizer and interpreter. For the reader's convenience these are listed alphabetically in Section 3.5, the last section of the current chapter. For purposes of this research, these definitions are the official arbiter of any and all questions concerning Micro-Gypsy. The other sections of this chapter may be regarded as an informal and incomplete guide to the formal definition. This informal discussion will advertise certain "entry points" into the formal definition in the form of key functions. The reader is advised to study the formal definition by beginning with these functions and investigating their definitions fully.

[10] Gypsy allows some annotations to be validated at run-time. We do not consider these in any of our discussions.

3.2 Summary of Micro-Gypsy

The input to the compiler is an abstract prefix form of Micro-Gypsy. All subsequent discussion will refer to the abstract syntax as though it were the standard syntax of Micro-Gypsy. It should be remembered if the syntax seems verbose and awkward that we envision this syntax as being the output of a preprocessor and much of this awkwardness would not be visible to the end user of the language. For example, consider the following program fragment in the Gypsy-style syntax:

```
loop
   if B then leave
        else X := U; Y := 23; signal FOO;
   end; {if}
end; {loop}
```

The same program fragment in our abstract syntax form would appear as

```
(LOOP-MG
  (IF-MG B
     (SIGNAL-MG LEAVE)
     (PROG2-MG (PREDEFINED-PROC-CALL-MG MG-VARIABLE-ASSIGNMENT (X U))
               (PROG2-MG (PREDEFINED-PROC-CALL-MG MG-CONSTANT-ASSIGNMENT
                                                  (Y (INT-MG 23)))
                         (SIGNAL-MG FOO))))).
```

Micro-Gypsy is a von Neumannn language; a program has effect only insofar as it makes changes to a global state consisting of a collection of data structures. Computation is performed by operating on these data structures with procedures predefined by the language or defined by the programmer.

One distinguished component of the Micro-Gypsy state is the *current condition*. Each Micro-Gypsy operation potentially updates the current condition which in turn affects the execution of subsequent statements. As described in Section 3.4 whenever a condition is *signaled*--by setting the current condition to some value other than **'NORMAL**--control flows lexically outward until a handler for the condition is encountered or the program is exited. Much of the mechanism of interpreting and compiling Micro-Gypsy is directed toward achieving this result in a way which is faithful to Gypsy semantics.

Micro-Gypsy is a strongly typed language. Data types include the three simple types **INT-MG**, **BOOLEAN-MG**, and **CHARACTER-MG** and one dimensional fixed-length homogeneous arrays of the simple types. Literals can be defined for each of the types. Simple literals are tagged with their types; array literals are lists of simple literals.

The only expressions allowed in the language are variable references and literals of the simple types. Since the input to the compiler is the product of a preprocessor, this is a less onerous restriction than it at first appears. Expressions in the source code are translated by the preprocessor into an appropriate sequence of calls to predefined routines. The preprocessor can make this process entirely transparent to the programmer.

Micro-Gypsy has eight statement types: **NO-OP-MG**, **SIGNAL-MG**, **PROG2-MG**, **LOOP-MG**, **IF-MG**, **BEGIN-MG**, **PROC-CALL-MG**, and **PREDEFINED-PROC-CALL-MG**. The syntactic requirements on these statement types are described in section 3.3 and the semantics described in Section 3.4.

3.3 The Recognizer

The collection of Boyer-Moore functions which we term *the recognizer* defines the abstract syntax of Micro-Gypsy syntactic units including statements and procedures. The principal functions described in this section are **OK-MG-STATEMENT** (see page 50), the recognizer for Micro-Gypsy instructions, and **OK-MG-DEF-PLISTP** (see page 48), the recognizer for procedure definitions.

3.3.1 Identifiers

Identifiers in Micro-Gypsy are simply Boyer-Moore litatoms which are neither Micro-Gypsy reserved words nor contain the character "-". We make the latter restriction so that we can define special identifiers at will and be assured that they will not conflict with any user-defined identifiers. Micro-Gypsy identifiers are a strict superset of Gypsy identifiers. The only reserved words of Micro-Gypsy are the three special condition names **'LEAVE**, **'ROUTINEERROR**, and **'NORMAL**.

3.3.2 Types and Literals

The simple types are the **INT-MG**, **BOOLEAN-MG**, and **CHARACTER-MG** types. One dimensional arrays of simple types are also allowed. Array type references have the form **(ARRAY-MG ARRAY-ELEMENT-TYPE ARRAY-LENGTH)** where **ARRAY-ELEMENT-TYPE** is a simple type and **ARRAY-LENGTH** is a positive integer. All Micro-Gypsy arrays are indexed from [0..**ARRAY-LENGTH** - 1]. Legal types are recognized by the function **MG-TYPE-REFP** (see page 46).

Micro-Gypsy allows literal values of each of the allowed types. Literals are tagged with their types. Some examples are given below.

```
(INT-MG 2)                                      integers
(INT-MG -1024)

(BOOLEAN-MG FALSE-MG)                           booleans
(BOOLEAN-MG TRUE-MG)

(CHARACTER-MG 23)                               characters
(CHARACTER-MG 127)

((INT-MG 2) (INT-MG -1024) (INT-MG 0))          arrays
((BOOLEAN-MG FALSE-MG) (BOOLEAN-MG TRUE-MG))
```

The **INT-MG** type is the subset of Gypsy integers representable on the target machine, in this case Piton's 32-bit two's complement integers. The **CHARACTER-MG** type is the 128-member ASCII character set of Gypsy; character literals are represented in terms of their ASCII character codes. An array literal of **ARRAY-TYPE** is a proper list of simple literals. Each element must be of type **(ARRAY-ELEMTYPE ARRAY-TYPE)** and the list must be of length **(ARRAY-LENGTH ARRAY-TYPE)**.

3.3.3 Recognizing Statements

The predicate **OK-MG-STATEMENT** (figure 3-1) is the recognizer for Micro-Gypsy instructions. **OK-MG-STATEMENT** checks the syntactic form of the statement and checks that it meets a set of minimal requirements necessary for carrying out the proof.

A Micro-Gypsy statement is recognized within a context which consists of the following components: the **NAME-ALIST**, **COND-LIST**, and **PROC-LIST**. The **NAME-ALIST** is an association list of pairs of the form **<VARIABLE NAME, TYPE>**. This is used to determine if a variable is defined and whether uses of a variable within the code are consistent with its type. The **NAME-ALIST** which applies to a statement is constructed from the declarations of the enclosing procedure as described in Section 3.3.4 below. Types of variables are determined by looking up their values on the **NAME-ALIST**. The following **NAME-ALIST**, for example, is part of a recognizer context in which five simple and two array variables are declared.

```
((X INT-MG) (Y INT-MG) (CH1 CHARACTER-MG) (B BOOLEAN-MG)
 (A1 (ARRAY-MG INT-MG 10)) (A2 (ARRAY-MG BOOLEAN-MG 4))
 (CH2 CHARACTER-MG)).
```

The **COND-LIST** gives the possible conditions which may be raised by the execution of a statement. This list is initially constructed from the formal condition parameters and condition locals of the enclosing procedure (see Section 3.3.4) and is updated recursively within **OK-MG-STATEMENT**. The litatom **'LEAVE**, for example, is placed on this list when recognizing the body of a loop; **'LEAVE** is a special condition permissibly signaled only within loop bodies. The condition list is simply a list of literal atoms.

The **PROC-LIST** is the list of user-defined procedures. Predefined procedures are treated specially. The particular form of the **PROC-LIST** is described in Section 3.3.4 below.

As with most of the other major functions we will consider, **OK-MG-STATEMENT** recurses on the structure of a Micro-Gypsy statement, splitting into various cases depending upon the operator of the statement. The text of **OK-MG-STATEMENT** is given in figure 3-1. **OK-MG-STATEMENT** defines the syntactic requirements placed on each of the statement types in our formalism. Each is required to be a proper list[11] of fixed length. In addition, there are specific requirements on the arguments to each statement constructor. These are summarized below.

- **(NO-OP-MG)**: no arguments.
- **(SIGNAL-MG SIGNALLED-CONDITION)**: **SIGNALLED-CONDITION** must be either **'ROUTINEERROR** or be a member of **COND-LIST**.
- **(PROG2-MG PROG2-LEFT-BRANCH PROG2-RIGHT-BRANCH)**: both branches must be Micro-Gypsy statements legal in the current context.
- **(LOOP-MG LOOP-BODY)**: **LOOP-BODY** is a Micro-Gypsy statement legal in the current context with **'LEAVE** added to the list of permissible signals.
- **(IF-MG IF-CONDITION IF-TRUE-BRANCH IF-FALSE-BRANCH)**: **IF-CONDITION** is a Boolean variable; both branches are Micro-Gypsy statements legal in the current context.
- **(BEGIN-MG BEGIN-BODY WHEN-LABELS WHEN-HANDLER)**: **BEGIN-BODY** is a legal statement in the current context with the **WHEN-LABELS** added to the list of permissible conditions. These labels are a non-empty subset of the **COND-LIST**. The **WHEN-HANDLER** is a legal statement in the current context.
- **(PROC-CALL-MG CALL-NAME CALL-ACTUALS CALL-CONDS)**: This must be a call to a user-defined procedure with formals compatible with these actuals. **CALL-CONDS** is the list of actual condition parameters to the call and must agree in length with the list of formal condition parameters. No aliasing is permitted in the data actuals.
- **(PREDEFINED-PROC-CALL-MG CALL-NAME CALL-ACTUALS)**: This must be a call to one of the Micro-Gypsy predefined routines with actuals appropriate for that procedure.

Only the clauses in the recognizer for user-defined and predefined procedure calls are at all complex. We consider each of these separately.

[11] A proper list is a list whose last CDR is NIL.

DEFINITION.

```
(OK-MG-STATEMENT STMT R-COND-LIST ALIST PROC-LIST)
   =
(CASE (CAR STMT)

  (NO-OP-MG (EQUAL (CDR STMT) NIL))

  (SIGNAL-MG
     (AND (LENGTH-PLISTP STMT 2)
          (OK-CONDITION (SIGNALLED-CONDITION STMT) R-COND-LIST)))

  (PROG2-MG
     (AND (LENGTH-PLISTP STMT 3)
          (OK-MG-STATEMENT (PROG2-LEFT-BRANCH STMT)
                           R-COND-LIST ALIST PROC-LIST)
          (OK-MG-STATEMENT (PROG2-RIGHT-BRANCH STMT)
                           R-COND-LIST ALIST PROC-LIST)))

  (LOOP-MG
     (AND (LENGTH-PLISTP STMT 2)
          (OK-MG-STATEMENT (LOOP-BODY STMT)
                           (CONS 'LEAVE R-COND-LIST) ALIST PROC-LIST)))

  (IF-MG
     (AND (LENGTH-PLISTP STMT 4)
          (BOOLEAN-IDENTIFIERP (IF-CONDITION STMT) ALIST)
          (OK-MG-STATEMENT (IF-TRUE-BRANCH STMT)
                           R-COND-LIST ALIST PROC-LIST)
          (OK-MG-STATEMENT (IF-FALSE-BRANCH STMT)
                           R-COND-LIST ALIST PROC-LIST)))

  (BEGIN-MG
     (AND (LENGTH-PLISTP STMT 4)
          (OK-MG-STATEMENT (BEGIN-BODY STMT)
                           (APPEND (WHEN-LABELS STMT) R-COND-LIST)
                           ALIST PROC-LIST)
          (NONEMPTY-COND-IDENTIFIER-PLISTP (WHEN-LABELS STMT)
                                           R-COND-LIST)
          (OK-MG-STATEMENT (WHEN-HANDLER STMT)
                           R-COND-LIST ALIST PROC-LIST)))

  (PROC-CALL-MG
     (OK-PROC-CALL STMT R-COND-LIST ALIST PROC-LIST))

  (PREDEFINED-PROC-CALL-MG
     (OK-PREDEFINED-PROC-CALL STMT ALIST))

  (OTHERWISE F)))
```

Figure 3-1: OK-MG-STATEMENT

3.3.3-A Recognizing User-Defined Procedure Calls

The **PROC-LIST** argument to the recognizer is a list of all of the user-defined procedures. The particular form of this list is discussed in Section 3.3.4 below, but for now we need to know that a member of the list is of the form:

```
(PROC-NAME FORMAL-DATA-PARAMS FORMAL-CONDITION-PARAMS ... ).
```

Recognizing a call requires fetching the template of the called routine from the **PROC-LIST** to determine that the formals and actuals correspond in number and type. Fetching the procedure definition is done with the function **FETCH-CALLED-DEF** (page 37) which simply looks up the procedure definition on the **PROC-LIST** using the **PROC-NAME** as a key.

A call to a user-defined procedure is recognized by the function **OK-PROC-CALL**.

DEFINITION.

```
(OK-PROC-CALL STMT R-COND-LIST ALIST PROC-LIST)
   =
(AND (LENGTH-PLISTP STMT 4)
     (IDENTIFIERP (CALL-NAME STMT))
     (USER-DEFINED-PROCP (CALL-NAME STMT) PROC-LIST)
     (OK-ACTUAL-PARAMS-LIST (CALL-ACTUALS STMT) ALIST)
     (NO-DUPLICATES (CALL-ACTUALS STMT))
     (DATA-PARAM-LISTS-MATCH
            (CALL-ACTUALS STMT)
            (DEF-FORMALS (FETCH-CALLED-DEF STMT PROC-LIST))
            ALIST)
     (COND-IDENTIFIER-PLISTP (CALL-CONDS STMT) R-COND-LIST)
     (COND-PARAMS-MATCH (CALL-CONDS STMT)
                        (DEF-CONDS (FETCH-CALLED-DEF STMT PROC-LIST))))
```

Roughly speaking, **OK-PROC-CALL** requires that a call statement

```
(PROC-CALL-MG CALL-NAME CALL-ACTUALS CALL-CONDS)
```

satisfy the following constraints.

- **CALL-NAME** must be the name of a user-defined procedure on the **PROC-LIST**.
- The actual parameters must be variable identifiers defined on **NAME-ALIST** and each actual must have the same type as the corresponding formal. The formal and actual parameter lists must agree in length.
- The actual parameters must be distinct identifiers. This assures that there is no dangerous aliasing among the parameters to a routine.
- The actual condition parameters must all be either **'ROUTINEERROR** or members of **COND-LIST**. The lengths of the actual and formal condition parameter lists must match.

3.3.3-B Recognizing Predefined Procedure Calls

It would be possible to subsume Micro-Gypsy predefined procedure calls under the same syntactic umbrella as user-defined procedure calls. However, this would require forcing them to adhere to all of the same restrictions placed on user-defined procedures. In particular, we insist that the actuals for user-defined procedures be distinct variables defined in the state alist. We would like to relax this restriction for some predefined procedures to permit us to easily code common Gypsy statements such as **X := X + Y** and **Z := 12**. For this reason we syntactically distinguish predefined procedures from user-defined procedures and permit, in certain cases, non-dangerous aliasing and literal arguments.

The recognizer for Micro-Gypsy predefined procedure calls is the function **OK-PREDEFINED-PROC-CALL**. This predicate checks that the call is of the appropriate form, that the called procedure is one of the permissible predefined routines, and that the arguments are correct for that particular routine.

DEFINITION.
```
(OK-PREDEFINED-PROC-CALL STMT ALIST)
   =
(AND (LENGTH-PLISTP STMT 3)
     (PREDEFINED-PROCP (CALL-NAME STMT))
     (OK-PREDEFINED-PROC-ARGS (CALL-NAME STMT)
                              (CALL-ACTUALS STMT)
                              ALIST))
```

Currently 13 predefined procedures are included in the subset. These are listed by the function **PREDEFINED-PROCEDURE-LIST** (page 52). This is a fairly small subset of the predefined functions and procedures available in Gypsy 2.05 and is mainly intended to demonstrate the variety of different potential predefined procedures which could be added. Our experience is that a new predefined procedure can be added to Micro-Gypsy and proven correct with a few hours effort.

Some of the predefined routines such as the assignment procedures are obvious requirements of an imperative language. Others, such as the arithmetic procedure **MG-INTEGER-ADD**, are the analogs of Gypsy predefined functions and are necessitated by the absence of complex expressions in Micro-Gypsy. Where a Gypsy programmer would write **X := Z + Y - W**, the Micro-Gypsy programmer (or more likely the preprocessor) must construct

```
(PROG2-MG (PREDEFINED-PROC-CALL-MG MG-INTEGER-ADD (TEMP Z Y))
          (PREDEFINED-PROC-CALL-MG MG-INTEGER-SUBTRACT (X TEMP W))).
```

Checking of the call's actual parameters is done with the function **OK-PREDEFINED-PROC-ARGS** (page 18). This predicate does a case split on the call-name and invokes an appropriate predicate to check the particular syntactic requirements for the arguments of the called routine.

DEFINITION.
```
(OK-PREDEFINED-PROC-ARGS NAME ARGS ALIST)
   =
(CASE NAME
      (MG-SIMPLE-VARIABLE-ASSIGNMENT
           (OK-MG-SIMPLE-VARIABLE-ASSIGNMENT-ARGS ARGS ALIST))
      (MG-SIMPLE-CONSTANT-ASSIGNMENT
           (OK-MG-SIMPLE-CONSTANT-ASSIGNMENT-ARGS ARGS ALIST))
      (MG-SIMPLE-VARIABLE-EQ (OK-MG-SIMPLE-VARIABLE-EQ-ARGS ARGS ALIST))
      ...
      (MG-ARRAY-ELEMENT-ASSIGNMENT
           (OK-MG-ARRAY-ELEMENT-ASSIGNMENT-ARGS ARGS ALIST))
      (OTHERWISE F))
```

Each of the predefined procedures has distinct syntactic requirements for its argument list; this allows maximum flexibility. Consider for example the requirements on arguments to calls of the predefined procedure **MG-SIMPLE-VARIABLE-ASSIGNMENT**. This is a "generic" operation which implements Gypsy statements of the form **X := Y** where both arguments are simple variables of the same type. These constraints are checked by the function **OK-MG-SIMPLE-VARIABLE-ASSIGNMENT-ARGS**.

DEFINITION.
```
(OK-MG-SIMPLE-VARIABLE-ASSIGNMENT-ARGS ARGS ALIST)
   =
(AND (LENGTH-PLISTP ARGS 2)
     (SIMPLE-IDENTIFIERP (CAR ARGS) ALIST)
     (SIMPLE-IDENTIFIERP (CADR ARGS) ALIST)
     (EQUAL (GET-M-TYPE (CAR ARGS) ALIST)
            (GET-M-TYPE (CADR ARGS) ALIST)))
```

Another predefined procedure **MG-INTEGER-LE** is the Micro-Gypsy analog of the Gypsy statement **B := X LE Y**. It requires three arguments, a Boolean variable and two integer variables.

DEFINITION.
```
(OK-MG-INTEGER-LE-ARGS ARGS ALIST)
   =
(AND (LENGTH-PLISTP ARGS 3)
     (BOOLEAN-IDENTIFIERP (CAR ARGS) ALIST)
     (INT-IDENTIFIERP (CADR ARGS) ALIST)
     (INT-IDENTIFIERP (CADDR ARGS) ALIST))
```

The reader should investigate the syntactic requirements of each of the Micro-Gypsy predefined procedures. The requirements on the arguments for predefined *proc-name* are defined by the function **OK-***proc-name***-ARGS**.

3.3.4 Recognizing Procedures

One of the requirements of our proof is that the procedure list be a list of legal Micro-Gypsy user-defined procedures. A Micro-Gypsy user-defined procedure has the form:

```
(PROC-NAME FORMAL-DATA-PARAMS
           FORMAL-CONDITION-PARAMS
           LOCAL-VARIABLES
           LOCAL-CONDITIONS
           BODY)
```

The syntactic constraints of a procedure definition are checked by the predicate **OK-MG-DEF** (page 48). To be acceptable, a procedure definition must satisfy the following constraints.

- The procedure name is a legal identifier.
- The formal parameter list is a proper list of pairs of the form **<IDENTIFIER, TYPE>**.
- The formal and local condition lists are proper lists of identifiers.
- The local variable list is a proper list of triples of the form **<IDENTIFIER, TYPE, INITIAL VALUE>** where **INITIAL VALUE** is appropriate for **TYPE**.[12]
- The names of formal parameters and locals variables are all distinct.
- The number of formal conditions is less than 4,294,967,293.[13]
- The procedure body is a Micro-Gypsy statement legal in the context defined by the procedure "header". The **NAME-ALIST** of this context is simply the concatenation of the formal and local variable lists. The **COND-LIST** is the concatenation of the formal and local condition lists.

A legal procedure list is simply a proper list of legal procedures. This is formalized in the function **OK-MG-DEF-PLISTP** (page 48). Our definitions allow recursive and mutually recursive procedures.

---

[12]All data types in Gypsy have a default initial value. The preprocessor may fill in the initial value field with the appropriate default initial value for the type.

[13]This restriction has to do with the way in which Micro-Gypsy conditions are represented in Piton and is discussed further in Chapter 5.

3.4 The Micro-Gypsy Interpreter

In this section we outline the operational semantics of Micro-Gypsy. Our goal is an understanding of the Micro-Gypsy state and the semantics of the permissible statement types. The formal characterization of the Micro-Gypsy semantics is provided in Section 3.5 in the form of an alphabetical listing of the Boyer-Moore functions defining the semantic definitions.

In a way our treatment is somewhat misleading because we concentrate on the meaning of *statements*. In truth, a Micro-Gypsy statement never occurs in isolation; statements appear in procedures and are executed only in the context of executing a procedure. However, it is awkward to talk about the meaning of a procedure without first discussing the meanings of its constituent statements. This in turn leads us into discussing the context for statement execution which is derived largely from the procedure definition. Thus, there are some unavoidable "forward references" in our discussion. Hopefully, these are all ultimately resolved to the reader's satisfaction.

Our interpreter function **`MG-MEANING`** gives an operational semantics to a Micro-Gypsy statement with respect to a certain *execution environment*. The execution environment can best thought of as a "snapshot" of some Micro-Gypsy world taken at the point just before the execution of that statement and containing all aspects of the Micro-Gypsy world which could possibly be relevant to the execution. It contains, for example, values of all of the variables which could be touched in the statement execution and procedures which might be called. Some components of the execution environment are static and some are dynamic. The procedure list, for example, is not changed by execution, whereas the values of variables on the variable alist may change. The dynamic components (with the exception of the clock which is treated separately) are bundled together into a structure which we call the **`MG-STATE`**. Given any statement and execution environment "snapshot," our interpreter definition tells us what the following snapshot must look like.

We provide two interpreter functions **`MG-MEANING`** and **`MG-MEANING-R`** which define the operational semantics of Micro-Gypsy. These functions are identical except that one of them takes into account the resource limitations of the target machine and the other does not. An important result we prove is that in the absence of resource errors these functions are equivalent. We first discuss **`MG-MEANING`** which does not handle resource errors and then treat resource errors and **`MG-MEANING-R`** separately.

3.4.1 The Context of a Statement

A statement is meaningful in a context consisting of the following arguments to **`MG-MEANING`**.

- **`MG-STATE`**: the current execution environment consisting of the variable alist, current condition and psw;
- **`PROC-LIST`**: the list of user-defined procedures;
- **`N`**: a natural number "clock".

3.4.1-A The MG-STATE

The current execution environment is represented by an **`MG-STATE`** triple. The semantics of the various Micro-Gypsy statement types is given in terms of their effect on this state. It has three components and is formally defined by the following shell in the Boyer-Moore logic.[14]

---

[14]See Appendix A for a discussion of Boyer-Moore shells.

Shell Definition.
Add the shell **MG-STATE** of **3** arguments, with
recognizer function symbol **MG-STATEP**, and
accessors **MG-ALIST**, **CC** and **MG-PSW**.

The **MG-ALIST** component of the state is a list of triples of the form **<NAME, TYPE, VALUE>**. This list encodes the data component of the current execution environment. Each triple represents a declared variable, its type, and its current value. An important invariant that is maintained by the interpreter is that the **MG-ALIST** is internally consistent--each variable has a legal Micro-Gypsy type and a value which is permissible for that type.

The **CC** component of the state has as its value a litatom which represents the current condition. The typical case is that **CC** has value **'NORMAL**. Any other value has a special significance to the interpreter. Intuitively, signaling a condition (by the Micro-Gypsy **SIGNAL-MG** statement or by other means to be explained shortly) causes execution to flow lexically outward from the site until a handler for the condition is encountered or the program is exited. Crossing a routine boundary may cause a formal to actual condition mapping as explained in Section 3.4.2-G below.

Condition handling is a part of the semantics of the Gypsy language and the **CC** component of the state part of its implementation in the Micro-Gypsy interpreter. In contrast, there are two exceptional situations which can occur in the execution of a program which are not part of the Micro-Gypsy language definition. Timing out of the interpreter "clock" and exhaustion of resources on the target machine cause the **MG-PSW** component of the state to be set to something other than its typical value of **'RUN**. No Micro-Gypsy instruction inspects the psw and there is no way for a Micro-Gypsy program to trap or mask a psw error. The psw is a metatheoretic concept in Micro-Gypsy; it is used to define the language but is not part of the language. Timing out is discussed further in Section 3.4.1-C; treatment of resource errors is considered in Section 3.4.3.

The following is a sample **MG-STATE**:

```
(MG-STATE 'ROUTINEERROR
          ((B BOOLEAN-MG (BOOLEAN-MG TRUE-MG))
           (I INT-MG (INT-MG -24))
           (A (ARRAY-MG CHARACTER-MG 3) ((CHARACTER-MG 78)
                                          (CHARACTER-MG 73)
                                          (CHARACTER-MG 76))))
          'RUN).
```

This state has **CC** equal to **'ROUTINEERROR**, an **MG-PSW** value of **'RUN**, and an **MG-ALIST** in which 3 variables are declared.

3.4.1-B The Procedure List

The **PROC-LIST** argument to the interpreter is exactly the same as that to the recognizer described in Section 3.3. It is required since a legal Micro-Gypsy statement may be a call to a user-defined procedure. The meaning of such a statement is the meaning of the procedure body with the appropriate substitution of actual parameters for formal parameters. To describe this we must be able to fetch the procedure definition from this procedure list.

3.4.1-C The Clock

The clock argument **N** is an artifact of the formal system in which our interpreter is constructed. The

Boyer-Moore logic is a constructive[15] formal logic in which all function definitions are required to be total. The "natural" interpreter semantics of Micro-Gypsy programs would not define a total function since Micro-Gypsy programs can be written which are non-terminating. We force termination of the interpreter by adding the additional argument **`N`** which is decremented in every recursive call. This can loosely be thought of as the maximum number of steps which the interpreter is allowed to execute; however, it is really an accretion to Micro-Gypsy semantics required for technical reasons and has no strong intuitive appeal.[16] If **`N`** is decremented to zero before the interpreter terminates normally the **`MG-PSW`** is set to **`'TIMED-OUT`** and execution terminates.

3.4.2 MG-MEANING

The operational semantics of Micro-Gypsy statements (in the absence of concern over resource limitations) is given by the function **`MG-MEANING`** (see figure 3-2). In general, **`MG-MEANING`** returns the state which results from executing **`STMT`** in the current environment. It is structured primarily as a case split on the various Micro-Gypsy statement types. However, there are two special cases in the interpreter definition.

The clock argument **`N`** is decremented in each recursive call; if it becomes zero (or is a non-number), the current state is returned with **`MG-PSW`** set to **`'TIMED-OUT`**. Since the **`MG-PSW`** is never reset to **`'RUN`**, this "meta-error" condition persists permanently. Our correctness proofs have as an hypothesis that the final **`MG-PSW`** value is **`'RUN`**. A psw value other than **`'RUN`** indicates an aberrant condition and all bets are off. It is clear, however, that for any *terminating* program, there is a value of **`N`** which is large enough to keep the program from timing out. For particular programs, it may be possible to prove that certain values are adequate. This issue is discussed further in Chapter 8.

If the **`CC`** of the current state is anything other than **`'NORMAL`**, the current state is returned. It might seem from this that once a condition is raised it can never be handled. However, the current call to **`MG-MEANING`** may be a recursive call made while interpreting a **`LOOP`** or **`BEGIN`** statement and a handler may be encountered as the recursion unwinds. This is discussed further below.

We now discuss informally the meaning of the Micro-Gypsy statement types as defined by **`MG-MEANING`**. For each statement type, the reader is advised to scrutinize carefully the corresponding lines in the interpreter definition and investigate the definitions of the subordinate functions.

3.4.2-A NO-OP-MG

The **`NO-OP-MG`** statement has no effect on the state. It is present in the language purely as a convenience in coding such constructs as the single-branch **`IF`** statement.

3.4.2-B SIGNAL-MG

Recall that a **`SIGNAL-MG`** statement has the form **`(SIGNAL-MG SIGNALLED-CONDITION)`**. The effect of the statement is to set the **`CC`** component of the state to **`SIGNALLED-CONDITION`**. If this occurs inside a recursive call to **`MG-MEANING`**, the effect of the non-**`'NORMAL`** current condition will cause the recursion to

---

[15]The logic has recently been extended with the addition of partial functions and bounded quantification so that it is not strictly constructive. However, we make no use of these features and the version of the prover used in our work does not include them.

[16]The most recent release of the Boyer-Moore system [BoyerMoore 88] allows partial functions and its use would obviate this device. However, proofs are typically more difficult in the extended logic.

DEFINITION.

```
(MG-MEANING STMT PROC-LIST MG-STATE N)
   =
(IF (ZEROP N)
    (SIGNAL-SYSTEM-ERROR MG-STATE 'TIMED-OUT)

(IF (NOT (NORMAL MG-STATE))
    MG-STATE

(CASE (CAR STMT)
  (NO-OP-MG   MG-STATE)
  (SIGNAL-MG (SET-CONDITION MG-STATE (SIGNALLED-CONDITION STMT)))
  (PROG2-MG
    (MG-MEANING (PROG2-RIGHT-BRANCH STMT) PROC-LIST
                (MG-MEANING (PROG2-LEFT-BRANCH STMT)
                            PROC-LIST MG-STATE (SUB1 N))
                (SUB1 N)))
  (LOOP-MG
    (REMOVE-LEAVE (MG-MEANING STMT PROC-LIST
                              (MG-MEANING (LOOP-BODY STMT) PROC-LIST
                                          MG-STATE (SUB1 N))
                              (SUB1 N))))
  (IF-MG
    (IF (MG-EXPRESSION-FALSEP (IF-CONDITION STMT) MG-STATE)
        (MG-MEANING (IF-FALSE-BRANCH STMT) PROC-LIST MG-STATE (SUB1 N))
        (MG-MEANING (IF-TRUE-BRANCH STMT) PROC-LIST MG-STATE (SUB1 N))))
  (BEGIN-MG
    (IF (MEMBER (CC (MG-MEANING (BEGIN-BODY STMT)
                                PROC-LIST MG-STATE (SUB1 N)))
                (WHEN-LABELS STMT))
        (MG-MEANING (WHEN-HANDLER STMT) PROC-LIST
                    (SET-CONDITION (MG-MEANING (BEGIN-BODY STMT)
                                               PROC-LIST MG-STATE (SUB1 N))
                                   'NORMAL)
                    (SUB1 N))
        (MG-MEANING (BEGIN-BODY STMT) PROC-LIST MG-STATE (SUB1 N))))
  (PROC-CALL-MG
    (MAP-CALL-EFFECTS
     (MG-MEANING (DEF-BODY (FETCH-CALLED-DEF STMT PROC-LIST))
                 PROC-LIST
                 (MAKE-CALL-ENVIRONMENT MG-STATE STMT
                                        (FETCH-CALLED-DEF STMT PROC-LIST))
                 (SUB1 N))
     (FETCH-CALLED-DEF STMT PROC-LIST)
     STMT
     MG-STATE))
  (PREDEFINED-PROC-CALL-MG
    (MG-MEANING-PREDEFINED-PROC-CALL STMT MG-STATE))

  (OTHERWISE  MG-STATE))))
```

Figure 3-2: MG-MEANING

be unwound until a handler for **SIGNALLED-CONDITION** is encountered or the program is exited. Syntactic requirements prevent the signaling of **'NORMAL**.

3.4.2-C PROG2-MG

A **PROG2-MG** statement has the form **(PROG2-MG LEFT-BRANCH RIGHT-BRANCH)**. **PROG2-MG** is the explicit sequencing operator of Micro-Gypsy and its presence in the language eliminates the need for reasoning about arbitrary lists of statements.[17]

Executing a **PROG2-MG** statement merely means executing the right branch in the state resulting from executing the left branch. Notice that the interpreter structure is such that if the left branch has a non-normal condition, the right branch is semantically a no-op.

3.4.2-D LOOP-MG

The Micro-Gypsy **LOOP-MG** statement has the form **(LOOP-MG LOOP-BODY)**. The loop is executed repetitively until something causes it to terminate. Termination occurs either because the clock argument **N** has been decremented to zero causing execution to time-out or because some condition has been raised.

In Gypsy, "clean" termination of a loop is handled by the **LEAVE** statement as illustrated in the following program fragment:

```
loop
   if N le 0 then leave end; {if}
   FACT := FACT * N;
   N := N - 1;
end; {loop}
```

Execution of the **LEAVE** statement causes termination of the innermost inclosing loop. Rather than have a separate **LEAVE** statement type in Micro-Gypsy this is modeled in the interpreter by treating the predefined condition **'LEAVE** in a special way. Recall that the recognizer adds **'LEAVE** to the list of permissible conditions when recognizing a loop body. A **(SIGNAL-MG LEAVE)** within the loop body "short circuits" the iteration by making the subsequent recursive call a no-op (since the **CC** is not **'NORMAL**). However, **'LEAVE** is unlike other conditions in that we don't want a **'LEAVE** condition to propagate outward until a handler is encountered.[18] We want execution to proceed normally following the loop. This is accomplished by calling the function **REMOVE-LEAVE** (page 53) on the resulting state. **REMOVE-LEAVE** resets the condition to **'NORMAL** if it is **'LEAVE** and otherwise leaves it alone. This has the desired effect of preserving any of the regular conditions while treating a **'LEAVE** exit from the loop in the appropriate fashion.

3.4.2-E IF-MG

The **IF** statement in Micro-Gypsy has form

(IF-MG IF-CONDITION IF-TRUE-BRANCH IF-FALSE-BRANCH).

The **IF-CONDITION** is required to be a Boolean variable defined in the state. If the current value of the variable is **(BOOLEAN-MG FALSE-MG)** then the meaning of the statement is the meaning of the false branch; otherwise, it is the meaning of the true branch.

---

[17]It is trivial for such lists in the input to be translated into a nested sequence of **PROG2-MG**'s by the preprocessor.

[18]In fact, our syntactic restrictions guarantee that **'LEAVE** could never be handled as a regular condition.

3.4.2-F BEGIN-MG

The Micro-Gypsy **BEGIN-MG** statement has form

```
(BEGIN-MG BEGIN-BODY WHEN-LABELS WHEN-HANDLER).
```

It is the Micro-Gypsy analog of the Begin statement in Gypsy, which takes the form

```
BEGIN
   BEGIN-BODY-STATEMENT-LIST
WHEN
   CONDITION-LIST_1: HANDLER-STATEMENT-LIST_1;
   CONDITION-LIST_2: HANDLER-STATEMENT-LIST_2;
   ...
   CONDITION-LIST_N: HANDLER-STATEMENT-LIST_N;
END;
```

and allows the association of condition handlers with statement lists. If any of the members of **CONDITION-LIST**$_1$**, ..., CONDITION-LIST**$_N$ are signaled in executing the begin body, the condition is set to **'NORMAL** and the associated handler is executed. The condition lists are required to be disjoint. The Micro-Gypsy form is syntactically more restrictive allowing only a single list of conditions with a single handler. Cohen [Cohen 86] shows how the more general form can be translated to our restricted form.[19]

If the state resulting from executing the **BEGIN-BODY** has a **CC** which is anything other than one of the conditions in the **WHEN-LABELS** list, that state is returned. Otherwise, one of the designated conditions has been signaled. We reset the **CC** to **'NORMAL** and execute the condition handler in the resulting state. Syntactic restrictions guarantee that **'NORMAL** is not a member of the **WHEN-LABELS** list.

3.4.2-G PROC-CALL-MG

User-defined procedure calls are without question the most difficult part of the Micro-Gypsy semantics. A procedure call has the form

```
(PROC-CALL-MG DATA-ACTUALS COND-ACTUALS).
```

We interpret a call to a user-defined procedure as follows:

1. Fetch the definition of the called procedure from the procedure list.
2. Create a new **MG-ALIST** by binding the formal names to the values of the actuals fetched from the current variable alist. Locals are given their initial values from the procedure definition. The current condition is necessarily **'NORMAL** at this point. The **MG-PSW** of the calling environment is preserved.
3. Execute the body of the procedure in this new execution environment.
4. Copy the values associated with the formal parameter names on the resulting alist into the corresponding actuals on the alist of the calling environment. Set the current condition in the calling environment according to rules discussed below.

Step 2 is accomplished by the function **MAKE-CALL-ENVIRONMENT** (page 39) which creates an **MG-STATE** for the execution of the procedure body. The execution of the procedure body is simply the recursive call to **MG-MEANING** on the procedure body in the context of the execution environment created in step 2. Copying out of the results is handled by the function **MAP-CALL-EFFECTS**.

---

[19] It is more complicated than simply nesting the **BEGIN-MG** statements since the handlers may themselves raise conditions.

DEFINITION.

```
(MAP-CALL-EFFECTS BODY-RESULT DEF STMT CALLING-ENVIRONMENT)
   =
(MG-STATE
    (CONVERT-CONDITION (CC BODY-RESULT)
                       (DEF-CONDS DEF)
                       (CALL-CONDS STMT))
    (COPY-OUT-PARAMS (DEF-FORMALS DEF)
                     (CALL-ACTUALS STMT)
                     (MG-ALIST BODY-RESULT)
                     (MG-ALIST CALLING-ENVIRONMENT))
    (MG-PSW BODY-RESULT)))
```

MAP-CALL-EFFECTS defines the effects of the procedure call on each of the three components of the calling environment: **CC**, **MG-ALIST**, and **MG-PSW**. The **MG-PSW** is the easiest; if the call timed-out or exhausted the system resources the error is reflected back into the calling environment.

The **MG-ALIST** of the calling environment is updated with the effects of the procedure body upon the actual parameters.[20] The function **COPY-OUT-PARAMS** (page 35) maps the new values of the formals in the state resulting from the execution of the body into the actuals in the calling environment.

Finally, the condition resulting from the call must be translated to one appropriate for the calling environment. This is accomplished by the function **CONVERT-CONDITION** (page 35) using the following rules. If the **CC** returned by the execution of the procedure body is one of the formal condition parameters, return the corresponding actual condition parameter. This is always well-defined since the two lists are required by the recognizer to match in length. The only other possibilities syntactically are for the procedure body to return a **CC** value of **'ROUTINEERROR** or one of the local conditions of the procedure definition. For these, return **'ROUTINEERROR**.

Notice that **MG-MEANING** defines a style of semantics for procedure calls which is typically labeled *call by value-result* or *copy-in copy-out* semantics. However, the non-concurrent fragment of Gypsy, and Micro-Gypsy as well, are very carefully defined to insure that call by value-result and call by reference give identical results [Cohen 86]. This is the reason for enforcing very strong aliasing restrictions on parameters to user-defined procedures. Consequently, the code generator is free to use an efficient call-by-reference implementation rather than copying parameters as would be expected from the interpreter definition. This is discussed further in Chapter 5.

3.4.2-H PREDEFINED-PROC-CALL-MG

The weakness of the expression language, which contains only variables and simple literals, requires that Micro-Gypsy have a large collection of predefined procedures. For our prototype implementation only 13 predefined procedures are included. However, because the specification and proof are extremely modular, this set can be extended easily. The semantics of the predefined procedures are defined by the function **MG-MEANING-PREDEFINED-PROC-CALL**, which simply does a case split on the called routine name.

---

[20] If we drew a distinction between **VAR** and **CONST** parameters, this is the point at which the difference would show. We would only have to copy out the **VAR** parameters since **CONST** parameters are guaranteed syntactically to be unchanged by the call. Also, we could allow literals in **CONST** positions rather than insisting that all of the actual parameters be variables. This is conceptually easy but complicates the proof substantially. However, it is also easy for the preprocessor to translate literal parameters into variable references after the parser has checked that **CONST** positions are unchanged in the code. This makes our restrictions on the form of the parameter list less onerous than they might at first appear.

DEFINITION.
```
(MG-MEANING-PREDEFINED-PROC-CALL STMT MG-STATE)
   =
(CASE (CALL-NAME STMT)
      (MG-SIMPLE-VARIABLE-ASSIGNMENT
          (MG-MEANING-MG-SIMPLE-VARIABLE-ASSIGNMENT STMT MG-STATE))
      (MG-SIMPLE-CONSTANT-ASSIGNMENT
          (MG-MEANING-MG-SIMPLE-CONSTANT-ASSIGNMENT STMT MG-STATE))
      ...
      (MG-ARRAY-ELEMENT-ASSIGNMENT
          (MG-MEANING-MG-ARRAY-ELEMENT-ASSIGNMENT STMT MG-STATE))
      (OTHERWISE MG-STATE)))
```

The semantics of each of the predefined operations is defined in turn by a Boyer-Moore function. The semantics of the predefined operation *predefined-name* is characterized by the function **MG-MEANING-***predefined-name*.

Consider, for example, the **MG-SIMPLE-VARIABLE-ASSIGNMENT** operation. The statement

```
(PREDEFINED-PROC-CALL-MG MG-SIMPLE-VARIABLE-ASSIGNMENT (X Y))
```

is the Micro-Gypsy abstract prefix analog of the Gypsy statement **X := Y**, where **X** and **Y** are variables of the same simple type. The meaning of this operation is defined by the function **MG-MEANING-MG-SIMPLE-VARIABLE-ASSIGNMENT**.

DEFINITION.
```
(MG-MEANING-MG-SIMPLE-VARIABLE-ASSIGNMENT STMT MG-STATE)
   =
(MG-STATE 'NORMAL
          (SET-ALIST-VALUE (CAR (CALL-ACTUALS STMT))
                           (GET-M-VALUE (CADR (CALL-ACTUALS STMT))
                                        (MG-ALIST MG-STATE))
                           (MG-ALIST MG-STATE))
          (MG-PSW MG-STATE))
```

which describes the state resulting from the execution of the simple assignment. The resulting state contains these components.

- **CC** is set to **'NORMAL**; no errors occur from a simple assignment.
- **MG-ALIST** has its value from the calling environment except that **X** now has the value of **Y** in the calling environment.
- **MG-PSW** has the same value as in the calling environment.

A more involved example is the predefined operation **MG-INDEX-ARRAY**. The Micro-Gypsy statement

```
(PREDEFINED-PROC-CALL-MG MG-INDEX-ARRAY (X A I N))
```

is semantically equivalent to the Gypsy statement **X := A[I]**, where **N** is the size of the array.[21] The semantics is defined by the function **MG-MEANING-MG-INDEX-ARRAY**.

[21]**N** is supplied as an additional argument for the sake of the code generator. The translator must know the array size to lay down code for range checking on the index. This is the one place where the translator needs type information. Passing **N** as an extra parameter on the two predefined operations involving array indexing eliminates passing the entire variable alist to the translator. This array size argument can easily be supplied by the preprocessor.

DEFINITION.

```
(MG-MEANING-MG-INDEX-ARRAY STMT MG-STATE)
   =
(LET ((INDEX (UNTAG (GET-M-VALUE (CADDR (CALL-ACTUALS STMT))
                                 (MG-ALIST MG-STATE)))))
  (IF (AND (NUMBERP INDEX)
           (LESSP INDEX
                  (ARRAY-LENGTH (GET-M-TYPE (CADR (CALL-ACTUALS STMT))
                                            (MG-ALIST MG-STATE)))))
      (MG-STATE 'NORMAL
                (SET-ALIST-VALUE
                    (CAR (CALL-ACTUALS STMT))
                    (FETCH-ARRAY-ELEMENT (CADR (CALL-ACTUALS STMT))
                                         INDEX
                                         (MG-ALIST MG-STATE))
                    (MG-ALIST MG-STATE))
                (MG-PSW MG-STATE))
    (SET-CONDITION MG-STATE 'ROUTINEERROR)))
```

Since **`I`** is an integer variable, its value in the current state will be a tagged integer literal of the form **`(INT-MG M)`**. Let **`INDEX`** be its untagged value. We check to see that **`INDEX`** is both a natural number and is less than the size of **`A`**, which we obtain from the type information on the **`MG-ALIST`**.[22] If not, then return with the condition **`'ROUTINEERROR`**. If so, then we return with a normal state in which the variable **`X`** has been set to the value of **`A[I]`**.

3.4.3 Handling Resource Errors

Each of the aspects of the Micro-Gypsy semantics discussed thus far is independent of the particular machine upon which Micro-Gypsy is implemented.[23] This is not true of the treatment of resource errors. Handling of resource errors at the level of the Micro-Gypsy interpreter requires wrestling with some issues that are very much tied to the implementation because it is the limitations of the target machine which result in resource errors such as overflow of the evaluation stack.

Recalling that we wish to prove that the translation preserves the semantics of the Micro-Gypsy program, there are several ways in which resource errors could be addressed in the Micro-Gypsy interpreter.

1. We could ignore them entirely. This would leave us vulnerable to the criticism that our code generator is a "toy" which does not address one of the key issues which real compilers must face. We feel that any realistic implementation must take into account the fact that there are finite resources available on the target machine.
2. We could treat resource errors at the Micro-Gypsy level as potential but essentially unpredictable effects of certain Micro-Gypsy operations. Some operations, procedure calls for example, could result in resource errors at any time. This approach abstracts the Micro-Gypsy semantics from the implementation. The proof is predicated on the assumption that no resource errors occur; in the presence of resource errors, all bets are off. This is similar to the semantics underlying Gypsy **`SPACEERROR`**.
3. Another approach is to make the resource limitations of the target machine visible to the Micro-Gypsy interpreter. The interpreter tracks the resource utilization of the program being interpreted and sets a flag in those cases where the resources would be exhausted. This flag is

---

[22] We could have used the fourth argument since this is guaranteed to be the array size. However, this argument was added solely for the benefit of the implementation and we preferred not to clutter up the semantics by reference to it.

[23] With some small exceptions. The meanings of the predefined arithmetic procedures, for example, check that the results can be stored in a machine word. The word size is defined to be 32-bits because that is the word-size of the FM8502 upon which Piton is implemented.

set in the Micro-Gypsy world exactly when resource errors would occur in the world of the Piton implementation. The proof of correctness is predicated on the assumption that this flag is never set.

It is the final approach which is followed in this research. It has the apparent disadvantage of adding an unfortunate amount of clutter to the semantic definition. The interpreter must know how much of each type of resource each Micro-Gypsy statement type consumes in the implementation and keep track of resource utilization in recursive calls. However, we will show that the "uncluttered" definition is adequate under the assumption that no resource errors occur.

We are able to characterize the occasions when resource errors occur fairly cleanly because our target language is Piton, which has well defined resource limitations and explicitly formalizable resource requirements for each of the operations of the target machine. A different implementation could require significantly revamping the interpreter.

3.4.4 MG-MEANING-R

In the remainder of this section we describe the function **MG-MEANING-R**, the interpreter for Micro-Gypsy which takes account of resource limitations of the Piton machine. This function is very closely related to **MG-MEANING** discussed in Section 3.4. An important theorem discussed below relates the two interpreters.

The overall structure of **MG-MEANING-R** is illustrated in figure 3-3. The similarity of **MG-MEANING-R** to **MG-MEANING** is apparent. In fact, **MG-MEANING-R** is exactly **MG-MEANING** with one additional argument **SIZES** and some additional mechanism to track the resource utilization of the program being interpreted.

The **SIZES** argument to **MG-MEANING-R** is intended to be a pair **<T-SIZE, C-SIZE>** representing the current sizes of the two critical resources in Piton which may be exhausted by the execution of the Piton image of a Micro-Gypsy program.[24] These are the Piton temporary stack and Piton control stack discussed in Chapter 4. Since we store the *current* sizes rather than the available space, to compute the amount of available stack space requires knowing the maximum sizes of the two resources. For our purposes it suffices that there be two constants of unspecified value which represent these maximum values. We declare these constants **MG-MAX-CTRL-STK-SIZE** and **MG-MAX-TEMP-STK-SIZE**, and specify with axioms that their values are numbers in the range [0...2^{32}-1].

For any statement type, resources are inadequate if the amount required for the current statement is less than the maximum amount minus the amount currently in use. Formally, this computation is defined as follows.

DEFINITION.
```
(RESOURCES-INADEQUATEP STMT PROC-LIST SIZE-PAIR)
    =
(OR (NOT (LESSP (TEMP-STK-REQUIREMENTS STMT PROC-LIST)
                (DIFFERENCE (MG-MAX-TEMP-STK-SIZE) (T-SIZE SIZE-PAIR))))
    (NOT (LESSP (CTRL-STK-REQUIREMENTS STMT PROC-LIST)
                (DIFFERENCE (MG-MAX-CTRL-STK-SIZE) (C-SIZE SIZE-PAIR)))))
```

Computation of the resource requirements of the various statement types is very closely tied to the particular implementation. It happens, for instance, in our implementation that a **SIGNAL-MG** statement requires that one item be pushed onto the Piton **TEMP-STK** and none onto the **CTRL-STK**. Hence the

---

[24]We ignore resources which have to do with *loading* rather than *executing* programs. For example, the translation of a legal Micro-Gypsy program may be too large to fit into the memory of the FM8502. Piton allows us to address this question by compiling the program and seeing if the resulting Piton code meets a particular set of "loadability" constraints. These are discussed in the Piton report [Moore 88] and briefly in Chapter 8.

DEFINITION.

```
(MG-MEANING-R STMT PROC-LIST MG-STATE N SIZES)
   =
(IF (ZEROP N)
    (SIGNAL-SYSTEM-ERROR MG-STATE 'TIMED-OUT)

(IF (NOT (NORMAL MG-STATE))
    MG-STATE

(IF (RESOURCES-INADEQUATEP STMT PROC-LIST SIZES)
    (SIGNAL-SYSTEM-ERROR MG-STATE 'RESOURCE-ERROR)

(CASE (CAR STMT)
  (NO-OP-MG MG-STATE)

  (SIGNAL-MG (SET-CONDITION MG-STATE (SIGNALLED-CONDITION STMT)))

  (PROG2-MG
    (MG-MEANING-R (PROG2-RIGHT-BRANCH STMT) PROC-LIST
                  (MG-MEANING-R (PROG2-LEFT-BRANCH STMT)
                                PROC-LIST MG-STATE (SUB1 N) SIZES)
                  (SUB1 N)
                  SIZES))

  (LOOP-MG
    (REMOVE-LEAVE (MG-MEANING-R STMT PROC-LIST
                                (MG-MEANING-R (LOOP-BODY STMT) PROC-LIST
                                              MG-STATE (SUB1 N) SIZES)
                                (SUB1 N)
                                SIZES)))

  (IF-MG
   (IF (MG-EXPRESSION-FALSEP (IF-CONDITION STMT) MG-STATE)
       (MG-MEANING-R (IF-FALSE-BRANCH STMT) PROC-LIST MG-STATE (SUB1 N) SIZES)
     (MG-MEANING-R (IF-TRUE-BRANCH STMT) PROC-LIST MG-STATE (SUB1 N) SIZES)))

  (BEGIN-MG
   (IF (MEMBER (CC (MG-MEANING-R (BEGIN-BODY STMT) PROC-LIST
                                 MG-STATE (SUB1 N) SIZES))
               (WHEN-LABELS STMT))
       (MG-MEANING-R
        (WHEN-HANDLER STMT) PROC-LIST
        (SET-CONDITION (MG-MEANING-R (BEGIN-BODY STMT)
                                     PROC-LIST MG-STATE (SUB1 N) SIZES)
                       'NORMAL)
        (SUB1 N)
        SIZES)
     (MG-MEANING-R (BEGIN-BODY STMT) PROC-LIST MG-STATE (SUB1 N) SIZES)))
```

Figure 3-3: MG-MEANING-R

```
(PROC-CALL-MG
  (MAP-CALL-EFFECTS
    (MG-MEANING-R
     (DEF-BODY (FETCH-CALLED-DEF STMT PROC-LIST))
     PROC-LIST
     (MAKE-CALL-ENVIRONMENT MG-STATE STMT
                           (FETCH-CALLED-DEF STMT PROC-LIST))
     (SUB1 N)
     (LIST (PLUS (T-SIZE SIZES)
                 (DATA-LENGTH (DEF-LOCALS
                               (FETCH-CALLED-DEF STMT PROC-LIST))))
           (PLUS (C-SIZE SIZES)
                 2
                 (LENGTH (DEF-LOCALS (FETCH-CALLED-DEF STMT PROC-LIST)))
                 (LENGTH (DEF-FORMALS
                           (FETCH-CALLED-DEF STMT PROC-LIST))))))
    (FETCH-CALLED-DEF STMT PROC-LIST)
    STMT
    MG-STATE))

(PREDEFINED-PROC-CALL-MG
  (MG-MEANING-PREDEFINED-PROC-CALL STMT MG-STATE))

(OTHERWISE  MG-STATE)))))
```

Figure 3-3, concluded

temp-stk requirement for **SIGNAL-MG** is 1 and the ctrl-stk requirement 0. Another implementation might have very different resource requirements.

Figure 3-4 summarizes the temp-stk and ctrl-stk requirements of the various Micro-Gypsy statement types, including the various predefined procedure calls.

```
  Statement type               temp-stk              ctrl-stk
                             requirements          requirements

NO-OP-MG                          0                     0
SIGNAL-MG                         1                     0
PROG2-MG                          0                     0
LOOP-MG                           1                     0
IF-MG                             1                     0
BEGIN-MG                          1                     0
PROC-CALL-MG            max (1, locals-data-size      locals-length
                              + locals-length        + formals-length
                              + actuals-length)      + 2
PREDEFINEDP-PROC-CALL-MG
  MG-SIMPLE-VARIABLE-ASSIGNMENT   2                     4
  MG-SIMPLE-CONSTANT-ASSIGNMENT   2                     4
  MG-SIMPLE-VARIABLE-EQ           3                     5
  MG-SIMPLE-CONSTANT-EQ           3                     5
  MG-INTEGER-LE                   3                     5
  MG-INTEGER-UNARY-MINUS          2                     6
  MG-INTEGER-ADD                  3                     6
  MG-INTEGER-SUBTRACT             3                     6
  MG-BOOLEAN-OR                   3                     5
  MG-BOOLEAN-AND                  3                     5
  MG-BOOLEAN-NOT                  2                     5
  MG-INDEX-ARRAY                  4                     7
  MG-ARRAY-ELEMENT-ASSIGNMENT     4                     7
```

Figure 3-4: Temp-Stk and Ctrl-Stk Requirements

The reasons for these numbers will only become clear after we discuss the translation of Micro-Gypsy statements to Piton in Chapter 5. They can be thought of as computing the maximum number of items placed on the stack at any time by the execution of the statement.

Notice that the resource requirements we compute are only those required by the current statement type. A **PROG2-MG** statement, for example, requires no stack space, even though its two branches may require an arbitrary amount. This is adequate since resource errors always propagate; if a resource error occurs anywhere within a computation, it will persist for the remainder of the computation.[25] Our main theorem has as a hypothesis that no such errors occur.

Looking again at **MG-MEANING-R** (page 32) we see that if available resources are inadequate to execute the current statement, the **MG-PSW** is set to **'RESOURCE-ERROR**. The **SIZES** argument lists the amount of temp-stk and ctrl-stk space currently in use. These are adjusted in recursive calls to account for additional stack space being held when the recursive call is entered. Only user-defined procedure calls utilize resources which are held at the time the recursive call is made. Procedure calls use temp-stk space for storing local data and passing parameters; a frame is pushed onto the ctrl-stk defining the execution environment for procedure body. Thus, user defined procedure calls pass an altered **<T-SIZE, C-SIZE>**

[25]Actually, it is more accurate to say that *some* resource error will persist. **'TIMED-OUT** and **'RESOURCE-ERROR** are both considered resource errors in this sense. If stack space and the clock both run out at some point in a computation, we make no promises about which error determines the final value of the **MG-PSW**. We only care if it is something other than **'RUN**.

pair in the recursive call. The current temp-stk size is increased by the size of the procedure's locals which are pushed onto the temp-stk; the ctrl-stk size is increased by the size of a Piton frame for the routine. In all of the other statement types, **SIZES** is passed unchanged to the recursive calls within **MG-MEANING-R**.

The definition of **MG-MEANING-R** is somewhat undesirable as a semantic definition for Micro-Gypsy since it is so tied to the Piton implementation. Its similarity to the definition of **MG-MEANING** suggests that in some cases **MG-MEANING** may suffice. The following theorem provides a very useful result about the relation between the two interpreter definitions.

```
THEOREM.  MG-MEANING-EQUIVALENCE
(IMPLIES (NOT (RESOURCE-ERRORP (MG-MEANING-R STMT PROC-LIST
                                             MG-STATE N SIZES)))
         (EQUAL (MG-MEANING-R STMT PROC-LIST MG-STATE N SIZES)
                (MG-MEANING STMT PROC-LIST MG-STATE N)))
```

Since we are most often interested in cases where no resource errors occur, this theorem permits us to replace calls to **MG-MEANING-R** by calls to **MG-MEANING** in many cases of interest. We use this result heavily in our proof.

It could be argued that our overall approach to resource errors is not entirely satisfying. Our assertion that the software runs correctly *if no resource errors occur* would be unsatisfactory if we were verifying a pacemaker or SDI, say. It would be straightforward to compute, for particular execution environments, the maximum resource utilization and to prove that resource errors will not occur. This issue is discussed at more length in Chapter8.

3.5 Alphabetical Listing of the Micro-Gypsy Definition

This section contains the complete formal definition of the Micro-Gypsy recognizer and interpreter along with the necessary subordinate functions. This list is complete in the sense that it can be "run" in the Kaufmann-enhanced Boyer-Moore theorem prover described in Appendix A with all definitions being accepted. The only prerequisite is that it be reordered such that functions are defined before they are referenced. The alphabetical ordering is merely a convenience for the reader.

```
DEFINITION.
(APPEND X Y)
   =
(IF (LISTP X)
    (CONS (CAR X) (APPEND (CDR X) Y))
    Y)

DEFINITION.
(ARRAY-ELEMTYPE TYPE) = (CADR TYPE)

DEFINITION.
(ARRAY-IDENTIFIERP NAME ALIST)
   =
(AND (DEFINED-IDENTIFIERP NAME ALIST)
     (HAS-ARRAY-TYPE NAME ALIST))

DEFINITION.
(ARRAY-LENGTH TYPE) = (CADDR TYPE)

DEFINITION.
(ARRAY-LITERALP EXP LENGTH ELEMTYPE)
   =
(AND (SIMPLE-TYPED-LITERAL-PLISTP EXP ELEMTYPE)
     (EQUAL (LENGTH EXP) LENGTH))
```

DEFINITION.
```
(ARRAY-MG-TYPE-REFP TYPREF)
   =
(AND (LENGTH-PLISTP TYPREF 3)
     (EQUAL (CAR TYPREF) 'ARRAY-MG)
     (SIMPLE-MG-TYPE-REFP (ARRAY-ELEMTYPE TYPREF))
     (NOT (ZEROP (ARRAY-LENGTH TYPREF))))
```

DEFINITION.
```
(ASSOC X Y)
   =
(IF (LISTP Y)
    (IF (EQUAL X (CAAR Y))
        (CAR Y)
        (ASSOC X (CDR Y)))
    F)
```

DEFINITION.
```
(BEGIN-BODY STMT) = (CADR STMT)
```

DEFINITION.
```
(BOOLEAN-IDENTIFIERP NAME ALIST)
   =
(AND (IDENTIFIERP NAME)
     (EQUAL (GET-M-TYPE NAME ALIST)
            'BOOLEAN-MG))
```

DEFINITION.
```
(BOOLEAN-LITERALP EXP)
   =
(AND (EQUAL 'BOOLEAN-MG (CAR EXP))
     (LENGTH-PLISTP EXP 2)
     (MEMBER (CADR EXP)
             '(TRUE-MG FALSE-MG)))
```

DEFINITION.
```
(C-SIZE X) = (CADR X)
```

DEFINITION.
```
(CALL-ACTUALS STMT) = (CADDR STMT)
```

DEFINITION.
```
(CALL-CONDS STMT) = (CADDDR STMT)
```

DEFINITION.
```
(CALL-NAME STMT) = (CADR STMT)
```

DEFINITION.
```
(CHARACTER-IDENTIFIERP NAME ALIST)
   =
(AND (IDENTIFIERP NAME)
     (EQUAL (GET-M-TYPE NAME ALIST)
            'CHARACTER-MG))
```

DEFINITION.
```
(CHARACTER-LITERALP EXP)
   =
(AND (EQUAL 'CHARACTER-MG (CAR EXP))
     (LENGTH-PLISTP EXP 2)
     (NUMBERP (CADR EXP))
     (IF (LESSP 127 (CADR EXP)) F T))
```

DEFINITION.
```
(COLLECT-LOCAL-NAMES DEF)
   =
(APPEND (LISTCARS (DEF-FORMALS DEF))
        (LISTCARS (DEF-LOCALS DEF)))
```

DEFINITION.
```
(COND-IDENTIFIER-PLISTP LST COND-LIST)
   =
(IF (NLISTP LST)
    (EQUAL LST NIL)
    (AND (COND-IDENTIFIERP (CAR LST) COND-LIST)
         (COND-IDENTIFIER-PLISTP (CDR LST)
                                 COND-LIST)))
```

DEFINITION.
```
(COND-IDENTIFIERP X COND-LIST)
   =
(OR (EQUAL X 'ROUTINEERROR)
    (AND (IDENTIFIERP X)
         (MEMBER X COND-LIST)))
```

DEFINITION.
```
(COND-PARAMS-MATCH COND-ACTUALS CONDS)
   =
(EQUAL (LENGTH COND-ACTUALS)
       (LENGTH CONDS))
```

DEFINITION.
```
(CONVERT-CONDITION COND FORMALS ACTUALS)
   =
(IF (MEMBER COND '(NORMAL ROUTINEERROR))
    COND
    (CONVERT-CONDITION1 COND FORMALS ACTUALS))
```

DEFINITION.
```
(CONVERT-CONDITION1 COND FORMALS ACTUALS)
   =
(COND ((NLISTP FORMALS) 'ROUTINEERROR)
      ((EQUAL COND (CAR FORMALS))
       (CAR ACTUALS))
      (T (CONVERT-CONDITION1 COND
                             (CDR FORMALS)
                             (CDR ACTUALS))))
```

DEFINITION.
```
(COPY-OUT-PARAMS FORMALS ACTUALS NEW-VAR-ALIST OLD-VAR-ALIST)
   =
(IF (NLISTP FORMALS)
    OLD-VAR-ALIST
    (COPY-OUT-PARAMS
     (CDR FORMALS)
     (CDR ACTUALS)
     NEW-VAR-ALIST
     (SET-ALIST-VALUE (CAR ACTUALS)
                      (CADDR (ASSOC (CAAR FORMALS) NEW-VAR-ALIST))
                      OLD-VAR-ALIST)))
```

DEFINITION.
```
(CTRL-STK-REQUIREMENTS STMT PROC-LIST)
   =
(CASE
 (CAR STMT)
 (NO-OP-MG 0)
 (SIGNAL-MG 0)
 (PROG2-MG 0)
 (LOOP-MG 0)
 (IF-MG 0)
 (BEGIN-MG 0)
 (PROC-CALL-MG
  (PLUS 2
        (LENGTH (DEF-LOCALS (FETCH-CALLED-DEF STMT PROC-LIST)))
        (LENGTH (DEF-FORMALS (FETCH-CALLED-DEF STMT PROC-LIST)))))
 (PREDEFINED-PROC-CALL-MG
  (PREDEFINED-PROC-CALL-P-FRAME-SIZE (CALL-NAME STMT)))
 (OTHERWISE 0))
```

DEFINITION.
```
(DATA-LENGTH LOCALS)
   =
(COND ((NLISTP LOCALS) 0)
      ((SIMPLE-MG-TYPE-REFP (CADAR LOCALS))
       (ADD1 (DATA-LENGTH (CDR LOCALS))))
      (T (PLUS (ARRAY-LENGTH (CADAR LOCALS))
               (DATA-LENGTH (CDR LOCALS))))))
```

DEFINITION.
```
(DATA-PARAM-LISTS-MATCH ACTUALS FORMALS ALIST)
   =
(IF (OR (NLISTP ACTUALS) (NLISTP FORMALS))
    (AND (EQUAL FORMALS NIL)
         (EQUAL ACTUALS NIL))
    (AND (DATA-PARAMS-MATCH (CAR ACTUALS)
                            (CAR FORMALS)
                            ALIST)
         (DATA-PARAM-LISTS-MATCH (CDR ACTUALS)
                                 (CDR FORMALS)
                                 ALIST)))
```

DEFINITION.
```
(DATA-PARAMS-MATCH ACTUAL FORMAL ALIST)
   =
(OK-IDENTIFIER-ACTUAL ACTUAL FORMAL ALIST)
```

DEFINITION.
```
(DEF-BODY DEF) = (CADDDDDR DEF)
```

DEFINITION.
```
(DEF-COND-LOCALS DEF) = (CADDDDR DEF)
```

DEFINITION.
```
(DEF-CONDS DEF) = (CADDR DEF)
```

DEFINITION.
```
(DEF-FORMALS DEF) = (CADR DEF)
```

DEFINITION.
```
(DEF-LOCALS DEF) = (CADDDR DEF)
```

DEFINITION.
```
(DEF-NAME DEF) = (CAR DEF)
```

DEFINITION.
```
(DEFINED-IDENTIFIERP NAME ALIST)
   =
(AND (IDENTIFIERP NAME)
     (DEFINEDP NAME ALIST))
```

DEFINITION.
```
(DEFINED-PROCP NAME PROC-LIST)
   =
(OR (PREDEFINED-PROCP NAME)
    (USER-DEFINED-PROCP NAME PROC-LIST))
```

DEFINITION.
```
(DEFINEDP NAME ALIST)
   =
(COND ((NLISTP ALIST) F)
      ((EQUAL NAME (CAAR ALIST)) T)
      (T (DEFINEDP NAME (CDR ALIST))))
```

DEFINITION.
```
(EXP X Y)
   =
(IF (ZEROP Y)
    1
    (TIMES X (EXP X (SUB1 Y))))
```

DEFINITION.
```
(FETCH-ARRAY-ELEMENT A I ALIST)
   =
(GET I (CADDR (ASSOC A ALIST)))
```

DEFINITION.
```
(FETCH-CALLED-DEF STMT PROC-LIST)
   =
(FETCH-DEF (CALL-NAME STMT)
           PROC-LIST)
```

DEFINITION.
```
(FETCH-DEF NAME PROC-LIST) = (ASSOC NAME PROC-LIST)
```

DEFINITION.
```
(FORMAL-INITIAL-VALUE LOCAL) = (CADDR LOCAL)
```

DEFINITION.
```
(FORMAL-TYPE EXP) = (CADR EXP)
```

DEFINITION.
```
(GET N LST)
   =
(IF (ZEROP N)
    (CAR LST)
    (GET (SUB1 N) (CDR LST)))
```

DEFINITION.
```
(GET-M-TYPE NAME ALIST) = (M-TYPE (ASSOC NAME ALIST))
```

DEFINITION.
```
(GET-M-VALUE NAME ALIST) = (M-VALUE (ASSOC NAME ALIST))
```

DEFINITION.
```
(HAS-ARRAY-TYPE NAME ALIST)
   =
(EQUAL (CAR (GET-M-TYPE NAME ALIST))
       'ARRAY-MG)
```

DEFINITION.
```
(IDENTIFIER-PLISTP LST)
   =
(IF (NLISTP LST)
    (EQUAL LST NIL)
    (AND (IDENTIFIERP (CAR LST))
         (IDENTIFIER-PLISTP (CDR LST))))
```

DEFINITION.
```
(IDENTIFIERP NAME) = (OK-MG-NAMEP NAME)
```

DEFINITION.
```
(IDIFFERENCE I J)
   =
(IPLUS I (INEGATE J))
```

DEFINITION.
```
(IF-CONDITION STMT) = (CADR STMT)
```

DEFINITION.
```
(IF-FALSE-BRANCH STMT) = (CADDDR STMT)
```

DEFINITION.
```
(IF-TRUE-BRANCH STMT) = (CADDR STMT)
```

DEFINITION.
```
(ILEQ X Y) = (NOT (ILESSP Y X))
```

DEFINITION.

```
(ILESSP X Y)
   =
(COND ((NEGATIVEP X)
       (IF (NEGATIVEP Y)
           (LESSP (NEGATIVE-GUTS Y)
                  (NEGATIVE-GUTS X))
           T))
      ((NEGATIVEP Y) F)
      (T (LESSP X Y)))
```

DEFINITION.

```
(INEGATE I)
   =
(COND ((NEGATIVEP I) (NEGATIVE-GUTS I))
      ((ZEROP I) 0)
      (T (MINUS I)))
```

DEFINITION.

```
(INT-IDENTIFIERP NAME ALIST)
   =
(AND (IDENTIFIERP NAME)
     (EQUAL (GET-M-TYPE NAME ALIST)
            'INT-MG))
```

DEFINITION.

```
(INT-LITERALP EXP)
   =
(AND (EQUAL 'INT-MG (CAR EXP))
     (LENGTH-PLISTP EXP 2) (SMALL-INTEGERP (CADR EXP) (MG-WORD-SIZE)))
```

DEFINITION.

```
(INTEGERP X)
   =
(OR (AND (NEGATIVEP X)
         (NOT (EQUAL (NEGATIVE-GUTS X) 0)))
    (NUMBERP X))
```

DEFINITION.

```
(IPLUS I J)
   =
(COND ((NEGATIVEP I)
       (COND ((NEGATIVEP J)
              (MINUS (PLUS (NEGATIVE-GUTS I)
                           (NEGATIVE-GUTS J))))
             ((LESSP J (NEGATIVE-GUTS I))
              (MINUS (DIFFERENCE (NEGATIVE-GUTS I) J)))
             (T (DIFFERENCE J (NEGATIVE-GUTS I)))))
      ((NEGATIVEP J)
       (IF (LESSP I (NEGATIVE-GUTS J))
           (MINUS (DIFFERENCE (NEGATIVE-GUTS J) I))
           (DIFFERENCE I (NEGATIVE-GUTS J))))
      (T (PLUS I J)))
```

DEFINITION.

```
(LENGTH-PLISTP LST N)
   =
(AND (PLISTP LST)
     (EQUAL (LENGTH LST) N))
```

DEFINITION.

```
(LISTCARS LST)
   =
(IF (NLISTP LST)
    NIL
    (CONS (CAAR LST)
          (LISTCARS (CDR LST))))
```

DEFINITION.

```
(LOOP-BODY STMT) = (CADR STMT)
```

DEFINITION.
```
(M-TYPE X) = (CADR X)
```

DEFINITION.
```
(M-VALUE X) = (CADDR X)
```

DEFINITION.
```
(MAKE-ALIST-FROM-FORMALS LST)
   =
(IF (NLISTP LST)
    NIL
    (CONS (LIST (CAAR LST)
                (FORMAL-TYPE (CAR LST)))
          (MAKE-ALIST-FROM-FORMALS (CDR LST))))
```

DEFINITION.
```
(MAKE-CALL-ENVIRONMENT MG-STATE STMT DEF)
   =
(MG-STATE 'NORMAL
          (MAKE-CALL-VAR-ALIST (MG-ALIST MG-STATE)
                               STMT DEF)
          (MG-PSW MG-STATE))
```

DEFINITION.
```
(MAKE-CALL-PARAM-ALIST FORMALS ACTUALS MG-ALIST)
   =
(IF (NLISTP FORMALS)
    NIL
    (CONS (LIST (CAAR FORMALS)
                (CADAR FORMALS)
                (CADDR (ASSOC (CAR ACTUALS) MG-ALIST)))
          (MAKE-CALL-PARAM-ALIST (CDR FORMALS)
                                 (CDR ACTUALS)
                                 MG-ALIST)))
```

DEFINITION.
```
(MAKE-CALL-VAR-ALIST MG-ALIST STMT DEF)
   =
(APPEND (MAKE-CALL-PARAM-ALIST (DEF-FORMALS DEF)
                               (CALL-ACTUALS STMT)
                               MG-ALIST)
        (DEF-LOCALS DEF))
```

DEFINITION.
```
(MAKE-COND-LIST DEF)
   =
(APPEND (DEF-CONDS DEF)
        (DEF-COND-LOCALS DEF))
```

DEFINITION.
```
(MAKE-NAME-ALIST DEF)
   =
(APPEND (MAKE-ALIST-FROM-FORMALS (DEF-FORMALS DEF))
        (MAKE-ALIST-FROM-FORMALS (DEF-LOCALS DEF)))
```

DEFINITION.
```
(MAP-CALL-EFFECTS NEW-STATE DEF STMT OLD-STATE)
   =
(MG-STATE (CONVERT-CONDITION (CC NEW-STATE)
                             (DEF-CONDS DEF)
                             (CALL-CONDS STMT))
          (COPY-OUT-PARAMS (DEF-FORMALS DEF)
                           (CALL-ACTUALS STMT)
                           (MG-ALIST NEW-STATE)
                           (MG-ALIST OLD-STATE))
          (MG-PSW NEW-STATE))
```

DEFINITION.
```
(MAX X Y) = (IF (LESSP X Y) Y X)
```

DEFINITION.
```
(MAXINT) = (SUB1 (EXP 2 (SUB1 (MG-WORD-SIZE))))
```

DEFINITION.
```
(MG-ALIST-ELEMENTP X)
   =
(AND (LENGTH-PLISTP X 3)
     (OK-MG-NAMEP (CAR X))
     (MG-TYPE-REFP (M-TYPE X))
     (OK-MG-VALUEP (M-VALUE X)
                   (M-TYPE X)))
```

DEFINITION.
```
(MG-ALISTP LST)
   =
(IF (NLISTP LST)
    (EQUAL LST NIL)
    (AND (MG-ALIST-ELEMENTP (CAR LST))
         (MG-ALISTP (CDR LST))))
```

DEFINITION.
```
(MG-AND-BOOL X Y)
   =
(IF (EQUAL X 'FALSE-MG)
    'FALSE-MG
    Y)
```

DEFINITION.
```
(MG-BOOL X)
   =
(TAG 'BOOLEAN-MG
     (IF X 'TRUE-MG 'FALSE-MG))
```

DEFINITION.
```
(MG-EXPRESSION-FALSEP EXP MG-STATE)
   =
(EQUAL (GET-M-VALUE EXP (MG-ALIST MG-STATE))
       '(BOOLEAN-MG FALSE-MG))
```

DECLARATION.
```
(MG-MAX-CTRL-STK-SIZE)
```

DECLARATION.
```
(MG-MAX-TEMP-STK-SIZE)
```

DEFINITION.
```
(MG-MEANING STMT PROC-LIST MG-STATE N)
   =
(COND
 ((ZEROP N)
  (SIGNAL-SYSTEM-ERROR MG-STATE 'TIMED-OUT))
 ((NOT (NORMAL MG-STATE)) MG-STATE)
 ((EQUAL (CAR STMT) 'NO-OP-MG)
  MG-STATE)
 ((EQUAL (CAR STMT) 'SIGNAL-MG)
  (SET-CONDITION MG-STATE
                 (SIGNALLED-CONDITION STMT)))
 ((EQUAL (CAR STMT) 'PROG2-MG)
  (MG-MEANING (PROG2-RIGHT-BRANCH STMT)
              PROC-LIST
              (MG-MEANING (PROG2-LEFT-BRANCH STMT)
                          PROC-LIST MG-STATE
                          (SUB1 N))
              (SUB1 N)))
 ((EQUAL (CAR STMT) 'LOOP-MG)
  (REMOVE-LEAVE (MG-MEANING STMT PROC-LIST
                            (MG-MEANING (LOOP-BODY STMT)
                                        PROC-LIST MG-STATE
                                        (SUB1 N))
                            (SUB1 N))))
```

```
 ((EQUAL (CAR STMT) 'IF-MG)
  (IF (MG-EXPRESSION-FALSEP (IF-CONDITION STMT)
                            MG-STATE)
      (MG-MEANING (IF-FALSE-BRANCH STMT)
                  PROC-LIST MG-STATE
                  (SUB1 N))
      (MG-MEANING (IF-TRUE-BRANCH STMT)
                  PROC-LIST MG-STATE
                  (SUB1 N))))
 ((EQUAL (CAR STMT) 'BEGIN-MG)
  (IF (MEMBER (CC (MG-MEANING (BEGIN-BODY STMT)
                              PROC-LIST MG-STATE
                              (SUB1 N)))
              (WHEN-LABELS STMT))
      (MG-MEANING (WHEN-HANDLER STMT)
                  PROC-LIST
                  (SET-CONDITION (MG-MEANING (BEGIN-BODY STMT)
                                             PROC-LIST MG-STATE
                                             (SUB1 N))
                                 'NORMAL)
                  (SUB1 N))
      (MG-MEANING (BEGIN-BODY STMT)
                  PROC-LIST MG-STATE
                  (SUB1 N))))
 ((EQUAL (CAR STMT) 'PROC-CALL-MG)
  (MAP-CALL-EFFECTS
   (MG-MEANING (DEF-BODY (FETCH-CALLED-DEF STMT PROC-LIST))
               PROC-LIST
               (MAKE-CALL-ENVIRONMENT MG-STATE STMT
                                      (FETCH-CALLED-DEF STMT PROC-LIST))
               (SUB1 N))
   (FETCH-CALLED-DEF STMT PROC-LIST)
   STMT MG-STATE))
 ((EQUAL (CAR STMT)
        'PREDEFINED-PROC-CALL-MG)
  (MG-MEANING-PREDEFINED-PROC-CALL STMT MG-STATE))
 (T MG-STATE))
```

DEFINITION.

```
(MG-MEANING-MG-ARRAY-ELEMENT-ASSIGNMENT STMT MG-STATE)
   =
(IF (AND (NUMBERP (UNTAG (GET-M-VALUE (CADR (CALL-ACTUALS STMT))
                                      (MG-ALIST MG-STATE))))
         (LESSP (UNTAG (GET-M-VALUE (CADR (CALL-ACTUALS STMT))
                                    (MG-ALIST MG-STATE)))
                (ARRAY-LENGTH (GET-M-TYPE (CAR (CALL-ACTUALS STMT))
                                          (MG-ALIST MG-STATE)))))
    (MG-STATE 'NORMAL
              (SET-ALIST-VALUE
               (CAR (CALL-ACTUALS STMT))
               (PUT-ARRAY-ELEMENT (CAR (CALL-ACTUALS STMT))
                                  (UNTAG (GET-M-VALUE (CADR (CALL-ACTUALS STMT))
                                                      (MG-ALIST MG-STATE)))
                                  (GET-M-VALUE (CADDR (CALL-ACTUALS STMT))
                                               (MG-ALIST MG-STATE))
                                  (MG-ALIST MG-STATE))
               (MG-ALIST MG-STATE))
              (MG-PSW MG-STATE))
    (SET-CONDITION MG-STATE 'ROUTINEERROR))
```

DEFINITION.

```
(MG-MEANING-MG-BOOLEAN-AND STMT MG-STATE)
   =
(MG-STATE 'NORMAL
          (SET-ALIST-VALUE
           (CAR (CALL-ACTUALS STMT))
           (TAG 'BOOLEAN-MG
                (MG-AND-BOOL (UNTAG (GET-M-VALUE (CADR (CALL-ACTUALS STMT))
                                                 (MG-ALIST MG-STATE)))
                             (UNTAG (GET-M-VALUE (CADDR (CALL-ACTUALS STMT))
                                                 (MG-ALIST MG-STATE)))))
           (MG-ALIST MG-STATE))
          (MG-PSW MG-STATE))
```

DEFINITION.

```
(MG-MEANING-MG-BOOLEAN-NOT STMT MG-STATE)
   =
(MG-STATE 'NORMAL
          (SET-ALIST-VALUE
           (CAR (CALL-ACTUALS STMT))
           (TAG 'BOOLEAN-MG
                (MG-NOT-BOOL (UNTAG (GET-M-VALUE (CADR (CALL-ACTUALS STMT))
                                                 (MG-ALIST MG-STATE)))))
           (MG-ALIST MG-STATE))
          (MG-PSW MG-STATE))
```

DEFINITION.

```
(MG-MEANING-MG-BOOLEAN-OR STMT MG-STATE)
   =
(MG-STATE 'NORMAL
          (SET-ALIST-VALUE
           (CAR (CALL-ACTUALS STMT))
           (TAG 'BOOLEAN-MG
                (MG-OR-BOOL (UNTAG (GET-M-VALUE (CADR (CALL-ACTUALS STMT))
                                                (MG-ALIST MG-STATE)))
                            (UNTAG (GET-M-VALUE (CADDR (CALL-ACTUALS STMT))
                                                (MG-ALIST MG-STATE)))))
           (MG-ALIST MG-STATE))
          (MG-PSW MG-STATE))
```

DEFINITION.

```
(MG-MEANING-MG-INDEX-ARRAY STMT MG-STATE)
   =
(IF (AND (NUMBERP (UNTAG (GET-M-VALUE (CADDR (CALL-ACTUALS STMT))
                                      (MG-ALIST MG-STATE))))
         (LESSP (UNTAG (GET-M-VALUE (CADDR (CALL-ACTUALS STMT))
                                    (MG-ALIST MG-STATE)))
                (ARRAY-LENGTH (GET-M-TYPE (CADR (CALL-ACTUALS STMT))
                                          (MG-ALIST MG-STATE))))))
    (MG-STATE 'NORMAL
              (SET-ALIST-VALUE
               (CAR (CALL-ACTUALS STMT))
               (FETCH-ARRAY-ELEMENT
                (CADR (CALL-ACTUALS STMT))
                (UNTAG (GET-M-VALUE (CADDR (CALL-ACTUALS STMT))
                                    (MG-ALIST MG-STATE)))
                (MG-ALIST MG-STATE))
               (MG-ALIST MG-STATE))
              (MG-PSW MG-STATE))
    (SET-CONDITION MG-STATE 'ROUTINEERROR))
```

DEFINITION.

```
(MG-MEANING-MG-INTEGER-ADD STMT MG-STATE)
   =
(IF (SMALL-INTEGERP (IPLUS (UNTAG (GET-M-VALUE (CADR (CALL-ACTUALS STMT))
                                               (MG-ALIST MG-STATE)))
                           (UNTAG (GET-M-VALUE (CADDR (CALL-ACTUALS STMT))
                                               (MG-ALIST MG-STATE))))
                    (MG-WORD-SIZE))
```

```
    (MG-STATE 'NORMAL
              (SET-ALIST-VALUE
                 (CAR (CALL-ACTUALS STMT))
                 (TAG 'INT-MG
                      (IPLUS (UNTAG (GET-M-VALUE (CADR (CALL-ACTUALS STMT))
                                                 (MG-ALIST MG-STATE)))
                             (UNTAG (GET-M-VALUE (CADDR (CALL-ACTUALS STMT))
                                                 (MG-ALIST MG-STATE)))))
                 (MG-ALIST MG-STATE))
              (MG-PSW MG-STATE))
    (SET-CONDITION MG-STATE 'ROUTINEERROR))
```

DEFINITION.

```
(MG-MEANING-MG-INTEGER-LE STMT MG-STATE)
   =
(MG-STATE 'NORMAL
          (SET-ALIST-VALUE
           (CAR (CALL-ACTUALS STMT))
           (MG-BOOL (ILEQ (UNTAG (GET-M-VALUE (CADR (CALL-ACTUALS STMT))
                                             (MG-ALIST MG-STATE)))
                          (UNTAG (GET-M-VALUE (CADDR (CALL-ACTUALS STMT))
                                             (MG-ALIST MG-STATE)))))
           (MG-ALIST MG-STATE))
          (MG-PSW MG-STATE))
```

DEFINITION.

```
(MG-MEANING-MG-INTEGER-SUBTRACT STMT MG-STATE)
   =
(IF (SMALL-INTEGERP (IDIFFERENCE (UNTAG (GET-M-VALUE (CADR (CALL-ACTUALS STMT))
                                                     (MG-ALIST MG-STATE)))
                                 (UNTAG (GET-M-VALUE (CADDR (CALL-ACTUALS STMT))
                                                     (MG-ALIST MG-STATE))))
                    (MG-WORD-SIZE))
    (MG-STATE 'NORMAL
              (SET-ALIST-VALUE
               (CAR (CALL-ACTUALS STMT))
               (TAG 'INT-MG
                    (IDIFFERENCE (UNTAG (GET-M-VALUE (CADR (CALL-ACTUALS STMT))
                                                     (MG-ALIST MG-STATE)))
                                 (UNTAG (GET-M-VALUE (CADDR (CALL-ACTUALS STMT))
                                                     (MG-ALIST MG-STATE)))))
               (MG-ALIST MG-STATE))
              (MG-PSW MG-STATE))
    (SET-CONDITION MG-STATE 'ROUTINEERROR))
```

DEFINITION.

```
(MG-MEANING-MG-INTEGER-UNARY-MINUS STMT MG-STATE)
   =
(IF (SMALL-INTEGERP (INEGATE (UNTAG (GET-M-VALUE (CADR (CALL-ACTUALS STMT))
                                                 (MG-ALIST MG-STATE))))
                    (MG-WORD-SIZE))
    (MG-STATE 'NORMAL
              (SET-ALIST-VALUE
               (CAR (CALL-ACTUALS STMT))
               (TAG 'INT-MG
                    (INEGATE (UNTAG (GET-M-VALUE (CADR (CALL-ACTUALS STMT))
                                                 (MG-ALIST MG-STATE)))))
               (MG-ALIST MG-STATE))
              (MG-PSW MG-STATE))
    (SET-CONDITION MG-STATE 'ROUTINEERROR))
```

DEFINITION.

```
(MG-MEANING-MG-SIMPLE-CONSTANT-ASSIGNMENT STMT MG-STATE)
   =
(MG-STATE 'NORMAL
          (SET-ALIST-VALUE (CAR (CALL-ACTUALS STMT))
                           (CADR (CALL-ACTUALS STMT))
                           (MG-ALIST MG-STATE))
          (MG-PSW MG-STATE))
```

DEFINITION.

```
(MG-MEANING-MG-SIMPLE-CONSTANT-EQ STMT MG-STATE)
   =
(MG-STATE 'NORMAL
          (SET-ALIST-VALUE
           (CAR (CALL-ACTUALS STMT))
           (MG-BOOL (EQUAL (UNTAG (GET-M-VALUE (CADR (CALL-ACTUALS STMT))
                                               (MG-ALIST MG-STATE)))
                           (UNTAG (CADDR (CALL-ACTUALS STMT)))))
           (MG-ALIST MG-STATE))
          (MG-PSW MG-STATE))
```

DEFINITION.

```
(MG-MEANING-MG-SIMPLE-VARIABLE-ASSIGNMENT STMT MG-STATE)
   =
(MG-STATE 'NORMAL
          (SET-ALIST-VALUE (CAR (CALL-ACTUALS STMT))
                           (GET-M-VALUE (CADR (CALL-ACTUALS STMT))
                                        (MG-ALIST MG-STATE))
                           (MG-ALIST MG-STATE))
          (MG-PSW MG-STATE))
```

DEFINITION.

```
(MG-MEANING-MG-SIMPLE-VARIABLE-EQ STMT MG-STATE)
   =
(MG-STATE 'NORMAL
          (SET-ALIST-VALUE
           (CAR (CALL-ACTUALS STMT))
           (MG-BOOL (EQUAL (UNTAG (GET-M-VALUE (CADR (CALL-ACTUALS STMT))
                                               (MG-ALIST MG-STATE)))
                           (UNTAG (GET-M-VALUE (CADDR (CALL-ACTUALS STMT))
                                               (MG-ALIST MG-STATE)))))
           (MG-ALIST MG-STATE))
          (MG-PSW MG-STATE))
```

DEFINITION.

```
(MG-MEANING-PREDEFINED-PROC-CALL STMT MG-STATE)
   =
(CASE
 (CALL-NAME STMT)
 (MG-SIMPLE-VARIABLE-ASSIGNMENT
  (MG-MEANING-MG-SIMPLE-VARIABLE-ASSIGNMENT STMT
                                            MG-STATE))
 (MG-SIMPLE-CONSTANT-ASSIGNMENT
  (MG-MEANING-MG-SIMPLE-CONSTANT-ASSIGNMENT STMT
                                            MG-STATE))
 (MG-SIMPLE-VARIABLE-EQ (MG-MEANING-MG-SIMPLE-VARIABLE-EQ STMT MG-STATE))
 (MG-SIMPLE-CONSTANT-EQ (MG-MEANING-MG-SIMPLE-CONSTANT-EQ STMT MG-STATE))
 (MG-INTEGER-LE (MG-MEANING-MG-INTEGER-LE STMT MG-STATE))
 (MG-INTEGER-UNARY-MINUS (MG-MEANING-MG-INTEGER-UNARY-MINUS STMT MG-STATE))
 (MG-INTEGER-ADD (MG-MEANING-MG-INTEGER-ADD STMT MG-STATE))
 (MG-INTEGER-SUBTRACT (MG-MEANING-MG-INTEGER-SUBTRACT STMT MG-STATE))
 (MG-BOOLEAN-OR (MG-MEANING-MG-BOOLEAN-OR STMT MG-STATE))
 (MG-BOOLEAN-AND (MG-MEANING-MG-BOOLEAN-AND STMT MG-STATE))
 (MG-BOOLEAN-NOT (MG-MEANING-MG-BOOLEAN-NOT STMT MG-STATE))
 (MG-INDEX-ARRAY (MG-MEANING-MG-INDEX-ARRAY STMT MG-STATE))
 (MG-ARRAY-ELEMENT-ASSIGNMENT
  (MG-MEANING-MG-ARRAY-ELEMENT-ASSIGNMENT STMT
                                          MG-STATE))
 (OTHERWISE MG-STATE))
```

DEFINITION.

```
(MG-MEANING-R STMT PROC-LIST MG-STATE N SIZES)
   =
(COND
 ((ZEROP N)
  (SIGNAL-SYSTEM-ERROR MG-STATE 'TIMED-OUT))
 ((NOT (NORMAL MG-STATE)) MG-STATE)
 ((RESOURCES-INADEQUATEP STMT PROC-LIST SIZES)
  (SIGNAL-SYSTEM-ERROR MG-STATE 'RESOURCE-ERROR))
 ((EQUAL (CAR STMT) 'NO-OP-MG)
  MG-STATE)
 ((EQUAL (CAR STMT) 'SIGNAL-MG)
  (SET-CONDITION MG-STATE
                (SIGNALLED-CONDITION STMT)))
 ((EQUAL (CAR STMT) 'PROG2-MG)
  (MG-MEANING-R (PROG2-RIGHT-BRANCH STMT)
               PROC-LIST
               (MG-MEANING-R (PROG2-LEFT-BRANCH STMT)
                             PROC-LIST MG-STATE
                             (SUB1 N)
                             SIZES)
               (SUB1 N)
               SIZES))
 ((EQUAL (CAR STMT) 'LOOP-MG)
  (REMOVE-LEAVE (MG-MEANING-R STMT PROC-LIST
                             (MG-MEANING-R (LOOP-BODY STMT)
                                           PROC-LIST MG-STATE
                                           (SUB1 N)
                                           SIZES)
                             (SUB1 N)
                             SIZES)))
 ((EQUAL (CAR STMT) 'IF-MG)
  (IF (MG-EXPRESSION-FALSEP (IF-CONDITION STMT)
                           MG-STATE)
      (MG-MEANING-R (IF-FALSE-BRANCH STMT)
                   PROC-LIST MG-STATE
                   (SUB1 N)
                   SIZES)
      (MG-MEANING-R (IF-TRUE-BRANCH STMT)
                   PROC-LIST MG-STATE
                   (SUB1 N)
                   SIZES)))
 ((EQUAL (CAR STMT) 'BEGIN-MG)
  (IF (MEMBER (CC (MG-MEANING-R (BEGIN-BODY STMT)
                               PROC-LIST MG-STATE
                               (SUB1 N)
                               SIZES))
             (WHEN-LABELS STMT))
      (MG-MEANING-R (WHEN-HANDLER STMT)
                   PROC-LIST
                   (SET-CONDITION (MG-MEANING-R (BEGIN-BODY STMT)
                                                PROC-LIST MG-STATE
                                                (SUB1 N)
                                                SIZES)
                                  'NORMAL)
                   (SUB1 N)
                   SIZES)
      (MG-MEANING-R (BEGIN-BODY STMT)
                   PROC-LIST MG-STATE
                   (SUB1 N)
                   SIZES)))
```

```
 ((EQUAL (CAR STMT) 'PROC-CALL-MG)
  (MAP-CALL-EFFECTS
   (MG-MEANING-R
    (DEF-BODY (FETCH-CALLED-DEF STMT PROC-LIST))
    PROC-LIST
    (MAKE-CALL-ENVIRONMENT MG-STATE STMT
                           (FETCH-CALLED-DEF STMT PROC-LIST))
    (SUB1 N)
    (LIST (PLUS (T-SIZE SIZES)
                (DATA-LENGTH (DEF-LOCALS (FETCH-CALLED-DEF STMT PROC-LIST))))
          (PLUS (C-SIZE SIZES)
                2
                (LENGTH (DEF-LOCALS (FETCH-CALLED-DEF STMT PROC-LIST)))
                (LENGTH (DEF-FORMALS (FETCH-CALLED-DEF STMT PROC-LIST))))))
   (FETCH-CALLED-DEF STMT PROC-LIST)
   STMT MG-STATE))
 ((EQUAL (CAR STMT)
         'PREDEFINED-PROC-CALL-MG)
  (MG-MEANING-PREDEFINED-PROC-CALL STMT MG-STATE))
 (T MG-STATE))
```

DEFINITION.

```
(MG-NAME-ALIST-ELEMENTP X)
   =
(AND (OK-MG-NAMEP (CAR X))
     (MG-TYPE-REFP (M-TYPE X)))
```

DEFINITION.

```
(MG-NAME-ALISTP ALIST)
   =
(IF (NLISTP ALIST)
    (EQUAL ALIST NIL)
    (AND (MG-NAME-ALIST-ELEMENTP (CAR ALIST))
         (MG-NAME-ALISTP (CDR ALIST))))
```

DEFINITION.

```
(MG-NOT-BOOL X)
   =
(IF (EQUAL X 'FALSE-MG)
    'TRUE-MG
    'FALSE-MG)
```

DEFINITION.

```
(MG-OR-BOOL X Y)
   =
(IF (EQUAL X 'FALSE-MG) Y 'TRUE-MG)
```

SHELL DEFINITION.
Add the shell **MG-STATE** of **3** arguments, with recognizer function symbol **MG-STATEP**, and accessors **CC**, **MG-ALIST** and **MG-PSW**.

DEFINITION.

```
(MG-TYPE-REFP TYPREF)
   =
(OR (SIMPLE-MG-TYPE-REFP TYPREF)
    (ARRAY-MG-TYPE-REFP TYPREF))
```

DEFINITION.

```
(MG-WORD-SIZE) = 32
```

DEFINITION.

```
(MININT) = (MINUS (EXP 2 (SUB1 (MG-WORD-SIZE))))
```

DEFINITION.

```
(NO-DUPLICATES LST)
   =
(COND ((NLISTP LST) T)
      ((MEMBER (CAR LST) (CDR LST)) F)
      (T (NO-DUPLICATES (CDR LST))))
```

DEFINITION.
```
(NONEMPTY-COND-IDENTIFIER-PLISTP LST COND-LIST)
   =
(AND (COND-IDENTIFIER-PLISTP LST COND-LIST)
     (NOT (EQUAL LST NIL)))
```

DEFINITION.
```
(NORMAL MG-STATE) = (EQUAL (CC MG-STATE) 'NORMAL)
```

DEFINITION.
```
(OK-ACTUAL-PARAMS-LIST LST ALIST)
   =
(IF (NLISTP LST)
    (EQUAL LST NIL)
    (AND (DEFINED-IDENTIFIERP (CAR LST) ALIST)
         (OK-ACTUAL-PARAMS-LIST (CDR LST)
                                ALIST)))
```

DEFINITION.
```
(OK-CC C COND-LIST)
   =
(AND (LITATOM C)
     (OR (MEMBER C '(NORMAL ROUTINEERROR))
         (MEMBER C COND-LIST)))
```

DEFINITION.
```
(OK-CONDITION EXP COND-LIST)
   =
(OR (EQUAL EXP 'ROUTINEERROR)
    (AND (OR (OK-MG-NAMEP EXP)
             (EQUAL EXP 'LEAVE))
         (MEMBER EXP COND-LIST)))
```

DEFINITION.
```
(OK-IDENTIFIER-ACTUAL ACTUAL FORMAL ALIST)
   =
(AND (IDENTIFIERP ACTUAL)
     (EQUAL (GET-M-TYPE ACTUAL ALIST)
            (FORMAL-TYPE FORMAL)))
```

DEFINITION.
```
(OK-MG-ARRAY-ELEMENT-ASSIGNMENT-ARGS ARGS ALIST)
   =
(AND (LENGTH-PLISTP ARGS 4)
     (ARRAY-IDENTIFIERP (CAR ARGS) ALIST)
     (INT-IDENTIFIERP (CADR ARGS) ALIST)
     (EQUAL (CADDDR ARGS)
            (ARRAY-LENGTH (CADR (ASSOC (CAR ARGS) ALIST))))
     (LESSP (CADDDR ARGS) (MAXINT))
     (SIMPLE-TYPED-IDENTIFIERP (CADDR ARGS)
                               (ARRAY-ELEMTYPE (CADR (ASSOC (CAR ARGS) ALIST)))
                               ALIST))
```

DEFINITION.
```
(OK-MG-ARRAY-VALUE EXP TYPE)
   =
(ARRAY-LITERALP EXP
                (ARRAY-LENGTH TYPE)
                (ARRAY-ELEMTYPE TYPE))
```

DEFINITION.
```
(OK-MG-BOOLEAN-AND-ARGS ARGS ALIST)
   =
(AND (LENGTH-PLISTP ARGS 3)
     (BOOLEAN-IDENTIFIERP (CAR ARGS) ALIST)
     (BOOLEAN-IDENTIFIERP (CADR ARGS)
                          ALIST)
     (BOOLEAN-IDENTIFIERP (CADDR ARGS)
                          ALIST))
```

DEFINITION.
```
(OK-MG-BOOLEAN-NOT-ARGS ARGS ALIST)
   =
(AND (LENGTH-PLISTP ARGS 2)
     (BOOLEAN-IDENTIFIERP (CAR ARGS) ALIST)
     (BOOLEAN-IDENTIFIERP (CADR ARGS)
                          ALIST))
```

DEFINITION.
```
(OK-MG-BOOLEAN-OR-ARGS ARGS ALIST)
   =
(AND (LENGTH-PLISTP ARGS 3)
     (BOOLEAN-IDENTIFIERP (CAR ARGS) ALIST)
     (BOOLEAN-IDENTIFIERP (CADR ARGS)
                          ALIST)
     (BOOLEAN-IDENTIFIERP (CADDR ARGS)
                          ALIST))
```

DEFINITION.
```
(OK-MG-DEF DEF PROC-LIST)
   =
(AND (LENGTH-PLISTP DEF 6)
     (OK-MG-NAMEP (DEF-NAME DEF))
     (OK-MG-FORMAL-DATA-PARAMS-PLISTP (DEF-FORMALS DEF))
     (IDENTIFIER-PLISTP (DEF-CONDS DEF))
     (OK-MG-LOCAL-DATA-PLISTP (DEF-LOCALS DEF))
     (IDENTIFIER-PLISTP (DEF-COND-LOCALS DEF))
     (NO-DUPLICATES (COLLECT-LOCAL-NAMES DEF))
     (LESSP (PLUS (LENGTH (DEF-CONDS DEF))
                  (LENGTH (DEF-COND-LOCALS DEF)))
            (SUB1 (SUB1 (SUB1 (EXP 2 (MG-WORD-SIZE))))))
     (OK-MG-STATEMENT (DEF-BODY DEF)
                      (MAKE-COND-LIST DEF)
                      (MAKE-NAME-ALIST DEF)
                      PROC-LIST))
```

DEFINITION.
```
(OK-MG-DEF-PLISTP PROC-LIST)
   =
(OK-MG-DEF-PLISTP1 PROC-LIST PROC-LIST)
```

DEFINITION.
```
(OK-MG-DEF-PLISTP1 LST1 LST2)
   =
(IF (NLISTP LST1)
    (EQUAL LST1 NIL)
    (AND (OK-MG-DEF (CAR LST1) LST2)
         (OK-MG-DEF-PLISTP1 (CDR LST1) LST2)))
```

DEFINITION.
```
(OK-MG-FORMAL-DATA-PARAM EXP)
   =
(AND (LENGTH-PLISTP EXP 2)
     (OK-MG-NAMEP (CAR EXP))
     (MG-TYPE-REFP (FORMAL-TYPE EXP)))
```

DEFINITION.
```
(OK-MG-FORMAL-DATA-PARAMS-PLISTP LST)
   =
(IF (NLISTP LST)
    (EQUAL LST NIL)
    (AND (OK-MG-FORMAL-DATA-PARAM (CAR LST))
         (OK-MG-FORMAL-DATA-PARAMS-PLISTP (CDR LST))))
```

DEFINITION.

```
(OK-MG-INDEX-ARRAY-ARGS ARGS ALIST)
   =
(AND (LENGTH-PLISTP ARGS 4)
     (ARRAY-IDENTIFIERP (CADR ARGS) ALIST)
     (INT-IDENTIFIERP (CADDR ARGS) ALIST)
     (SIMPLE-TYPED-IDENTIFIERP (CAR ARGS)
                               (ARRAY-ELEMTYPE (CADR (ASSOC (CADR ARGS) ALIST)))
                               ALIST)
     (EQUAL (CADDDR ARGS)
            (ARRAY-LENGTH (CADR (ASSOC (CADR ARGS) ALIST))))
     (LESSP (CADDDR ARGS) (MAXINT)))
```

DEFINITION.

```
(OK-MG-INTEGER-ADD-ARGS ARGS ALIST)
   =
(AND (LENGTH-PLISTP ARGS 3)
     (INT-IDENTIFIERP (CAR ARGS) ALIST)
     (INT-IDENTIFIERP (CADR ARGS) ALIST)
     (INT-IDENTIFIERP (CADDR ARGS) ALIST))
```

DEFINITION.

```
(OK-MG-INTEGER-LE-ARGS ARGS ALIST)
   =
(AND (LENGTH-PLISTP ARGS 3)
     (BOOLEAN-IDENTIFIERP (CAR ARGS) ALIST)
     (INT-IDENTIFIERP (CADR ARGS) ALIST)
     (INT-IDENTIFIERP (CADDR ARGS) ALIST))
```

DEFINITION.

```
(OK-MG-INTEGER-SUBTRACT-ARGS ARGS ALIST)
   =
(AND (LENGTH-PLISTP ARGS 3)
     (INT-IDENTIFIERP (CAR ARGS) ALIST)
     (INT-IDENTIFIERP (CADR ARGS) ALIST)
     (INT-IDENTIFIERP (CADDR ARGS) ALIST))
```

DEFINITION.

```
(OK-MG-INTEGER-UNARY-MINUS-ARGS ARGS ALIST)
   =
(AND (LENGTH-PLISTP ARGS 2)
     (INT-IDENTIFIERP (CAR ARGS) ALIST)
     (INT-IDENTIFIERP (CADR ARGS) ALIST))
```

DEFINITION.

```
(OK-MG-LOCAL-DATA-DECL EXP)
   =
(AND (LENGTH-PLISTP EXP 3)
     (OK-MG-NAMEP (CAR EXP))
     (MG-TYPE-REFP (FORMAL-TYPE EXP))
     (OK-MG-VALUEP (FORMAL-INITIAL-VALUE EXP)
                   (FORMAL-TYPE EXP)))
```

DEFINITION.

```
(OK-MG-LOCAL-DATA-PLISTP LST)
   =
(IF (NLISTP LST)
    (EQUAL LST NIL)
    (AND (OK-MG-LOCAL-DATA-DECL (CAR LST))
         (OK-MG-LOCAL-DATA-PLISTP (CDR LST))))
```

DEFINITION.

```
(OK-MG-NAMEP IDENT)
   =
(AND (LITATOM IDENT)
     (NOT (MEMBER 45 (UNPACK IDENT)))
     (NOT (RESERVED-WORD IDENT)))
```

DEFINITION.
```
(OK-MG-SIMPLE-CONSTANT-ASSIGNMENT-ARGS ARGS ALIST)
   =
(AND (LENGTH-PLISTP ARGS 2)
     (SIMPLE-IDENTIFIERP (CAR ARGS) ALIST)
     (SIMPLE-TYPED-LITERALP (CADR ARGS)
                            (CADR (ASSOC (CAR ARGS) ALIST))))
```

DEFINITION.
```
(OK-MG-SIMPLE-CONSTANT-EQ-ARGS ARGS ALIST)
   =
(AND (LENGTH-PLISTP ARGS 3)
     (BOOLEAN-IDENTIFIERP (CAR ARGS) ALIST)
     (SIMPLE-IDENTIFIERP (CADR ARGS) ALIST)
     (SIMPLE-TYPED-LITERALP (CADDR ARGS)
                            (CADR (ASSOC (CADR ARGS) ALIST))))
```

DEFINITION.

```
(OK-MG-SIMPLE-VARIABLE-ASSIGNMENT-ARGS ARGS ALIST)
   =
(AND (LENGTH-PLISTP ARGS 2)
     (SIMPLE-IDENTIFIERP (CAR ARGS) ALIST)
     (SIMPLE-IDENTIFIERP (CADR ARGS) ALIST)
     (EQUAL (CADR (ASSOC (CAR ARGS) ALIST))
            (CADR (ASSOC (CADR ARGS) ALIST))))
```

DEFINITION.
```
(OK-MG-SIMPLE-VARIABLE-EQ-ARGS ARGS ALIST)
   =
(AND (LENGTH-PLISTP ARGS 3)
     (BOOLEAN-IDENTIFIERP (CAR ARGS) ALIST)
     (SIMPLE-IDENTIFIERP (CADR ARGS) ALIST)
     (SIMPLE-IDENTIFIERP (CADDR ARGS)
                         ALIST)
     (EQUAL (CADR (ASSOC (CADR ARGS) ALIST))
            (CADR (ASSOC (CADDR ARGS) ALIST))))
```

DEFINITION.
```
(OK-MG-STATEMENT STMT R-COND-LIST ALIST PROC-LIST)
   =
(CASE (CAR STMT)
      (NO-OP-MG (EQUAL (CDR STMT) NIL))
      (SIGNAL-MG
       (AND (LENGTH-PLISTP STMT 2)
            (OK-CONDITION (SIGNALLED-CONDITION STMT) R-COND-LIST)))
      (PROG2-MG
       (AND (LENGTH-PLISTP STMT 3)
            (OK-MG-STATEMENT (PROG2-LEFT-BRANCH STMT) R-COND-LIST ALIST PROC-LIST)
            (OK-MG-STATEMENT (PROG2-RIGHT-BRANCH STMT)
                             R-COND-LIST ALIST PROC-LIST)))
      (LOOP-MG
       (AND (LENGTH-PLISTP STMT 2)
            (OK-MG-STATEMENT (LOOP-BODY STMT)
                             (CONS 'LEAVE R-COND-LIST) ALIST PROC-LIST)))
      (IF-MG
       (AND (LENGTH-PLISTP STMT 4)
            (BOOLEAN-IDENTIFIERP (IF-CONDITION STMT) ALIST)
            (OK-MG-STATEMENT (IF-TRUE-BRANCH STMT) R-COND-LIST ALIST PROC-LIST)
            (OK-MG-STATEMENT (IF-FALSE-BRANCH STMT)
                             R-COND-LIST ALIST PROC-LIST)))
      (BEGIN-MG
       (AND (LENGTH-PLISTP STMT 4)
            (OK-MG-STATEMENT (BEGIN-BODY STMT)
                             (APPEND (WHEN-LABELS STMT) R-COND-LIST)
                             ALIST PROC-LIST)
            (NONEMPTY-COND-IDENTIFIER-PLISTP (WHEN-LABELS STMT) R-COND-LIST)
            (OK-MG-STATEMENT (WHEN-HANDLER STMT) R-COND-LIST ALIST PROC-LIST)))
```

```
      (PROC-CALL-MG
       (OK-PROC-CALL STMT R-COND-LIST ALIST PROC-LIST))
      (PREDEFINED-PROC-CALL-MG
       (OK-PREDEFINED-PROC-CALL STMT ALIST))
      (OTHERWISE F))
```

DEFINITION.

```
(OK-MG-STATEP MG-STATE COND-LIST)
   =
(AND (OK-CC (CC MG-STATE) COND-LIST)
     (MG-ALISTP (MG-ALIST MG-STATE)))
```

DEFINITION.

```
(OK-MG-VALUEP EXP TYPE)
   =
(COND ((SIMPLE-MG-TYPE-REFP TYPE)
       (SIMPLE-TYPED-LITERALP EXP TYPE))
      ((ARRAY-MG-TYPE-REFP TYPE)
       (OK-MG-ARRAY-VALUE EXP TYPE))
      (T F))
```

DEFINITION.

```
(OK-PREDEFINED-PROC-ARGS NAME ARGS ALIST)
   =
(CASE NAME
      (MG-SIMPLE-VARIABLE-ASSIGNMENT
           (OK-MG-SIMPLE-VARIABLE-ASSIGNMENT-ARGS ARGS ALIST))
      (MG-SIMPLE-CONSTANT-ASSIGNMENT
           (OK-MG-SIMPLE-CONSTANT-ASSIGNMENT-ARGS ARGS ALIST))
      (MG-SIMPLE-VARIABLE-EQ (OK-MG-SIMPLE-VARIABLE-EQ-ARGS ARGS ALIST))
      (MG-SIMPLE-CONSTANT-EQ (OK-MG-SIMPLE-CONSTANT-EQ-ARGS ARGS ALIST))
      (MG-INTEGER-LE (OK-MG-INTEGER-LE-ARGS ARGS ALIST))
      (MG-INTEGER-UNARY-MINUS (OK-MG-INTEGER-UNARY-MINUS-ARGS ARGS ALIST))
      (MG-INTEGER-ADD (OK-MG-INTEGER-ADD-ARGS ARGS ALIST))
      (MG-INTEGER-SUBTRACT (OK-MG-INTEGER-SUBTRACT-ARGS ARGS ALIST))
      (MG-BOOLEAN-OR (OK-MG-BOOLEAN-OR-ARGS ARGS ALIST))
      (MG-BOOLEAN-AND (OK-MG-BOOLEAN-AND-ARGS ARGS ALIST))
      (MG-BOOLEAN-NOT (OK-MG-BOOLEAN-NOT-ARGS ARGS ALIST))
      (MG-INDEX-ARRAY (OK-MG-INDEX-ARRAY-ARGS ARGS ALIST))
      (MG-ARRAY-ELEMENT-ASSIGNMENT (OK-MG-ARRAY-ELEMENT-ASSIGNMENT-ARGS ARGS
                                                                      ALIST))
      (OTHERWISE F))
```

DEFINITION.

```
(OK-PREDEFINED-PROC-CALL STMT ALIST)
   =
(AND (LENGTH-PLISTP STMT 3)
     (PREDEFINED-PROCP (CALL-NAME STMT))
     (OK-PREDEFINED-PROC-ARGS (CALL-NAME STMT)
                              (CALL-ACTUALS STMT)
                              ALIST))
```

DEFINITION.

```
(OK-PROC-CALL STMT R-COND-LIST ALIST PROC-LIST)
   =
(AND (LENGTH-PLISTP STMT 4)
     (IDENTIFIERP (CALL-NAME STMT))
     (USER-DEFINED-PROCP (CALL-NAME STMT)
                         PROC-LIST)
     (OK-ACTUAL-PARAMS-LIST (CALL-ACTUALS STMT)
                            ALIST)
     (NO-DUPLICATES (CALL-ACTUALS STMT))
     (DATA-PARAM-LISTS-MATCH (CALL-ACTUALS STMT)
                             (DEF-FORMALS (FETCH-CALLED-DEF STMT PROC-LIST))
                             ALIST)
     (COND-IDENTIFIER-PLISTP (CALL-CONDS STMT)
                             R-COND-LIST)
     (COND-PARAMS-MATCH (CALL-CONDS STMT)
                        (DEF-CONDS (FETCH-CALLED-DEF STMT PROC-LIST))))
```

DEFINITION.
```
(PLISTP X)
   =
(IF (NLISTP X)
    (EQUAL X NIL)
    (PLISTP (CDR X)))
```

DEFINITION.
```
(PREDEFINED-PROC-CALL-BINDINGS-COUNT NAME)
   =
(CASE NAME
      (MG-SIMPLE-VARIABLE-ASSIGNMENT 2)
      (MG-SIMPLE-CONSTANT-ASSIGNMENT 2)
      (MG-SIMPLE-VARIABLE-EQ 3)
      (MG-SIMPLE-CONSTANT-EQ 3)
      (MG-INTEGER-LE 3)
      (MG-INTEGER-UNARY-MINUS 4)
      (MG-INTEGER-ADD 4)
      (MG-INTEGER-SUBTRACT 4)
      (MG-BOOLEAN-OR 3)
      (MG-BOOLEAN-AND 3)
      (MG-BOOLEAN-NOT 3)
      (MG-INDEX-ARRAY 5)
      (MG-ARRAY-ELEMENT-ASSIGNMENT 5)
      (OTHERWISE 0))
```

DEFINITION.
```
(PREDEFINED-PROC-CALL-P-FRAME-SIZE NAME)
   =
(ADD1 (ADD1 (PREDEFINED-PROC-CALL-BINDINGS-COUNT NAME)))
```

DEFINITION.
```
(PREDEFINED-PROC-CALL-TEMP-STK-REQUIREMENT NAME)
   =
(CASE NAME
      (MG-SIMPLE-VARIABLE-ASSIGNMENT 2)
      (MG-SIMPLE-CONSTANT-ASSIGNMENT 2)
      (MG-SIMPLE-VARIABLE-EQ 3)
      (MG-SIMPLE-CONSTANT-EQ 3)
      (MG-INTEGER-LE 3)
      (MG-INTEGER-UNARY-MINUS 2)
      (MG-INTEGER-ADD 3)
      (MG-INTEGER-SUBTRACT 3)
      (MG-BOOLEAN-OR 3)
      (MG-BOOLEAN-AND 3)
      (MG-BOOLEAN-NOT 2)
      (MG-INDEX-ARRAY 4)
      (MG-ARRAY-ELEMENT-ASSIGNMENT 4)
      (OTHERWISE 0))
```

DEFINITION.
```
(PREDEFINED-PROCEDURE-LIST)
   =
'(MG-SIMPLE-VARIABLE-ASSIGNMENT MG-SIMPLE-CONSTANT-ASSIGNMENT
                                MG-SIMPLE-VARIABLE-EQ
                                MG-SIMPLE-CONSTANT-EQ MG-INTEGER-LE
                                MG-INTEGER-UNARY-MINUS MG-INTEGER-ADD
                                MG-INTEGER-SUBTRACT MG-BOOLEAN-OR
                                MG-BOOLEAN-AND MG-BOOLEAN-NOT
                                MG-INDEX-ARRAY
                                MG-ARRAY-ELEMENT-ASSIGNMENT)
```

DEFINITION.
```
(PREDEFINED-PROCP NAME)
   =
(MEMBER NAME (PREDEFINED-PROCEDURE-LIST))
```

DEFINITION.
```
(PROG2-LEFT-BRANCH STMT) = (CADR STMT)
```

DEFINITION.
```
(PROG2-RIGHT-BRANCH STMT) = (CADDR STMT)
```

DEFINITION.
```
(PUT VAL N LST)
   =
(IF (ZEROP N)
    (IF (LISTP LST)
        (CONS VAL (CDR LST))
        (LIST VAL))
    (CONS (CAR LST)
          (PUT VAL (SUB1 N) (CDR LST))))
```

DEFINITION.
```
(PUT-ARRAY-ELEMENT A I VAL ALIST)
   =
(PUT VAL I (CADDR (ASSOC A ALIST)))
```

DEFINITION.
```
(REMOVE-LEAVE MG-STATE)
   =
(IF (EQUAL (CC MG-STATE) 'LEAVE)
    (SET-CONDITION MG-STATE 'NORMAL)
    MG-STATE)
```

DEFINITION.
```
(RESERVED-NAMES-LIST) = '(LEAVE NORMAL ROUTINEERROR)
```

DEFINITION.
```
(RESERVED-WORD WD) = (MEMBER WD (RESERVED-NAMES-LIST))
```

DEFINITION.
```
(RESOURCE-ERRORP MG-STATE)
   =
(NOT (EQUAL (MG-PSW MG-STATE) 'RUN))
```

DEFINITION.
```
(RESOURCES-INADEQUATEP STMT PROC-LIST SIZE-PAIR)
   =
(OR (NOT (LESSP (TEMP-STK-REQUIREMENTS STMT PROC-LIST)
                (DIFFERENCE (MG-MAX-TEMP-STK-SIZE)
                            (T-SIZE SIZE-PAIR))))
    (NOT (LESSP (CTRL-STK-REQUIREMENTS STMT PROC-LIST)
                (DIFFERENCE (MG-MAX-CTRL-STK-SIZE)
                            (C-SIZE SIZE-PAIR)))))
```

DEFINITION.
```
(SET-ALIST-VALUE NAME VAL ALIST)
   =
(COND ((NLISTP ALIST) NIL)
      ((EQUAL (CAAR ALIST) NAME)
       (CONS (CONS NAME
                   (CONS (M-TYPE (CAR ALIST))
                         (CONS VAL (CDDDAR ALIST))))
             (CDR ALIST)))
      (T (CONS (CAR ALIST)
               (SET-ALIST-VALUE NAME VAL
                                (CDR ALIST)))))
```

DEFINITION.
```
(SET-CONDITION MG-STATE CONDITION-NAME)
   =
(MG-STATE CONDITION-NAME
          (MG-ALIST MG-STATE)
          (MG-PSW MG-STATE))
```

DEFINITION.
```
(SIGNAL-SYSTEM-ERROR MG-STATE ERROR)
   =
(MG-STATE (CC MG-STATE)
          (MG-ALIST MG-STATE)
          ERROR)
```

DEFINITION.
```
(SIGNALLED-CONDITION STMT) = (CADR STMT)
```

DEFINITION.
```
(SIMPLE-IDENTIFIERP NAME ALIST)
   =
(OR (BOOLEAN-IDENTIFIERP NAME ALIST)
    (INT-IDENTIFIERP NAME ALIST)
    (CHARACTER-IDENTIFIERP NAME ALIST))
```

DEFINITION.
```
(SIMPLE-MG-TYPE-REFP TYPREF)
   =
(MEMBER TYPREF '(INT-MG BOOLEAN-MG CHARACTER-MG))
```

DEFINITION.
```
(SIMPLE-TYPED-IDENTIFIERP IDENT TYPE ALIST)
   =
(CASE TYPE
      (INT-MG (INT-IDENTIFIERP IDENT ALIST))
      (BOOLEAN-MG (BOOLEAN-IDENTIFIERP IDENT ALIST))
      (CHARACTER-MG (CHARACTER-IDENTIFIERP IDENT ALIST))
      (OTHERWISE F))
```

DEFINITION.
```
(SIMPLE-TYPED-LITERAL-PLISTP LST TYPE)
   =
(IF (NLISTP LST)
    (EQUAL LST NIL)
    (AND (SIMPLE-TYPED-LITERALP (CAR LST) TYPE)
         (SIMPLE-TYPED-LITERAL-PLISTP (CDR LST)
                                      TYPE)))
```

DEFINITION.
```
(SIMPLE-TYPED-LITERALP LIT TYPE)
   =
(CASE TYPE
      (INT-MG (INT-LITERALP LIT))
      (BOOLEAN-MG (BOOLEAN-LITERALP LIT))
      (CHARACTER-MG (CHARACTER-LITERALP LIT))
      (OTHERWISE F))
```

DEFINITION.
```
(SMALL-INTEGERP I WORD-SIZE)
   =
(AND (INTEGERP I)
     (NOT (ILESSP I (MINUS (EXP 2 (SUB1 WORD-SIZE)))))
     (ILESSP I (EXP 2 (SUB1 WORD-SIZE))))
```

DEFINITION.
```
(T-SIZE X) = (CAR X)
```

DEFINITION.
```
(TAG TYPE OBJ) = (LIST TYPE OBJ)
```

DEFINITION.
```
(TEMP-STK-REQUIREMENTS STMT PROC-LIST)
   =
(CASE
 (CAR STMT)
 (NO-OP-MG 0)
 (SIGNAL-MG 1)
 (PROG2-MG 0)
 (LOOP-MG 1)
 (IF-MG 1)
 (BEGIN-MG 1)
 (PROC-CALL-MG
  (MAX (PLUS (DATA-LENGTH (DEF-LOCALS (FETCH-CALLED-DEF STMT PROC-LIST)))
             (LENGTH (DEF-LOCALS (FETCH-CALLED-DEF STMT PROC-LIST)))
             (LENGTH (CALL-ACTUALS STMT)))
       1))
 (PREDEFINED-PROC-CALL-MG
  (PREDEFINED-PROC-CALL-TEMP-STK-REQUIREMENT (CALL-NAME STMT)))
 (OTHERWISE 0))
```

DEFINITION.
```
(UNTAG CONST) = (CADR CONST)
```

DEFINITION.
```
(USER-DEFINED-PROCP NAME PROC-LIST)
   =
(AND (NOT (MEMBER NAME (PREDEFINED-PROCEDURE-LIST)))
     (DEFINEDP NAME PROC-LIST))
```

DEFINITION.
```
(WHEN-HANDLER STMT) = (CADDDR STMT)
```

DEFINITION.
```
(WHEN-LABELS STMT) = (CADDR STMT)
```

Chapter 4
PITON

Piton[26] is a high-level assembly language designed for verified applications and as the target language for high-level language compilers. It provides execute-only programs, recursive subroutine call and return, stack based parameter passing, local variables, global variables and arrays, a user-visible stack for intermediate computations, and seven abstract data types including integers, data addresses, program addresses and subroutine names. Piton is formally specified by an interpreter written for it in the computational logic of Boyer and Moore. It is this interpreter which is the target level interpreter for our Micro-Gypsy to Piton interpreter equivalence proof.

Piton has been implemented on the FM8502, a general purpose microprocessor whose gate-level design has been mechanically proved to implement its machine code interpreter. [Hunt 85] The FM8502 implementation of Piton is via a function in the Boyer-Moore logic which maps a Piton initial state into an FM8502 binary core image. The compiler, assembler and linker are all defined as functions in the logic. The implementation has been mechanically proved correct. In particular, if a Piton state can be run to completion without error, then the final values of all the global data structures can be ascertained from an inspection of an FM8502 core image obtained by running the core image produced by the compiler, assembler, and linker. Thus, verified Piton programs running on FM8502 can be thought of as having been verified down to the gate level. The implementation and proof are described in the Piton manual [Moore 88] and not discussed further here.

In the current chapter, we first provide an informal overview of Piton and then in Section 4.2 give the formal characterization of Piton in the form of an alphabetically arranged list of the Boyer-Moore definitions defining the Piton interpreter. In both our formal and informal characterization, we limit ourselves to the subset of Piton which was used in the Micro-Gypsy translator.

4.1 An Informal Sketch of Piton

Among the features provided by Piton are:

- execute-only program space
- named read/write global data spaces randomly accessed as one-dimensional arrays
- recursive subroutine call and return
- provision of named formal parameters and stack-based parameter passing

[26]This entire chapter is taken, with permission, from Chapter 3 of the Piton report by J Moore. [Moore 88] We have done only very minor editing to eliminate discussion of some aspects not relevant to the Micro-Gypsy work.

- provision of named temporary variables allocated and initialized to constants on call
- a user-visible temporary stack
- seven abstract data types:
    - integers
    - natural numbers
    - bit vectors
    - Booleans
    - data addresses
    - program addresses (labels)
    - subroutine names
- stack-based instructions for manipulating the various abstract objects
- standard flow-of-control instructions
- instructions for determining resource limitations

4.1.1 An Example Piton Program

We begin our presentation of Piton with an example programming problem and its solution in Piton. The problem is to write a program for doing "big number addition." A "big number" is a fixed length array of "digits," each digit being a natural number less than a fixed "base." The intended interpretation of such an array is that it represents the natural number obtained by summing the product of the successive digits and successive powers of the base. In our representation of big numbers we put the least significant digit in position 0. For example, a big number array of length 5 representing the number 123 in base 10 is `(3 2 1 0 0)`. Of course, normally the base of a big number system is the first unrepresentable natural on the host machine. For example, in a 32-bit wide machine, the natural base for big number arithmetic is 2^{32}, so that each digit is a full word.

Big number addition is the process that takes as input two big number arrays and produces as output the big number array representing their sum. For example, the table below shows two naturals, i and j, their corresponding base 100 big-number arrays of length 5 and the two sums (the natural sum and the corresponding big number sum).

naturals	corresponding big numbers
10473250	(34 207 159 0)
+ 3321928714	(10 156 0 198)
3332401964	(44 107 160 198)

In Figure 4-0 we show a Piton subroutine for big number addition. The program is a list constant in the computational logic of Boyer and Moore [BoyerMoore 88] and is displayed in the traditional Lisp-like notation. Comments are written in the right-hand column, bracketed by the comment delimiters semi-colon and end-of-line. The name of the program is `BIG-ADD`. It expects three arguments: `A`, the address of the least significant digit in the first big number array, `B`, the address of the least significant digit in the second big number array, and `N`, the lengths of the two big number arrays. The base of the big number system is implicitly the first unrepresentable natural on the Piton machine. The subroutine sums the two arrays and writes the sum into the first big number array. It leaves on the stack a Boolean indicating whether the sum "carried out" of the array.

```
(BIG-ADD    (A B N)                         ; Formal parameters
            NIL                             ; Temporary variables
                                            ; Body
            (PUSH-CONSTANT (BOOL F))        ; Push the input carry flag for
                                            ; the first ADD-NAT-WITH-CARRY
            (PUSH-LOCAL A)                  ; Push the address A

   (DL LOOP ()                              ; This is the top level loop.
                                            ; Every time we get here the carry
                                            ; flag from the last addition and
                                            ; the current value of A will be
                                            ; on the stack.
            (FETCH))                        ; Fetch next digit from A
            (PUSH-LOCAL B)                  ; Push the address B
            (FETCH)                         ; Fetch next digit from B
            (ADD-NAT-WITH-CARRY)            ; Add the two digits and flag
            (PUSH-LOCAL A)                  ; Deposit the sum digit in A
            (DEPOSIT)                       ; (but leave carry flag)
            (PUSH-LOCAL N)                  ; Decrement N by 1
            (SUB1-NAT)
            (SET-LOCAL N)                   ; (but leave N on the stack)
            (TEST-NAT-AND-JUMP ZERO DONE)   ; If N=0, go to DONE
            (PUSH-LOCAL B)                  ; Increment B by 1
            (PUSH-CONSTANT (NAT 1))
            (ADD-ADDR)
            (POP-LOCAL B)
            (PUSH-LOCAL A)                  ; Increment A by 1
            (PUSH-CONSTANT (NAT 1))
            (ADD-ADDR)
            (SET-LOCAL A)                   ; (but leave A on the stack)
            (JUMP LOOP)                     ; goto LOOP
   (DL DONE ()
            (RET)))                         ; Exit.
```

Figure 4-1: A Piton Program for Big Number Addition

Suppose that **BN1** and **BN2** are the names of two big number arrays of length 80. The following sequence of Piton instructions adds the two big numbers together, overwriting **BN1**.

```
(PUSH-CONSTANT (ADDR (BN1 . 0)))
(PUSH-CONSTANT (ADDR (BN2 . 0)))
(PUSH-CONSTANT (NAT 80))
(CALL BIG-ADD)
```

In addition to its effect on **BN1**, the call of **BIG-ADD** removes the three arguments from the stack and pushes onto the stack the Boolean indicating whether the addition "carried out" of the array.

4.1.2 Piton States

The Piton machine is a conventional von Neumann state transition machine. Roughly speaking, a particular instruction is singled out as the "current instruction" in any Piton state. When "executed" each instruction changes the state in some way, including changing the identity of the current instruction. The Piton machine operates on an initial state by iteratively executing the current instruction until some termination condition is met.

A Piton state, or *p-state*, is a 9-tuple with the following components:

- a *program segment*, defining a system of Piton programs or subroutines;
- a *data segment*, defining a collection of disjoint named indexed data spaces (i.e., global arrays);
- a *temporary stack*;
- three control fields, consisting of
    - a *control stack*, consisting of a stack of *frames*, the top-most frame describing the currently active subroutine invocation and the successive frames describing the hierarchy of suspended invocations;
    - a *program counter*, indicating which instruction in which subroutine is the next to be executed;
    - a *program status word* (*psw*); and
- three resource limitation fields,
    - a *word size*, which governs the size of numeric constants and bit vectors,
    - a *maximum control stack size*, and
    - a *maximum temporary stack size*.

The formalization of this concept is embodied in the function **P-STATE** which is defined on page 84.

The program counter of a p-state names one of the programs in the program segment, which we call the *current program*, and gives the position of one of the instructions in that program's body, which we call the *current instruction*. We say *control* is *in* the current program and *at* the current instruction.

The control stack of the p-state records the history of subroutine invocations leading to the current p-state. The top-most frame of the control stack (which is the only frame directly accessible to any Piton instruction) has two fields in it. One contains the *current bindings* of the local variables of the current program. The other contains the *return program counter*, which is the program counter to which control is to return when the current subroutine exits.

When a subroutine is *called* or *invoked*, a new frame is pushed onto the control stack. The local variables of the called subroutine are bound to the appropriate values and the return program counter is saved. Then control is transferred to the first instruction in the body of the subroutine. All references to local variables in the instructions of the called subroutine refer implicitly to the current bindings. When the subroutine returns to its caller, the top frame of the control stack is popped off, thus restoring the current bindings of the caller extant at the time of call. In short, the values assigned to the local variables of a subroutine are local to a particular invocation and cannot be accessed or changed by any other subroutine or recursive invocation. We define "local variables" and what we mean by the "appropriate values" when we discuss Piton programs.

Recall the program **`BIG-ADD`** shown earlier. Suppose we wished to add the big number (246838082 3116233281 42632655 0) (base 2^{32}) to (3579363592 3979696680 7693250 0) (base 2^{32}). The two big numbers represent the naturals 786,433,689,351,873,913,098,236,738 and 141,915,430,937,733,100,148,932,872, respectively. A suitable Piton initial state for this computation is shown in Figure 4-1.

The nine fields of the p-state in Figure 4-1 are enumerated and named in the comments of the figure. We discuss each field in turn. Field (1) is the program counter. It is **`(PC (MAIN . 0))`**. The tag **`PC`** indicates that the object is a program counter; we discuss tags and types below. The pair **`(MAIN . 0)`** is an address, pointing to the the 0^{th} instruction in the **`MAIN`** program. Field (2) is the control stack. It is a list of frames. The top frame--and in this example, the only frame--describes the local variables and return program counter for the current subroutine invocation. Since the current program counter in is **`MAIN`**, the single frame on the control stack describes the invocation of **`MAIN`**. Since **`MAIN`** has no local variables, the frame has the empty list, **`NIL`**, as the local variable bindings. Since there is only one frame on the stack, it describes the top-level entry into Piton and hence the "return program counter" is completely irrelevant. If control is ever "returned" from this invocation of **`MAIN`** the Piton machine will halt rather than "return control" outside of Piton. However, despite the fact that the initial return program counter is irrelevant we insist that it be a legal program counter and so in this example we chose **`(PC (MAIN . 0))`**. Field (3) is the temporary stack. In this example it is empty. Field (4) is the program segment. It contains two programs. The first is named **`MAIN`** and has no formals, no temporaries, and five instructions. The second is the previously exhibited **`BIG-ADD`** program. Observe that **`MAIN`** calls **`BIG-ADD`**, passing it (a) the address of the 0^{th} element of an array named **`A`**, (b) the address of the 0^{th} element of an array named **`B`**, and (c) the value of the global variable **`N`**. After calling **`BIG-ADD`**, **`MAIN`** "returns," which in this state means the Piton machine halts. Field (5) is the data segment. It contains three "global arrays" named, respectively, **`A`**, **`B`**, and **`N`**. **`A`** and **`B`** are both arrays of length four. **`N`** is an array of length 1 (which we think of as simply a global variable). The **`A`** array contains the first of the two big numbers we wish to add, namely (246838082 3116233281 42632655 0)--except in Piton all data objects are tagged and so instead of writing the raw naturals we tag each with the type **`NAT`** and write

```
((NAT 246838082) (NAT 3116233281) (NAT 42632655) (NAT 0)).
```

The **`B`** array contains the second big number, appropriately tagged. **`N`** contains the (tagged) length of the two arrays. Fields (6)-(8) are, respectively, the maximum control stack size, 10, the maximum temporary stack size, 8, and the word size, 32. The stack sizes declared in this example are unusually small but sufficient for the computation described. Finally, field (9) is the program status word **`'RUN`**.

4.1.3 Type Checking

Piton programs manipulate seven types of data: integers, natural numbers, Booleans, fixed length bit vectors, data addresses, program addresses, and subroutine names.

All objects are "first class" in the sense that they can be passed around and stored into arbitrary variable, stack, and data locations. *There is no type checking in the Piton syntax.* A variable can hold an integer value now and a Boolean value later, for example.

```
(P-STATE '(PC (MAIN . 0))                          ; (1) program counter
         '((NIL (PC (MAIN . 0))))                  ; (2) control stack
         NIL                                       ; (3) temporary stack

         '((MAIN NIL NIL                           ; (4) program segment
                 (PUSH-CONSTANT (ADDR (A . 0)))
                 (PUSH-CONSTANT (ADDR (B . 0)))
                 (PUSH-GLOBAL N)
                 (CALL BIG-ADD)
                 (RET))
           (BIG-ADD (A B N) NIL
                 (PUSH-CONSTANT (BOOL F))
                 (PUSH-LOCAL A)
             (DL LOOP NIL (FETCH))
                 (PUSH-LOCAL B)
                 (FETCH)
                 (ADD-NAT-WITH-CARRY)
                 (PUSH-LOCAL A)
                 (DEPOSIT)
                 (PUSH-LOCAL N)
                 (SUB1-NAT)
                 (SET-LOCAL N)
                 (TEST-NAT-AND-JUMP ZERO DONE)
                 (PUSH-LOCAL B)
                 (PUSH-CONSTANT (NAT 1))
                 (ADD-ADDR)
                 (POP-LOCAL B)
                 (PUSH-LOCAL A)
                 (PUSH-CONSTANT (NAT 1))
                 (ADD-ADDR)
                 (SET-LOCAL A)
                 (JUMP LOOP)
             (DL DONE NIL (RET))))

         '((A    (NAT 246838082)                   ; (5) data segment
                 (NAT 3116233281)
                 (NAT 42632655)
                 (NAT 0))
           (B    (NAT 3579363592)
                 (NAT 3979696680)
                 (NAT 7693250)
                 (NAT 0))
           (N    (NAT 4)))

         10                                        ; (6) max ctrl stk size
         8                                         ; (7) max temp stk size
         32                                        ; (8) word size
         'RUN)                                     ; (9) psw
```

Figure 4-2: An Initial Piton State for Big Number Addition

Each type comes with a set of Piton instructions designed to manipulate objects of that type. For example, the **ADD-NAT** instruction adds two naturals together to produce a natural; the **ADD-ADDR** instruction increments a data address by a natural to produce a new data address. The effects of most instructions are defined only when the operands are of the expected type. For example, the formal definition of Piton does not specify what the **ADD-NAT** instruction does if given a non-natural. However, our implementation of Piton *has no runtime type checking facilities.* The programmer must know what he is doing.

Such cavalier runtime treatment of types--i.e., no syntactic type checking and no runtime type checking--would normally be an invitation to disaster. In most programming languages the definition of the language is embedded in only two mechanical devices: the compiler (where syntactic checks are made) and the runtime system (where semantic checks are made). If some feature of the language (e.g., correct use of the type system) is not checked by either of these two devices, then the programmer had better read the language manual and his program very carefully because he bears the entire responsibility.

But the Piton programmer is relieved of this burden by an unconventional third mechanical device. In addition to a compiler and a run-time system, Piton has a mechanized formal semantics. This device--actually the Boyer-Moore theorem prover initialized with the formal definition of Piton--completely embodies the formal semantics of Piton. If a programmer wishes to establish that he has not violated Piton's type restrictions, he can undertake to prove it mechanically.

As programmers we find this a marvelous state of affairs. We are relieved of the burden of syntactic restrictions in the language--objects can be slung around any way we please. We are relieved of the inefficiency of checking types at runtime. But we don't have to worry about having made mistakes. The price, of course, is that we must be willing to prove our programs correct.

4.1.4 Data Types

As noted, Piton supports seven primitive data types. The syntax of Piton requires that all data objects be tagged by their type. Thus, **(INT 5)** is the way we write the integer 5, while **(NAT 5)** is the way we write the natural number 5. The question "are they the same?" cannot arise in Piton because no operation compares them.

Below we characterize all of the legal instances of each type. However, this must be done with respect to a given p-state, since the p-state determines the resource limitations, legal addresses, etc. We use w as the word size of the p-state implicit in our discussion. In the examples of this section we assume the word size is 8.[27] The formalization of the concept of "legal Piton data object" is embodied in the function **P-OBJECTP** which is defined on page 81.

4.1.4-A Integers

Piton provides the integers, i, in the range $-2^{w-1} \le i < 2^{w-1}$. We say such integers are *representable* in the given p-state. Observe that there is one more representable negative integer than representable positive integers. Integers are written down in the form **(INT I)**, where **I** is an optionally signed integer in decimal notation. For example, **(INT -4)** and **(INT 3)** are Piton integers. Piton provides instructions for adding, subtracting, and comparing integers. It is also possible to convert non-negative integers into naturals.

---

[27]In our FM8502 implementation of Piton we fix the word size at 32.

4.1.4-B Natural Numbers

Piton provides the natural numbers, n, in the range $0 \leq n < 2^w$. We say such naturals are *representable* in the given p-state. Naturals are written down in the form **(NAT N)**, where **N** is an unsigned integer in decimal notation. For example, **(NAT 0)** and **(NAT 7)** are Piton naturals. Piton provides instructions for adding, subtracting, doubling, halving, and comparing naturals. Naturals also play a role in those instructions that do address manipulation, random access into the temporary stack, and some control functions.

4.1.4-C Booleans

There are two Boolean objects, called **T** and **F**. They are written down **(BOOL T)** and **(BOOL F)**.[28] Piton provides the logical operations of conjunction, disjunction, negation and equivalence. Several Piton instructions generate Boolean objects (e.g., the "less than" operators for integers and naturals).

4.1.4-D Bit Vectors

A Piton bit vector is an array of 1's and 0's as long as the word size. Bit vectors are written in the form **(BITV V)** where v is a list of length w, enclosed in parentheses, containing only 1's and 0's. For example (BITV (1 1 1 1 0 0 0 0)) is a bit vector when w is 8. Operations on bit vectors include componentwise conjunction, disjunction, negation, exclusive-or, and equivalence.

4.1.4-E Data Addresses

A Piton data address is a pair consisting of a name and a number. To be legal in a given p-state, the name must be the name of some data area in the data segment of the state and the number must be non-negative and less than the length of the array associated with the named data area. Data addresses are written **(ADDR (NAME . N))**. Such an address refers to the **N**th element of array associated with **NAME**, where enumeration is 0 based, starting at the left hand end of the array. For example, if the data segment of the state contains a data area named **DELTA1** that has an associated array of length 128, then **(ADDR (DELTA1 . 122))** is a data address. The operations on data addresses include incrementing, decrementing, and comparing addresses, fetching the object at an address, and depositing an object at an address.

4.1.4-F Program Addresses

A Piton program address is a pair consisting of a name and a number. To be legal in a given p-state, the name must be the name of some program in the program segment of the state and the number must be non-negative and less than the length of the body of the named program. Program addresses are written **(PC (NAME . N))**. Such an address refers to the **N**th instruction in the body of the program named **NAME**, where enumeration is 0 based starting with the first instruction in the body. For example, if the program segment of the state contains a program named **SETUP** that has 200 instructions in its body, then **(PC (SETUP . 27))** is a legal program address. Program addresses can be compared and control can be transferred to (the instruction at) a program address. Some instructions generate program addresses. But it is impossible to deposit anything at a program address (just as it is impossible to transfer control to a data address).

---

[28] Note to those familiar with the Boyer-Moore logic: The **T** and **F** used in the representation of the Piton Booleans are *not* the **(TRUE)** and **(FALSE)** of the logic but the literal atoms **'T** and **'F** of the logic.

The program counter component of a p-state is an object of this type. For example, to start a computation at the first instruction of the program named **MAIN**, the program counter in the state should be set to **(PC (MAIN . 0))**.

4.1.4-G Subroutines

A Piton subroutine name is just a name. To be legal, it must be the name of some program in the program segment. Subroutine names are written **(SUBR NAME)**. For example, if **SETUP** is the name of a program in the program segment, then **(SUBR SETUP)** is a subroutine object in Piton. The only operation on subroutine objects is to call them.

4.1.5 The Data Segment

The Piton data segment contains all of the global data in a p-state. The data segment is a list of *data areas*. Each data area consists of a literal atom *data area name* followed by one or more Piton objects, called the *array* associated with the name. The objects in the array are implicitly indexed from 0, starting with the leftmost. Using data addresses, which specify a name and an index, Piton programs can access and change the elements in an array.

We sometimes call a data area name a *global variable*. Some Piton instructions expect global variables as their arguments and operate on the 0^{th} position of the named data area. We define the *value* of a global variable to be the contents of the 0^{th} location in its associated array. This is a pleasant convention if the data area only has one element but tends to be confusing otherwise.

Here, for example, is a data segment:

```
((LEN (NAT 5))
 (A   (NAT 0)
      (NAT 1)
      (NAT 2)
      (NAT 3)
      (NAT 4))
 (X   (INT -23)
      (NAT 256)
      (BOOL T)
      (BITV (1 0 1 0 1 1 0 0))
      (ADDR (A . 3))
      (PC (SETUP . 25))
      (SUBR MAIN))).
```

This segment contains three data areas, **LEN**, **A**, and **X**. The **LEN** area has only one element and so is naturally thought of as a global variable. Its value is the natural number 5. The **A** array is of length 5 and contains the consecutive naturals starting from 0. While **A** is of homogeneous type as shown, Piton programs may write arbitrary objects into **A**. The third data area, **X**, has an associated array of length 7. It happens that this array contains one object of every Piton type.

Let **ADDR** be the Piton data address object **(ADDR (X . 1))**. If we fetch from **ADDR** we get **(NAT 256)**. If we deposit **(NAT 7)** at **ADDR** the data segment becomes

```
((LEN (NAT 5))
 (A   (NAT 0)
      (NAT 1)
      (NAT 2)
      (NAT 3)
      (NAT 4))
 (X   (INT -23)
      (NAT 7)
      (BOOL T)
      (BITV (1 0 1 0 1 1 0 0))
      (ADDR (A . 3))
      (PC (SETUP . 25))
      (SUBR MAIN))).
```

If we increment **ADDR** by one and then fetch from **ADDR** we get **(BOOL T)**.

The individual data areas are totally isolated from each other. Despite the fact that addresses can be incremented and decremented, there is no way for a Piton program to manipulate **ADDR**, which addresses the area named **X**, so as to obtain an address into the area named **A**.

4.1.6 The Program Segment

The program segment of a p-state is a list of "program definitions". A *program definition* (or, interchangeably, a *subroutine definition*) is an object of the following form

$$
\begin{array}{ll}
\texttt{(name} & \texttt{(v}_0\ \texttt{v}_1\ \ldots\ \texttt{v}_{n-1}\texttt{)} \\
 & \texttt{((v}_n\ \texttt{i}_n\texttt{)}\ \texttt{(v}_{n+1}\ \texttt{i}_{n+1}\texttt{)}\ \ \ldots\ \texttt{(v}_{n+k-1}\ \texttt{i}_{n+k-1}\texttt{))} \\
 & \texttt{ins}_0 \\
 & \texttt{ins}_1 \\
 & \ldots \\
 & \texttt{ins}_m\texttt{),}
\end{array}
$$

where **NAME** is the *name* of the program and is some literal atom; V_0, ..., V_{N-1} are the $N \geq 0$ *formal parameters* of the program and are literal atoms; V_N, ..., V_{N+K-1} are the $K \geq 0$ *temporary variables* of the program and are literal atoms; I_N, ..., I_{N+K-1} are the *initial values* of the corresponding temporary variables and are data objects in the state in which the program occurs; and INS_0, ... INS_M are **M+1** optionally labeled Piton instructions, called the *body* of the program. The body must be non-empty.

The *local variables* of a program are the formal parameters together with the temporary variables. The values of the local variables of a subroutine may be accessed and changed by position as well as by name. For this purpose we enumerate the local variables starting from 0 in the same order they are displayed above.

As noted previously, upon subroutine call the local variables of the called subroutine are bound to the "appropriate values" in the stack frame created for that invocation. The **N** formal parameters are initialized from the temporary stack. The top-most **N** elements of the temporary stacks are called the *actuals* for the call. They are removed from the temporary stack and become the values of formals. The association is in reverse order of the formals; i.e., the last formal, V_{N-1}, is bound to the object on the top of the temporary stack at the time of the call and V_0 is bound to the object **N** down from the top at the time of the call. The **K** temporary variables are bound to their respective initial values at the time of call.

The instructions in the body of a Piton program may be optionally labeled. A *label* is a literal atom. To attach label **LAB** to an instruction, **INS**, write

```
(DL LAB COMMENT INS)
```

where **COMMENT** is any object in the logic and is totally ignored by the Piton semantics and implementation. We say **LAB** is *defined* in a program if the body of the program contains **(DL LAB ...)** as one of its members. Such a form is called a *def-label form* because it defines a label. Label definitions are local to the program in which they occur. Use of the atom **LOOP**, for example, as a label in some instruction in a program refers to the (first) point in that program at which **LOOP** is defined in a def-label form.

Because of the local nature of label definitions it is not possible for one program to jump to a label in another. A similar effect can be obtained efficiently using the data objects of type **PC**. The **POPJ** instruction transfers control to the program address on the top of the temporary stack--provided that address is in the current program.[29] Thus, if subroutine **MASTER** wants to jump to the 23rd instruction of subroutine **SLAVE**, it could **CALL SLAVE** and pass it the argument **(PC (SLAVE . 22))**, and **SLAVE** could do a **POPJ** as its first instruction to branch to the desired location.

The last instruction, $\mathbf{INS_M}$ must be a return or some form of unconditional jump. It is not permitted to "fall off" the end of a Piton program.

4.1.7 Instructions

We now list the current Piton instructions together with a brief summary of their semantics. The language as currently defined provides 65 instructions. The Micro-Gypsy translator requires only a subset of these and we delete the descriptions of some of the Piton instructions. The reader interested in the remaining instructions should consult the Piton manual. We do not regard the current instruction set as fixed in granite; we imagine Piton will continue to evolve to suit the needs of its users.

We describe each instruction informally by explaining the syntactic form of the instruction, the preconditions on its execution, and the effects of executing an acceptable instance of the instruction. Unless otherwise indicated, every instruction increments the program counter by one so that the next instruction to be executed is the instruction following the current one in the current subroutine. All references to "the stack" refer to the temporary stack unless otherwise specified. When we say "push" or "pop" without mentioning a particular stack we mean to push or pop the temporary stack. The formal characterization of the Piton instuctions is given in Section 4.2 and is a precise manual for the subset of Piton used in our translator. The references to page numbers below refer to the formal definitions of the corresponding functions or predicates.

(CALL SUBR) *Precondition* (page 75): Suppose that **SUBR** has n formal variables and k temporary variables. Then the temporary stack must contain at least n items and the control stack must have at least 2+n+k free slots. *Effect* (page 76): Transfer control to the first instruction in the body of **SUBR** after removing the top-most n elements from the temporary stack and constructing a new frame on the control stack. In the new frame the formals of **SUBR** are bound to the n elements removed from the temporary stack, in reverse order, the temporaries of **SUBR** are bound to their declared initial values, and the return program counter points to the instruction after the **CALL**.

(RET) *Precondition* (page 83): None. *Effect* (page 84): If the control stack contains only one frame (i.e., if the current invocation is the top-level entry into Piton) **'HALT** the machine. Otherwise, set the program counter to the return program counter in the top-most frame of the control stack and pop that frame off the control stack.

(PUSH-CONSTANT CONST)

Precondition (page 82): There is room to push at least one item. *Effect* (page 82): If **CONST** is a Piton object, push **CONST**; if **CONST** is the atom **PC**, push the program counter of the next instruction; otherwise push the program counter corresponding to the label **CONST**.

[29]There is no way, in Piton, to transfer control into another subroutine except via the call/return mechanism.

(PUSH-LOCAL LVAR)
Precondition (page 83): There is room to push at least one item. *Effect* (page 83): Push the value of the local variable **LVAR**.

(PUSH-GLOBAL GVAR)
Precondition (page 83): There is room to push at least one item. *Effect* (page 83): Push the value of the global variable **GVAR**, i.e., the contents of position 0 in the array associated with **GVAR**.

(PUSH-TEMP-STK-INDEX N)
Precondition (page 83): **N** is less than the length of the temporary stack and there is room to push at least one item. *Effect* (page 83): Push the natural number (length-**N**)-1, where length is the current length of the temporary stack. Note: We permit the temporary stack to be accessed randomly as an array. The elements in the stack are enumerated from 0 starting at the *bottom-most* so that pushes and pops do not change the positions of undisturbed elements. This instruction converts from a top-most-first enumeration to our enumeration. That is, it pushes onto the temporary stack the index of the element **N** removed from the top. See also **FETCH-TEMP-STK** and **DEPOSIT-TEMP-STK**.

(POP) *Precondition* (page 82): There is at least one item on the stack. *Effect* (page 82): Pop and discard the top of the stack.

(POP* N) *Precondition* (page 81): there are at least **N** items on the stack. *Effect* (page 81): Pop and discard the top-most **N** items.

(POP-LOCAL LVAR) *Precondition* (page 82): There is an object, val, on top of the stack. *Effect* (page 82): Pop and assign val to the local variable **LVAR**.

(POP-GLOBAL GVAR)
Precondition (page 81): There is an object, val, on top of the stack. *Effect* (page 82): Pop and assign val to the (0^{th} position of the array associated with the) global variable **GVAR**.

(FETCH-TEMP-STK) *Precondition* (page 77): There is a natural number, n, on top of the stack and n is less than the length of the stack. *Effect* (page 77): Let val be the n^{th} element of the stack, where elements are enumerated from 0 starting at the bottom-most element. Pop once and then push val.

(DEPOSIT-TEMP-STK)
Precondition (page 76): There is a natural number, n, on top of the stack and some object, val, immediately below it. Furthermore, n is less than the length of the stack after popping two elements. *Effect* (page 76): Pop twice and then deposit val at the n^{th} position in the temporary stack, where positions are enumerated from 0 starting at the bottom.

(JUMP LAB) *Precondition* (page 79): None. *Effect* (page 79): Jump to **LAB**.

(JUMP-CASE LAB0 LAB1 ... LABN)
Precondition (page 79): There is a natural, i, on top of the stack and i is less than **N**. *Effect* (page 79): Pop once and then jump to **LABI**.

(SET-LOCAL LVAR) *Precondition* (page 84): There is an object, val, on top of the stack. *Effect* (page 84): Assign val to the local variable **LVAR**. The stack is not popped.

(TEST-NAT-AND-JUMP TEST LAB)

Precondition (page 87): There is a natural, n, on top of the stack. *Effect* (page 87): Pop once and then jump to **LAB** if **TEST** is satisfied, as indicated below.

test	condition tested
ZERO	$n = 0$
NOT-ZERO	$n \neq 0$

(TEST-INT-AND-JUMP TEST LAB)

Precondition (page 87): There is an integer, i, on top of the stack. *Effect* (page 87): Pop once and then jump to **LAB** if **TEST** is satisfied, as indicated below.

test	condition tested
NEG	$i < 0$
NOT-NEG	$i \geq 0$
ZERO	$i = 0$
NOT-ZERO	$i \neq 0$
POS	$i > 0$
NOT-POS	$i \leq 0$

(TEST-BOOL-AND-JUMP TEST LAB)

Precondition (page 86): There is a Boolean, b, on top of the stack. *Effect* (page 87): Pop once and then jump to **LAB** if **TEST** is satisfied, as indicated below.

test	condition tested
T	b = **T**
F	b = **F**

(NO-OP) *Precondition* (page 80): None. *Effect* (page 80): Do nothing; continue execution.

(EQ) *Precondition* (page 76): The temporary stack contains at least two items and the top two are of the same type. *Effect* (page 77): Pop twice and then push the Boolean **T** if they are the same and the Boolean **F** if they are not.

(ADD-INT-WITH-CARRY)

Precondition (page 74): There is an integer, i, on top of the stack, an integer, j, immediately below it, and a Boolean, c, below that. *Effect* (page 74): Pop three times. Let k be 1 if c is **T** and 0 otherwise. Let sum be i+j+k. If sum is representable in the word size, w, of this p-state, push the Boolean **F** and then the integer sum; if sum is not representable and is negative, push the Boolean **T** and the integer sum+2^w; if sum is not representable and positive, push the Boolean **T** and the integer sum-2^w.

(SUB-INT) *Precondition* (page 84): There is an integer, i, on top of the stack and an integer, j, immediately below it. j-i is representable. *Effect* (page 85): Pop twice and then push the integer j-i.

(SUB-INT-WITH-CARRY)

Precondition (page 85): There is an integer, i, on top of the stack, an integer, j, immediately below it, and a Boolean, c, below that. *Effect* (page 85): Pop three times. Let k be 1 if c is **T** and 0 otherwise. Let diff be the integer j-(i+k). If diff is representable in the word size, w, of this p-state, push the Boolean **F** and the integer diff; if diff is not representable and is negative, push the Boolean **T** and the integer diff+2^w; if diff is not representable and is positive, push the Boolean **T** and the integer diff-2^w.

(NEG-INT) *Precondition* (page 80): There is an integer, i, on top of the stack and -i is representable. *Effect* (page 80): Pop once and then push the integer -i.

(LT-INT) *Precondition* (page 79): There is an integer, i, on top of the stack and an integer, j, immediately below it. *Effect* (page 80): Pop twice and then push the Boolean **T** if j < i and the Boolean **F** otherwise.

(INT-TO-NAT) *Precondition* (page 79): There is a non-negative integer, i, on top of the stack. *Effect* (page 79): Pop and then push the natural i.

(ADD-NAT) *Precondition* (page 75):There is a natural, i, on top of the stack and a natural, j, immediately below it. j+i is representable. *Effect* (page 75): Pop twice and then push the natural j+i.

(SUB-NAT) *Precondition* (page 85): There is a natural, i, on top of the stack and natural, j, immediately below it. Furthermore, j ≥ i. *Effect* (page 85): Pop twice and then push the natural j-i.

(SUB1-NAT) *Precondition* (page 86): There is a non-zero natural, i, on top of the stack. *Effect* (page 86): Pop and then push the natural i-1.

(OR-BOOL) *Precondition* (page 81): There is a Boolean, 1, on top of the stack and a Boolean, b2, immediately below it. *Effect* (page 81): Pop twice and then push the Boolean disjunction of b1 and b2.

(AND-BOOL) *Precondition* (page 75): There is a Boolean, b1, on top of the stack and a Boolean, b2, immediately below it. *Effect* (page 75): Pop twice and then push the Boolean conjunction of b1 and b2.

(NOT-BOOL) *Precondition* (page 80): There is a Boolean, b, on top of the stack. *Effect* (page 80): Pop once and then push the Boolean negation of b.

4.1.8 The Piton Interpreter

Associated with each instruction is a predicate on p-states called the *ok predicate* or the *precondition* for the instruction. This predicate insures that it is legal to execute the instruction in the current p-state. Generally speaking, the precondition of an instruction checks that the operands exist, have the appropriate types and do not cause the machine to exceed its resource limits.

Also associated with each instruction is a function from p-states to p-states called the *step* or *effects* function. The step function for an instruction defines the state produced by executing the instruction, provided the precondition is satisfied. Most of the step functions increment the program counter by one and manipulate the stacks and/or global data segment.

The Piton interpreter is a typical von Neumann state transition machine. The interpreter iteratively constructs the new current state by applying the step function for the current instruction to the current state, provided the precondition is satisfied. This process stops, if at all, either when a precondition is unsatisfied or a top-level return instruction is executed.[30] The property of being a proper p-state is preserved by the Piton interpreter. That is, if the initial state is proper, so is the final state. The formalization of the Piton interpreter is the function **P** which is defined on page 74.

Recall the **BIG-ADD** program and the example initial state shown in Figure 4-1, page 64. The state was set

[30]We formalize this machine constructively by defining the function that iterates the process a given number of times.

up to compute the big number sum of (246838082 3116233281 42632655 0) and (3579363592 3979696680 7693250 0), base 2^{32}. These big numbers represent the naturals 786,433,689,351,873,913, 098,236,738 and 141,915,430,937,733,100,148,932,872, respectively. By running the Piton machine 75 steps from that initial state one obtains the state shown in Figure 4-2.

Observe that the psw in Figure 4-2 is **'HALT**. This tells us the computation terminated without error. The program counter points to the **RET** statement in the **MAIN** program, the last instruction executed. The control stack is exactly as it was in the initial state. The temporary stack has the Boolean value **F** on it, indicating that the addition did not carry out of the big number arrays. The program segment and resource limits are exactly as they were in the initial state--they are never changed. The final value of the **A** array in the data segment is now the big number (3826201674 2800962665 50325906 0). A little arithmetic will confirm that this big number represents the natural 928,349,120,289,607,013,247,169,610, which is the sum of 786,433,689,351,873,913,098,236,738 and 141,915,430,937,733,100,148,932,872, as desired.

4.1.9 Erroneous States

What does the Piton machine do when the precondition for the current instruction is not satisfied? This brings us to the role of the program status word, psw, in the state and its use in error handling. This has important consequences in the design, implementation, and proof of Piton.

The psw is normally set to the literal atom **'RUN**, which indicates that the computation is proceeding normally. The psw is set to **'HALT** by the **RET** (return) instruction when executed in the top level program; the **'HALT** psw indicates successful termination of the computation. The psw is set to one of many *error conditions* whenever the precondition for the current instruction is not satisfied. Any state with a psw other than **'RUN** or **'HALT** is called an *erroneous* state. The Piton interpreter is defined as an identity function on erroneous states.

No Piton instruction (i.e., no precondition or step function) inspects the psw. It is impossible for a Piton program to trap or mask an error. The psw and the notion of erroneous states are metatheoretic concepts in Piton; they are used to define the language but are not part of the language.

We consider an implementation of Piton correct if it has the property that it can successfully carry out every computation that produces a non-erroneous state. This is made formal in the Piton manual. But the consequences to the implementation should be clear now. For example, the **ADD-NAT** instruction requires that two natural numbers be on top of the stack and that their sum is representable. This need not be checked at run-time by the implementation of Piton. The run-time code for **ADD-NAT** can simply add together the top two elements of the stack and increment the program counter. If the Piton machine produces a non-erroneous state on the **ADD-NAT** instruction, then the implementation follows it faithfully. If the Piton machine produces an erroneous state, then it does not matter what the implementation does. For example, our implementation of **ADD-NAT** does not check that the stack has two elements, that the top two elements are naturals, or that their sum is representable. It is difficult even to characterize the damage that might be caused if these conditions are not satisfied when our code is executed. As noted in our discussion of type checking, mechanical proof can be used to certify that no such errors occur.

The language contains adequate facilities to program explicit checks for all resource errors. For example, **ADD-NAT-WITH-CARRY** will not only add two naturals together, it will push a Boolean which indicates whether the result is the true sum. If you have to test whether the sum is representable, use **ADD-NAT-WITH-CARRY**. On the other hand, if you *know* the result is representable, use **ADD-NAT**.

But, unless you are adding constants together, how can you possibly know the result is representable?

```
(P-STATE '(PC (MAIN . 4))                          ; program counter
         '((NIL (PC (MAIN . 0))))                  ; control stack
         '((BOOL F))                               ; temporary stack
         '((MAIN NIL NIL                           ; program segment
                 (PUSH-CONSTANT (ADDR (A . 0)))
                 (PUSH-CONSTANT (ADDR (B . 0)))
                 (PUSH-GLOBAL N)
                 (CALL BIG-ADD)
                 (RET))
           (BIG-ADD (A B N) NIL
                 (PUSH-CONSTANT (BOOL F))
                 (PUSH-LOCAL A)
             (DL LOOP NIL (FETCH))
                 (PUSH-LOCAL B)
                 (FETCH)
                 (ADD-NAT-WITH-CARRY)
                 (PUSH-LOCAL A)
                 (DEPOSIT)
                 (PUSH-LOCAL N)
                 (SUB1-NAT)
                 (SET-LOCAL N)
                 (TEST-NAT-AND-JUMP ZERO DONE)
                 (PUSH-LOCAL B)
                 (PUSH-CONSTANT (NAT 1))
                 (ADD-ADDR)
                 (POP-LOCAL B)
                 (PUSH-LOCAL A)
                 (PUSH-CONSTANT (NAT 1))
                 (ADD-ADDR)
                 (SET-LOCAL A)
                 (JUMP LOOP)
             (DL DONE NIL (RET))))

         '((A    (NAT 3826201674)                  ; data segment
                 (NAT 2800962665)
                 (NAT 50325906)
                 (NAT 0))
           (B    (NAT 3579363592)
                 (NAT 3979696680)
                 (NAT 7693250)
                 (NAT 0))
           (N    (NAT 4)))

         10                                        ; max ctrl stk size
         8                                         ; max temp stk size
         32                                        ; word size
         'HALT)                                    ; psw
```

Figure 4-3: A Final Piton State for Big Number Addition

That is, under what conditions can you to use **ADD-NAT** and still prove the absence of errors? This brings us to the crux of the problem. When you write a Piton program and prove it non-erroneous you do not have to prove the total absence of errors. You *do* have to state the conditions under which the program may be called and prove the absence of errors under those conditions. For example, a typical hypothesis about the initial state might be that the sum of the top two elements of the stack is representable and the stack contains at least 5 free cells. These conditions are expressed in the logic, not in Piton.

4.2 An Alphabetical Listing of the Piton Interpreter Definition

This section contains an alphabetical listing of the Boyer-Moore definitions defining the subset of Piton used in the compiler. Various parts of the language--bit vectors, for example--are not used at all by the Micro-Gypsy work. The definitions have been adjusted to expunge any references to bit vectors and the other parts of Piton which are not used. The reader should consult the Piton manual [Moore 88] for the full language definition.

This section is self contained except that some subordinate functions which were used in defining Micro-Gypsy are not repeated here. These are the functions **APPEND**, **ASSOC**, **DEFINEDP**, **EXP**, **GET**, **IDIFFERENCE**, **ILESSP**, **INEGATE**, **INTEGERP**, **IPLUS**, **PUT**, **SMALL-INTEGERP**, **TAG**, and **UNTAG**. These definitions are found in Section 3.5.

```
DEFINITION.
(ADD-ADDR ADDR N) = (TAG (TYPE ADDR) (ADD-ADP (UNTAG ADDR) N))

DEFINITION.
(ADD-ADP ADP N) = (CONS (ADP-NAME ADP) (PLUS (ADP-OFFSET ADP) N))

DEFINITION.
(ADD1-ADDR ADDR) = (ADD-ADDR ADDR 1)

DEFINITION.
(ADD1-ADP ADP) = (ADD-ADP ADP 1)

DEFINITION.
(ADD1-NAT NAT)
   =
(TAG 'NAT (ADD1 (UNTAG NAT)))

DEFINITION.
(ADD1-P-PC P) = (ADD1-ADDR (P-PC P))

DEFINITION.
(ADP-NAME ADP) = (CAR ADP)

DEFINITION.
(ADP-OFFSET ADP) = (CDR ADP)

DEFINITION.
(ADPP X SEGMENT)
   =
(AND (LISTP X)
     (NUMBERP (ADP-OFFSET X))
     (DEFINEDP (ADP-NAME X) SEGMENT)
     (LESSP (ADP-OFFSET X)
            (LENGTH (VALUE (ADP-NAME X) SEGMENT))))

DEFINITION.
(AND-BOOL X Y) = (IF (EQUAL X 'F) 'F Y)

DEFINITION.
(AREA-NAME X) = (ADP-NAME (UNTAG X))

DEFINITION.
(BINDINGS FRAME) = (CAR FRAME)

DEFINITION.
(BOOL X) = (TAG 'BOOL (IF X 'T 'F))
```

DEFINITION.

```
(BOOL-TO-NAT FLG) = (IF (EQUAL FLG 'F) 0 1)
```

DEFINITION.

```
(BOOLEANP X) = (OR (EQUAL X 'T) (EQUAL X 'F))
```

DEFINITION.

```
(DEFINITION NAME ALIST) = (ASSOC NAME ALIST)
```

DEFINITION.

```
(DEPOSIT VAL ADDR SEGMENT)
   =
(DEPOSIT-ADP VAL (UNTAG ADDR) SEGMENT)
```

DEFINITION.

```
(DEPOSIT-ADP VAL ADP SEGMENT)
   =
(PUT-VALUE (PUT VAL
                (ADP-OFFSET ADP)
                (VALUE (ADP-NAME ADP) SEGMENT))
           (ADP-NAME ADP)
           SEGMENT)
```

DEFINITION.

```
(FETCH ADDR SEGMENT)
   =
(FETCH-ADP (UNTAG ADDR) SEGMENT)
```

DEFINITION.

```
(FETCH-ADP ADP SEGMENT)
   =
(GET (ADP-OFFSET ADP)
     (VALUE (ADP-NAME ADP) SEGMENT))
```

DEFINITION.

```
(FIND-LABEL X LST)
   =
(COND ((NLISTP LST) 0)
      ((AND (LABELLEDP (CAR LST))
            (EQUAL X (CADAR LST)))
       0)
      (T (ADD1 (FIND-LABEL X (CDR LST)))))
```

DEFINITION.

```
(FIND-LABELP X LST)
   =
(COND ((NLISTP LST) F)
      ((AND (LABELLEDP (CAR LST))
            (EQUAL X (CADAR LST)))
       T)
      (T (FIND-LABELP X (CDR LST))))
```

DEFINITION.

```
(FIRST-N N X)
   =
(IF (ZEROP N)
    NIL
    (CONS (CAR X)
          (FIRST-N (SUB1 N) (CDR X))))
```

DEFINITION.

```
(FIX-SMALL-INTEGER I WORD-SIZE)
   =
(COND ((SMALL-INTEGERP I WORD-SIZE) I)
      ((NEGATIVEP I)
       (IPLUS I (EXP 2 WORD-SIZE)))
      (T (IPLUS I (MINUS (EXP 2 WORD-SIZE)))))
```

DEFINITION.

```
(FORMAL-VARS D) = (CADR D)
```

DEFINITION.

```
(LABELLEDP X) = (EQUAL (CAR X) 'DL)
```

DEFINITION.
```
(LOCAL-VAR-VALUE VAR CTRL-STK)
   =
(VALUE VAR (BINDINGS (TOP CTRL-STK)))
```

DEFINITION.
```
(MAKE-P-CALL-FRAME FORMAL-VARS TEMP-STK TEMP-VAR-DCLS RET-PC)
   =
(P-FRAME (APPEND (PAIR-FORMAL-VARS-WITH-ACTUALS FORMAL-VARS TEMP-STK)
                 (PAIR-TEMPS-WITH-INITIAL-VALUES TEMP-VAR-DCLS))
         RET-PC)
```

DEFINITION.
```
(NOT-BOOL X) = (IF (EQUAL X 'F) 'T 'F)
```

DEFINITION.
```
(OFFSET X) = (ADP-OFFSET (UNTAG X))
```

DEFINITION.
```
(OR-BOOL X Y) = (IF (EQUAL X 'F) Y 'T)
```

DEFINITION.
```
(P P N)
   =
(IF (ZEROP N)
    P
    (P (P-STEP P) (SUB1 N)))
```

DEFINITION.
```
(P-ADD-INT-WITH-CARRY-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (LISTP (POP (P-TEMP-STK P)))
     (LISTP (POP (POP (P-TEMP-STK P))))
     (P-OBJECTP-TYPE 'INT (TOP (P-TEMP-STK P)) P)
     (P-OBJECTP-TYPE 'INT (TOP1 (P-TEMP-STK P)) P)
     (P-OBJECTP-TYPE 'BOOL (TOP2 (P-TEMP-STK P)) P))
```

DEFINITION.
```
(P-ADD-INT-WITH-CARRY-STEP INS P)
   =
(P-STATE
 (ADD1-P-PC P)
 (P-CTRL-STK P)
 (PUSH
  (TAG 'INT
       (FIX-SMALL-INTEGER (IPLUS (BOOL-TO-NAT (UNTAG (TOP2 (P-TEMP-STK P))))
                                 (IPLUS (UNTAG (TOP1 (P-TEMP-STK P)))
                                        (UNTAG (TOP (P-TEMP-STK P)))))
                          (P-WORD-SIZE P)))
  (PUSH
   (BOOL
    (NOT (SMALL-INTEGERP (IPLUS (BOOL-TO-NAT (UNTAG (TOP2 (P-TEMP-STK P))))
                                (IPLUS (UNTAG (TOP1 (P-TEMP-STK P)))
                                       (UNTAG (TOP (P-TEMP-STK P)))))
                         (P-WORD-SIZE P))))
   (POP (POP (POP (P-TEMP-STK P))))))
 (P-PROG-SEGMENT P)
 (P-DATA-SEGMENT P)
 (P-MAX-CTRL-STK-SIZE P)
 (P-MAX-TEMP-STK-SIZE P)
 (P-WORD-SIZE P)
 'RUN)
```

DEFINITION.
```
(P-ADD-NAT-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (LISTP (POP (P-TEMP-STK P)))
     (P-OBJECTP-TYPE 'NAT (TOP (P-TEMP-STK P)) P)
     (P-OBJECTP-TYPE 'NAT (TOP1 (P-TEMP-STK P)) P)
     (SMALL-NATURALP (PLUS (UNTAG (TOP1 (P-TEMP-STK P)))
                           (UNTAG (TOP (P-TEMP-STK P))))
                     (P-WORD-SIZE P)))
```

DEFINITION.
```
(P-ADD-NAT-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (TAG 'NAT
                    (PLUS (UNTAG (TOP1 (P-TEMP-STK P)))
                          (UNTAG (TOP (P-TEMP-STK P)))))
               (POP (POP (P-TEMP-STK P))))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

DEFINITION.
```
(P-AND-BOOL-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (LISTP (POP (P-TEMP-STK P)))
     (P-OBJECTP-TYPE 'BOOL (TOP (P-TEMP-STK P)) P)
     (P-OBJECTP-TYPE 'BOOL (TOP1 (P-TEMP-STK P)) P))
```

DEFINITION.
```
(P-AND-BOOL-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (TAG 'BOOL
                    (AND-BOOL (UNTAG (TOP1 (P-TEMP-STK P)))
                              (UNTAG (TOP (P-TEMP-STK P)))))
               (POP (POP (P-TEMP-STK P))))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

DEFINITION.
```
(P-CALL-OKP INS P)
   =
(AND
 (NOT
  (LESSP
   (P-MAX-CTRL-STK-SIZE P)
   (P-CTRL-STK-SIZE
    (PUSH (MAKE-P-CALL-FRAME (FORMAL-VARS (DEFINITION (CADR INS)
                                                      (P-PROG-SEGMENT P)))
                             (P-TEMP-STK P)
                             (TEMP-VAR-DCLS (DEFINITION (CADR INS)
                                                        (P-PROG-SEGMENT P)))
                             (ADD1-ADDR (P-PC P)))
          (P-CTRL-STK P)))))
 (NOT (LESSP (LENGTH (P-TEMP-STK P))
             (LENGTH (FORMAL-VARS (DEFINITION (CADR INS)
                                              (P-PROG-SEGMENT P)))))))
```

DEFINITION.
```
(P-CALL-STEP INS P)
   =
(P-STATE
 (TAG 'PC (CONS (CADR INS) 0))
 (PUSH (MAKE-P-CALL-FRAME (FORMAL-VARS (DEFINITION (CADR INS)
                                                   (P-PROG-SEGMENT P)))
                          (P-TEMP-STK P)
                          (TEMP-VAR-DCLS (DEFINITION (CADR INS)
                                                     (P-PROG-SEGMENT P)))
                          (ADD1-ADDR (P-PC P)))
       (P-CTRL-STK P))
 (POPN (LENGTH (FORMAL-VARS (DEFINITION (CADR INS)
                              (P-PROG-SEGMENT P))))
       (P-TEMP-STK P))
 (P-PROG-SEGMENT P)
 (P-DATA-SEGMENT P)
 (P-MAX-CTRL-STK-SIZE P)
 (P-MAX-TEMP-STK-SIZE P)
 (P-WORD-SIZE P)
 'RUN)
```

DEFINITION.
```
(P-CTRL-STK-SIZE CTRL-STK)
   =
(IF (NLISTP CTRL-STK)
    0
    (PLUS (P-FRAME-SIZE (TOP CTRL-STK))
          (P-CTRL-STK-SIZE (CDR CTRL-STK))))
```

DEFINITION.
```
(P-CURRENT-INSTRUCTION P)
   =
(UNLABEL (GET (OFFSET (P-PC P))
              (PROGRAM-BODY (P-CURRENT-PROGRAM P))))
```

DEFINITION.
```
(P-CURRENT-PROGRAM P) = (DEFINITION (AREA-NAME (P-PC P)) (P-PROG-SEGMENT P))
```

DEFINITION.
```
(P-DEPOSIT-TEMP-STK-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (LISTP (POP (P-TEMP-STK P)))
     (P-OBJECTP-TYPE 'NAT (TOP (P-TEMP-STK P)) P)
     (LESSP (UNTAG (TOP (P-TEMP-STK P)))
            (LENGTH (POP (POP (P-TEMP-STK P))))))
```

DEFINITION.
```
(P-DEPOSIT-TEMP-STK-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (RPUT (TOP1 (P-TEMP-STK P))
               (UNTAG (TOP (P-TEMP-STK P)))
               (POP (POP (P-TEMP-STK P))))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

DEFINITION.
```
(P-EQ-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (LISTP (POP (P-TEMP-STK P)))
     (EQUAL (TYPE (TOP (P-TEMP-STK P)))
            (TYPE (TOP1 (P-TEMP-STK P)))))
```

DEFINITION.
```
(P-EQ-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (BOOL (EQUAL (UNTAG (TOP1 (P-TEMP-STK P)))
                            (UNTAG (TOP (P-TEMP-STK P)))))
               (POP (POP (P-TEMP-STK P))))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

DEFINITION.
```
(P-FETCH-TEMP-STK-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (P-OBJECTP-TYPE 'NAT (TOP (P-TEMP-STK P)) P)
     (LESSP (UNTAG (TOP (P-TEMP-STK P)))
            (LENGTH (P-TEMP-STK P))))
```

DEFINITION.
```
(P-FETCH-TEMP-STK-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (RGET (UNTAG (TOP (P-TEMP-STK P)))
                     (P-TEMP-STK P))
               (POP (P-TEMP-STK P)))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

DEFINITION.
```
(P-FRAME BINDINGS RET-PC) = (LIST BINDINGS RET-PC)
```

DEFINITION.
```
(P-FRAME-SIZE FRAME) = (PLUS 2 (LENGTH (BINDINGS FRAME)))
```

DEFINITION.
```
(P-HALT P PSW)
   =
(P-STATE (P-PC P)
         (P-CTRL-STK P)
         (P-TEMP-STK P)
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         PSW)
```

DEFINITION.
```
(P-INS-OKP INS P)
   =
(CASE (CAR INS)
      (CALL (P-CALL-OKP INS P))
      (RET (P-RET-OKP INS P))
      (PUSH-CONSTANT (P-PUSH-CONSTANT-OKP INS P))
      (PUSH-LOCAL (P-PUSH-LOCAL-OKP INS P))
      (PUSH-GLOBAL (P-PUSH-GLOBAL-OKP INS P))
      (PUSH-TEMP-STK-INDEX (P-PUSH-TEMP-STK-INDEX-OKP INS P))
      (POP (P-POP-OKP INS P))
      (POP* (P-POP*-OKP INS P))
      (POP-LOCAL (P-POP-LOCAL-OKP INS P))
      (POP-GLOBAL (P-POP-GLOBAL-OKP INS P))
```

```
      (FETCH-TEMP-STK (P-FETCH-TEMP-STK-OKP INS P))
      (DEPOSIT-TEMP-STK (P-DEPOSIT-TEMP-STK-OKP INS P))
      (JUMP (P-JUMP-OKP INS P))
      (JUMP-CASE (P-JUMP-CASE-OKP INS P))
      (SET-LOCAL (P-SET-LOCAL-OKP INS P))
      (TEST-NAT-AND-JUMP (P-TEST-NAT-AND-JUMP-OKP INS P))
      (TEST-INT-AND-JUMP (P-TEST-INT-AND-JUMP-OKP INS P))
      (TEST-BOOL-AND-JUMP (P-TEST-BOOL-AND-JUMP-OKP INS P))
      (NO-OP (P-NO-OP-OKP INS P))
      (EQ (P-EQ-OKP INS P))
      (ADD-INT-WITH-CARRY (P-ADD-INT-WITH-CARRY-OKP INS P))
      (SUB-INT (P-SUB-INT-OKP INS P))
      (SUB-INT-WITH-CARRY (P-SUB-INT-WITH-CARRY-OKP INS P))
      (NEG-INT (P-NEG-INT-OKP INS P))
      (LT-INT (P-LT-INT-OKP INS P))
      (INT-TO-NAT (P-INT-TO-NAT-OKP INS P))
      (ADD-NAT (P-ADD-NAT-OKP INS P))
      (SUB-NAT (P-SUB-NAT-OKP INS P))
      (SUB1-NAT (P-SUB1-NAT-OKP INS P))
      (OR-BOOL (P-OR-BOOL-OKP INS P))
      (AND-BOOL (P-AND-BOOL-OKP INS P))
      (NOT-BOOL (P-NOT-BOOL-OKP INS P))
      (OTHERWISE F))
```

DEFINITION.

```
(P-INS-STEP INS P)
   =
(CASE (CAR INS)
      (CALL (P-CALL-STEP INS P))
      (RET (P-RET-STEP INS P))
      (PUSH-CONSTANT (P-PUSH-CONSTANT-STEP INS P))
      (PUSH-LOCAL (P-PUSH-LOCAL-STEP INS P))
      (PUSH-GLOBAL (P-PUSH-GLOBAL-STEP INS P))
      (PUSH-TEMP-STK-INDEX (P-PUSH-TEMP-STK-INDEX-STEP INS P))
      (POP (P-POP-STEP INS P))
      (POP* (P-POP*-STEP INS P))
      (POP-LOCAL (P-POP-LOCAL-STEP INS P))
      (POP-GLOBAL (P-POP-GLOBAL-STEP INS P))
      (FETCH-TEMP-STK (P-FETCH-TEMP-STK-STEP INS P))
      (DEPOSIT-TEMP-STK (P-DEPOSIT-TEMP-STK-STEP INS P))
      (JUMP (P-JUMP-STEP INS P))
      (JUMP-CASE (P-JUMP-CASE-STEP INS P))
      (SET-LOCAL (P-SET-LOCAL-STEP INS P))
      (TEST-NAT-AND-JUMP (P-TEST-NAT-AND-JUMP-STEP INS P))
      (TEST-INT-AND-JUMP (P-TEST-INT-AND-JUMP-STEP INS P))
      (TEST-BOOL-AND-JUMP (P-TEST-BOOL-AND-JUMP-STEP INS P))
      (NO-OP (P-NO-OP-STEP INS P))
      (EQ (P-EQ-STEP INS P))
      (ADD-INT-WITH-CARRY (P-ADD-INT-WITH-CARRY-STEP INS P))
      (SUB-INT (P-SUB-INT-STEP INS P))
      (SUB-INT-WITH-CARRY (P-SUB-INT-WITH-CARRY-STEP INS P))
      (NEG-INT (P-NEG-INT-STEP INS P))
      (LT-INT (P-LT-INT-STEP INS P))
      (INT-TO-NAT (P-INT-TO-NAT-STEP INS P))
      (ADD-NAT (P-ADD-NAT-STEP INS P))
      (SUB-NAT (P-SUB-NAT-STEP INS P))
      (SUB1-NAT (P-SUB1-NAT-STEP INS P))
      (OR-BOOL (P-OR-BOOL-STEP INS P))
      (AND-BOOL (P-AND-BOOL-STEP INS P))
      (NOT-BOOL (P-NOT-BOOL-STEP INS P))
      (OTHERWISE (P-HALT P 'RUN)))
```

DEFINITION.
```
(P-INT-TO-NAT-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (P-OBJECTP-TYPE 'INT
                     (TOP (P-TEMP-STK P))
                     P)
     (NOT (NEGATIVEP (UNTAG (TOP (P-TEMP-STK P))))))
```

DEFINITION.
```
(P-INT-TO-NAT-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (TAG 'NAT
                    (UNTAG (TOP (P-TEMP-STK P))))
               (POP (P-TEMP-STK P)))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

DEFINITION.
```
(P-JUMP-CASE-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (P-OBJECTP-TYPE 'NAT (TOP (P-TEMP-STK P)) P)
     (LESSP (UNTAG (TOP (P-TEMP-STK P)))
            (LENGTH (CDR INS))))
```

DEFINITION.
```
(P-JUMP-CASE-STEP INS P)
   =
(P-STATE (PC (GET (UNTAG (TOP (P-TEMP-STK P)))
                  (CDR INS))
             (P-CURRENT-PROGRAM P))
         (P-CTRL-STK P)
         (POP (P-TEMP-STK P))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

DEFINITION.
```
(P-JUMP-OKP INS P) = T
```

DEFINITION.
```
(P-JUMP-STEP INS P)
   =
(P-STATE (PC (CADR INS) (P-CURRENT-PROGRAM P))
         (P-CTRL-STK P)
         (P-TEMP-STK P)
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

DEFINITION.
```
(P-LT-INT-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (LISTP (POP (P-TEMP-STK P)))
     (P-OBJECTP-TYPE 'INT (TOP (P-TEMP-STK P)) P)
     (P-OBJECTP-TYPE 'INT (TOP1 (P-TEMP-STK P)) P))
```

DEFINITION.
```
(P-LT-INT-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (BOOL (ILESSP (UNTAG (TOP1 (P-TEMP-STK P)))
                             (UNTAG (TOP (P-TEMP-STK P)))))
               (POP (POP (P-TEMP-STK P))))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

DEFINITION.
```
(P-NEG-INT-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (P-OBJECTP-TYPE 'INT (TOP (P-TEMP-STK P)) P)
     (SMALL-INTEGERP (INEGATE (UNTAG (TOP (P-TEMP-STK P))))
                     (P-WORD-SIZE P)))
```

DEFINITION.
```
(P-NEG-INT-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (TAG 'INT
                    (INEGATE (UNTAG (TOP (P-TEMP-STK P)))))
               (POP (P-TEMP-STK P)))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

DEFINITION.
```
(P-NO-OP-OKP INS P) = T
```

DEFINITION.
```
(P-NO-OP-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (P-TEMP-STK P)
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

DEFINITION.
```
(P-NOT-BOOL-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (P-OBJECTP-TYPE 'BOOL (TOP (P-TEMP-STK P)) P))
```

DEFINITION.
```
(P-NOT-BOOL-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (TAG 'BOOL
                    (NOT-BOOL (UNTAG (TOP (P-TEMP-STK P)))))
               (POP (P-TEMP-STK P)))
```

```
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

DEFINITION.

```
(P-OBJECTP X P)
   =
(AND (LISTP X)
     (EQUAL (CDDR X) NIL)
     (CASE (TYPE X)
           (NAT (SMALL-NATURALP (UNTAG X) (P-WORD-SIZE P)))
           (INT (SMALL-INTEGERP (UNTAG X) (P-WORD-SIZE P)))
           (BOOL (BOOLEANP (UNTAG X)))
           (ADDR (ADPP (UNTAG X) (P-DATA-SEGMENT P)))
           (PC (PCPP (UNTAG X) (P-PROG-SEGMENT P)))
           (SUBR (DEFINEDP (UNTAG X)
                   (P-PROG-SEGMENT P)))
           (OTHERWISE F)))
```

DEFINITION.

```
(P-OBJECTP-TYPE TYPE X P) = (AND (EQUAL (TYPE X) TYPE) (P-OBJECTP X P))
```

DEFINITION.

```
(P-OR-BOOL-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (LISTP (POP (P-TEMP-STK P)))
     (P-OBJECTP-TYPE 'BOOL (TOP (P-TEMP-STK P)) P)
     (P-OBJECTP-TYPE 'BOOL (TOP1 (P-TEMP-STK P)) P))
```

DEFINITION.

```
(P-OR-BOOL-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (TAG 'BOOL
                    (OR-BOOL (UNTAG (TOP1 (P-TEMP-STK P)))
                             (UNTAG (TOP (P-TEMP-STK P)))))
               (POP (POP (P-TEMP-STK P))))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

DEFINITION.

```
(P-POP*-OKP INS P)
   =
(NOT (LESSP (LENGTH (P-TEMP-STK P))
            (CADR INS)))
```

DEFINITION.

```
(P-POP*-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (POPN (CADR INS) (P-TEMP-STK P))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

DEFINITION.

```
(P-POP-GLOBAL-OKP INS P) = (LISTP (P-TEMP-STK P))
```

DEFINITION.
```
(P-POP-GLOBAL-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (POP (P-TEMP-STK P))
         (P-PROG-SEGMENT P)
         (DEPOSIT (TOP (P-TEMP-STK P))
                  (TAG 'ADDR (CONS (CADR INS) 0))
                  (P-DATA-SEGMENT P))
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

DEFINITION.
```
(P-POP-LOCAL-OKP INS P) = (LISTP (P-TEMP-STK P))
```

DEFINITION.
```
(P-POP-LOCAL-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (SET-LOCAL-VAR-VALUE (TOP (P-TEMP-STK P))
                              (CADR INS)
                              (P-CTRL-STK P))
         (POP (P-TEMP-STK P))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

DEFINITION.
```
(P-POP-OKP INS P) = (LISTP (P-TEMP-STK P))
```

DEFINITION.
```
(P-POP-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (POP (P-TEMP-STK P))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

DEFINITION.
```
(P-PUSH-CONSTANT-OKP INS P)
   =
(LESSP (LENGTH (P-TEMP-STK P))
       (P-MAX-TEMP-STK-SIZE P))
```

DEFINITION.
```
(P-PUSH-CONSTANT-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (UNABBREVIATE-CONSTANT (CADR INS) P)
               (P-TEMP-STK P))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

DEFINITION.

```
(P-PUSH-GLOBAL-OKP INS P)
   =
(LESSP (LENGTH (P-TEMP-STK P))
       (P-MAX-TEMP-STK-SIZE P))
```

DEFINITION.

```
(P-PUSH-GLOBAL-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (FETCH (TAG 'ADDR (CONS (CADR INS) 0))
                      (P-DATA-SEGMENT P))
               (P-TEMP-STK P))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

DEFINITION.

```
(P-PUSH-LOCAL-OKP INS P)
   =
(LESSP (LENGTH (P-TEMP-STK P))
       (P-MAX-TEMP-STK-SIZE P))
```

DEFINITION.

```
(P-PUSH-LOCAL-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (LOCAL-VAR-VALUE (CADR INS)
                                (P-CTRL-STK P))
               (P-TEMP-STK P))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

DEFINITION.

```
(P-PUSH-TEMP-STK-INDEX-OKP INS P)
   =
(AND (LESSP (LENGTH (P-TEMP-STK P))
            (P-MAX-TEMP-STK-SIZE P))
     (LESSP (CADR INS)
            (LENGTH (P-TEMP-STK P))))
```

DEFINITION.

```
(P-PUSH-TEMP-STK-INDEX-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (TAG 'NAT
                    (SUB1 (DIFFERENCE (LENGTH (P-TEMP-STK P))
                                      (CADR INS))))
               (P-TEMP-STK P))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

DEFINITION.

```
(P-RET-OKP INS P) = T
```

DEFINITION.
```
(P-RET-STEP INS P)
    =
(IF (LISTP (POP (P-CTRL-STK P)))
    (P-STATE (RET-PC (TOP (P-CTRL-STK P)))
             (POP (P-CTRL-STK P))
             (P-TEMP-STK P)
             (P-PROG-SEGMENT P)
             (P-DATA-SEGMENT P)
             (P-MAX-CTRL-STK-SIZE P)
             (P-MAX-TEMP-STK-SIZE P)
             (P-WORD-SIZE P)
             'RUN)
    (P-HALT P 'HALT))
```

DEFINITION.
```
(P-SET-LOCAL-OKP INS P)
    =
(LISTP (P-TEMP-STK P))
```

DEFINITION.
```
(P-SET-LOCAL-STEP INS P)
    =
(P-STATE (ADD1-P-PC P)
         (SET-LOCAL-VAR-VALUE (TOP (P-TEMP-STK P))
                              (CADR INS)
                              (P-CTRL-STK P))
         (P-TEMP-STK P)
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

SHELL DEFINITION.
Add the shell **P-STATE** of **9** arguments, with recognizer function symbol **P-STATEP**, and accessors **P-PC**, **P-CTRL-STK**, **P-TEMP-STK**, **P-PROG-SEGMENT**, **P-DATA-SEGMENT**, **P-MAX-CTRL-STK-SIZE**, **P-MAX-TEMP-STK-SIZE**, **P-WORD-SIZE** and **P-PSW**.

DEFINITION.
```
(P-STEP P)
    =
(IF (EQUAL (P-PSW P) 'RUN)
    (P-STEP1 (P-CURRENT-INSTRUCTION P) P)
    P)
```

DEFINITION.
```
(P-STEP1 INS P)
    =
(IF (P-INS-OKP INS P)
    (P-INS-STEP INS P)
    (P-HALT P (X-Y-ERROR-MSG 'P (CAR INS))))
```

DEFINITION.
```
(P-SUB-INT-OKP INS P)
    =
(AND (LISTP (P-TEMP-STK P))
     (LISTP (POP (P-TEMP-STK P)))
     (P-OBJECTP-TYPE 'INT (TOP (P-TEMP-STK P)) P)
     (P-OBJECTP-TYPE 'INT (TOP1 (P-TEMP-STK P)) P)
     (SMALL-INTEGERP (IDIFFERENCE (UNTAG (TOP1 (P-TEMP-STK P)))
                                  (UNTAG (TOP (P-TEMP-STK P))))
                     (P-WORD-SIZE P)))
```

DEFINITION.
```
(P-SUB-INT-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (TAG 'INT
                    (IDIFFERENCE (UNTAG (TOP1 (P-TEMP-STK P)))
                                 (UNTAG (TOP (P-TEMP-STK P)))))
               (POP (POP (P-TEMP-STK P))))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

DEFINITION.
```
(P-SUB-INT-WITH-CARRY-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (LISTP (POP (P-TEMP-STK P)))
     (LISTP (POP (POP (P-TEMP-STK P))))
     (P-OBJECTP-TYPE 'INT (TOP (P-TEMP-STK P)) P)
     (P-OBJECTP-TYPE 'INT (TOP1 (P-TEMP-STK P)) P)
     (P-OBJECTP-TYPE 'BOOL (TOP2 (P-TEMP-STK P)) P))
```

DEFINITION.
```
(P-SUB-INT-WITH-CARRY-STEP INS P)
   =
(P-STATE
 (ADD1-P-PC P)
 (P-CTRL-STK P)
 (PUSH
  (TAG 'INT
       (FIX-SMALL-INTEGER
        (IDIFFERENCE (UNTAG (TOP1 (P-TEMP-STK P)))
                     (IPLUS (UNTAG (TOP (P-TEMP-STK P)))
                            (BOOL-TO-NAT (UNTAG (TOP2 (P-TEMP-STK P))))))
        (P-WORD-SIZE P)))
  (PUSH
   (BOOL
    (NOT
     (SMALL-INTEGERP
      (IDIFFERENCE (UNTAG (TOP1 (P-TEMP-STK P)))
                   (IPLUS (UNTAG (TOP (P-TEMP-STK P)))
                          (BOOL-TO-NAT (UNTAG (TOP2 (P-TEMP-STK P))))))
      (P-WORD-SIZE P))))
   (POP (POP (POP (P-TEMP-STK P))))))
 (P-PROG-SEGMENT P)
 (P-DATA-SEGMENT P)
 (P-MAX-CTRL-STK-SIZE P)
 (P-MAX-TEMP-STK-SIZE P)
 (P-WORD-SIZE P)
 'RUN)
```

DEFINITION.
```
(P-SUB-NAT-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (LISTP (POP (P-TEMP-STK P)))
     (P-OBJECTP-TYPE 'NAT (TOP (P-TEMP-STK P)) P)
     (P-OBJECTP-TYPE 'NAT (TOP1 (P-TEMP-STK P)) P)
     (NOT (LESSP (UNTAG (TOP1 (P-TEMP-STK P)))
                 (UNTAG (TOP (P-TEMP-STK P))))))
```

DEFINITION.
```
(P-SUB-NAT-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
```

```
         (PUSH (TAG 'NAT
                    (DIFFERENCE (UNTAG (TOP1 (P-TEMP-STK P)))
                                (UNTAG (TOP (P-TEMP-STK P)))))
               (POP (POP (P-TEMP-STK P))))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

DEFINITION.

```
(P-SUB1-NAT-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (P-OBJECTP-TYPE 'NAT (TOP (P-TEMP-STK P)) P)
     (NOT (ZEROP (UNTAG (TOP (P-TEMP-STK P))))))
```

DEFINITION.

```
(P-SUB1-NAT-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (TAG 'NAT (SUB1 (UNTAG (TOP (P-TEMP-STK P)))))
               (POP (P-TEMP-STK P)))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

DEFINITION.

```
(P-TEST-AND-JUMP-OKP INS TYPE TEST P)
   =
(AND (LISTP (P-TEMP-STK P))
     (P-OBJECTP-TYPE TYPE (TOP (P-TEMP-STK P)) P))
```

DEFINITION.

```
(P-TEST-AND-JUMP-STEP TEST LAB P)
   =
(IF TEST
    (P-STATE (PC LAB (P-CURRENT-PROGRAM P))
             (P-CTRL-STK P)
             (POP (P-TEMP-STK P))
             (P-PROG-SEGMENT P)
             (P-DATA-SEGMENT P)
             (P-MAX-CTRL-STK-SIZE P)
             (P-MAX-TEMP-STK-SIZE P)
             (P-WORD-SIZE P)
             'RUN)
    (P-STATE (ADD1-P-PC P)
             (P-CTRL-STK P)
             (POP (P-TEMP-STK P))
             (P-PROG-SEGMENT P)
             (P-DATA-SEGMENT P)
             (P-MAX-CTRL-STK-SIZE P)
             (P-MAX-TEMP-STK-SIZE P)
             (P-WORD-SIZE P)
             'RUN))
```

DEFINITION.

```
(P-TEST-BOOL-AND-JUMP-OKP INS P)
   =
(P-TEST-AND-JUMP-OKP INS 'BOOL
                     (P-TEST-BOOLP (CADR INS)
                                   (UNTAG (TOP (P-TEMP-STK P))))
                     P)
```

DEFINITION.

```
(P-TEST-BOOL-AND-JUMP-STEP INS P)
   =
(P-TEST-AND-JUMP-STEP (P-TEST-BOOLP (CADR INS) (UNTAG (TOP (P-TEMP-STK P))))
                      (CADDR INS)
                      P)
```

DEFINITION.

```
(P-TEST-BOOLP FLG X)
   =
(IF (EQUAL FLG 'T) (EQUAL X 'T) (EQUAL X 'F))
```

DEFINITION.

```
(P-TEST-INT-AND-JUMP-OKP INS P)
   =
(P-TEST-AND-JUMP-OKP INS 'INT
                     (P-TEST-INTP (CADR INS)
                                  (UNTAG (TOP (P-TEMP-STK P))))
                     P)
```

DEFINITION.

```
(P-TEST-INT-AND-JUMP-STEP INS P)
   =
(P-TEST-AND-JUMP-STEP (P-TEST-INTP (CADR INS) (UNTAG (TOP (P-TEMP-STK P))))
                      (CADDR INS)
                      P)
```

DEFINITION.

```
(P-TEST-INTP FLG X)
   =
(CASE FLG
      (ZERO (EQUAL X 0))
      (NOT-ZERO (NOT (EQUAL X 0)))
      (NEG (NEGATIVEP X))
      (NOT-NEG (NOT (NEGATIVEP X)))
      (POS (AND (NUMBERP X) (NOT (EQUAL X 0))))
      (OTHERWISE (OR (EQUAL X 0) (NEGATIVEP X))))
```

DEFINITION.

```
(P-TEST-NAT-AND-JUMP-OKP INS P)
   =
(P-TEST-AND-JUMP-OKP INS 'NAT
                     (P-TEST-NATP (CADR INS) (UNTAG (TOP (P-TEMP-STK P))))
                     P)
```

DEFINITION.

```
(P-TEST-NAT-AND-JUMP-STEP INS P)
   =
(P-TEST-AND-JUMP-STEP (P-TEST-NATP (CADR INS) (UNTAG (TOP (P-TEMP-STK P))))
                      (CADDR INS)
                      P)
```

DEFINITION.

```
(P-TEST-NATP FLG X)
   =
(IF (EQUAL FLG 'ZERO) (EQUAL X 0) (NOT (EQUAL X 0)))
```

DEFINITION.

```
(PAIR-FORMAL-VARS-WITH-ACTUALS FORMAL-VARS TEMP-STK)
   =
(PAIRLIST FORMAL-VARS
          (REVERSE (FIRST-N (LENGTH FORMAL-VARS) TEMP-STK)))
```

DEFINITION.

```
(PAIR-TEMPS-WITH-INITIAL-VALUES TEMP-VAR-DCLS)
   =
(IF (NLISTP TEMP-VAR-DCLS)
    NIL
    (CONS (CONS (CAAR TEMP-VAR-DCLS)
                (CADAR TEMP-VAR-DCLS))
          (PAIR-TEMPS-WITH-INITIAL-VALUES (CDR TEMP-VAR-DCLS))))
```

DEFINITION.
```
(PAIRLIST LST1 LST2)
   =
(IF (NLISTP LST1)
    NIL
    (CONS (CONS (CAR LST1) (CAR LST2))
          (PAIRLIST (CDR LST1) (CDR LST2))))
```

DEFINITION.
```
(PC LABEL PROGRAM)
   =
(TAG 'PC
     (CONS (CAR PROGRAM)
           (FIND-LABEL LABEL
                       (PROGRAM-BODY PROGRAM))))
```

DEFINITION.
```
(PCPP X SEGMENT)
   =
(AND (LISTP X)
     (NUMBERP (ADP-OFFSET X))
     (DEFINEDP (ADP-NAME X) SEGMENT)
     (LESSP (ADP-OFFSET X)
            (LENGTH (PROGRAM-BODY (DEFINITION (ADP-NAME X) SEGMENT)))))
```

DEFINITION.
```
(POP STK) = (CDR STK)
```

DEFINITION.
```
(POPN N X)
   =
(IF (ZEROP N)
    X
    (POPN (SUB1 N) (CDR X)))
```

DEFINITION.
```
(PROGRAM-BODY D) = (CDDDR D)
```

DEFINITION.
```
(PUSH X STK) = (CONS X STK)
```

DEFINITION.
```
(PUT-ASSOC VAL NAME ALIST)
   =
(COND ((NLISTP ALIST) ALIST)
      ((EQUAL NAME (CAAR ALIST))
       (CONS (CONS NAME VAL) (CDR ALIST)))
      (T (CONS (CAR ALIST)
               (PUT-ASSOC VAL NAME (CDR ALIST)))))
```

DEFINITION.
```
(PUT-VALUE VAL NAME ALIST) = (PUT-ASSOC VAL NAME ALIST)
```

DEFINITION.
```
(RET-PC FRAME) = (CADR FRAME)
```

DEFINITION.
```
(REVERSE X)
   =
(IF (LISTP X)
    (APPEND (REVERSE (CDR X))
            (LIST (CAR X)))
    NIL)
```

DEFINITION.
```
(RGET N LST) = (GET (SUB1 (DIFFERENCE (LENGTH LST) N))
     LST)
```

DEFINITION.
```
(RPUT VAL N LST)
   =
(PUT VAL (SUB1 (DIFFERENCE (LENGTH LST) N)) LST)
```

DEFINITION.
```
(SET-LOCAL-VAR-VALUE VAL VAR CTRL-STK)
   =
(PUSH (P-FRAME (PUT-VALUE VAL VAR
                          (BINDINGS (TOP CTRL-STK)))
               (RET-PC (TOP CTRL-STK)))
      (POP CTRL-STK))
```

DEFINITION.
```
(SMALL-NATURALP I WORD-SIZE)
   =
(AND (NUMBERP I)
     (LESSP I (EXP 2 WORD-SIZE)))
```

DEFINITION.
```
(SUB-ADDR ADDR N) = (TAG (TYPE ADDR) (SUB-ADP (UNTAG ADDR) N))
```

DEFINITION.
```
(SUB-ADP ADP N) = (CONS (ADP-NAME ADP) (DIFFERENCE (ADP-OFFSET ADP) N))
```

DEFINITION.
```
(SUB1-ADDR ADDR) = (SUB-ADDR ADDR 1)
```

DEFINITION.
```
(TEMP-VAR-DCLS D) = (CADDR D)
```

DEFINITION.
```
(TOP STK) = (CAR STK)
```

DEFINITION.
```
(TOP1 STK) = (TOP (POP STK))
```

DEFINITION.
```
(TOP2 STK) = (TOP (POP (POP STK)))
```

DEFINITION.
```
(TYPE CONST) = (CAR CONST)
```

DEFINITION.
```
(UNABBREVIATE-CONSTANT C P)
   =
(COND ((EQUAL C 'PC) (ADD1-P-PC P))
      ((NLISTP C)
       (PC C (P-CURRENT-PROGRAM P)))
      (T C))
```

DEFINITION.
```
(UNLABEL X) = (IF (LABELLEDP X) (CADDDR X) X)
```

DEFINITION.
```
(VALUE NAME ALIST) = (CDR (ASSOC NAME ALIST))
```

DEFINITION.
```
(X-Y-ERROR-MSG X Y)
   =
(PACK (APPEND (UNPACK 'ILLEGAL-)
              (APPEND (UNPACK Y)
                      (CDR (UNPACK 'G-INSTRUCTION)))))
```

Chapter 5
TRANSLATING MICRO-GYPSY INTO PITON

In Chapter 2, we described a means of expressing a relationship between execution environments for two languages in the form of an *interpreter equivalence theorem*. This involves defining interpreters for the languages and mapping functions *Map-Down* and *Map-Up* between the two abstract spaces in which programs in the languages execute. In Chapters 3 and 4 we have seen how to define interpreters for our two languages of interest, Micro-Gypsy and Piton, and how to characterize the respective execution environments.

We now turn our attention to mapping Micro-Gypsy execution environments into Piton environments, that is, to defining our version of the abstract *Map-Down* function. Given a Micro-Gypsy execution environment, our goal for the translator is the ability to define a Piton execution environment which *implements* it in the sense described in Chapter 2. A large part of this process is what is commonly regarded as compilation--translating Micro-Gypsy statements into lines of Piton code. However, that is not all. We must also decide how to characterize the other components of a Micro-Gypsy execution environment in Piton. This will occupy quite a bit of our attention in this chapter.

Recall from Chapter 3 that a total Micro-Gypsy execution environment includes a diversity of structures including a statement, procedure list, current condition, variable alist, psw, and resource limitations. In addition, there is a "clock" parameter which gives us a maximum number of steps we may run our Micro-Gypsy program. In Chapter 4 we saw that the Piton execution environment is more compact; many structures separate in Micro-Gypsy are bundled into a single state. There is also a clock parameter to the Piton interpreter which tells us how often to step the Piton state.

Our translation must show how each of the components of the Micro-Gypsy execution environment are represented in the Piton state. In this chapter we discuss the representation of Micro-Gypsy data and code in Piton. We then show how a Piton state can be constructed which represents an entire Micro-Gypsy execution environment. The final section of the chapter contains the formal characterization of the mapping from Micro-Gypsy to Piton in the form of an alphabetical listing of the Boyer-Moore functions involved in the definition.

5.1 Representing Micro-Gypsy Data in Piton

5.1.1 Translating Micro-Gypsy Literals

It is straightforward to convert the Micro-Gypsy simple literals into Piton data structures. The following table illustrates the correspondences.

Micro-Gypsy literal	Piton literal
(INT-MG n)	(INT n)
(CHARACTER-MG m)	(INT m)
(BOOLEAN-MG FALSE-MG)	(BOOL F)
(BOOLEAN-MG TRUE-MG)	(BOOL T)

Notice that a Piton integer literal may represent either a Micro-Gypsy integer or character. Thus, though the mapping from Micro-Gypsy to Piton is well-defined; the inverse mapping requires the Micro-Gypsy type information.[31] This point is discussed further in Chapter 7. Micro-Gypsy array literals are merely lists of simple literals. We can obviously translate them on an element-wise basis.

5.1.2 Storing Data in Piton

Given the structure of the Piton state as described in Chapter 4, we have some freedom in deciding *where* to store data. The obvious choice is to store all data in the data-segment component of the p-state; for technical reasons we chose not to do this. We store all data values on the Piton temp-stk. Simple literals take up one slot on the temp-stk; arrays of length *n* take *n* contiguous positions in the temp-stk. The Piton temp-stk is our "memory."

We will discuss in Section 5.6.1 how the data gets there initially. How do we access it? The Piton operations **FETCH-TEMP-STK** and **DEPOSIT-TEMP-STK** allow random access to positions within the temp-stk, given a Piton natural which represents the index of that element in the stack. This provides a convenient way of storing and accessing data. We need merely maintain an association list with a pair **<NAME INDEX>** for each of the data structures. For arrays, the index indicates the beginning of the array on the stack. Array elements are accessible at an offset from the index.

For example, suppose we have the **MG-ALIST** containing the following data:

```
((X INT-MG (INT-MG -12))
 (Y BOOLEAN-MG (BOOLEAN-MG TRUE-MG))
 (A (ARRAY-MG INT-MG 3) ((INT-MG 4) (INT-MG 0) (INT-MG -6)))).
```

Suppose that **X** is stored at location 10, **Y** at location 3, and the array **A** beginning at location 6. This temp-stk is illustrated below. The ellipsis indicate other data.

index	contents	represents
11	...	
10	(int -12)	X
9	...	
8	(int -6)	A[2]
7	(int 0)	A[1]
6	(int 4)	A[0]
5	...	
4	...	
3	(bool t)	Y
2	...	
1	...	
0	...	

[31] We might have avoided an instance of this problem by representing Micro-Gypsy characters as Piton naturals. This would patch only small part of a more pervasive problem.

We maintain access to this data by keeping available the alist

```
((X . 10) (Y . 3) (A . 6))
```

which associates names of data structures with pointers to their locations on the temp-stk. We call this list the *pointer-alist*. This alist, along with the type information which tells us the size of the structure, allows us to access any of the simple data structures and, by an appropriate index computation, any element of an array.

Recall that a frame on the Piton ctrl-stk is of the form **<BINDINGS, RET-PC>** where **BINDINGS** is a list of **<NAME, VALUE>** pairs. We use this feature for storing the pointer-alist. We arrange to store the translation of each of the **MG-ALIST** elements on the temp-stk and a **<NAME, POINTER>** pair for that structure in the bindings component of the top element of the Piton ctrl-stk. These pointers thus become the values of Piton *local* data items which can be accessed via Piton operations such as **PUSH-LOCAL**, **POP-LOCAL**, etc. Given this arrangement, the following Piton program fragment allows us to fetch the value of **X**, for instance.

```
(PUSH-LOCAL X)                 ; push the index of  X  onto the temp-stk
(FETCH-TEMP-STK)               ; pop the index and use it to fetch
                               ;       the value from within the stk
```

The following computation allows us to fetch **A[1]**.

```
(PUSH-LOCAL A)                 ; push the index of  A
(PUSH-CONSTANT (NAT 1))        ; push the literal index
(ADD-NAT)                      ; pop two natural values and add
(FETCH-TEMP-STK)               ; fetch temp-stk value  A+1
```

We explain in Section 5.6.1 below how the **MG-ALIST** data is initially stored on the temp-stk and how the pointer-alist is created and stored in the bindings component of the top ctrl-stk frame.

5.2 Representing Conditions in Piton

In Micro-Gypsy conditions are represented as Boyer-Moore literal atoms. There is no similar Piton data type which is the obvious choice for representing conditions in Piton. However, if we assume that we have a list of all of the possible conditions which could arise, we could simply use the index of a condition in that list to represent the condition as a Piton natural number. That is the strategy we follow.

For various reasons we represent the conditions **'NORMAL**, **'ROUTINEERROR**, and **'LEAVE** specially. Assume that we have available a list containing all of the other conditions which could possibly be raised.[32] If our condition list has the form **{CONDITION$_1$, ..., CONDITION$_K$}**, we represent Micro-Gypsy conditions in Piton with the following scheme:

condition	Piton representation
'LEAVE	(NAT 0)
'ROUTINEERROR	(NAT 1)
'NORMAL	(NAT 2)
condition$_i$	(NAT i+2)

There is exactly one condition stored in the Micro-Gypsy state at any time, the value of the **CC** component of the state. Consequently, it is only necessary to store a single natural in the Piton state to represent the current condition. We allocate the global variable **CC** in the Piton data-segment for the representation of the current condition. This is the only use that we make of the Piton data-segment. A Micro-Gypsy state

[32]Recall from Chapter 3 that for any statement the only conditions which it can raise are **'ROUTINEERROR**, one of the formal or local conditions of the enclosing procedure, and possibly **'LEAVE**.

with a **CC** value of **'NORMAL** will be represented by a Piton state with data segment **((CC (NAT 2)))**. The representation of the current condition in the Piton state then is Piton *global* data and is accessible via instructions such as **PUSH-GLOBAL** and **POP-GLOBAL**. Setting the current condition to **'ROUTINEERROR**, for example, requires the instructions

```
(PUSH-CONSTANT (NAT 1))
(POP-GLOBAL CC)
```

Notice that since conditions are represented as Piton naturals, only a fixed finite number can be represented. In particular, our scheme would fail if the length of the condition list exceeded $(2^{word\text{-}size} - 3)$. We make this restriction an hypothesis to our proof, though in practice it is extremely unlikely that any program will require 4,294,967,293 separate conditions.

5.3 Translating Micro-Gypsy Statements

The primary function for translating Micro-Gypsy code into Piton code is **TRANSLATE** (page 121). In this section we discuss the parameters to **TRANSLATE** and its output for each of the Micro-Gypsy statement types.

5.3.1 The Translation Context

A Micro-Gypsy statement is translated in a context consisting of the following:

- **CINFO**: a record structure which encodes the current "state" of the translation;
- **COND-LIST**: a list of condition names used for translating Micro-Gypsy conditions to Piton naturals as described in Section 5.2;
- **PROC-LIST**: the Micro-Gypsy procedure list as described in Section 3.3.4.

The **CINFO** structure is formally defined by a Boyer-Moore shell definition:

Shell Definition.
Add the shell **MAKE-CINFO** of **3** arguments, with
recognizer function symbol **CINFOP**, and
accessors **CODE**, **LABEL-ALIST** and **LABEL-CNT**.

A **CINFO** is returned by the **TRANSLATE** function; the fields of this record are utilized as follows.

- **CODE** is used to accumulate the Piton instruction list being generated. At the end of the translation of a Micro-Gypsy statement **STMT**, **CODE** should contain the Piton code fragment which is the translation of **STMT**.
- **LABEL-CNT** is a counter incremented with each use so that new labels can be generated at will.
- **LABEL-ALIST** is an association list of pairs **<CONDITION-NAME, LABEL>**. We maintain the invariant that at any point in the translation, any condition which might be raised is represented on the **LABEL-ALIST**. The associated label marks the position in the code to which a signal of the condition should branch. The code at that position might be a handler for the condition or possibly the end of the enclosing procedure.

TRANSLATE is illustrated in figure 5-1. **TRANSLATE** does a large case split on the statement operator and, for each statement type, returns an updated **CINFO** reflecting the results of translating **STMT**. We now consider each of the statement types in turn.

Definition.

```
(TRANSLATE CINFO COND-LIST STMT PROC-LIST)
   =
(CASE (CAR STMT)
 (NO-OP-MG CINFO)
 (SIGNAL-MG
    (MAKE-CINFO
     (APPEND (CODE CINFO)
             (LIST (LIST 'PUSH-CONSTANT
                         (MG-COND-TO-P-NAT (SIGNALLED-CONDITION STMT)
                                           COND-LIST))
                   (LIST 'POP-GLOBAL 'C-C)
                   (LIST 'JUMP (FETCH-LABEL (SIGNALLED-CONDITION STMT)
                                            (LABEL-ALIST CINFO)))))
     (LABEL-ALIST CINFO)
     (LABEL-CNT CINFO)))
 (PROG2-MG
  (TRANSLATE (TRANSLATE CINFO COND-LIST
                        (PROG2-LEFT-BRANCH STMT) PROC-LIST)
             COND-LIST
             (PROG2-RIGHT-BRANCH STMT) PROC-LIST))

 (LOOP-MG
  (DISCARD-LABEL
   (ADD-CODE
    (TRANSLATE
     (MAKE-CINFO (APPEND (CODE CINFO)
                         (LIST (LIST 'DL (LABEL-CNT CINFO) NIL '(NO-OP))))
                 (CONS (CONS 'LEAVE (ADD1 (LABEL-CNT CINFO)))
                       (LABEL-ALIST CINFO))
                 (ADD1 (ADD1 (LABEL-CNT CINFO))))
     COND-LIST
     (LOOP-BODY STMT)
     PROC-LIST)
    (LIST (LIST 'JUMP (LABEL-CNT CINFO))
          (LIST 'DL (ADD1 (LABEL-CNT CINFO)) NIL '(PUSH-CONSTANT (NAT 2)))
          '(POP-GLOBAL C-C)))))
 (IF-MG
  (ADD-CODE
   (TRANSLATE
    (ADD-CODE
     (TRANSLATE
      (MAKE-CINFO
       (APPEND (CODE CINFO)
               (LIST (LIST 'PUSH-LOCAL (IF-CONDITION STMT))
                     '(FETCH-TEMP-STK)
                     (LIST 'TEST-BOOL-AND-JUMP 'FALSE (LABEL-CNT CINFO))))
       (LABEL-ALIST CINFO)
       (ADD1 (ADD1 (LABEL-CNT CINFO))))
      COND-LIST
      (IF-TRUE-BRANCH STMT)
      PROC-LIST)
     (LIST (LIST 'JUMP (ADD1 (LABEL-CNT CINFO)))
           (LIST 'DL (LABEL-CNT CINFO) NIL '(NO-OP))))
    COND-LIST
    (IF-FALSE-BRANCH STMT)
    PROC-LIST)
   (LIST (LIST 'DL (ADD1 (LABEL-CNT CINFO)) NIL '(NO-OP)))))
```

Figure 5-1: Translate

```
(BEGIN-MG
 (ADD-CODE
  (TRANSLATE
   (ADD-CODE
    (SET-LABEL-ALIST
     (TRANSLATE
      (MAKE-CINFO (CODE CINFO)
                  (APPEND (MAKE-LABEL-ALIST (WHEN-LABELS STMT)
                                            (LABEL-CNT CINFO))
                          (LABEL-ALIST CINFO))
                  (ADD1 (ADD1 (LABEL-CNT CINFO))))
      COND-LIST
      (BEGIN-BODY STMT)
      PROC-LIST)
     (LABEL-ALIST CINFO))
    (LIST (LIST 'JUMP (ADD1 (LABEL-CNT CINFO)))
          (LIST 'DL (LABEL-CNT CINFO) NIL '(PUSH-CONSTANT (NAT 2)))
          '(POP-GLOBAL C-C)))
   COND-LIST
   (WHEN-HANDLER STMT)
   PROC-LIST)
  (LIST (LIST 'DL (ADD1 (LABEL-CNT CINFO)) NIL '(NO-OP)))))

(PROC-CALL-MG
 (MAKE-CINFO
  (APPEND
     (CODE CINFO)
     (PROC-CALL-CODE
        CINFO STMT COND-LIST
        (DEF-LOCALS (FETCH-CALLED-DEF STMT PROC-LIST))
        (LENGTH (DEF-COND-LOCALS (FETCH-CALLED-DEF STMT PROC-LIST)))))
  (LABEL-ALIST CINFO)
  (PLUS (LABEL-CNT CINFO)
        (ADD1 (ADD1 (LENGTH (CALL-CONDS STMT)))))))

(PREDEFINED-PROC-CALL-MG
 (ADD-CODE CINFO (PREDEFINED-PROC-CALL-SEQUENCE
                             STMT (LABEL-ALIST CINFO))))
(OTHERWISE CINFO))
```

Figure 5-1, `concluded`

5.3.2 NO-OP-MG

No code is generated for the Micro-Gypsy **NO-OP-MG** statement at the Piton level. The **CINFO** returned is simply the **CINFO** parameter to **TRANSLATE**.

5.3.3 SIGNAL-MG

The effect of a **(SIGNAL-MG SIGNALLED-CONDITION)** statement is to set the current condition to **SIGNALLED-CONDITION**. In Piton this is implemented by setting the global variable **CC** to the Piton natural number representation of **SIGNALLED-CONDITION**. Signaling also causes flow of control to branch to a handler for the condition or to the end of the enclosing routine. This is implemented by maintaining on the **LABEL-ALIST** an association between each condition and the appropriate target of the branch for that condition. The code which is generated is

```
(PUSH-CONSTANT n)
(POP-GLOBAL CC)
(JUMP label)
```

where **N** is the Piton natural number representation of **SIGNALLED-CONDITION** and **LABEL** is the associated label from the **LABEL-ALIST**.

To understand the formal mechanism, consider the clause in **TRANSLATE** for **SIGNAL-MG** which describes the **CINFO** generated for the translation of a **SIGNAL-MG** statement.

```
(MAKE-CINFO
   (APPEND (CODE CINFO)
           (LIST (LIST 'PUSH-CONSTANT
                       (MG-COND-TO-P-NAT (SIGNALLED-CONDITION STMT)
                                         COND-LIST))
                 (LIST 'POP-GLOBAL 'CC)
                 (LIST 'JUMP (FETCH-LABEL (SIGNALLED-CONDITION STMT)
                                          (LABEL-ALIST CINFO)))))
   (LABEL-ALIST CINFO)
   (LABEL-CNT CINFO))
```

TRANSLATE is passed an initial **CINFO** value and returns a **CINFO** with the same **LABEL-ALIST** and **LABEL-CNT** fields but with three Piton statements adjoined to the end of the **CODE** field. These are the statements listed above; the translation of the signalled condition is computed with respect to the current **COND-LIST** and the associated label for that condition is retrieved from the current **LABEL-ALIST**.

5.3.4 PROG2-MG

The Micro-Gypsy statement

```
(PROG2-MG prog2-left-branch prog2-right-branch)
```

results in the Piton code:

```
<code for prog2-left-branch>
<code for prog2-right-branch>
```

The code generated is simply the concatenation of the code for the two branches. We achieve this by translating **PROG2-RIGHT-BRANCH** with the **CINFO** resulting from translating **PROG2-LEFT-BRANCH**.

As an aside, suppose that some condition is raised during the execution of **PROG2-LEFT-BRANCH**. An invariant maintained during the translation is that every condition which might be signalled has an associated label on the **LABEL-ALIST** and that label designates a legal "target" for a branch. A signal within **PROG2-LEFT-BRANCH** must either be handled within **PROG2-LEFT-BRANCH** or must branch outside of the **PROG2** statement entirely. There is no case in which a branch is made into a statement block from outside. An implication of this is that any block is entered at the beginning or not at all. Thus, if execution of **PROG2-LEFT-BRANCH** does not return a **'NORMAL** current condition **PROG2-RIGHT-BRANCH** is not executed. Any signal within **PROG2-LEFT-BRANCH** is either handled or causes a branch outside the **PROG2** statement. This, of course, is exactly what the semantics requires.

5.3.5 LOOP-MG

The Micro-Gypsy statement **(LOOP-MG LOOP-BODY)** executes **LOOP-BODY** repetitively until some condition causes termination of the loop. For the implementation we generate code of the form

```
(DL n NIL (NO-OP))                          ; 1
<code for loop body>                        ; 2
(JUMP n)                                    ; 3
(DL n+1 NIL (PUSH-CONSTANT (NAT 2)))        ; 4
(POP-GLOBAL CC)                             ; 5
```

where **N** is the current value of the **LABEL-CNT**. The code for the loop body (line 2) is generated by a recursive call to **TRANSLATE**. Unless execution of that code raises a condition, we reach the **(JUMP N)** statement at line 3 which takes us back to the beginning of the loop.

Recall that within a loop body, Signaling the condition **'LEAVE** causes a "clean" termination of the loop. This is handled in the translator by adding to the front of the **LABEL-ALIST** for the recursive call the pair **<LEAVE, N+1>**. A **(SIGNAL-MG LEAVE)** statement in the body will result in code to set **CC** to **(NAT 0)** and a jump to the label **N+1** at line 4, exiting the loop. Since we don't want the current condition to remain **'LEAVE** outside the loop, we reset **CC** to **'NORMAL** (lines 4 and 5).

5.3.6 IF-MG

The code generated for the statement **(IF-MG B TRUE-BRANCH FALSE-BRANCH)** is

```
(PUSH-LOCAL b)                              ; 1
(FETCH-TEMP-STK)                            ; 2
(TEST-BOOL-AND-JUMP FALSE n)                ; 3
<code for true-branch>                      ; 4
(JUMP n+1)                                  ; 5
(DL n NIL (NO-OP))                          ; 6
<code for false-branch>                     ; 7
(DL n+1 NIL (NO-OP))                        ; 8
```

The code is quite straightforward except for the lines 1 to 3. Recall from Section 3.4.2-E that the condition of an **IF-MG** statement must be a Boolean variable. Our convention is that the values of variables are stored on the temp-stk with pointers in the pointer-alist. The **PUSH-LOCAL** statement at line 1 fetches the pointer which is the local value of **B** and pushes it onto the temp-stk. Using that pointer, we fetch the Boolean value of **B** from the temp-stk. The **TEST-BOOL-AND-JUMP** instruction at line 3 pops the temp-stk and jumps if the result is **(BOOL F)**.

5.3.7 BEGIN-MG

The Micro-Gypsy **BEGIN-MG** statement takes the form

```
(BEGIN-MG begin-body when-labels when-handler)
```

and allows us to associate a condition handler with a block of code. The code generated is

```
<code for begin-body>                       ; 1
(JUMP n+1)                                  ; 2
(DL n NIL (PUSH-CONSTANT (NAT 2)))          ; 3
(POP-GLOBAL CC)                             ; 4
<code for when-handler>                     ; 5
(DL n+1 NIL (NO-OP))                        ; 6
```

The code for the **BEGIN-BODY**, represented by line 1, must be translated in such a way that raising any of the conditions in **WHEN-LABELS** will cause a branch to the when handler. This is accomplished by translating the **BEGIN-BODY** in a context in which pairs of the form **<WHEN-CONDITION, N>** have been added to the **LABEL-ALIST**, for each **WHEN-CONDITION** in the **WHEN-LABELS**. A signal to any of these conditions will result in code to branch to label **N** at line 3. Here, the current condition is reset to **'NORMAL** (lines 3-4) and the **WHEN-HANDLER** is executed. If the **BEGIN-BODY** executes without raising a condition, the jump at line 2 causes a branch over the **WHEN-HANDLER** code.

What happens if some condition is signalled which is not a member of the **WHEN-LABELS**? Recall that every condition which can be signalled has an associated label, which in this case must be a label outside of the **BEGIN-MG** statement. Consequently, flow of control branches out of the **BEGIN-MG** statement code before the end of the **BEGIN-BODY** is reached.

5.3.8 PROC-CALL-MG

Recall that the form of a Micro-Gypsy user-defined procedure definition is

```
(proc-name formal-data-params
           formal-condition-params
           local-variables
           local-conditions
           body)
```

We describe the translation of procedure definitions in detail in Section 5.4 below. For now, it is enough to say that we generate a single Piton procedure of the same name for each Micro-Gypsy procedure. The formals of the Piton procedure will correspond to the formal parameters *and* local variables of the Micro-Gypsy procedure. The Piton procedure has the form

```
(proc-name (local_1 ... local_n
            formal_1 ... formal_m)
           nil
           <code for procedure body>)
```

Suppose, for example, that we have a Micro-Gypsy procedure **SORT**

```
(SORT ((L INT-MG) (H INT-MG) (ARR (ARRAY-MG INT-MG 5)))
      (ERR1 ERR2)
      ((CH CHARACTER-MG (CHARACTER-MG 27))
       (B1 BOOL-MG (BOOL-MG F))
       (A2 (ARRAY-MG INT-MG 3) ((INT-MG -12)
                                (INT-MG 0)
                                (INT-MG 3926)))
       (B2 BOOL-MG (BOOL-MG T)))
      (L-ERR1 L-ERR2 L-ERR3)
      <body>).
```

The corresponding Piton procedure will be

```
(SORT (CH B1 A2 B2 L H ARR)
      NIL
      <translation of body>)
```

Notice that the Piton procedure has formal parameters corresponding to both the formals and locals of the Micro-Gypsy procedure and no locals of its own. Finally, for the discussion below suppose that we are translating the following call to the Micro-Gypsy procedure **SORT**

```
(PROC-CALL-MG SORT (U V ARR) (SORT-ERROR ROUTINEERROR)).
```

5.3.8-A Passing Data

All parameters are passed by *reference*, that is, as pointers to the actual data stored on the temp-stk. This means that we must pass to the Piton procedure $n + m$ data items, where m is the length of **FORMAL-DATA-PARAMS** (and consequently the length of the actuals list) and n is the length of **LOCAL-VARIABLES** in the procedure definition. As described in Chapter 4, we pass parameters in Piton by leaving them on the temp-stk. We pass all parameters, including the locals, by *reference*. For this, we must arrange for $n + m$ pointers to be on top of the temp-stk when we make the call. It had better be the case that these are pointers to the corresponding data in the temp-stk. For the actual parameters, this is no problem since the actuals are guaranteed to be defined identifiers in the calling environment. For the locals, however, the data cannot be expected to already be on the temp-stk. We now discuss the treatment of locals and then return to the actuals.

Because we allow recursive procedures, storage for locals must be allocated at the call site. Since locals are treated as parameters and passed by reference we must allocate storage for the locals, initialize them, and push pointers to their newly allocated locations. As with all other data, we want the values of the locals to be stored within the temp-stk. Allocating space and initializing them merely means pushing their values onto the stack. The Micro-Gypsy initial values of the locals are part of the Micro-Gypsy procedure definition's **LOCAL-VARIABLES** list. We fetch these values, convert them to Piton values, and push them onto the temp-stk. The code which accomplishes this is a series of Piton **PUSH-CONSTANT** instructions generated by the function **PUSH-LOCALS-VALUES-CODE** (see page 121).

Suppose also that before our call to the **SORT** routine, the topmost value on the temp-stk was stored at location 10. After pushing the values of the locals, the temp-stk will appear as follows:

index	contents	represents
17		unused above here
16	(bool f)	B2
15	(int 3926)	A2[2]
14	(int 0)	A2[1]
13	(int -12)	A2[0]
12	(bool f)	B1
11	(int 27)	CH
10	...	other data below here
9	...	

After storing the values of the locals we arrange to push pointers (i.e., indices into the temp-stk) to these local values we have just pushed. This requires some care since not all of the locals take the same amount of space. Ideal for this purpose is the Piton instruction **(PUSH-TEMP-STK-INDEX N)** which has the effect of pushing onto the temp-stk a pointer to a position $n+1$ slots down from the current top in the temp-stk. Adjusting **N** appropriately allows us to push pointers to the values of the locals. The function **PUSH-LOCALS-ADDRESSES-CODE** (page 120) lays down the necessary instructions. For our example, the code generated is

```
(PUSH-TEMP-STK-INDEX 5)          ; push pointer to CH
(PUSH-TEMP-STK-INDEX 5)          ; push pointer to B1
(PUSH-TEMP-STK-INDEX 5)          ; push pointer to A2
(PUSH-TEMP-STK-INDEX 3)          ; push pointer to B2
```

Notice that in the fourth line, the value of **N** is bumped to account for the extra elements of the array **A2**. I.e., the pointer for **B1** in the temp-stk is six locations above the value of **B1**, but the pointer for **B2** is only four locations above it's value.

After pushing the pointers for the locals in our example, the temp-stk looks like

index	contents	represents
21		unused above here
20	(nat 16)	pointer to B2
19	(nat 13)	pointer to A2
18	(nat 12)	pointer to B1
17	(nat 11)	pointer to CH
16	(bool f)	B2
15	(int 3926)	A2[2]
14	(int 0)	A2[1]
13	(int -12)	A2[0]
12	(bool f)	B1
11	(int 27)	CH
10	...	
9	...	

For each formal we must push a pointer to the corresponding actual parameter. The actual is necessarily a variable name, say **X**, defined in the current context. The value of **X** is somewhere in the temp-stk, and a pointer to that value is stored in the bindings component of the top frame on the ctrl-stk. Consequently, the statement **(PUSH-LOCAL X)** suffices to push the appropriate pointer onto the temp-stk. Notice that this is adequate for actuals of both simple and structured types. The function **PUSH-ACTUALS-CODE** (see page 120) accomplishes this. For our **SORT** example, the code generated is

```
(PUSH-LOCAL U)
(PUSH-LOCAL V)
(PUSH-LOCAL ARR)
```

Suppose that **U**, **V**, and **ARR** have local values of **(NAT 3)**, **(NAT 1)**, and **(NAT 5)**, respectively.[33] Following the execution of the code to push the actuals we have the following temp-stk,

index	contents	represents
24		unused above here
23	(nat 5)	pointer to ARR
22	(nat 1)	pointer to V
21	(nat 3)	pointer to U
20	(nat 16)	pointer to B2
19	(nat 13)	pointer to A2
18	(nat 12)	pointer to B1
17	(nat 11)	pointer to CH
16	(bool f)	B2
15	(int 3926)	A2[2]
14	(int 0)	A2[1]
13	(int -12)	A2[0]
12	(bool f)	B1
11	(int 27)	CH
10	...	
9	...	

5.3.8-B The Call

Once the local values, local pointers, and actual pointers are pushed, we make the procedure call **(CALL SORT)**. The Piton procedure expects 7 values as parameters corresponding to the 3 formals and 4 locals of the Micro-Gypsy procedure. As explained in Chapter 4, during the call these are popped off and a new frame is created on the ctrl-stk associating each of the formal and local names with their pointers. The effect of this is to create an appropriate environment for the call-body in which each of the Piton formals names is associated with a pointer to in the temp-stk. For our example, the bindings component of

[33] As a reminder, this means that **U**'s value is stored beginning at location 3 on the temp-stk and the pair **<U, (NAT 3)>** can be found in the bindings of the topmost frame on the ctrl-stk.

the new frame is

```
((CH  . (NAT 11))
 (B1  . (NAT 12))
 (A2  . (NAT 13))
 (B2  . (NAT 16))
 (L   . (NAT 3))
 (H   . (NAT 1))
 (ARR . (NAT 5))).
```

Also stored as part of the frame is the return pc, which allows execution to resume at the statement following the `CALL`. Notice that the *values* of the locals remain on the stack as they must. It is the responsibility of the called routine to remove them before returning. This is discussed in Section 5.4.

5.3.8-C Translating the Condition

The only non-`'NORMAL` conditions which can be returned by a procedure are `'ROUTINEERROR` or a member of the formal or local condition lists. The condition returned is translated to a condition in the calling environment according to the following rule. If the `CC` of the body result is one of the formal condition parameters, return the corresponding actual condition parameter. Otherwise, return `'ROUTINEERROR`. The code to accomplish this mapping process in Piton is unfortunately quite complicated. We use the `SORT` example again to illustrate.

We have the following condition lists:

`(ERR1 ERR2)`	formal conditions
`(L-ERR1 L-ERR2 L-ERR3)`	local conditions
`(SORT-ERROR ROUTINEERROR)`	actual conditions

Suppose that the `CINFO` in which we are translating has `LABEL-CNT` equal to 212 and a `LABEL-ALIST` containing the pairs **<SORT-ERROR, 25>** and **<ROUTINEERROR, 102>**. Finally suppose that the `COND-LIST` in the calling environment is `(COND1 COND2 SORT-ERROR COND3)`. The code generated for mapping the current condition for this CALL statement is

```
(PUSH-GLOBAL CC)                              ; 1
(JUMP-CASE 212 212 213 214 215 212 212 212)   ; 2
(DL 212 NIL (PUSH-CONSTANT (NAT 1)))          ; 3
(POP-GLOBAL CC)                               ; 4
(JUMP 102)                                    ; 5
(DL 214 NIL (PUSH-CONSTANT (NAT 5)))          ; 6
(POP-GLOBAL CC)                               ; 7
(JUMP 25)                                     ; 8
(DL 215 NIL (PUSH-CONSTANT (NAT 1)))          ; 9
(POP-GLOBAL)                                  ; 10
(JUMP 102)                                    ; 11
(DL 213 NIL (NO-OP))                          ; 12
```

We first push the value of the current condition onto the temp-stk (at line 1). Recall that this could be the Piton representation of any of the Micro-Gypsy conditions `'NORMAL`, `'ROUTINEERROR`, a member of the formal conditions, or a member of the local conditions. The Piton representation will be a natural number which is `(NAT 2)` for `'NORMAL`, `(NAT 1)` for `'ROUTINEERROR`, and an index into the procedure body `COND-LIST` for the others.[34] This is the Piton representation of conditions explained in Section 5.2 above. We use this fact to branch to an appropriate bit of code for doing the condition mapping.

The Piton instruction $(\texttt{JUMP-CASE LAB}_0 \ \texttt{...} \ \texttt{LAB}_N)$ pops the stack to obtain a natural number `I` between 0 and `N`. A jump is then made to label $\texttt{LAB}_I$. We construct a `JUMP-CASE` instruction at line 2 which will

---

[34] The procedure body `COND-LIST` is `(ERR1 ERR2 L-ERR1 L-ERR2 L-ERR3)` created by appending the formal condition and local condition lists.

cause us to branch appropriately depending on the **CC** value. The value of **CC** for our example could be certain values between 0 and 7 which correspond to a condition returned from the procedure body as follows.

condition	CC value	Maps To	with CC value
LEAVE	(NAT 0)	ROUTINEERROR	(NAT 1)
ROUTINEERROR	(NAT 1)	ROUTINEERROR	(NAT 1)
NORMAL	(NAT 2)	NORMAL	(NAT 2)
ERR1	(NAT 3)	SORT-ERROR	(NAT 5)
ERR2	(NAT 4)	ROUTINEERROR	(NAT 1)
L-ERR1	(NAT 5)	ROUTINEERROR	(NAT 1)
L-ERR2	(NAT 6)	ROUTINEERROR	(NAT 1)
L-ERR3	(NAT 7)	ROUTINEERROR	(NAT 1)

(**'LEAVE** can't actually be returned but we must have some value in the 0^{th} position in the **JUMP-CASE** instruction.) The third and fourth columns show the condition we want to return in the calling environment with its Piton representation.

The **JUMP-CASE** instruction has 8 labels, one for each of the possible values of the current-condition. If the condition value is **(NAT 2)** for **'NORMAL** a jump is made to the final **NO-OP** instruction at line 12 and execution continues following the procedure call. Any other condition causes a jump to a 3-statement block of code which

1. pushes the desired new value of the **CC** onto the temp-stk;
2. pops the value into **CC**;
3. jumps to the label associated with the new condition on the **LABEL-ALIST**.

There is one such block for each of the formal conditions and one for **'ROUTINEERROR**.

For instance, if the formal condition **ERR1** is returned by the procedure body, this corresponds in the calling environment to the actual **'SORT-ERROR**. The Piton **CC** value will be **(NAT 3)**, *i.e.*, two plus the index of **ERR1** in the **COND-LIST**. The **JUMP-CASE** statement for a value of **(NAT 3)** causes a jump to label 214, the label of the code for mapping this condition (line 6). The Piton representation of the condition **SORT-ERROR** is **(NAT 5)**--two plus the index of **SORT-ERROR** in the **COND-LIST** of the calling environment. In lines 6 and 7, we set the current condition to **SORT-ERROR**. Finally, we jump to the label associated with **SORT-ERROR** on the **LABEL-ALIST** for the calling environment.

The reader is advised to study the function **CONDITION-MAP-CODE** (page 111) for a more thorough understanding of the algorithm for mapping conditions at the routine boundary.

5.3.8-D The Procedure Call Code

For completeness we list the entire code list generated for the Micro-Gypsy statement

```
(PROC-CALL-MG SORT (U V ARR) (SORT-ERROR ROUTINEERROR))
```

in our example.

```
(PUSH-CONSTANT (INT 27))                      ; push the values of the four
(PUSH-CONSTANT (BOOL F))                      ; locals of the routine onto
(PUSH-CONSTANT (INT -12))                     ; the temp-stk
(PUSH-CONSTANT (INT 0))
(PUSH-CONSTANT (INT 3926))
(PUSH-CONSTANT (BOOL F))
(PUSH-TEMP-STK-INDEX 5)                       ; push pointers to the values
(PUSH-TEMP-STK-INDEX 5)                       ; of the four locals stored on
(PUSH-TEMP-STK-INDEX 5)                       ; the temp-stk above
(PUSH-TEMP-STK-INDEX 3)
(PUSH-LOCAL U)                                ; push the pointers of the
(PUSH-LOCAL V)                                ; three actual parameters
(PUSH-LOCAL ARR)
(CALL SORT)                                   ; jump to the subroutine,
                                              ; execution pops pointers and
                                              ; local data from stack and may
                                              ; side-effect U, V, and ARR

(PUSH-GLOBAL CC)                              ; push returned condition
(JUMP-CASE 212 212 213 214 215 212 212 212)
                                              ; jump to condition mapping code
(DL 212 NIL (PUSH-CONSTANT (NAT 1)))          ; mapping for ROUTINEERROR
(POP-GLOBAL CC)
(JUMP 102)
(DL 214 NIL (PUSH-CONSTANT (NAT 5)))          ; mapping ERR1 → SORT-ERROR
(POP-GLOBAL CC)
(JUMP 25)
(DL 215 NIL (PUSH-CONSTANT (NAT 1)))          ; mapping ERR2 → ROUTINEERROR
(POP-GLOBAL)
(JUMP 102)
(DL 213 NIL (NO-OP))                          ; condition was NORMAL
```

5.3.9 Predefined Procedure Calls

The translation of calls to predefined procedures is significantly easier than that of user-defined procedures. We have "hand-coded" the call sequence for each of the predefineds to allow for the variability in the instruction lists. The call sequence for redefined procedure *proc-name* is defined by the function *proc-name*-**CALL-SEQUENCE**. For instance, for the procedure **MG-SIMPLE-VARIABLE-ASSIGNMENT** we define the function

Definition.

```
(MG-SIMPLE-VARIABLE-ASSIGNMENT-CALL-SEQUENCE stmt)
  =
(LIST (LIST 'PUSH-LOCAL (CAR (CALL-ACTUALS STMT)))
      (LIST 'PUSH-LOCAL (CADR (CALL-ACTUALS STMT)))
      '(CALL MG-SIMPLE-VARIABLE-ASSIGNMENT))
```

Given the call statement

```
(PREDEFINED-PROC-CALL-MG MG-SIMPLE-VARIABLE-ASSIGNMENT (X Y))
```

we generate the code

```
(PUSH-LOCAL X)
(PUSH-LOCAL Y)
(CALL MG-SIMPLE-VARIABLE-ASSIGNMENT)
```

Each of the predefineds expects its arguments on the top of the temp-stk. As with user-defined routines, the parameters we are pushing are really pointers to positions in the temp-stk. The body of the predefined routine may fetch or store values from the temp-stk via these pointers. This is illustrated in Section 5.5 below.

The simple call sequence above is adequate for assignment since it always returns normally. Some of the predefined routines can return with **'ROUTINEERROR**. For these we must cause a branch. The code generated for a call statement

```
(PREDEFINED-PROC-CALL-MG MG-INTEGER-ADD (X Y Z))
```

for example, is

```
(PUSH-LOCAL X)                                          ; 1
(PUSH-LOCAL Y)                                          ; 2
(PUSH-LOCAL Z)                                          ; 3
(CALL MG-INTEGER-ADD)                                   ; 4
(PUSH-GLOBAL CC)                                        ; 5
(SUB1 NAT)                                              ; 6
(TEST-NAT-AND-JUMP ZERO lab)                            ; 7
```

where **lab** is the label associated with **'ROUTINEERROR** on the **LABEL-ALIST**. At line 5 we push the value of the condition returned onto the temp-stk. We know from the semantics of the routine that this value must be either **(NAT 1)** for **'ROUTINEERROR** or **(NAT 2)** for **'NORMAL**. In lines 6-7 we decrement the value and branch if the result is 0 to the appropriate location.

5.4 Translating User-Defined Procedures

We illustrated briefly in Section 5.3.8 part of the translation of procedures on the Micro-Gypsy **PROC-LIST** into Piton procedures. We now explain the mechanism for this translation in more detail. We have already mentioned the following.

1. Each Micro-Gypsy procedure is translated to a single Piton procedure with the same name.
2. The Piton procedure has formal parameters corresponding to the formal parameters and local variables of the Micro-Gypsy procedure.
3. The Piton procedure expects on the temp-stk pointers into the temp-stk for each of parameters. Each of these pointers is really the index for the temp-stk location where the corresponding data begins. In the case of the Micro-Gypsy local data, we placed the data into the temp-stk prior to pushing the parameters, as explained above.
4. The Piton procedure has no locals.

Thus, corresponding to the Micro-Gypsy procedure

```
(proc-name ((formal_1 f-type_1) ... (formal_n f-type_n))
           (f-cond_1 ... f-cond_k)
           ((local_1 l-type_1 l-value_1) ... (local_m l-type_m l-value_m))
           (l-cond_1 ... l-cond_j)
           Micro-Gypsy-body)
```

we have the Piton procedure

```
(proc-name (local_1 ... local_m formal_1 ... formal_n)
           NIL
           Piton-body).
```

We need only say how **Piton-body** is derived from **Micro-Gypsy-body**. **Micro-Gypsy-body** is some Micro-Gypsy statement legal with respect to the **NAME-ALIST** created from the formal parameter and local variable lists and **COND-LIST** created from the formal and local condition lists.

Micro-Gypsy-body is translated in a **CINFO** with the following fields:

- **CODE**: is **NIL**;
- **LABEL-CNT**: is 1;[35]
- **LABEL-ALIST**: has each possible condition (**'ROUTINEERROR**, all formal conditions, all local-conditions) associated with 0.

---

[35]Labels in Piton need only be unique within procedures. Hence, we can reset the **LABEL-CNT** for each new procedure.

The other parameters to the translation are the **PROC-LIST** and the **COND-LIST** generated by appending the routine's formal and local condition lists together.

We append to the translated procedure body the following three statements:

```
(DL 0 NIL (NO-OP))                          ; destination for
                                            ;       unhandled conditions
(POP* n)                                    ; remove local data from temp-stk
(RET)                                       ; return from proc body
```

where **N** is the size of the local data we pushed onto the stack in preparing for the call as described in Section 5.3.8-A.

Our construction of the **LABEL-ALIST** assures that all conditions not handled within the procedure body cause a branch to the end of the routine. The **POP*** instruction pops **N** locations off the stack. This removes from the stack the storage of the local data we used in preparing for the call.[36]

To illustrate the translation of Micro-Gypsy procedures into Piton, consider the simple Micro-Gypsy procedure in figure 5-2. This procedure multiplies a Micro-Gypsy integer by a non-negative integer using repeated addition.[37] In our abstract prefix syntax, this procedure has the form given in figure 5-3. Finally, the translation of this routine into Piton is shown in figure 5-4.

```
procedure MULTIPLY_BY_POSITIVE (var ANS: int;
                                I, J: int) =
begin
   var K: int := 0;
   K := J;
   ANS := 0;
   loop
      if K le 0 then leave end;
      ANS := ANS + I;
      K := K - 1;
   end;
end; {mult}
```

Figure 5-2: A Simple Micro-Gypsy Procedure

5.5 Translating Predefined Procedures

It is in the predefined procedures where much of the work of Micro-Gypsy is accomplished. The predefined procedures are individually coded. The code for predefined procedure *proc-name* is defined by the function *proc-name***-TRANSLATION**. We examine the translations of two predefineds to illustrate the coding techniques.

The predefined procedure **MG-SIMPLE-VARIABLE-ASSIGNMENT** has the following body

---

[36] This could just as easily have been accomplished by the caller.

[37] We return to this procedure in Chapter 8.

```
(MG_MULTIPLY_BY_POSITIVE
  ((ANS INT-MG)
   (I   INT-MG)
   (J   INT-MG))
  NIL
  ((K    INT-MG     (INT-MG 0))
   (ZERO INT-MG     (INT-MG 0))
   (ONE  INT-MG     (INT-MG 1))
   (B    BOOLEAN-MG (BOOLEAN-MG FALSE-MG)))
  NIL
  (PROG2-MG
   (PREDEFINED-PROC-CALL-MG MG-SIMPLE-CONSTANT-ASSIGNMENT (ANS (INT-MG 0)))
   (PROG2-MG
    (PREDEFINED-PROC-CALL-MG MG-SIMPLE-VARIABLE-ASSIGNMENT (K J))
    (LOOP-MG
     (PROG2-MG
      (PREDEFINED-PROC-CALL-MG MG-INTEGER-LE (B K ZERO))
      (PROG2-MG
       (IF-MG B (SIGNAL-MG LEAVE) (NO-OP-MG))
       (PROG2-MG
        (PREDEFINED-PROC-CALL-MG MG-INTEGER-ADD (ANS ANS I))
        (PREDEFINED-PROC-CALL-MG MG-INTEGER-SUBTRACT (K K ONE)))))))))
```

Figure 5-3: The Procedure in Abstract Prefix Form

```
(MG-SIMPLE-VARIABLE-ASSIGNMENT
   (DEST SOURCE)                                   ; formals
   NIL                                             ; locals
   (PUSH-LOCAL SOURCE)                             ; 1
   (FETCH-TEMP-STK)                                ; 2
   (PUSH-LOCAL DEST)                               ; 3
   (DEPOSIT-TEMP-STK)                              ; 4
   (RET))                                          ; 5
```

It has two formals and no local data. **MG-SIMPLE-VARIABLE-ASSIGNMENT**, like each of the other predefined procedures expects its arguments to be passed by reference on top of the temp-stk. The call statement will push a frame onto the control stack with bindings associating the names **DEST** and **SOURCE** with pointers to the locations of the actuals' values in the temp-stk. The pointer for **SOURCE** is used to fetch the data (lines 1-2), which is left on top of the temp-stk. The destination pointer is fetched (line 3) and used to store the data (line 4). Finally, we return using the return pc stored in the topmost frame which we pop off of the ctrl-stk. This predefined procedure is particularly simple since no conditions can be raised.

A more complicated example is the **MG-INTEGER-ADD** routine. The code is

```
(MG-INTEGER-ADD
   (ANS Y Z)                                  ; formals
   ((T1 (INT 0)))                             ; local (initialized to 0)
   (PUSH-CONSTANT (BOOL F))                   ; 1
   (PUSH-LOCAL Y)                             ; 2
   (FETCH-TEMP-STK)                           ; 3
   (PUSH-LOCAL Z)                             ; 4
   (FETCH-TEMP-STK)                           ; 5
   (ADD-INT-WITH-CARRY)                       ; 6
   (POP-LOCAL T1)                             ; 7
   (TEST-BOOL-AND-JUMP T 0)                   ; 8
   (PUSH-LOCAL T1)                            ; 9
   (PUSH-LOCAL ANS)                           ; 10
   (DEPOSIT-TEMP-STK)                         ; 11
   (JUMP 1)                                   ; 12
   (DL 0 NIL (PUSH-CONSTANT (NAT 1)))         ; 13
   (POP-GLOBAL CC)                            ; 14
   (DL 1 NIL (RET)))                          ; 15
```

We have three formal parameters and one local variable used as a temporary. Fetch the addends (lines

```
(MG_MULTIPLY_BY_POSITIVE
     (K ZERO ONE B ANS I J)                              ; formals
     NIL                                                 ; locals
     (PUSH-LOCAL ANS)                                    ; ans := 0;
     (PUSH-CONSTANT (INT 0))
     (CALL MG-SIMPLE-CONSTANT-ASSIGNMENT)
     (PUSH-LOCAL K)                                      ; k := j;
     (PUSH-LOCAL J)
     (CALL MG-SIMPLE-VARIABLE-ASSIGNMENT)
     (DL 1 NIL (NO-OP))                                  ; loop
     (PUSH-LOCAL B)                                      ;   b := k le 0
     (PUSH-LOCAL K)
     (PUSH-LOCAL ZERO)
     (CALL MG-INTEGER-LE)
     (PUSH-LOCAL B)                                      ;   if b then leave
     (FETCH-TEMP-STK)
     (TEST-BOOL-AND-JUMP FALSE 3)
     (PUSH-CONSTANT (NAT 0))
     (POP-GLOBAL C-C)
     (JUMP 2)
     (JUMP 4)
     (DL 3 NIL (NO-OP))
     (DL 4 NIL (NO-OP))
     (PUSH-LOCAL ANS)                                    ;     ans := ans + i;
     (PUSH-LOCAL ANS)
     (PUSH-LOCAL I)
     (CALL MG-INTEGER-ADD)
     (PUSH-GLOBAL C-C)
     (SUB1-NAT)
     (TEST-NAT-AND-JUMP ZERO 0)
     (PUSH-LOCAL K)                                      ;     k := k - 1;
     (PUSH-LOCAL K)
     (PUSH-LOCAL ONE)
     (CALL MG-INTEGER-SUBTRACT)
     (PUSH-GLOBAL C-C)
     (SUB1-NAT)
     (TEST-NAT-AND-JUMP ZERO 0)
     (JUMP 1)
     (DL 2 NIL (PUSH-CONSTANT (NAT 2)))
     (POP-GLOBAL C-C)                                    ; end; {loop}
     (DL 0 NIL (NO-OP))
     (POP* 4)
     (RET)))
```

Figure 5-4: The Translation of the Procedure

2-5) and add them (line 6) along with a carry bit set to 0 (line 1). Pushed onto the temp-stk as a result of the **ADD-INT-WITH-CARRY** instruction at line 6 is a Boolean **B** indicating whether the sum overflows and the (possibly incorrect) sum. Store the sum in the temporary (line 7). If **B** indicates that overflow has occurred, jump to line 13, set the condition to **'ROUTINEERROR**, and return. Otherwise, fetch the sum from the temporary (line 9), get the target address (line 10), and store the result. Then jump to the line 15 and return.

There is little doubt that more elegant codings could be found for some of the predefined operations. The reader is invited to try.

5.6 Creating a Piton State

We have considered the mapping of Micro-Gypsy data into Piton and the translation of Micro-Gypsy code into Piton code. Now we are ready to consider the mapping from entire Micro-Gypsy execution environments to Piton execution environments. The components of the Micro-Gypsy environment include the statement being interpreted, procedure list, current condition, variable alist, psw, and resource limitations. We must show how each of these is represented in a Piton state.

5.6.1 Mapping the Micro-Gypsy Data

Recall our convention that all of the data of our Micro-Gypsy world would be represented in the corresponding Piton world as values on the temp-stk. The data component of the Micro-Gypsy execution environment is exactly the **MG-ALIST** component of the **MG-STATE**. We have already seen in Section 5.3.8-A how we can convert this data to Piton values and store it on the temp-stk. Thus, our initial Piton temp-stk is created simply by pushing the Piton representations of each of the data values on the Micro-Gypsy **MG-ALIST**. This is done with the function **INITIAL-TEMP-STK** (page 133).

We also saw in Section 5.3.8-A how we might generate a pointer-alist which would allow us to access the values on the temp-stk. We need merely arrange to generate these bindings and place them into the topmost ctrl-stk frame. The function **INITIAL-BINDINGS** (page 133) is used for this.

Suppose, that our Micro-Gypsy execution environment contains the following **MG-ALIST**:

```
((B1 BOOLEAN-MG (BOOLEAN-MG FALSE-MG))
 (CH CHARACTER-MG (CHARACTER-MG 25))
 (A (ARRAY-MG INT-MG 5) ((INT-MG -294)
                         (INT-MG 38)
                         (INT-MG 0)
                         (INT-MG 12)
                         (INT-MG -2983)))
 (B2 BOOLEAN-MG (BOOLEAN-MG TRUE-MG))
 (I INT-MG (INT-MG 4202))).
```

The initial value of the temp-stk in our Piton execution environment would be:

```
index          contents           represents

  9                               unused above here
  8            (INT 4202)             I
  7            (BOOL T)               B2
  6            (INT -2983)            A[4]
  5            (INT 12)               A[3]
  4            (INT 0)                A[2]
  3            (INT 38)               A[1]
  2            (INT -294)             A[0]
  1            (INT 25)               CH
  0            (BOOL f)               B1
```

The following list would be created for storage in the bindings component of the top frame of the temp-stk:

```
((B1 . 0) (CH . 1) (A  . 2) (B2 . 7) (I  . 8))
```

5.6.2 The Micro-Gypsy Statement and Procedure List

The program is really represented in the Micro-Gypsy execution environment by the list of procedures and the statement to be interpreted. The statement is best thought of as an *entry point* into our program. Piton has no analogous concept of an isolated statement to be interpreted. The current point of execution is designated by a value of the **P-PC** pointing somewhere into the Piton program-segment.

Given the Micro-Gypsy statement **STMT**, condition list **COND-LIST**, **MG-ALIST**, and a new identifier **SUBR**[38], we begin by constructing the following Micro-Gypsy procedure.

```
(subr                        ; proc-name
 NIL                         ; formals
 cond-list                   ; formal-conditions
 mg-alist                    ; locals
 NIL                         ; local-conditions
 stmt)                       ; body
```

This is a legal Micro-Gypsy procedure of which the following is true.

1. Its name is distinct from that of any user-defined or predefined procedure name. This implies that the new definition does not supercede any existing user-defined procedure or affect in any way the semantics of procedures on the procedure list.
2. The body of the procedure is simply the statement to be interpreted;
3. The conditions which can be signalled are exactly **'ROUTINEERROR** and members of **COND-LIST**.
4. The local data are exactly the data structures on the **MG-ALIST**.

Using the method described in Section 5.4, we translate this new procedure and each of the members of the Micro-Gypsy user-defined procedure list and create a list of the translations. We append to the front of this list the list of translations of the Micro-Gypsy predefined procedure definitions as described in Section 5.5. The result is a list of Piton procedure definitions containing the translations of all Micro-Gypsy user-defined and predefined procedures, and the one special "entry point" procedure. Our syntactic restrictions guarantee that all of the names are distinct. This list becomes our Piton **P-PROG-SEGMENT**.

The point of control in the Piton state should correspond to the beginning of the **SUBR** special procedure.

---

[38]We could generate such an identifier by taking all of the user-defined procedure names and concatenating them together. Instead we simply supply a name and make as an hypothesis to our theorem that it is distinct.

This is easily accomplished by setting the **`P-PC`** component of the Piton state to **`(SUBR . 0)`**. The first statement of the Piton procedure **`SUBR`** in the **`P-PROG-SEGMENT`** is guaranteed by construction to be precisely the beginning of the Piton code corresponding to **`STMT`** in the translation.

5.6.3 The Complete Piton State

We can now say how to construct the complete Piton state corresponding to a Micro-Gypsy execution environment. We fill in the fields in the Piton state as follows.

- **`P-PC`**: **`(SUBR . 0)`** where **`SUBR`** is the name of our special Micro-Gypsy procedure (and its Piton translation) described in Section 5.6.2.
- **`P-CTRL-STK`**: contains a single frame with bindings as described in Section 5.6.1 and return pc of **`(SUBR . 0)`**.[39]
- **`P-TEMP-STK`**: contains the representations of the Micro-Gypsy **`MG-ALIST`** data as described in Section 5.6.1.
- **`P-PROG-SEGMENT`**: contains the translations of the Micro-Gypsy user-defined, predefined, and special **`SUBR`** procedures as described in Section 5.6.2.
- **`P-DATA-SEGMENT`**: contains a single segment **`CC`** whose value is the Piton representation of the Micro-Gypsy current condition.
- **`P-MAX-CTRL-STK-SIZE`**: the value of **`MG-MAX-CTRL-STK-SIZE`**.
- **`P-MAX-TEMP-STK-SIZE`**: the value of **`MG-MAX-TEMP-STK-SIZE`**.
- **`P-WORD-SIZE`**: the value of **`MG-WORD-SIZE`**.
- **`P-PSW`**: **`'RUN`**.

This Piton state is constructed by the function **`MAP-DOWN1`** (page 113). We have motivated our choices for some of the fields; others require some explaining.

The **`P-CTRL-STK`** and **`P-TEMP-STK`** are initialized to provide the appropriate data environment for the execution of **`STMT`**. The values from **`MG-ALIST`**, which is all of the data available to **`STMT`**, are made available following our convention that data values are stored on the temp-stk and accessible via pointers stored on the ctrl-stk. These initial values of **`P-CTRL-STK`** and **`P-TEMP-STK`** are similar to those which would have been created by the Piton statement **`(CALL SUBR)`**. The return pc value in the ctrl-stk frame is arbitrary; a return from the top-most procedure in Piton ignores the return pc.

The fields **`P-MAX-CTRL-STK-SIZE`**, **`P-MAX-TEMP-STK-SIZE`**, and **`P-WORD-SIZE`** define the resource limitations of the Piton machine. These must obviously be set to the resource limits we assumed for the Micro-Gypsy machine as discussed in Section 3.4.3. The **`P-PSW`** must be set to **`'RUN`** to allow any computation in the Piton state.

5.7 An Alphabetical Listing of the Translator Definition

This section contains the functions in the formal definition of the translator. Several of these function definitions refer to functions defined with respect to Micro-Gypsy in Section 3.5 and Piton in Section 4.2.

---

[39] The return pc is not used but must be a legal Piton pc value.

DEFINITION
```
(ADD-CODE CINFO CODE)
   =
(MAKE-CINFO (APPEND (CODE CINFO) CODE)
            (LABEL-ALIST CINFO)
            (LABEL-CNT CINFO))
```

DEFINITION
```
(COND-CASE-JUMP-LABEL-LIST LC N)
   =
(IF (ZEROP N)
    NIL
    (CONS LC
          (COND-CASE-JUMP-LABEL-LIST (ADD1 LC)
                                     (SUB1 N))))
```

DEFINITION
```
(COND-CONVERSION ACTUAL-CONDS LC COND-LIST LABEL-ALIST)
   =
(IF (NLISTP ACTUAL-CONDS)
    NIL
    (CONS (LIST 'DL
                LC NIL
                (LIST 'PUSH-CONSTANT
                      (MG-COND-TO-P-NAT (CAR ACTUAL-CONDS)
                                        COND-LIST)))
          (CONS '(POP-GLOBAL C-C)
                (CONS (LIST 'JUMP
                            (FETCH-LABEL (CAR ACTUAL-CONDS)
                                         LABEL-ALIST))
                      (COND-CONVERSION (CDR ACTUAL-CONDS)
                                       (ADD1 LC)
                                       COND-LIST LABEL-ALIST)))))
```

DEFINITION
```
(CONDITION-INDEX COND COND-LIST)
   =
(CASE COND
      (LEAVE 0)
      (ROUTINEERROR 1)
      (NORMAL 2)
      (OTHERWISE (ADD1 (ADD1 (INDEX COND COND-LIST)))))
```

DEFINITION
```
(CONDITION-MAP-CODE ACTUAL-CONDS LC COND-LIST LABEL-ALIST PROC-LOCALS-LNGTH)
   =
(APPEND
 (LIST
  '(PUSH-GLOBAL C-C)
  (APPEND
   (CONS 'JUMP-CASE
         (CONS LC
               (CONS LC
                     (COND-CASE-JUMP-LABEL-LIST (ADD1 LC)
                                                (ADD1 (LENGTH ACTUAL-CONDS))))))
   (LABEL-CNT-LIST LC PROC-LOCALS-LNGTH))
  (CONS 'DL
        (CONS LC
              '(NIL (PUSH-CONSTANT (NAT 1)))))
  '(POP-GLOBAL C-C)
  (LIST 'JUMP
        (FETCH-LABEL 'ROUTINEERROR
                     LABEL-ALIST)))
 (APPEND (COND-CONVERSION ACTUAL-CONDS
                          (ADD1 (ADD1 LC))
                          COND-LIST LABEL-ALIST)
         (LIST (CONS 'DL
                     (CONS (ADD1 LC) '(NIL (NO-OP)))))))
```

DEFINITION
```
(DEPOSIT-ALIST-VALUE MG-ALIST-ELEMENT BINDINGS TEMP-STK)
   =
(IF (SIMPLE-MG-TYPE-REFP (M-TYPE MG-ALIST-ELEMENT))
    (DEPOSIT-TEMP (MG-TO-P-SIMPLE-LITERAL (M-VALUE MG-ALIST-ELEMENT))
                  (CDR (ASSOC (CAR MG-ALIST-ELEMENT)
                              BINDINGS))
                  TEMP-STK)
    (DEPOSIT-ARRAY-VALUE (M-VALUE MG-ALIST-ELEMENT)
                         (CDR (ASSOC (CAR MG-ALIST-ELEMENT)
                                     BINDINGS))
                         TEMP-STK))
```

DEFINITION
```
(DEPOSIT-ARRAY-VALUE LIT-LIST NAT TEMP-STK)
   =
(IF (NLISTP LIT-LIST)
    TEMP-STK
    (DEPOSIT-ARRAY-VALUE (CDR LIT-LIST)
                         (ADD1-NAT NAT)
                         (DEPOSIT-TEMP (MG-TO-P-SIMPLE-LITERAL (CAR LIT-LIST))
                                       NAT TEMP-STK)))
```

DEFINITION
```
(DEPOSIT-TEMP VAL NAT TEMP-STK)
   =
(RPUT VAL (UNTAG NAT) TEMP-STK)
```

DEFINITION
```
(DISCARD-LABEL CINFO)
   =
(MAKE-CINFO (CODE CINFO)
            (CDR (LABEL-ALIST CINFO))
            (LABEL-CNT CINFO))
```

DEFINITION
```
(FETCH-LABEL CONDITION LABEL-ALIST)
   =
(CDR (ASSOC CONDITION LABEL-ALIST))
```

DEFINITION
```
(FETCH-N-TEMP-STK-ELEMENTS TEMP-STK NAT N)
   =
(IF (ZEROP N)
    NIL
    (CONS (FETCH-TEMP NAT TEMP-STK)
          (FETCH-N-TEMP-STK-ELEMENTS TEMP-STK
                                     (ADD1-NAT NAT)
                                     (SUB1 N))))
```

DEFINITION
```
(FETCH-TEMP NAT TEMP-STK)
   =
(RGET (UNTAG NAT) TEMP-STK)
```

DEFINITION
```
(LABEL-CNT-LIST LC N)
   =
(IF (ZEROP N)
    NIL
    (CONS LC
          (LABEL-CNT-LIST LC (SUB1 N))))
```

SHELL DEFINITION.
Add the shell **MAKE-CINFO** of **3** arguments, with
recognizer function symboL **CINFOP**, and
accessors **CODE**, **LABEL-ALIST** and **LABEL-CNT**.

DEFINITION
```
(MAKE-LABEL-ALIST NAME-LIST LABEL)
   =
(IF (NLISTP NAME-LIST)
    NIL
    (CONS (CONS (CAR NAME-LIST) LABEL)
          (MAKE-LABEL-ALIST (CDR NAME-LIST)
                            LABEL)))
```

DEFINITION
```
(MAP-DOWN MG-STATE PROC-LIST CTRL-STK TEMP-STK ADDR COND-LIST)
   =
(P-STATE ADDR CTRL-STK
         (MAP-DOWN-VALUES (MG-ALIST MG-STATE)
                          (BINDINGS (TOP CTRL-STK))
                          TEMP-STK)
         (TRANSLATE-PROC-LIST PROC-LIST)
         (LIST (LIST 'C-C
                     (MG-COND-TO-P-NAT (CC MG-STATE)
                                       COND-LIST)))
         (MG-MAX-CTRL-STK-SIZE)
         (MG-MAX-TEMP-STK-SIZE)
         (MG-WORD-SIZE)
         'RUN)
```

Definition.
```
  (MAP-DOWN1 MG-STATE PROC-LIST COND-LIST SUBR STMT)
     =
  (MAP-DOWN MG-STATE
            (CONS (MAKE-MG-PROC (MG-ALIST MG-STATE) SUBR STMT COND-LIST)
                  PROC-LIST)
            (LIST (CONS (INITIAL-BINDINGS (MG-ALIST MG-STATE) 0)
                        (LIST (TAG 'PR (CONS SUBR 0)))))
            (INITIAL-TEMP-STK (MG-ALIST MG-STATE))
            (TAG 'PC (CONS SUBR 0))
            COND-LIST)
```

DEFINITION
```
(MAP-DOWN-VALUES MG-ALIST BINDINGS TEMP-STK)
   =
(IF (NLISTP MG-ALIST)
    TEMP-STK
    (MAP-DOWN-VALUES (CDR MG-ALIST)
                     BINDINGS
                     (DEPOSIT-ALIST-VALUE (CAR MG-ALIST)
                                          BINDINGS TEMP-STK)))
```

DEFINITION
```
(MG-ACTUALS-TO-P-ACTUALS MG-ACTUALS BINDINGS)
   =
(IF (NLISTP MG-ACTUALS)
    NIL
    (CONS (CDR (ASSOC (CAR MG-ACTUALS) BINDINGS))
          (MG-ACTUALS-TO-P-ACTUALS (CDR MG-ACTUALS)
                                   BINDINGS)))
```

DEFINITION
```
(MG-ARRAY-ELEMENT-ASSIGNMENT-CALL-SEQUENCE STMT LABEL-ALIST)
   =
(LIST (LIST 'PUSH-LOCAL
            (CAR (CALL-ACTUALS STMT)))
      (LIST 'PUSH-LOCAL
            (CADR (CALL-ACTUALS STMT)))
      (LIST 'PUSH-LOCAL
            (CADDR (CALL-ACTUALS STMT)))
      (LIST 'PUSH-CONSTANT
            (TAG 'INT
                 (CADDDR (CALL-ACTUALS STMT))))
      '(CALL MG-ARRAY-ELEMENT-ASSIGNMENT)
      '(PUSH-GLOBAL C-C)
```

```
      '(SUB1-NAT)
      (LIST 'TEST-NAT-AND-JUMP
            'ZERO
            (FETCH-LABEL 'ROUTINEERROR
                         LABEL-ALIST)))
```

DEFINITION
```
(MG-ARRAY-ELEMENT-ASSIGNMENT-TRANSLATION)
   =
'(MG-ARRAY-ELEMENT-ASSIGNMENT (A I VALUE ARRAY-SIZE)
                              ((TEMP-I (NAT 0)))
                              (PUSH-LOCAL I)
                              (FETCH-TEMP-STK)
                              (SET-LOCAL TEMP-I)
                              (TEST-INT-AND-JUMP NEG 0)
                              (PUSH-LOCAL ARRAY-SIZE)
                              (PUSH-LOCAL TEMP-I)
                              (SUB-INT)
                              (TEST-INT-AND-JUMP NOT-POS 0)
                              (PUSH-LOCAL VALUE)
                              (FETCH-TEMP-STK)
                              (PUSH-LOCAL A)
                              (PUSH-LOCAL TEMP-I)
                              (INT-TO-NAT)
                              (ADD-NAT)
                              (DEPOSIT-TEMP-STK)
                              (JUMP 1)
                              (DL 0 NIL (PUSH-CONSTANT (NAT 1)))
                              (POP-GLOBAL C-C)
                              (DL 1 NIL (RET)))
```

DEFINITION
```
(MG-BOOLEAN-AND-CALL-SEQUENCE STMT)
   =
(CONS (LIST 'PUSH-LOCAL
            (CAR (CALL-ACTUALS STMT)))
      (CONS (LIST 'PUSH-LOCAL
                  (CADR (CALL-ACTUALS STMT)))
            (CONS (LIST 'PUSH-LOCAL
                        (CADDR (CALL-ACTUALS STMT)))
                  '((CALL MG-BOOLEAN-AND)))))
```

DEFINITION
```
(MG-BOOLEAN-AND-TRANSLATION)
   =
'(MG-BOOLEAN-AND (ANS B1 B2)
                 NIL
                 (PUSH-LOCAL B1)
                 (FETCH-TEMP-STK)
                 (PUSH-LOCAL B2)
                 (FETCH-TEMP-STK)
                 (AND-BOOL)
                 (PUSH-LOCAL ANS)
                 (DEPOSIT-TEMP-STK)
                 (RET))
```

DEFINITION
```
(MG-BOOLEAN-NOT-CALL-SEQUENCE STMT)
   =
(CONS (LIST 'PUSH-LOCAL
            (CAR (CALL-ACTUALS STMT)))
      (CONS (LIST 'PUSH-LOCAL
                  (CADR (CALL-ACTUALS STMT)))
            '((CALL MG-BOOLEAN-NOT))))
```

DEFINITION
```
(MG-BOOLEAN-NOT-TRANSLATION)
   =
'(MG-BOOLEAN-NOT (ANS B1)
                 NIL
                 (PUSH-LOCAL B1)
```

```
                 (FETCH-TEMP-STK)
                 (NOT-BOOL)
                 (PUSH-LOCAL ANS)
                 (DEPOSIT-TEMP-STK)
                 (RET))
```

DEFINITION

```
(MG-BOOLEAN-OR-CALL-SEQUENCE STMT)
   =
(CONS (LIST 'PUSH-LOCAL
            (CAR (CALL-ACTUALS STMT)))
      (CONS (LIST 'PUSH-LOCAL
                  (CADR (CALL-ACTUALS STMT)))
            (CONS (LIST 'PUSH-LOCAL
                        (CADDR (CALL-ACTUALS STMT)))
                  '((CALL MG-BOOLEAN-OR)))))
```

DEFINITION

```
(MG-BOOLEAN-OR-TRANSLATION)
   =
'(MG-BOOLEAN-OR (ANS B1 B2)
                NIL
                (PUSH-LOCAL B1)
                (FETCH-TEMP-STK)
                (PUSH-LOCAL B2)
                (FETCH-TEMP-STK)
                (OR-BOOL)
                (PUSH-LOCAL ANS)
                (DEPOSIT-TEMP-STK)
                (RET))
```

DEFINITION

```
(MG-COND-TO-P-NAT C COND-LIST)
   =
(LIST 'NAT
      (CONDITION-INDEX C COND-LIST))
```

DEFINITION

```
(MG-INDEX-ARRAY-CALL-SEQUENCE STMT LABEL-ALIST)
   =
(LIST (LIST 'PUSH-LOCAL (CAR (CALL-ACTUALS STMT)))
      (LIST 'PUSH-LOCAL (CADR (CALL-ACTUALS STMT)))
      (LIST 'PUSH-LOCAL (CADDR (CALL-ACTUALS STMT)))
      (LIST 'PUSH-CONSTANT (TAG 'INT (CADDDR (CALL-ACTUALS STMT))))
      '(CALL MG-INDEX-ARRAY)
      '(PUSH-GLOBAL C-C)
      '(SUB1-NAT)
      (LIST 'TEST-NAT-AND-JUMP
            'ZERO
            (FETCH-LABEL 'ROUTINEERROR
                         LABEL-ALIST)))
```

DEFINITION

```
(MG-INDEX-ARRAY-TRANSLATION)
   =
'(MG-INDEX-ARRAY (ANS A I ARRAY-SIZE)
                 ((TEMP-I (NAT 0)))
                 (PUSH-LOCAL I)
                 (FETCH-TEMP-STK)
                 (SET-LOCAL TEMP-I)
                 (TEST-INT-AND-JUMP NEG 0)
                 (PUSH-LOCAL ARRAY-SIZE)
                 (PUSH-LOCAL TEMP-I)
                 (SUB-INT)
                 (TEST-INT-AND-JUMP NOT-POS 0)
                 (PUSH-LOCAL A)
                 (PUSH-LOCAL TEMP-I)
                 (INT-TO-NAT)
                 (ADD-NAT)
```

```
                 (FETCH-TEMP-STK)
                 (PUSH-LOCAL ANS)
                 (DEPOSIT-TEMP-STK)
                 (JUMP 1)
                 (DL 0 NIL (PUSH-CONSTANT (NAT 1)))
                 (POP-GLOBAL C-C)
                 (DL 1 NIL (RET)))
```

DEFINITION

```
(MG-INTEGER-ADD-CALL-SEQUENCE STMT LABEL-ALIST)
   =
(LIST (LIST 'PUSH-LOCAL (CAR (CALL-ACTUALS STMT)))
      (LIST 'PUSH-LOCAL (CADR (CALL-ACTUALS STMT)))
      (LIST 'PUSH-LOCAL (CADDR (CALL-ACTUALS STMT)))
      '(CALL MG-INTEGER-ADD)
      '(PUSH-GLOBAL C-C)
      '(SUB1-NAT)
      (LIST 'TEST-NAT-AND-JUMP
            'ZERO
            (FETCH-LABEL 'ROUTINEERROR
                         LABEL-ALIST)))
```

DEFINITION

```
(MG-INTEGER-ADD-TRANSLATION)
   =
'(MG-INTEGER-ADD (ANS Y Z)
                 ((T1 (INT 0)))
                 (PUSH-CONSTANT (BOOL F))
                 (PUSH-LOCAL Y)
                 (FETCH-TEMP-STK)
                 (PUSH-LOCAL Z)
                 (FETCH-TEMP-STK)
                 (ADD-INT-WITH-CARRY)
                 (POP-LOCAL T1)
                 (TEST-BOOL-AND-JUMP T 0)
                 (PUSH-LOCAL T1)
                 (PUSH-LOCAL ANS)
                 (DEPOSIT-TEMP-STK)
                 (JUMP 1)
                 (DL 0 NIL (PUSH-CONSTANT (NAT 1)))
                 (POP-GLOBAL C-C)
                 (DL 1 NIL (RET)))
```

DEFINITION

```
(MG-INTEGER-LE-CALL-SEQUENCE STMT)
   =
(CONS (LIST 'PUSH-LOCAL
            (CAR (CALL-ACTUALS STMT)))
      (CONS (LIST 'PUSH-LOCAL
                  (CADR (CALL-ACTUALS STMT)))
            (CONS (LIST 'PUSH-LOCAL
                        (CADDR (CALL-ACTUALS STMT)))
                  '((CALL MG-INTEGER-LE)))))
```

DEFINITION

```
(MG-INTEGER-LE-TRANSLATION)
   =
'(MG-INTEGER-LE (ANS X Y)
                NIL
                (PUSH-LOCAL Y)
                (FETCH-TEMP-STK)
                (PUSH-LOCAL X)
                (FETCH-TEMP-STK)
                (LT-INT)
                (NOT-BOOL)
                (PUSH-LOCAL ANS)
                (DEPOSIT-TEMP-STK)
                (RET))
```

DEFINITION

```
(MG-INTEGER-SUBTRACT-CALL-SEQUENCE STMT LABEL-ALIST)
   =
(LIST (LIST 'PUSH-LOCAL (CAR (CALL-ACTUALS STMT)))
      (LIST 'PUSH-LOCAL (CADR (CALL-ACTUALS STMT)))
      (LIST 'PUSH-LOCAL (CADDR (CALL-ACTUALS STMT)))
      '(CALL MG-INTEGER-SUBTRACT)
      '(PUSH-GLOBAL C-C)
      '(SUB1-NAT)
      (LIST 'TEST-NAT-AND-JUMP
            'ZERO
            (FETCH-LABEL 'ROUTINEERROR
                         LABEL-ALIST)))
```

DEFINITION

```
(MG-INTEGER-SUBTRACT-TRANSLATION)
   =
'(MG-INTEGER-SUBTRACT (ANS Y Z)
                      ((T1 (INT 0)))
                      (PUSH-CONSTANT (BOOL F))
                      (PUSH-LOCAL Y)
                      (FETCH-TEMP-STK)
                      (PUSH-LOCAL Z)
                      (FETCH-TEMP-STK)
                      (SUB-INT-WITH-CARRY)
                      (POP-LOCAL T1)
                      (TEST-BOOL-AND-JUMP T 0)
                      (PUSH-LOCAL T1)
                      (PUSH-LOCAL ANS)
                      (DEPOSIT-TEMP-STK)
                      (JUMP 1)
                      (DL 0 NIL (PUSH-CONSTANT (NAT 1)))
                      (POP-GLOBAL C-C)
                      (DL 1 NIL (RET)))
```

DEFINITION

```
(MG-INTEGER-UNARY-MINUS-CALL-SEQUENCE STMT LABEL-ALIST)
   =
(LIST (LIST 'PUSH-LOCAL (CAR (CALL-ACTUALS STMT)))
      (LIST 'PUSH-LOCAL (CADR (CALL-ACTUALS STMT)))
      '(CALL MG-INTEGER-UNARY-MINUS)
      '(PUSH-GLOBAL C-C)
      '(SUB1-NAT)
      (LIST 'TEST-NAT-AND-JUMP
            'ZERO
            (FETCH-LABEL 'ROUTINEERROR
                         LABEL-ALIST)))
```

DEFINITION

```
(MG-INTEGER-UNARY-MINUS-TRANSLATION)
   =
'(MG-INTEGER-UNARY-MINUS (ANS X)
                         ((MIN-INT (INT -2147483648))
                          (TEMP-X (INT 0)))
                         (PUSH-LOCAL X)
                         (FETCH-TEMP-STK)
                         (SET-LOCAL TEMP-X)
                         (PUSH-LOCAL MIN-INT)
                         (EQ)
                         (TEST-BOOL-AND-JUMP F 0)
                         (PUSH-CONSTANT (NAT 1))
                         (POP-GLOBAL C-C)
                         (JUMP 1)
                         (DL 0 NIL (PUSH-LOCAL TEMP-X))
                         (NEG-INT)
                         (PUSH-LOCAL ANS)
                         (DEPOSIT-TEMP-STK)
                         (DL 1 NIL (RET)))
```

DEFINITION
```
(MG-SIMPLE-CONSTANT-ASSIGNMENT-CALL-SEQUENCE STMT)
   =
(CONS (LIST 'PUSH-LOCAL (CAR (CALL-ACTUALS STMT)))
      (CONS (LIST 'PUSH-CONSTANT
                  (MG-TO-P-SIMPLE-LITERAL (CADR (CALL-ACTUALS STMT))))
            '((CALL MG-SIMPLE-CONSTANT-ASSIGNMENT))))
```

DEFINITION
```
(MG-SIMPLE-CONSTANT-ASSIGNMENT-TRANSLATION)
   =
'(MG-SIMPLE-CONSTANT-ASSIGNMENT (DEST SOURCE)
                                NIL
                                (PUSH-LOCAL SOURCE)
                                (PUSH-LOCAL DEST)
                                (DEPOSIT-TEMP-STK)
                                (RET))
```

DEFINITION
```
(MG-SIMPLE-CONSTANT-EQ-CALL-SEQUENCE STMT)
   =
(CONS (LIST 'PUSH-LOCAL
            (CAR (CALL-ACTUALS STMT)))
      (CONS (LIST 'PUSH-LOCAL (CADR (CALL-ACTUALS STMT)))
            (CONS (LIST 'PUSH-CONSTANT
                        (MG-TO-P-SIMPLE-LITERAL (CADDR (CALL-ACTUALS STMT))))
                  '((CALL MG-SIMPLE-CONSTANT-EQ)))))
```

DEFINITION
```
(MG-SIMPLE-CONSTANT-EQ-TRANSLATION)
   =
'(MG-SIMPLE-CONSTANT-EQ (ANS X Y)
                        NIL
                        (PUSH-LOCAL X)
                        (FETCH-TEMP-STK)
                        (PUSH-LOCAL Y)
                        (EQ)
                        (PUSH-LOCAL ANS)
                        (DEPOSIT-TEMP-STK)
                        (RET))
```

DEFINITION
```
(MG-SIMPLE-VARIABLE-ASSIGNMENT-CALL-SEQUENCE STMT)
   =
(CONS (LIST 'PUSH-LOCAL (CAR (CALL-ACTUALS STMT)))
      (CONS (LIST 'PUSH-LOCAL
                  (CADR (CALL-ACTUALS STMT)))
            '((CALL MG-SIMPLE-VARIABLE-ASSIGNMENT))))
```

DEFINITION
```
(MG-SIMPLE-VARIABLE-ASSIGNMENT-TRANSLATION)
   =
'(MG-SIMPLE-VARIABLE-ASSIGNMENT (DEST SOURCE)
                                NIL
                                (PUSH-LOCAL SOURCE)
                                (FETCH-TEMP-STK)
                                (PUSH-LOCAL DEST)
                                (DEPOSIT-TEMP-STK)
                                (RET))
```

DEFINITION
```
(MG-SIMPLE-VARIABLE-EQ-CALL-SEQUENCE STMT)
   =
(CONS (LIST 'PUSH-LOCAL
            (CAR (CALL-ACTUALS STMT)))
      (CONS (LIST 'PUSH-LOCAL
                  (CADR (CALL-ACTUALS STMT)))
            (CONS (LIST 'PUSH-LOCAL
                        (CADDR (CALL-ACTUALS STMT)))
                  '((CALL MG-SIMPLE-VARIABLE-EQ)))))
```

DEFINITION

```
(MG-SIMPLE-VARIABLE-EQ-TRANSLATION)
   =
'(MG-SIMPLE-VARIABLE-EQ (ANS X Y)
                        NIL
                        (PUSH-LOCAL X)
                        (FETCH-TEMP-STK)
                        (PUSH-LOCAL Y)
                        (FETCH-TEMP-STK)
                        (EQ)
                        (PUSH-LOCAL ANS)
                        (DEPOSIT-TEMP-STK)
                        (RET))
```

DEFINITION

```
(MG-TO-P-SIMPLE-LITERAL LIT)
   =
(COND ((INT-LITERALP LIT)
       (LIST 'INT (CADR LIT)))
      ((BOOLEAN-LITERALP LIT)
       (IF (EQUAL (CADR LIT) 'FALSE-MG)
           '(BOOL F)
           '(BOOL T)))
      ((CHARACTER-LITERALP LIT)
       (LIST 'INT (CADR LIT)))
      (T 0))
```

DEFINITION

```
(MG-TO-P-SIMPLE-LITERAL-LIST LST)
   =
(IF (NLISTP LST)
    NIL
    (CONS (MG-TO-P-SIMPLE-LITERAL (CAR LST))
          (MG-TO-P-SIMPLE-LITERAL-LIST (CDR LST))))
```

DEFINITION

```
(P-NAT-TO-MG-COND P-NAT COND-LIST)
   =
(CASE P-NAT
      ((NAT 0) 'LEAVE)
      ((NAT 1) 'ROUTINEERROR)
      ((NAT 2) 'NORMAL)
      (OTHERWISE (CAR (NTH COND-LIST
                           (SUB1 (SUB1 (SUB1 (CADR P-NAT))))))))
```

DEFINITION

```
(PREDEFINED-PROC-CALL-SEQUENCE STMT LABEL-ALIST)
   =
(CASE
 (CALL-NAME STMT)
 (MG-SIMPLE-VARIABLE-ASSIGNMENT
  (MG-SIMPLE-VARIABLE-ASSIGNMENT-CALL-SEQUENCE STMT))
 (MG-SIMPLE-CONSTANT-ASSIGNMENT
  (MG-SIMPLE-CONSTANT-ASSIGNMENT-CALL-SEQUENCE STMT))
 (MG-SIMPLE-VARIABLE-EQ (MG-SIMPLE-VARIABLE-EQ-CALL-SEQUENCE STMT))
 (MG-SIMPLE-CONSTANT-EQ (MG-SIMPLE-CONSTANT-EQ-CALL-SEQUENCE STMT))
 (MG-INTEGER-LE (MG-INTEGER-LE-CALL-SEQUENCE STMT))
 (MG-INTEGER-UNARY-MINUS (MG-INTEGER-UNARY-MINUS-CALL-SEQUENCE STMT
                                                               LABEL-ALIST))
 (MG-INTEGER-ADD (MG-INTEGER-ADD-CALL-SEQUENCE STMT LABEL-ALIST))
 (MG-INTEGER-SUBTRACT (MG-INTEGER-SUBTRACT-CALL-SEQUENCE STMT LABEL-ALIST))
 (MG-BOOLEAN-OR (MG-BOOLEAN-OR-CALL-SEQUENCE STMT))
 (MG-BOOLEAN-AND (MG-BOOLEAN-AND-CALL-SEQUENCE STMT))
 (MG-BOOLEAN-NOT (MG-BOOLEAN-NOT-CALL-SEQUENCE STMT))
 (MG-INDEX-ARRAY (MG-INDEX-ARRAY-CALL-SEQUENCE STMT LABEL-ALIST))
 (MG-ARRAY-ELEMENT-ASSIGNMENT
  (MG-ARRAY-ELEMENT-ASSIGNMENT-CALL-SEQUENCE STMT
                                             LABEL-ALIST))
 (OTHERWISE NIL))
```

DEFINITION
```
(PREDEFINED-PROCEDURE-TRANSLATIONS-LIST)
   =
(LIST (MG-SIMPLE-VARIABLE-ASSIGNMENT-TRANSLATION)
      (MG-SIMPLE-CONSTANT-ASSIGNMENT-TRANSLATION)
      (MG-SIMPLE-VARIABLE-EQ-TRANSLATION)
      (MG-SIMPLE-CONSTANT-EQ-TRANSLATION)
      (MG-INTEGER-LE-TRANSLATION)
      (MG-INTEGER-UNARY-MINUS-TRANSLATION)
      (MG-INTEGER-ADD-TRANSLATION)
      (MG-INTEGER-SUBTRACT-TRANSLATION)
      (MG-BOOLEAN-OR-TRANSLATION)
      (MG-BOOLEAN-AND-TRANSLATION)
      (MG-BOOLEAN-NOT-TRANSLATION)
      (MG-INDEX-ARRAY-TRANSLATION)
      (MG-ARRAY-ELEMENT-ASSIGNMENT-TRANSLATION))
```

DEFINITION
```
(PROC-CALL-CODE CINFO STMT COND-LIST LOCALS COND-LOCALS-LNGTH)
   =
(APPEND (PUSH-PARAMETERS-CODE LOCALS
                              (CALL-ACTUALS STMT))
        (CONS (LIST 'CALL (CALL-NAME STMT))
              (CONDITION-MAP-CODE (CALL-CONDS STMT)
                                  (LABEL-CNT CINFO)
                                  COND-LIST
                                  (LABEL-ALIST CINFO)
                                  COND-LOCALS-LNGTH)))
```

DEFINITION
```
(PUSH-ACTUALS-CODE ACTUALS)
   =
(IF (NLISTP ACTUALS)
    NIL
    (CONS (LIST 'PUSH-LOCAL (CAR ACTUALS))
          (PUSH-ACTUALS-CODE (CDR ACTUALS))))
```

DEFINITION
```
(PUSH-LOCAL-ARRAY-VALUES-CODE ARRAY-VALUE)
   =
(IF (NLISTP ARRAY-VALUE)
    NIL
    (CONS (LIST 'PUSH-CONSTANT
                (MG-TO-P-SIMPLE-LITERAL (CAR ARRAY-VALUE)))
          (PUSH-LOCAL-ARRAY-VALUES-CODE (CDR ARRAY-VALUE))))
```

DEFINITION
```
(PUSH-LOCALS-ADDRESSES-CODE LOCALS N)
   =
(COND
 ((NLISTP LOCALS) NIL)
 ((SIMPLE-MG-TYPE-REFP (CADAR LOCALS))
  (CONS (LIST 'PUSH-TEMP-STK-INDEX N)
        (PUSH-LOCALS-ADDRESSES-CODE (CDR LOCALS)
                                    N)))
 (T
  (CONS
   (LIST 'PUSH-TEMP-STK-INDEX N)
   (PUSH-LOCALS-ADDRESSES-CODE
    (CDR LOCALS)
    (ADD1 (DIFFERENCE N
                      (ARRAY-LENGTH (CADAR LOCALS))))))))
```

DEFINITION
```
(PUSH-LOCALS-VALUES-CODE LOCALS)
   =
(COND ((NLISTP LOCALS) NIL)
      ((SIMPLE-MG-TYPE-REFP (CADAR LOCALS))
       (CONS (LIST 'PUSH-CONSTANT
                   (MG-TO-P-SIMPLE-LITERAL (CADDAR LOCALS)))
             (PUSH-LOCALS-VALUES-CODE (CDR LOCALS))))
      (T (APPEND (PUSH-LOCAL-ARRAY-VALUES-CODE (CADDAR LOCALS))
                 (PUSH-LOCALS-VALUES-CODE (CDR LOCALS)))))
```

DEFINITION
```
(PUSH-PARAMETERS-CODE LOCALS ACTUALS)
   =
(APPEND (PUSH-LOCALS-VALUES-CODE LOCALS)
        (APPEND (PUSH-LOCALS-ADDRESSES-CODE LOCALS
                                            (SUB1 (DATA-LENGTH LOCALS)))
                (PUSH-ACTUALS-CODE ACTUALS)))
```

DEFINITION
```
(SET-LABEL-ALIST CINFO NEW-LABEL-ALIST)
   =
(MAKE-CINFO (CODE CINFO)
            NEW-LABEL-ALIST
            (LABEL-CNT CINFO))
```

DEFINITION
```
(TRANSLATE CINFO COND-LIST STMT PROC-LIST)
   =
(CASE
 (CAR STMT)
 (NO-OP-MG CINFO)
 (SIGNAL-MG
  (MAKE-CINFO (APPEND (CODE CINFO)
                      (LIST (LIST 'PUSH-CONSTANT
                                  (MG-COND-TO-P-NAT (SIGNALLED-CONDITION STMT)
                                                    COND-LIST))
                            '(POP-GLOBAL C-C)
                            (LIST 'JUMP
                                  (FETCH-LABEL (SIGNALLED-CONDITION STMT)
                                               (LABEL-ALIST CINFO)))))
              (LABEL-ALIST CINFO)
              (LABEL-CNT CINFO)))
 (PROG2-MG (TRANSLATE (TRANSLATE CINFO COND-LIST
                                 (PROG2-LEFT-BRANCH STMT)
                                 PROC-LIST)
                      COND-LIST
                      (PROG2-RIGHT-BRANCH STMT)
                      PROC-LIST))
 (LOOP-MG
  (DISCARD-LABEL
   (ADD-CODE
    (TRANSLATE (MAKE-CINFO (APPEND (CODE CINFO)
                                   (LIST (CONS 'DL
                                               (CONS (LABEL-CNT CINFO)
                                                     '(NIL (NO-OP))))))
                           (CONS (CONS 'LEAVE
                                       (ADD1 (LABEL-CNT CINFO)))
                                 (LABEL-ALIST CINFO))
                           (ADD1 (ADD1 (LABEL-CNT CINFO))))
               COND-LIST
               (LOOP-BODY STMT)
               PROC-LIST)
    (CONS (LIST 'JUMP (LABEL-CNT CINFO))
          (CONS (CONS 'DL
                      (CONS (ADD1 (LABEL-CNT CINFO))
                            '(NIL (PUSH-CONSTANT (NAT 2)))))
                '((POP-GLOBAL C-C))))))
```

```
(IF-MG
 (ADD-CODE
  (TRANSLATE
   (ADD-CODE (TRANSLATE (MAKE-CINFO (APPEND (CODE CINFO)
                                            (LIST (LIST 'PUSH-LOCAL
                                                        (IF-CONDITION STMT))
                                                  '(FETCH-TEMP-STK)
                                                  (LIST 'TEST-BOOL-AND-JUMP
                                                        'FALSE
                                                        (LABEL-CNT CINFO))))
                                    (LABEL-ALIST CINFO)
                                    (ADD1 (ADD1 (LABEL-CNT CINFO))))
                        COND-LIST
                        (IF-TRUE-BRANCH STMT)
                        PROC-LIST)
             (LIST (LIST 'JUMP (ADD1 (LABEL-CNT CINFO)))
                   (CONS 'DL
                         (CONS (LABEL-CNT CINFO)
                               '(NIL (NO-OP))))))
   COND-LIST
   (IF-FALSE-BRANCH STMT)
   PROC-LIST)
  (LIST (CONS 'DL
              (CONS (ADD1 (LABEL-CNT CINFO))
                    '(NIL (NO-OP)))))))
(BEGIN-MG
 (ADD-CODE
  (TRANSLATE
   (ADD-CODE
    (SET-LABEL-ALIST
     (TRANSLATE (MAKE-CINFO (CODE CINFO)
                            (APPEND (MAKE-LABEL-ALIST (WHEN-LABELS STMT)
                                                      (LABEL-CNT CINFO))
                                    (LABEL-ALIST CINFO))
                            (ADD1 (ADD1 (LABEL-CNT CINFO))))
                COND-LIST
                (BEGIN-BODY STMT)
                PROC-LIST)
     (LABEL-ALIST CINFO))
    (CONS (LIST 'JUMP (ADD1 (LABEL-CNT CINFO)))
          (CONS (CONS 'DL
                      (CONS (LABEL-CNT CINFO)
                            '(NIL (PUSH-CONSTANT (NAT 2)))))
                '((POP-GLOBAL C-C)))))
   COND-LIST
   (WHEN-HANDLER STMT)
   PROC-LIST)
  (LIST (CONS 'DL
              (CONS (ADD1 (LABEL-CNT CINFO))
                    '(NIL (NO-OP)))))))
(PROC-CALL-MG
 (MAKE-CINFO
  (APPEND
   (CODE CINFO)
   (PROC-CALL-CODE CINFO STMT COND-LIST
                   (DEF-LOCALS (FETCH-CALLED-DEF STMT PROC-LIST))
                   (LENGTH (DEF-COND-LOCALS (FETCH-CALLED-DEF STMT PROC-LIST)))))
  (LABEL-ALIST CINFO)
  (PLUS (LABEL-CNT CINFO)
        (ADD1 (ADD1 (LENGTH (CALL-CONDS STMT)))))))
(PREDEFINED-PROC-CALL-MG
 (ADD-CODE CINFO
           (PREDEFINED-PROC-CALL-SEQUENCE STMT
                                          (LABEL-ALIST CINFO))))
(OTHERWISE CINFO))
```

DEFINITION
```
(TRANSLATE-DEF DEF PROC-LIST)
   =
(APPEND (CONS (DEF-NAME DEF)
              (CONS (APPEND (LISTCARS (DEF-LOCALS DEF))
                            (LISTCARS (DEF-FORMALS DEF)))
                    '(NIL)))
        (CODE (TRANSLATE-DEF-BODY DEF PROC-LIST)))
```

DEFINITION
```
(TRANSLATE-DEF-BODY PROC-DEF PROC-LIST)
   =
(ADD-CODE
 (TRANSLATE (MAKE-CINFO NIL
                        (CONS '(ROUTINEERROR . 0)
                              (MAKE-LABEL-ALIST (MAKE-COND-LIST PROC-DEF)
                                                0))
                        1)
            (MAKE-COND-LIST PROC-DEF)
            (DEF-BODY PROC-DEF)
            PROC-LIST)
 (CONS '(DL 0 NIL (NO-OP))
       (CONS (LIST 'POP*
                   (DATA-LENGTH (DEF-LOCALS PROC-DEF)))
             '((RET)))))
```

DEFINITION
```
(TRANSLATE-PROC-LIST PROC-LIST)
   =
(APPEND (PREDEFINED-PROCEDURE-TRANSLATIONS-LIST)
        (TRANSLATE-PROC-LIST1 PROC-LIST PROC-LIST))
```

DEFINITION
```
(TRANSLATE-PROC-LIST1 PROC-LIST1 PROC-LIST2)
   =
(IF (NLISTP PROC-LIST1)
    NIL
    (CONS (TRANSLATE-DEF (CAR PROC-LIST1)
                         PROC-LIST2)
          (TRANSLATE-PROC-LIST1 (CDR PROC-LIST1)
                                PROC-LIST2)))
```

Chapter 6
THE CORRECTNESS THEOREM

Chapter 2 painted in rather broad strokes a picture of how one goes about showing the correctness of a translator via an interpreter equivalence proof. In the current chapter, we fill in the details and state formally our claim that we can implement Micro-Gypsy programs correctly in Piton.

6.1 Mapping Up

Recall from Chapter 2 the form of interpreter equivalence theorem we are presenting, as illustrated by the following diagram.

```
MG-State1 ---------------------> MG-State2
      |                              ^
   |Map-Down                     |Map-Up          1
      |                              |
      |                              |
      V                              |
  P-initial --------------------> P-final
                 P(P-initial, clock)
```

Figure 6-1: Our Commuting Diagram

In Chapters 3 and 4 we saw the definitions of the source and target language interpreters. Chapter 5 described our version of the *Map-Down* function. To complete the diagram it only remains to describe the *Map-Up* function.

We are not concerned with mapping up all aspects of the Piton execution environment, only those dynamic aspects representing the components of the Micro-Gypsy **MG-STATE**. One of our hypotheses is that no time-out or resource-error occurs; consequently the psw must be **'RUN**. That leaves only the current condition and variable alist.

6.1.1 Mapping the Current Condition Up

From the discussion in Section 5.2, it should be apparent that the mapping from Micro-Gypsy conditions to Piton naturals with respect to a fixed condition list, is injective. It is trivial to compute the inverse mapping **P-NAT-TO-MG-COND** (page 122). The ability to map up the current condition relies upon the fact that we maintain as an invariant that the current condition is always legitimate. That is, the current condition is one of the special conditions--**'NORMAL**, **'ROUTINEERROR**, **'LEAVE**--or a member of the condition list. This assures that the map up for conditions is well defined. Notice that the inverse mapping requires the condition list as a parameter. This is an example of the need for source language information in the definition of the *Map-Up* function, to which we alluded earlier.

6.1.2 Mapping Variables Up

In Section 5.1 we discussed the scheme for storing Micro-Gypsy data values on the **MG-ALIST** in the Piton state. Data values are stored in the Piton temp-stk and accessible via pointers in the Piton ctrl-stk. We cannot reconstruct an **MG-ALIST** solely from the temp-stk and ctrl-stk for two reasons.

1. Distinct Micro-Gypsy simple literals may map down to the same Piton data value.
2. There is no indication in the Piton state of the size of the data structure pointed to by an index in a ctrl-stk frame.

Because of these reasons, we need the type information from the Micro-Gypsy level to be able to map up the variables. We could supply this in the form of the initial **MG-ALIST**. However, this might raise the suspicion that the *Map-Up* is somehow cheating, as described in Section 2.2. To allay this suspicion, we pass to the *Map-Up* function only the *signature* of the initial **MG-ALIST**. This is the list of **<NAME, TYPE>** pairs derived from the **MG-ALIST** by dropping the data values.

It is obvious that mapping up data structures is straightforward given the Micro-Gypsy type information. Given that the names of data structures are preserved by our *Map-Down* function we use the following procedure.

- For a simple variable **X**, use the pointer associated with **X** in the bindings component of the topmost frame of the Piton ctrl-stk as an index into the temp-stk. The value at that index is converted into its Micro-Gypsy analog according to the scheme in section 5.1
- For an array variable **A** of length *n*, fetch the pointer for **A** from the bindings. Form the list of *n* successive temp-stk elements beginning at that index and map to Micro-Gypsy values on an element-wise basis.

To illustrate, suppose that we have the following values for the **MG-ALIST** signature, Piton bindings, and Piton temp-stk:

MG-ALIST signature

```
((A (ARRAY 3 INT-MG)) (B BOOL-MG)
 (X INT-MG) (CH CHARACTER-MG))
```

Piton bindings

```
((A . (NAT 6)) (B . (NAT 3))
 (X . (NAT 10)) (CH . (NAT 1)))
```

Piton temp-stk

```
index          contents

 11            ...
 10            (INT -12)
  9            ...
  8            (INT -6)
  7            (INT 0)
  6            (INT 4)
  5            ...
  4            ...
  3            (BOOL T)
  2            ...
  1            (INT 78)
  0            ...
```

It is straightforward to map up to the following **MG-ALIST** value

```
((A (ARRAY 3 INT-MG) ((INT-MG 4) (INT-MG 0) (INT-MG -6)))
 (B BOOL-MG (BOOLEAN-MG TRUE-MG))
 (X INT-MG (INT-MG -12))
 (CH CHARACTER-MG (CHARACTER-MG 78))).
```

Notice that without the Micro-Gypsy type information we would not have known whether to map the Piton value of **CH** to the Micro-Gypsy integer 78 or to the character *N*.

6.2 The Correctness Theorem

Our main result is the theorem **TRANSLATION-IS-CORRECT5**, an interpreter equivalence theorem which proves that Micro-Gypsy programs are correctly implemented in Piton by our translation scheme.

Theorem. **TRANSLATION-IS-CORRECT5**

```
(IMPLIES
 (AND (OK-EXECUTION-ENVIRONMENT STMT COND-LIST PROC-LIST MG-STATE SUBR N)
      (NOT (RESOURCE-ERRORP
            (MG-MEANING-R STMT PROC-LIST MG-STATE N
                          (LIST (DATA-LENGTH (MG-ALIST MG-STATE))
                                (PLUS 2 (LENGTH (MG-ALIST MG-STATE))))))))
 (EQUAL (MAP-UP (P (MAP-DOWN1 MG-STATE PROC-LIST COND-LIST SUBR STMT)
                   (CLOCK STMT PROC-LIST MG-STATE N))
                (SIGNATURE (MG-ALIST MG-STATE))
                COND-LIST)
        (MG-MEANING STMT PROC-LIST MG-STATE N))))
```

This theorem can be seen to be a formalization of the interpreter equivalence diagram with which we began this chapter by observing that **MG-MEANING** is the interpreter for the source language and **P** the interpreter for the target language. However, the theorem is quite subtle and we devote the remainder of this chapter to explaining it.

6.2.1 The Hypotheses of the Correctness Theorem

We are willing to assert that our translation process is correct only for Micro-Gypsy execution environments in which all of the components are well-formed and only if certain aberrant conditions do not occur during execution. The first hypothesis bundles together a number of assumptions including the following.

1. **STMT** is a Micro-Gypsy statement legal in the current execution environment.
2. **PROC-LIST** is a legal list of Micro-Gypsy user-defined procedures.
3. The current Micro-Gypsy state **MG-STATE** has the appropriate structure and its various components have legal values.
4. **SUBR** is a "new" Micro-Gypsy identifier unused as a procedure name in the **PROC-LIST**.
5. **COND-LIST** is a list of legal Micro-Gypsy condition names and is not more than a fixed maximum length.

The theorem assumes with the second hypothesis that the final **MG-STATE** is non-erroneous in the sense that the **MG-PSW** of the final state is **'RUN**. Two other possible values might appear.

1. **'TIMED-OUT** would mean that the "clock" parameter **N** to **MG-MEANING** is not large enough for the computation. This situation is discussed in Section 3.4.1-C.
2. A value of **'RESOURCE-ERROR** implies that the resource limitations have been exceeded at some point in the computation. This is discussed in Section 3.4.3.

What is the appropriate response to these errors? For any terminating program, it should be possible to find an adequate value for **N**; for a non-terminating program there is no such value. We anticipate that for many programs of interest it may be possible to *prove* that a given value of **N** is large enough, where the clock will typically be some function of the inputs. This issue is discussed further in Chapter 8.

A **CC** of **'RESOURCE-ERROR** indicates that the computation requires more temp-stk or ctrl-stk space than is allowed by the constants **MG-MAX-TEMP-STK-SIZE** and **MG-MAX-CTRL-STK-SIZE**. These constants would typically be set to the maximum allowed by the implementation, i.e. are as big as possible given the resource limitations of the underlying machine. Encountering this situation either means that the problem cannot be solved in the current implementation or that the problem needs to be recoded to obtain a more space-efficient algorithm.

6.2.2 The Conclusion of the Correctness Theorem

The conclusion is a formalization of the commuting diagram (1) in Section 6.1 above. It asserts that we can obtain an identical Micro-Gypsy **MG-STATE** either by running the Micro-Gypsy interpreter or by going through the Piton implementation.

6.2.2-A The Micro-Gypsy Route

One way to obtain the final result is indicated in the right hand side of the conclusion; run the Micro-Gypsy interpreter **MG-MEANING** to obtain the final state. Notice that this final state is only the dynamic **MG-STATE** component of the total Micro-Gypsy execution environment. Other components, such as the statement being interpreted and the procedure list, are static and not considered in the final result. Notice also that the final result is computed with the desired **MG-MEANING**, not with **MG-MEANING-R**, the version of the interpreter with resource errors.

6.2.2-B The Piton Route

The left hand side of the conclusion describes a path using the Piton interpreter. The function **MAP-DOWN1** described in Section 5.6 creates a Piton execution environment representing the initial Micro-Gypsy execution environment. It takes as parameters the various components of the Micro-Gypsy execution environment and a new name **SUBR** for generating our special "entry point" procedure. We run the Piton interpreter **P** on this Piton state to obtain a final Piton state.

How many steps do we run the Piton interpreter? This value is computed by the function **CLOCK** (page 129). For the execution of a Micro-Gypsy statement, **CLOCK** calculates the number of Piton instructions executed in the implementation. Since this is dependent upon the inputs in most cases, this computation requires essentially running the Micro-Gypsy interpreter to emulate the computation and counting up the number of lower-level steps required.

The computation of these clock values is quite complicated and very sensitive to the implementation. Consider the **IF-MG** statement. Recall from Section 5.3.6 that the code generated for an **IF-MG** statement is

```
(PUSH-LOCAL b)                          ; 1
(FETCH-TEMP-STK)                        ; 2
(TEST-BOOL-AND-JUMP FALSE n)            ; 3
<code for true-branch>                  ; 4
(JUMP n+1)                              ; 5
(DL n NIL (NO-OP))                      ; 6
<code for false-branch>                 ; 7
(DL n+1 NIL (NO-OP))                    ; 8
```

The number of Piton instructions executed in the implementation of a Micro-Gypsy **IF-MG** statement depends on which branch is executed and upon whether or not a condition is raised in the execution of that branch. The value is defined by the following expression from the definition of **CLOCK**

```
(IF (MG-EXPRESSION-FALSEP (IF-CONDITION STMT) MG-STATE)
  (IF (NORMAL (MG-MEANING (IF-FALSE-BRANCH STMT)
                          PROC-LIST MG-STATE (SUB1 N)))
      (PLUS 5 (CLOCK (IF-FALSE-BRANCH STMT) PROC-LIST MG-STATE (SUB1 N)))
      (PLUS 4 (CLOCK (IF-FALSE-BRANCH STMT) PROC-LIST MG-STATE (SUB1 N))))
  (IF (NORMAL (MG-MEANING (IF-TRUE-BRANCH STMT)
                          PROC-LIST MG-STATE (SUB1 N)))
      (PLUS 5 (CLOCK (IF-TRUE-BRANCH STMT) PROC-LIST MG-STATE (SUB1 N)))
      (PLUS 3 (CLOCK (IF-TRUE-BRANCH STMT) PROC-LIST MG-STATE (SUB1 N))))).
```

For example, in the case where the test is false and no condition is raised the Piton instructions at lines 1, 2, 3, 6, and 8 are executed along with the code for the false branch at line 7. This is five instructions plus the number of clock "ticks" for the false branch. The reader is advised to study this expression and the code above to see the relationship.

After running the Piton interpreter for an appropriate number of "ticks" on the initial Piton state, we map the result back into the Micro-Gypsy world using the **MAP-UP** function outlined in Section 6.1. Notice that **MAP-UP** requires as parameters the Micro-Gypsy **COND-LIST** for mapping up the values of the current condition (see Section 6.1.1) and the type information from the Micro-Gypsy **MG-ALIST** (see Section 6.1.2).

In the next chapter, we outline the proof of **TRANSLATION-IS-CORRECT5**.

6.3 An Alphabetical Listing of Functions Used in the Correctness Theorem

Many of the definitions involved in the statement of the correctness theorem have been listed in previous chapters. This section contains the remainder of the formal definitions involved in our *Map-Down* function and in the statement of the correctness theorem.

```
DEFINITION.
(ALL-LABELS-UNIQUE CODELIST)
   =
(NO-DUPLICATES (COLLECT-LABELS CODELIST))

DEFINITION.
(ALL-POINTERS-BIGGER LST N)
   =
(IF (NLISTP LST)
    T
    (AND (IF (LESSP (CAR LST) N) F T)
         (ALL-POINTERS-BIGGER (CDR LST) N)))

DEFINITION.
(ALL-POINTERS-SMALLER LST N)
   =
(IF (NLISTP LST)
    T
    (AND (LESSP (CAR LST) N)
         (ALL-POINTERS-SMALLER (CDR LST) N)))
```

DEFINITION.

```
(ASCENDING-LOCAL-ADDRESS-SEQUENCE LOCALS N)
   =
(COND
 ((NLISTP LOCALS) NIL)
 ((SIMPLE-MG-TYPE-REFP (FORMAL-TYPE (CAR LOCALS)))
  (CONS (TAG 'NAT N)
        (ASCENDING-LOCAL-ADDRESS-SEQUENCE (CDR LOCALS)
                                          (ADD1 N))))
 (T
  (CONS
   (TAG 'NAT N)
   (ASCENDING-LOCAL-ADDRESS-SEQUENCE
    (CDR LOCALS)
    (PLUS (ARRAY-LENGTH (FORMAL-TYPE (CAR LOCALS)))
          N)))))
```

DEFINITION.

```
(CLOCK STMT PROC-LIST MG-STATE N)
   =
(COND
 ((OR (ZEROP N) (NOT (NORMAL MG-STATE)))
  0)
 ((EQUAL (CAR STMT) 'NO-OP-MG) 0)
 ((EQUAL (CAR STMT) 'SIGNAL-MG) 3)
 ((EQUAL (CAR STMT) 'PROG2-MG)
  (PLUS (CLOCK (PROG2-LEFT-BRANCH STMT)
               PROC-LIST MG-STATE
               (SUB1 N))
        (CLOCK (PROG2-RIGHT-BRANCH STMT)
               PROC-LIST
               (MG-MEANING (PROG2-LEFT-BRANCH STMT)
                           PROC-LIST MG-STATE
                           (SUB1 N))
               (SUB1 N))))
 ((EQUAL (CAR STMT) 'LOOP-MG)
  (IF (NOT (NORMAL (MG-MEANING (LOOP-BODY STMT)
                               PROC-LIST MG-STATE
                               (SUB1 N))))
      (IF (EQUAL (CC (MG-MEANING (LOOP-BODY STMT)
                                 PROC-LIST MG-STATE
                                 (SUB1 N)))
                 'LEAVE)
          (PLUS 3
                (CLOCK (LOOP-BODY STMT)
                       PROC-LIST MG-STATE
                       (SUB1 N)))
          (ADD1 (CLOCK (LOOP-BODY STMT)
                       PROC-LIST MG-STATE
                       (SUB1 N))))
      (ADD1 (PLUS (ADD1 (CLOCK (LOOP-BODY STMT)
                               PROC-LIST MG-STATE
                               (SUB1 N)))
                  (CLOCK STMT PROC-LIST
                         (MG-MEANING (LOOP-BODY STMT)
                                     PROC-LIST MG-STATE
                                     (SUB1 N))
                         (SUB1 N))))))
```

```
((EQUAL (CAR STMT) 'IF-MG)
 (COND ((MG-EXPRESSION-FALSEP (IF-CONDITION STMT)
                              MG-STATE)
        (IF (NORMAL (MG-MEANING (IF-FALSE-BRANCH STMT)
                                PROC-LIST MG-STATE
                                (SUB1 N)))
            (PLUS 5
                  (CLOCK (IF-FALSE-BRANCH STMT)
                         PROC-LIST MG-STATE
                         (SUB1 N)))
            (PLUS 4
                  (CLOCK (IF-FALSE-BRANCH STMT)
                         PROC-LIST MG-STATE
                         (SUB1 N)))))
       ((NORMAL (MG-MEANING (IF-TRUE-BRANCH STMT)
                            PROC-LIST MG-STATE
                            (SUB1 N)))
        (PLUS 5
              (CLOCK (IF-TRUE-BRANCH STMT)
                     PROC-LIST MG-STATE
                     (SUB1 N))))
       (T (PLUS 3
                (CLOCK (IF-TRUE-BRANCH STMT)
                       PROC-LIST MG-STATE
                       (SUB1 N))))))
((EQUAL (CAR STMT) 'BEGIN-MG)
 (COND ((MEMBER (CC (MG-MEANING (BEGIN-BODY STMT)
                                PROC-LIST MG-STATE
                                (SUB1 N)))
                (WHEN-LABELS STMT))
        (IF (NORMAL (MG-MEANING (WHEN-HANDLER STMT)
                                PROC-LIST
                                (SET-CONDITION (MG-MEANING (BEGIN-BODY STMT)
                                                           PROC-LIST
                                                           MG-STATE
                                                           (SUB1 N))
                                               'NORMAL)
                                (SUB1 N)))
            (PLUS (CLOCK (BEGIN-BODY STMT)
                         PROC-LIST MG-STATE
                         (SUB1 N))
                  3
                  (CLOCK (WHEN-HANDLER STMT)
                         PROC-LIST
                         (SET-CONDITION (MG-MEANING (BEGIN-BODY STMT)
                                                    PROC-LIST MG-STATE
                                                    (SUB1 N))
                                        'NORMAL)
                         (SUB1 N)))
            (PLUS (CLOCK (BEGIN-BODY STMT)
                         PROC-LIST MG-STATE
                         (SUB1 N))
                  2
                  (CLOCK (WHEN-HANDLER STMT)
                         PROC-LIST
                         (SET-CONDITION (MG-MEANING (BEGIN-BODY STMT)
                                                    PROC-LIST MG-STATE
                                                    (SUB1 N))
                                        'NORMAL)
                         (SUB1 N)))))
```

```
        ((NORMAL (MG-MEANING (BEGIN-BODY STMT)
                             PROC-LIST MG-STATE
                             (SUB1 N)))
         (PLUS 2
               (CLOCK (BEGIN-BODY STMT)
                      PROC-LIST MG-STATE
                      (SUB1 N))))
        (T (CLOCK (BEGIN-BODY STMT)
                  PROC-LIST MG-STATE
                  (SUB1 N)))))
 ((EQUAL (CAR STMT) 'PROC-CALL-MG)
  (PLUS
   (DATA-LENGTH (DEF-LOCALS (FETCH-CALLED-DEF STMT PROC-LIST)))
   (LENGTH (DEF-LOCALS (FETCH-CALLED-DEF STMT PROC-LIST)))
   (LENGTH (CALL-ACTUALS STMT))
   1
   (CLOCK (DEF-BODY (FETCH-CALLED-DEF STMT PROC-LIST))
          PROC-LIST
          (MAKE-CALL-ENVIRONMENT MG-STATE STMT
                                 (FETCH-CALLED-DEF STMT PROC-LIST))
          (SUB1 N))
   5
   (IF
    (NORMAL
     (MG-MEANING (DEF-BODY (FETCH-CALLED-DEF STMT PROC-LIST))
                 PROC-LIST
                 (MAKE-CALL-ENVIRONMENT MG-STATE STMT
                                        (FETCH-CALLED-DEF STMT PROC-LIST))
                 (SUB1 N)))
    1 3)))
 ((EQUAL (CAR STMT)
         'PREDEFINED-PROC-CALL-MG)
  (PREDEFINED-PROC-CALL-CLOCK STMT MG-STATE))
 (T 0))
```

DEFINITION.

```
(CLOCK-PREDEFINED-PROC-CALL-BODY-TRANSLATION STMT MG-STATE)
   =
(CASE
 (CALL-NAME STMT)
 (MG-SIMPLE-VARIABLE-ASSIGNMENT 5)
 (MG-SIMPLE-CONSTANT-ASSIGNMENT 4)
 (MG-SIMPLE-VARIABLE-EQ 8)
 (MG-SIMPLE-CONSTANT-EQ 7)
 (MG-INTEGER-LE 9)
 (MG-INTEGER-UNARY-MINUS
  (IF (SMALL-INTEGERP (INEGATE (UNTAG (CADDR (ASSOC (CADR (CALL-ACTUALS STMT))
                                                    (MG-ALIST MG-STATE)))))
                      (MG-WORD-SIZE))
      11 10))
 (MG-INTEGER-ADD
  (IF (SMALL-INTEGERP (IPLUS (UNTAG (CADDR (ASSOC (CADR (CALL-ACTUALS STMT))
                                                  (MG-ALIST MG-STATE))))
                             (UNTAG (CADDR (ASSOC (CADDR (CALL-ACTUALS STMT))
                                                  (MG-ALIST MG-STATE)))))
                      (MG-WORD-SIZE))
      13 11))
 (MG-INTEGER-SUBTRACT
  (IF
   (SMALL-INTEGERP
    (IDIFFERENCE (UNTAG (CADDR (ASSOC (CADR (CALL-ACTUALS STMT))
                                      (MG-ALIST MG-STATE))))
                 (UNTAG (CADDR (ASSOC (CADDR (CALL-ACTUALS STMT))
                                      (MG-ALIST MG-STATE)))))
    (MG-WORD-SIZE))
   13 11))
 (MG-BOOLEAN-OR 8)
 (MG-BOOLEAN-AND 8)
 (MG-BOOLEAN-NOT 6)
```

```
 (MG-INDEX-ARRAY
  (COND
   ((NEGATIVEP (CADADDR (ASSOC (CADDR (CALL-ACTUALS STMT))
                               (MG-ALIST MG-STATE))))
    7)
   ((OR (EQUAL (IDIFFERENCE (CADDDR (CALL-ACTUALS STMT))
                            (CADADDR (ASSOC (CADDR (CALL-ACTUALS STMT))
                                            (MG-ALIST MG-STATE))))
               0)
        (NEGATIVEP (IDIFFERENCE (CADDDR (CALL-ACTUALS STMT))
                                (CADADDR (ASSOC (CADDR (CALL-ACTUALS STMT))
                                                (MG-ALIST MG-STATE))))))
    11)
   (T 17)))
 (MG-ARRAY-ELEMENT-ASSIGNMENT
  (COND
   ((NEGATIVEP (CADADDR (ASSOC (CADR (CALL-ACTUALS STMT))
                               (MG-ALIST MG-STATE))))
    7)
   ((OR (EQUAL (IDIFFERENCE (CADDDR (CALL-ACTUALS STMT))
                            (CADADDR (ASSOC (CADR (CALL-ACTUALS STMT))
                                            (MG-ALIST MG-STATE))))
               0)
        (NEGATIVEP (IDIFFERENCE (CADDDR (CALL-ACTUALS STMT))
                                (CADADDR (ASSOC (CADR (CALL-ACTUALS STMT))
                                                (MG-ALIST MG-STATE))))))
    11)
   (T 17)))
 (OTHERWISE 0))
```

DEFINITION.

```
(CLOCK-PREDEFINED-PROC-CALL-SEQUENCE NAME)
   =
(CASE NAME
      (MG-SIMPLE-VARIABLE-ASSIGNMENT 3)
      (MG-SIMPLE-CONSTANT-ASSIGNMENT 3)
      (MG-SIMPLE-VARIABLE-EQ 4)
      (MG-SIMPLE-CONSTANT-EQ 4)
      (MG-INTEGER-LE 4)
      (MG-INTEGER-UNARY-MINUS 6)
      (MG-INTEGER-ADD 7)
      (MG-INTEGER-SUBTRACT 7)
      (MG-BOOLEAN-OR 4)
      (MG-BOOLEAN-AND 4)
      (MG-BOOLEAN-NOT 3)
      (MG-INDEX-ARRAY 8)
      (MG-ARRAY-ELEMENT-ASSIGNMENT 8)
      (OTHERWISE 0))
```

DEFINITION.

```
(COLLECT-LABELS CODELIST)
   =
(COND ((NLISTP CODELIST) NIL)
      ((EQUAL (CAAR CODELIST) 'DL)
       (CONS (CADAR CODELIST)
             (COLLECT-LABELS (CDR CODELIST))))
      (T (COLLECT-LABELS (CDR CODELIST))))
```

DEFINITION.

```
(COLLECT-POINTERS BINDINGS ALIST)
   =
(COND ((NLISTP ALIST) NIL)
      ((SIMPLE-MG-TYPE-REFP (CADAR ALIST))
       (CONS (UNTAG (CDR (ASSOC (CAAR ALIST) BINDINGS)))
             (COLLECT-POINTERS BINDINGS
                               (CDR ALIST))))
      (T (APPEND (N-SUCCESSIVE-POINTERS (CDR (ASSOC (CAAR ALIST) BINDINGS))
                                        (ARRAY-LENGTH (CADAR ALIST)))
                 (COLLECT-POINTERS BINDINGS
                                   (CDR ALIST)))))
```

DEFINITION.
```
(INITIAL-BINDINGS MG-ALIST N)
   =
(IF (NLISTP MG-ALIST)
    NIL
    (IF (SIMPLE-MG-TYPE-REFP (CADR (CAR MG-ALIST)))
        (CONS (CONS (CAAR MG-ALIST) (TAG 'NAT N))
              (INITIAL-BINDINGS (CDR MG-ALIST) (ADD1 N)))
        (CONS (CONS (CAAR MG-ALIST) (TAG 'NAT N))
              (INITIAL-BINDINGS (CDR MG-ALIST)
                                (PLUS N (ARRAY-LENGTH (CADR (CAR MG-ALIST))))))))
```

DEFINITION.
```
(INITIAL-TEMP-STK MG-ALIST)
   =
(REVERSE (INITIAL-TEMP-STK-REVERSED MG-ALIST))
```

DEFINITION.
```
(INITIAL-TEMP-STK-REVERSED MG-ALIST)
   =
(IF (NLISTP MG-ALIST)
    NIL
    (IF (SIMPLE-MG-TYPE-REFP (CADR (CAR MG-ALIST)))
        (CONS (MG-TO-P-SIMPLE-LITERAL (CADDR (CAR MG-ALIST)))
              (INITIAL-TEMP-STK-REVERSED (CDR MG-ALIST)))
        (APPEND (MG-TO-P-SIMPLE-LITERAL-LIST (CADDR (CAR MG-ALIST)))
                (INITIAL-TEMP-STK-REVERSED (CDR MG-ALIST)))))
```

DEFINITION.
```
(LABEL-CNT-BIG-ENOUGH LC CODE)
   =
(COND ((NLISTP CODE) T)
      ((EQUAL (CAAR CODE) 'DL)
       (AND (LESSP (CADAR CODE) LC)
            (LABEL-CNT-BIG-ENOUGH LC (CDR CODE))))
      (T (LABEL-CNT-BIG-ENOUGH LC
                               (CDR CODE))))
```

DEFINITION.
```
(LABEL-HOLE-BIG-ENOUGH CINFO COND-LIST STMT PROC-LIST Y)
   =
(ALL-LABELS-UNIQUE (APPEND (CODE (TRANSLATE CINFO COND-LIST STMT
                                            PROC-LIST))
                           Y))
```

DEFINITION.
```
(MAKE-MG-PROC ALIST SUBR STMT COND-LIST)
   =
(LIST SUBR NIL COND-LIST (MAKE-MG-LOCALS-LIST ALIST) NIL STMT)
```

DEFINITION.
```
(MAKE-FRAME-ALIST DEF STMT CTRL-STK TEMP-STK)
   =
(APPEND (MAP-CALL-LOCALS (DEF-LOCALS DEF)
                         (LENGTH TEMP-STK))
        (MAP-CALL-FORMALS (DEF-FORMALS DEF)
                          (CALL-ACTUALS STMT)
                          (BINDINGS (TOP CTRL-STK))))
```

DEFINITION.
```
(MAKE-MG-LOCALS-LIST MG-ALIST)
   =
(IF (NLISTP MG-ALIST)
    NIL
    (CONS (LIST (NAME (CAR MG-ALIST))
                (M-TYPE (CAR MG-ALIST))
                (M-VALUE (CAR MG-ALIST)))
          (MAKE-MG-LOCALS-LIST (CDR MG-ALIST))))
```

DEFINITION.
```
(MAP-CALL-FORMALS FORMALS ACTUALS BINDINGS)
   =
(IF (NLISTP FORMALS)
    NIL
    (CONS (CONS (CAAR FORMALS)
                (CDR (ASSOC (CAR ACTUALS) BINDINGS)))
          (MAP-CALL-FORMALS (CDR FORMALS)
                            (CDR ACTUALS)
                            BINDINGS)))
```

DEFINITION.
```
(MAP-CALL-LOCALS LOCALS N)
   =
(COND ((NLISTP LOCALS) NIL)
      ((SIMPLE-MG-TYPE-REFP (CADAR LOCALS))
       (CONS (CONS (CAAR LOCALS) (TAG 'NAT N))
             (MAP-CALL-LOCALS (CDR LOCALS)
                              (ADD1 N))))
      (T (CONS (CONS (CAAR LOCALS) (TAG 'NAT N))
               (MAP-CALL-LOCALS (CDR LOCALS)
                                (PLUS (ARRAY-LENGTH (CADAR LOCALS))
                                      N)))))
```

DEFINITION.
```
(MAP-UP P-STATE MG-SIGNATURE COND-LIST)
   =
(MG-STATE (P-NAT-TO-MG-COND (PITON-CC P-STATE)
                            COND-LIST)
          (MAP-UP-VARS-LIST (BINDINGS (TOP (P-CTRL-STK P-STATE)))
                            (P-TEMP-STK P-STATE)
                            MG-SIGNATURE)
          'RUN)
```

DEFINITION.
```
(MAP-UP-VARS-LIST P-VARS TEMP-STK SIGNATURE)
   =
(IF (NLISTP SIGNATURE)
    NIL
    (CONS (MAP-UP-VARS-LIST-ELEMENT (ASSOC (CAAR SIGNATURE) P-VARS)
                                    TEMP-STK
                                    (CAR SIGNATURE))
          (MAP-UP-VARS-LIST P-VARS TEMP-STK
                            (CDR SIGNATURE))))
```

DEFINITION.
```
(MAP-UP-VARS-LIST-ARRAY P-VAR TEMP-STK VAR-SIGNATURE)
   =
(LIST
 (CAR VAR-SIGNATURE)
 (CADR VAR-SIGNATURE)
 (P-TO-MG-SIMPLE-LITERAL-LIST
  (FETCH-N-TEMP-STK-ELEMENTS TEMP-STK
                             (CDR P-VAR)
                             (ARRAY-LENGTH (CADR VAR-SIGNATURE)))
  (ARRAY-ELEMTYPE (CADR VAR-SIGNATURE))))
```

DEFINITION.
```
(MAP-UP-VARS-LIST-ELEMENT P-VAR TEMP-STK VAR-SIGNATURE)
   =
(IF (SIMPLE-MG-TYPE-REFP (CADR VAR-SIGNATURE))
    (MAP-UP-VARS-LIST-SIMPLE-ELEMENT P-VAR TEMP-STK VAR-SIGNATURE)
    (MAP-UP-VARS-LIST-ARRAY P-VAR TEMP-STK VAR-SIGNATURE))
```

DEFINITION.
```
(MAP-UP-VARS-LIST-SIMPLE-ELEMENT P-VAR TEMP-STK VAR-SIGNATURE)
   =
(LIST (CAR VAR-SIGNATURE)
      (CADR VAR-SIGNATURE)
      (P-TO-MG-SIMPLE-LITERAL (FETCH-TEMP (CDR P-VAR) TEMP-STK)
                              (CADR VAR-SIGNATURE)))
```

DEFINITION.
```
(MG-TO-P-LOCAL-VALUES LOCALS)
   =
(COND ((NLISTP LOCALS) NIL)
      ((SIMPLE-MG-TYPE-REFP (CADAR LOCALS))
       (CONS (MG-TO-P-SIMPLE-LITERAL (CADDAR LOCALS))
             (MG-TO-P-LOCAL-VALUES (CDR LOCALS))))
      (T (APPEND (MG-TO-P-SIMPLE-LITERAL-LIST (CADDAR LOCALS))
                 (MG-TO-P-LOCAL-VALUES (CDR LOCALS)))))
```

DEFINITION.
```
(MG-VAR-OK-IN-P-STATE MG-VAR BINDINGS TEMP-STK)
   =
(AND (DEFINEDP (CAR MG-VAR) BINDINGS)
     (IF (SIMPLE-MG-TYPE-REFP (M-TYPE MG-VAR))
         (OK-TEMP-STK-INDEX (CDR (ASSOC (CAR MG-VAR) BINDINGS))
                            TEMP-STK)
         (OK-TEMP-STK-ARRAY-INDEX (CDR (ASSOC (CAR MG-VAR) BINDINGS))
                                  TEMP-STK
                                  (ARRAY-LENGTH (M-TYPE MG-VAR)))))
```

DEFINITION.
```
(MG-VARS-LIST-OK-IN-P-STATE MG-VARS BINDINGS TEMP-STK)
   =
(IF (NLISTP MG-VARS)
    T
    (AND (MG-VAR-OK-IN-P-STATE (CAR MG-VARS)
                               BINDINGS TEMP-STK)
         (MG-VARS-LIST-OK-IN-P-STATE (CDR MG-VARS)
                                     BINDINGS TEMP-STK)))
```

DEFINITION.
```
(N-SUCCESSIVE-POINTERS NAT N)
   =
(IF (ZEROP N)
    NIL
    (CONS (UNTAG NAT)
          (N-SUCCESSIVE-POINTERS (ADD1-NAT NAT)
                                 (SUB1 N))))
```

DEFINITION.
```
(NEW-PROC-NAME X PROC-LIST)
   =
(AND (OK-MG-NAMEP X)
     (NOT (DEFINED-PROCP X PROC-LIST)))
```

DEFINITION.
```
(NO-P-ALIASING BINDINGS ALIST)
   =
(NO-DUPLICATES (COLLECT-POINTERS BINDINGS ALIST))
```

DEFINITION.
```
(NULLIFY CINFO)
   =
(MAKE-CINFO NIL
            (LABEL-ALIST CINFO)
            (LABEL-CNT CINFO))
```

DEFINITION.
```
(OK-CINFOP CINFO)
   =
(PLISTP (CODE CINFO))
```

DEFINITION.
```
(OK-COND-LIST LST)
   =
(IF (NLISTP LST)
    (EQUAL LST NIL)
    (AND (OR (OK-MG-NAMEP (CAR LST))
             (MEMBER (CAR LST)
                     '(LEAVE ROUTINEERROR)))
         (OK-COND-LIST (CDR LST))))
```

DEFINITION.
```
(OK-EXECUTION-ENVIRONMENT STMT COND-LIST PROC-LIST MG-STATE SUBR N)
   =
(AND (OK-MG-STATEMENT STMT COND-LIST (MG-ALIST MG-STATE) PROC-LIST)
     (OK-MG-DEF-PLISTP PROC-LIST)
     (OK-MG-STATEP MG-STATE COND-LIST)
     (IDENTIFIER-PLISTP COND-LIST)
     (ALL-CARS-UNIQUE (MG-ALIST MG-STATE))
     (NEW-PROC-NAME SUBR PROC-LIST)
     (LESSP (LENGTH COND-LIST) (SUB1 (SUB1 (SUB1 (EXP 2 (P-WORD-SIZE)))))))
```

DEFINITION.
```
(OK-TEMP-STK-ARRAY-INDEX NAT TEMP-STK LNGTH)
   =
(AND (LENGTH-PLISTP NAT 2)
     (EQUAL (TYPE NAT) 'NAT)
     (NUMBERP (UNTAG NAT))
     (LESSP (PLUS (UNTAG NAT) (SUB1 LNGTH))
            (LENGTH TEMP-STK)))
```

DEFINITION.
```
(OK-TEMP-STK-INDEX NAT TEMP-STK)
   =
(AND (LENGTH-PLISTP NAT 2)
     (EQUAL (TYPE NAT) 'NAT)
     (NUMBERP (UNTAG NAT))
     (LESSP (UNTAG NAT)
            (LENGTH TEMP-STK)))
```

DEFINITION.
```
(OK-TRANSLATION-PARAMETERS CINFO COND-LIST STMT PROC-LIST Y)
   =
(AND (OK-CINFOP CINFO)
     (OK-COND-LIST COND-LIST)
     (LABEL-HOLE-BIG-ENOUGH CINFO COND-LIST STMT PROC-LIST Y))
```

DEFINITION.
```
(P-TO-MG-SIMPLE-LITERAL LIT TYPESPEC)
   =
(CASE TYPESPEC
      (INT-MG (LIST 'INT-MG (CADR LIT)))
      (BOOLEAN-MG (LIST 'BOOLEAN-MG
                        (IF (EQUAL (CADR LIT) 'F)
                            'FALSE-MG
                            'TRUE-MG)))
      (OTHERWISE (LIST 'CHARACTER-MG (CADR LIT))))
```

DEFINITION.
```
(P-TO-MG-SIMPLE-LITERAL-LIST LST TYPESPEC)
   =
(IF (NLISTP LST)
    NIL
    (CONS (P-TO-MG-SIMPLE-LITERAL (CAR LST)
                                  TYPESPEC)
          (P-TO-MG-SIMPLE-LITERAL-LIST (CDR LST)
                                       TYPESPEC)))
```

DEFINITION.
```
(PITON-CC P)
   =
(FETCH-ADP '(C-C . 0)
           (P-DATA-SEGMENT P))
```

DEFINITION.
```
(PREDEFINED-PROC-CALL-CLOCK STMT MG-STATE)
   =
(PLUS (CLOCK-PREDEFINED-PROC-CALL-SEQUENCE (CALL-NAME STMT))
      (CLOCK-PREDEFINED-PROC-CALL-BODY-TRANSLATION STMT MG-STATE))
```

Chapter 7
PROOF OF THE CORRECTNESS THEOREM

In Chapter 6 we described the interpreter equivalence theorem which asserts the correctness of our translation from Micro-Gypsy to Piton. The formal proof of this theorem is embodied in the script of events in Appendix B. The current chapter contains an informal overview of some of the main milestones in that proof.

Approximately 90% of the effort expended in our entire proof was directed toward the formulation and proof of a particular lemma, the **`EXACT-TIME-LEMMA`**. We discuss the proof of this lemma and show how our main theorem follows from it.

7.1 EXACT-TIME-LEMMA

It often happens that a conjecture of interest is too specific to be proven directly, yet can be proved as a corollary of a more general fact which subsumes it. This may be true because the specific conjecture is unsuitable for proof by induction. Our main theorem **`TRANSLATION-IS-CORRECT5`** is in exactly this class.

Recall from Chapters 5 and 6 that our main theorem involves the function **`MAP-DOWN1`** which describes the implementation of a Micro-Gypsy execution environment in Piton. **`MAP-DOWN1`** encapsulates our Micro-Gypsy entry point into a special procedure; we then compile and execute the body of that procedure. One might expect that we could prove **`TRANSLATION-IS-CORRECT5`** by an induction on the structure of the entry point statement. However, the position of this statement within the overall Micro-Gypsy program text--as the body of a distinguished procedure--is not general enough to allow us to carry out the induction. We need to be able to describe the effects of interpreting an *arbitrary* Micro-Gypsy statement wherever it might appear in the list of procedures which comprise the Micro-Gypsy procedure list. If so, then we can describe as a special case the effects of the single statement which is the body of our entry point procedure. This more general result is the content of the theorem **`EXACT-TIME-LEMMA`**.

`EXACT-TIME-LEMMA` essentially gives us the commuting diagram illustrated in figure 7-1. **`EXACT-TIME-LEMMA`** tells us that the P-state returned by mapping an initial Micro-Gypsy state down to an initial P-state and then running the Piton interpreter is identical to the p-state obtained by mapping down the final Micro-Gypsy state. This result is formulated in a very general context.

The trick in formulating **`EXACT-TIME-LEMMA`** is to characterize the translation of a truly arbitrary Micro-Gypsy statement into Piton in a way that will be amenable to inductive proof. Consider the point in the Micro-Gypsy world just before the execution of an arbitrary Micro-Gypsy statement **`STMT`**. Given our discussion of the translation from Micro-Gypsy execution environments into Piton execution environments in Chapter 5, what can we say about the Piton state which correspond to this arbitrary

```
MG-State1  ------------------> MG-State2
    |                              |
   |Map-Down                     |Map-Down
    |                              |
    |                              |
    V                              V
P-initial -------------------> P-final
              P(P-initial, clock)
```

Figure 7-1: EXACT-TIME-LEMMA Effect

"snapshot" of a Micro-Gypsy execution? **EXACT-TIME-LEMMA** is our attempt to say as much as possible about this snapshot. The lemma is illustrated in figure 7-2.

The statement of **EXACT-TIME-LEMMA** involves the following variables with these intended interpretations:

- **STMT**: an arbitrary Micro-Gypsy statement;
- **R-COND-LIST**: the cond-list used by the recognizer in recognizing **STMT**;
- **T-COND-LIST**: the cond-list used by the translator for converting conditions into Piton naturals;
- **NAME-ALIST**: the list of **<NAME, TYPE>** pairs used by the recognizer;
- **PROC-LIST**: the list of Micro-Gypsy user-defined procedures;
- **CINFO**: the **<CODE, LABEL-CNT, LABEL-ALIST>** triple used in translating **STMT**;
- **CODE2**: a list of Piton instructions;
- **SUBR**: a Micro-Gypsy procedure name;
- **TEMP-STK**: a Piton temp-stk;
- **CTRL-STK**: a Piton ctrl-stk;
- **MG-STATE**: a **<CC, MG-ALIST, MG-PSW>** triple as described in Section 3.4.1-A;
- **N**: the ubiquitous clock.

7.1.1 Hypotheses of the EXACT-TIME-LEMMA

Our goal is to characterize the behavior of the translation of an arbitrary Micro-Gypsy statement **STMT**. This is done to a large extent in the hypotheses of **EXACT-TIME-LEMMA**. Since **EXACT-TIME-LEMMA** is really the heart of the proof, we consider each of the hypotheses in some detail. For the readers' convenience, we have numbered the hypotheses in the comment field in figure 7-2; it is to these numbers that we refer in the following discussion.

Hypotheses 1-3 insure that the various Micro-Gypsy language structures are accepted by the recognizer.

1. **STMT** is a Micro-Gypsy statement legal in the context of the **R-COND-LIST**, **NAME-ALIST**, and **PROC-LIST** (see Section 3.3.3).
2. **PROC-LIST** is a legal list of Micro-Gypsy user-defined procedures (see Section 3.3.4).
3. **SUBR** is the name of one of the user-defined procedures in the **PROC-LIST**.

Our assumption is that **STMT** appears somewhere within the body of procedure **SUBR**. From 2 and 3 we can infer that it is legal to fetch procedure **SUBR** from the list and thereby obtain a legal Micro-Gypsy procedure and that therefore **STMT** must occur in a legitimate statement context.

Theorem. EXACT-TIME-LEMMA

```
(IMPLIES
 (AND (OK-MG-STATEMENT STMT R-COND-LIST NAME-ALIST PROC-LIST)          ; 1
      (OK-MG-DEF-PLISTP PROC-LIST)                                     ; 2
      (USER-DEFINED-PROCP SUBR PROC-LIST)                              ; 3
      (OK-MG-STATEP MG-STATE R-COND-LIST)                              ; 4
      (NORMAL MG-STATE)                                                ; 5
      (ALL-CARS-UNIQUE (MG-ALIST MG-STATE))                            ; 6
      (COND-SUBSETP R-COND-LIST T-COND-LIST)                           ; 7
      (SIGNATURES-MATCH (MG-ALIST MG-STATE) NAME-ALIST)                ; 8
      (EQUAL (CODE (TRANSLATE-DEF-BODY
                    (ASSOC SUBR PROC-LIST) PROC-LIST))                 ; 9
             (APPEND (CODE (TRANSLATE CINFO T-COND-LIST STMT PROC-LIST))
                     CODE2))
      (OK-TRANSLATION-PARAMETERS CINFO T-COND-LIST
                                 STMT PROC-LIST CODE2)                ; 10
      (PLISTP TEMP-STK)                                               ; 11
      (LISTP CTRL-STK)                                                ; 12
      (MG-VARS-LIST-OK-IN-P-STATE (MG-ALIST MG-STATE)                 ; 13
                                  (BINDINGS (TOP CTRL-STK))
                                  TEMP-STK)
      (NO-P-ALIASING (BINDINGS (TOP CTRL-STK))                        ; 14
                     (MG-ALIST MG-STATE))
      (NOT (RESOURCE-ERRORP
            (MG-MEANING-R STMT PROC-LIST MG-STATE N                   ; 15
                          (LIST (LENGTH TEMP-STK)
                                (P-CTRL-STK-SIZE CTRL-STK)))))))
 (EQUAL
  (P (MAP-DOWN MG-STATE PROC-LIST CTRL-STK TEMP-STK
               (TAG 'PC (CONS SUBR (LENGTH (CODE CINFO))))
               T-COND-LIST)
     (CLOCK STMT PROC-LIST MG-STATE N))
  (MAP-DOWN
   (MG-MEANING-R STMT PROC-LIST MG-STATE N
                 (LIST (LENGTH TEMP-STK)
                       (P-CTRL-STK-SIZE CTRL-STK)))
   PROC-LIST CTRL-STK TEMP-STK
   (TAG 'PC
        (CONS SUBR
              (IF
               (NORMAL (MG-MEANING-R STMT PROC-LIST MG-STATE N
                                     (LIST (LENGTH TEMP-STK)
                                           (P-CTRL-STK-SIZE CTRL-STK))))
               (LENGTH (CODE (TRANSLATE CINFO T-COND-LIST STMT PROC-LIST)))
               (FIND-LABEL
                (FETCH-LABEL
                 (CC (MG-MEANING-R STMT PROC-LIST MG-STATE N
                                   (LIST (LENGTH TEMP-STK)
                                         (P-CTRL-STK-SIZE CTRL-STK))))
                 (LABEL-ALIST CINFO))
                (APPEND (CODE (TRANSLATE CINFO T-COND-LIST STMT PROC-LIST))
                        CODE2)))))
   T-COND-LIST)))
```

Figure 7-2: EXACT-TIME-LEMMA

Hypotheses 4-6 establish that **MG-STATE** is a legal Micro-Gypsy state in which to interpret a statement.

4. **MG-STATE** is a **<CC, MG-ALIST, MG-PSW>** triple, with **CC** a condition legal with respect to **R-COND-LIST** and **MG-ALIST** a legal Micro-Gypsy variable alist.
5. **(CC MG-ALIST)** is **'NORMAL**.
6. The variable names in **MG-ALIST** are all unique.

The next two hypotheses relate the recognizer context to the interpreter context.

7. Each member of the **R-COND-LIST** is either a member of the **T-COND-LIST** or is **'LEAVE** or **'ROUTINEERROR**.
8. The **MG-ALIST** is a list of triples **<NAME, TYPE, VALUE>**; the **NAME-ALIST** contains pairs **<NAME, TYPE>**. Discounting the **VALUE** components, the lists should match.

In the main theorem **TRANSLATION-IS-CORRECT5**, there is only one condition list and the **MG-ALIST** is used as the recognizer's name-alist. We need distinct structures here because of the generality of the current theorem. For example, when recognizing the body of a **LOOP-MG** statement, the recognizer name-alist is of the form **(CONS 'LEAVE LST)** to indicate that **'LEAVE** can be signaled within the loop body. In translating, however, there is no need to add **'LEAVE** to the translator **COND-LIST** since **'LEAVE** is handled specially. Hypothesis 7 merely formalizes the relationship that must hold between the condition lists used by the recognizer and translator with respect to a given statement. Hypothesis 8 merely says that the list of variables and associated types assumed by the recognizer and by the translator are consistent.

9. **STMT** appears within the Micro-Gypsy procedure **SUBR** and, consequently, that the translation of **STMT** lies within the body of the translation of **SUBR**.

The statement of hypothesis 9 is somewhat subtle. Consider the following Micro-Gypsy procedure and its Piton translation.

```
      Micro-Gypsy procedure           Piton procedure

 (subr formals formal-conds          (subr
       locals local-conds             formals nil
       (...                           (code CINFO)
        STMT                          <translation of STMT>
        ...                           <CODE2>
            ))                                )
```

Assuming that **CINFO** is the context in which **STMT** is translated, the body of the Piton procedure is equal to

```
(APPEND (CODE CINFO)
        (APPEND (TRANSLATE CINFO STMT T-COND-LIST PROC-LIST)
                CODE2))
```

where **CODE2** is some list of Piton instructions. Now, this Piton procedure body should be equal to the translation of the body of the Micro-Gypsy user-defined procedure **SUBR**. This equality is the content of Hypothesis 9.

10. The translation is carried out in a legitimate translation context. **(CODE CINFO)** is a proper list; **T-COND-LIST** is a list of legal condition names; sandwiching **(TRANSLATE CINFO STMT T-COND-LIST PROC-LIST)** between **(CODE CINFO)** and **CODE2** yields a list of Piton instructions in which all labels are unique.

The Piton temp-stk and ctrl-stk are variables in **EXACT-TIME-LEMMA**. They are assumed to satisfy certain constraints formalized in hypotheses 11-14.

11. **TEMP-STK** is a proper list. This is a purely technical requirement.

12. **CTRL-STK** must contain at least one frame. We see why below.
13. Each variable **V** on the **MG-ALIST** must be represented by a pair **<V, NAT>** in the bindings component of the topmost frame on **CTRL-STK**. **NAT** must be a potential index into the temp-stk for **V**. If **V** is the name of an array, there must be space on the temp-stk starting at location **NAT** for an array of that length.
14. If the structures on **MG-ALIST** are placed onto the temp-stk using the bindings on **CTRL-STK**, there will be no overlapping.

Notice that 13 and 14 say that the bindings component of the topmost frame on the ctrl-stk *could be* a legal *pointer-alist* (see Section 5.1.2 for the representation of the data from **MG-ALIST**). It must also be impossible to access a single temp-stk location using two different pointers from this pointer-alist.

Finally, we make our standard disclaimer that all bets are off in the event of a meta-level error condition.

15. The final **MG-PSW** of the run of our Micro-Gypsy interpreter is **'RUN**.

7.1.2 Conclusion of the EXACT-TIME-LEMMA

The conclusion of **EXACT-TIME-LEMMA** tells us what to expect if we map a Micro-Gypsy state down and run the Piton interpreter on the result. Notice that the function we use for mapping down is *not* the function **MAP-DOWN1** used in **TRANSLATION-IS-CORRECT5**. **MAP-DOWN1** is an instance of this more general function **MAP-DOWN**.

Definition.
```
(MAP-DOWN MG-STATE PROC-LIST CTRL-STK TEMP-STK ADDR COND-LIST)
  =
(P-STATE ADDR CTRL-STK
         (MAP-DOWN-VALUES (MG-ALIST MG-STATE)
                          (BINDINGS (TOP CTRL-STK))
                          TEMP-STK)
         (TRANSLATE-PROC-LIST PROC-LIST)
         (LIST (LIST 'C-C
                     (MG-COND-TO-P-NAT (CC MG-STATE)
                                       COND-LIST)))
         (MG-MAX-CTRL-STK-SIZE)
         (MG-MAX-TEMP-STK-SIZE)
         (MG-WORD-SIZE)
         'RUN))
```

MAP-DOWN creates a Piton state in which the Micro-Gypsy **MG-ALIST**, **PROC-LIST**, and **CC** are represented. The values of the Micro-Gypsy variables on the **MG-ALIST** are converted to Piton values and written into the temp-stk using the indices in the pointer-alist. Recall that hypotheses 13 and 14 assure that this is possible and that there will be no overlapping. The **CC** and **PROC-LIST** are translated and stored in the Piton state just as with **MAP-DOWN1** (see Section 5.6).

We are interested in the execution of the Piton translation of the Micro-Gypsy statement **STMT**. From our discussion of hypothesis 9 above, we see that the pc in the p-state must be the Piton address **(PC (SUBR . N))**, where **N** is the length of the code which precedes the translation of **STMT** in the body of the *Piton* routine **SUBR**, i.e., **(CODE CINFO)**. By the definition of **MAP-DOWN** we insure that all of the variables from the **MG-ALIST** are properly represented in the Piton state, with appropriate values in the Piton temp-stk and pointers in the bindings component of the topmost frame of the Piton ctrl-stk. The Piton interpreter is run for exactly the same number of steps as in the main theorem.

The right hand side of the conclusion tells us the form of the final p-state explicitly. This should be exactly the p-state which would be returned, mapping the final values of the Micro-Gypsy data structures into a Piton state with the function **MAP-DOWN**. The only components of the final piton state which bear comment are the values of the Piton **P-PC**, **P-TEMP-STK**, **P-CTRL-STK**, and **P-DATA-SEGMENT**.

P-PC has one of two possible values. If the interpreter returns normally on **STMT**, then the Piton pc should be at the next statement following the translation of **STMT**. If some condition is raised in the interpretation of **STMT**, the pc should point to the label associated with that condition in the **LABEL-ALIST** of the translator. Both of these must by within the current procedure.

CTRL-STK is unchanged. Any frames pushed during execution have been popped off.

The initial p-state has the values of the initial Micro-Gypsy **MG-ALIST** stored on the temp-stk. The final p-state should have the **P-TEMP-STK** which would result from storing the *final* Micro-Gypsy **MG-ALIST**.

The **P-DATA-SEGMENT** contains only the Piton natural representing the value of the Micro-Gypsy **CC**. The final value must contain the translation of the final value of the **CC**.

7.1.3 Proof of the EXACT-TIME-LEMMA

The proof of **EXACT-TIME-LEMMA** is a very complicated induction on the structure of the Micro-Gypsy statement being interpreted/translated. The Boyer-Moore prover allows the user to suggest an induction schema. For the **EXACT-TIME-LEMMA** this involves suggesting an induction which treats every one of the recursive calls in just the right way.

Consider the **PROG2-MG** statement for example. The version of the **EXACT-TIME-LEMMA** commuting diagram for **PROG2-MG** is really the composition of the commuting diagrams for the **PROG2-LEFT-BRANCH** and **PROG2-RIGHT-BRANCH**. The induction hint is formulated so that there will be two inductive hypotheses

```
                prog2-left                     prog2-right

MG-initial ----------> state1    state2-------------> MG-final
    |                    |         |                      |
    |                    |         |                      |
    |                    |         |                      |
    |                    |         |                      |
    |                    |         |                      |
    V                    V         V                      V
 P-initial ----------> P-state1  P-state2------------> P-final
```

Figure 7-3: EXACT-TIME-LEMMA PROG2-MG Case

which exactly characterize the two halves of the commuting diagram in such a way that they fit together to yield the desired result. For example it must be the case that **STATE1** and **STATE2** are equal if **PROG2-LEFT** returns normally; otherwise **STATE1** and **MG-FINAL** must be identical. The inductive subcase of **EXACT-TIME-LEMMA** for **PROG2-MG** is isolated in the lemma **EXACT-TIME-LEMMA-PROG2-CASE** (see Appendix B) The reader is invited to examine this lemma to see an example of (the simplest of) the inductive cases. The reader who is concerned with the inductive structure of the proof should study the induction hint given in the form of the definition **EXACT-TIME-INDUCTION-HINT** (see Appendix B).[40]

[40]Be forewarned, however, that the induction hint has 12 parameters and is over 250 lines long.

7.2 The Proof of the Main Theorem

Recall from Chapter 2 our comment that we can infer the interpreter equivalence theorem which corresponds to a diagram

```
SL-initial  -----------------> SL-final
   |                              ^
   |                              |
Map-Down                       Map-Up                    (1)
   |                              |
   |                              |
   V                              |
TL-initial ------------------> TL-final
```

if we have the theorem corresponding to the diagram

```
SL-initial  -----------------> SL-final
   |                              |
   |                              |
Map-Down                       Map-Down                  (2)
   |                              |
   |                              |
   V                              V
TL-initial ------------------> TL-final
```

provided that we know that **MAP-UP** is a left-inverse for **MAP-DOWN**. We have noted that **EXACT-TIME-LEMMA** has a commuting diagram in the form of *(2)* and **TRANSLATION-IS-CORRECT5** a commuting diagram like *(1)*.

We finish the proof of **TRANSLATION-IS-CORRECT5** by proving the formal analogues of the following observations.

1. The **MAP-DOWN1** function used in **TRANSLATION-IS-CORRECT5** is an instance of the **MAP-DOWN** function used in **EXACT-TIME-LEMMA**. Moreover, it is an instance which allows us to satisfy the hypotheses of **EXACT-TIME-LEMMA**. For example, **MAP-DOWN1** creates an initial temp-stk and ctrl-stk in such a way that the ctrl-stk contains a pointer-alist appropriate for the **MG-ALIST** values stored on the initial temp-stk as required by hypothesis 13 of **EXACT-TIME-LEMMA**.
2. **MAP-UP** is an appropriate left inverse for **MAP-DOWN** under certain conditions. This is the content of the lemma **MAP-UP-INVERTS-MAP-DOWN** (see Appendix B). Moreover, these conditions are all satisfied in the context of the hypotheses of **TRANSLATION-IS-CORRECT5**.

Chapter 8
PROOF OF A MICRO-GYPSY PROGRAM

In this chapter we display a simple Micro-Gypsy program and shows how it can be proved in either of two ways.

1. We show how a Micro-Gypsy program can be verified using the GVE, translated to a program in our syntax, and then compiled.
2. We prove the program directly against the interpreter semantics described in Chapter 3.

We compare these two approaches and discuss the advantages and disadvantages of each.

8.1 A Simple Micro-Gypsy Program

Multiplication is a primitive operation in Gypsy but not in our simple Micro-Gypsy subset. Suppose that we wish to make multiplication available. We can do this by extending the language. Define a new predefined procedure, say, **MG-INTEGER-MULTIPLY** and go to the work of re-proving the correctness of the compiler for this extended subset. Alternatively, the current language is strong enough to allow a programmer to write a Micro-Gypsy multiplication procedure.

An implementation using a simple iterated addition scheme in standard Gypsy syntax is illustrated in figure 8-1. The procedure **MULTIPLY_BY_POSITIVE** implements multiplication by a non-negative integer. This is then called by **MULTIPLY** with a simple case analysis to obtain the correct sign for the result.

Several points are worth noting. This version is annotated as it might be for proof in the Gypsy Verification Environment. The specification is weaker than it might be. In particular, the **exit** specification on **MULTIPLY** is[41]

```
EXIT ANS = I * J;
```

This is really an abbreviation for

```
EXIT CASE (IS NORMAL: ANS = I * J;
           IS ROUTINEERROR: TRUE);
```

indicating that the only cases of interest to the programmer are those in which no condition is raised. Such incomplete specifications are quite common for Gypsy programs. Gypsy is quite flexible in allowing the programmer to specify the program in more or less detail. We deliberately chose an incomplete specification to see how this is reflected in our two proofs.

---

[41]Note that though multiplication is not available in the executable portion of Micro-Gypsy, it is available in the specification language. See the discussion of this point in Chapter 3.

```
scope MULTIPLICATION_ROUTINES =
begin

   const MININT := -2147483648;

   const MAXINT := 2147483647;

   type INT = integer [MININT .. MAXINT];

   procedure MULTIPLY (var ANS: INT;
                       I, J: INT) =
   begin
      exit ANS = I * J;
      if J ge 0
         then MULTIPLY_BY_POSITIVE (ANS, I, J)
         else MULTIPLY_BY_POSITIVE (ANS, I, -J);
              ANS := - ANS
      end; {if}
   end; {multiply}

   procedure MULTIPLY_BY_POSITIVE (var ANS: INT;
                                   I, J: INT) =
   begin
      entry J ge 0;
      exit  ANS = I * J;
      var K: INT := 0;
      K := J;
      ANS := 0;
      loop
         assert J ge 0
                & K in [0 .. j]
                & ANS = (J-K) * I;
         if K le 0 then leave end;
         ANS := ANS + I;
         K := K - 1;
      end;
   end; {multiply_by_positive}

end; {scope multiplication_routines}
```

Figure 8-1: A Micro-Gypsy Multiplication Routine

The program illustrated above is easily translated into our Micro-Gypsy abstract prefix notation to yield the two procedures shown in figure 8-2. This translation was carried out by hand, but could be handled fairly simply by a preprocessor as discussed in Chapter 3. You may recognize `MULTIPLY_BY_POSITIVE` as an example we used in Chapter 5 to illustrate the translation process; the Piton translation was given in figure 5-4.

8.2 Verifying the Multiplication Routine

We would like to verify our multiplication program with respect to the (incomplete) specification given in the **exit** assertion on the `MULTIPLY` routine. That is, *in the absence of errors* `MULTIPLY` computes a product and stores the result in its first argument.

8.2.1 The GVE Approach

One first approach is to prove the program given in figure 8-1 using the Gypsy Verification Environment [GoodDivitoSmith 88]. Verification conditions sufficient to guarantee the conformance of the program with its specification are generated and these are then proven interactively using the GVE proof checker. For the routine `MULTIPLY_BY_POSITIVE` four verification conditions (vc's) are generated, two of which are proven automatically by the vc generator. The remaining two vc's are show below.

```
VERIFICATION CONDITION MULTIPLY_BY_POSITIVE#3
  H1: ANS + I * K = I * J
  H2: ANS + I IN [MININT..MAXINT]
  H3: K - 1 IN [MININT..MAXINT]
  H4: K IN [0..J]
  H5: 1 LE K
  H6: 0 LE J
 -->
  C1: K - 1 IN [0..J]

VERIFICATION CONDITION MULTIPLY_BY_POSITIVE#4
  H1: ANS + I * K = I * J
  H2: K IN [0..J]
  H3: 0 LE - K
  H4: 0 LE J
 -->
  C1: ANS = I * J
```

These are proven very easily in the proof checker. For `MULTIPLY` only a single non-trivial vc is generated.

```
VERIFICATION CONDITION MULTIPLY#1
  H1:    J IN [MININT..MAXINT]
      -> - J IN [MININT..MAXINT]
  H2: J + 1 LE 0
 -->
  C1: 0 LE - J
```

This again is proven quite simply in the proof checker. The entire proof of the multiplication routine, from starting up the GVE to the completion of the proof of `MULTIPLY` took approximately 10 minutes.

Following this proof, we translate our verified program into the abstract prefix syntax either by hand or using a preprocessor. The resulting program is then compiled as described in Chapter 5 and executed with the Piton interpreter or further translated into code for the FM8502 as described in the Piton manual [Moore 88].

```
(MG_MULTIPLY
  ((ANS  INT-MG)
    (I    INT-MG)
    (J    INT-MG))
   NIL
   ((K    INT-MG    (INT-MG 0))
    (ZERO INT-MG    (INT-MG 0))
    (B  BOOLEAN-MG  (BOOLEAN-MG FALSE-MG)))
   NIL
   (PROG2-MG
    (PREDEFINED-PROC-CALL-MG MG-INTEGER-LE (B ZERO J))
    (IF-MG B
      (PROC-CALL-MG MG_MULTIPLY_BY_POSITIVE (ANS I J) NIL)
      (PROG2-MG
       (PREDEFINED-PROC-CALL-MG MG-INTEGER-UNARY-MINUS (K J))
       (PROG2-MG
        (PROC-CALL-MG MG_MULTIPLY_BY_POSITIVE (ANS I K) NIL)
        (PREDEFINED-PROC-CALL-MG
         MG-INTEGER-UNARY-MINUS (ANS ANS)))))))


(MG_MULTIPLY_BY_POSITIVE
  ((ANS INT-MG)
   (I   INT-MG)
   (J   INT-MG))
  NIL
  ((K    INT-MG     (INT-MG 0))
   (ZERO INT-MG     (INT-MG 0))
   (ONE  INT-MG     (INT-MG 1))
   (B    BOOLEAN-MG (BOOLEAN-MG FALSE-MG)))
  NIL
  (PROG2-MG
   (PREDEFINED-PROC-CALL-MG
           MG-SIMPLE-CONSTANT-ASSIGNMENT (ANS (INT-MG 0)))
   (PROG2-MG
    (PREDEFINED-PROC-CALL-MG
            MG-SIMPLE-VARIABLE-ASSIGNMENT (K J))
    (LOOP-MG
     (PROG2-MG
      (PREDEFINED-PROC-CALL-MG MG-INTEGER-LE (B K ZERO))
      (PROG2-MG
       (IF-MG B (SIGNAL-MG LEAVE) (NO-OP-MG))
        (PROG2-MG
         (PREDEFINED-PROC-CALL-MG MG-INTEGER-ADD (ANS ANS I))
         (PREDEFINED-PROC-CALL-MG
               MG-INTEGER-SUBTRACT (K K ONE)))))))))
```

Figure 8-2: The Multiplication Routine in Abstract Prefix

8.2.2 Verifying Against the MG-MEANING Semantics

Using the GVE means that the verification of a program is carried out with respect to the Gypsy semantics assumed in the verification condition generator. The verification conditions are appropriate for that version of the Gypsy semantics and the proof is valid if the underlying logic is soundly implemented in the GVE proof checker. We can avoid these assumptions by verifying our Micro-Gypsy program directly with respect to the semantics defined by the function **`MG-MEANING`**.

Consider again the procedure **`MULTIPLY_BY_POSITIVE`** and its specification. The correctness of this procedure can be formalized as in the lemma **`MG-MULTIPLY-BY-POSITIVE-CORRECTNESS`** in figure 8-3.

The conclusion asserts that a call of the form

```
'(PROC-CALL-MG MG_MULTIPLY_BY_POSITIVE (ANS X Y) NIL)
```

has the effect of setting **`ANS`** to the Micro-Gypsy representation of the integer product of the values of integer variables **`X`** and **`Y`** in the current state.

The hypotheses of **`MG-MULTIPLY-BY-POSITIVE-CORRECTNESS`** permit the following assumptions.

1. Our call statement is a legal Micro-Gypsy statement. Hence, the actual parameter lists are appropriate for the formal lists in the definition of **`MG_MULTIPLY_BY_POSITIVE`**.
2. The state in which we are interpreting is well-formed. In particular, the integer variable actuals have integer literal values in the **`MG-ALIST`**.
3. The **`CC`** is **`'NORMAL`**.
4. The definition of **`MG_MULTIPLY_BY_POSITIVE`** in the procedure list is exactly that given in figure 8-2.
5. The clock parameter **`N`** to the interpreter is adequate to carry out the multiplication without timing out. Calculating an appropriate lower bound is one of the most difficult aspects of formulating this theorem.
6. The multiplicand **`Y`** has a non-negative value. This corresponds to the entry specification **`entry j ge 0`** of **`MULTIPLY_BY_POSITIVE`**.
7. The product of **`X`** and **`Y`** is representable as a Micro-Gypsy **`INT-MG`** value and can be stored in the variable **`ANS`**.[42]

The lemma **`MG-MULTIPLY-BY-POSITIVE-CORRECTNESS`** has been proven in the Kaufmann-enhanced Boyer-Moore theorem prover. Refining the lemma, proving the supporting lemmas, and completing the proof took approximately two days of fairly intense effort.

8.2.3 Comparing the Two Approaches

Each of the two approaches to proving Micro-Gypsy programs has advantages and disadvantages. We have published elsewhere [KaufmannYoung 87] a general discussion comparing Gypsy and Boyer-Moore style specifications and proofs and most of those remarks apply here. Several points are particularly noteworthy in the current context.

The annotated Gypsy procedures in figure 8-1 define a collection of *program theorems*.

---

[42]As an aside, it follows from this that each of the intermediate results computed by **`MG_MULTIPLY_BY_POSITIVE`** is representable, and hence that no condition is raised if the final answer is in the right range.

```
THEOREM.  MG-MULTIPLY-BY-POSITIVE-CORRECTNESS
(IMPLIES
 (AND
  (OK-MG-STATEMENT                                                                    ; 1
   '(PROC-CALL-MG MG_MULTIPLY_BY_POSITIVE (ANS X Y) NIL)
   COND-LIST (MG-ALIST MG-STATE) PROC-LIST)
  (OK-MG-STATEP MG-STATE NIL)                                                         ; 2
  (NORMAL MG-STATE)                                                                   ; 3
  (EQUAL                                                                              ; 4
   (FETCH-DEF 'MG_MULTIPLY_BY_POSITIVE PROC-LIST)
   '(MG_MULTIPLY_BY_POSITIVE
     ((ANS INT-MG)
      (I   INT-MG)
      (J   INT-MG))
     NIL
     ((K    INT-MG     (INT-MG 0))
      (ZERO INT-MG     (INT-MG 0))
      (ONE  INT-MG     (INT-MG 1))
      (B    BOOLEAN-MG (BOOLEAN-MG FALSE-MG)))
     NIL
     (PROG2-MG
      (PREDEFINED-PROC-CALL-MG
            MG-SIMPLE-CONSTANT-ASSIGNMENT (ANS (INT-MG 0)))
      (PROG2-MG
       (PREDEFINED-PROC-CALL-MG
        MG-SIMPLE-VARIABLE-ASSIGNMENT (K J))
       (LOOP-MG
        (PROG2-MG
         (PREDEFINED-PROC-CALL-MG MG-INTEGER-LE (B K ZERO))
         (PROG2-MG
          (IF-MG B (SIGNAL-MG LEAVE) (NO-OP-MG))
          (PROG2-MG
           (PREDEFINED-PROC-CALL-MG MG-INTEGER-ADD (ANS ANS I))
           (PREDEFINED-PROC-CALL-MG
                   MG-INTEGER-SUBTRACT (K K ONE)))))))))))
  (LESSP (PLUS 4                                                                      ; 5
           (TIMES 4 (ADD1 (UNTAG (GET-M-VALUE
                                  'Y (MG-ALIST MG-STATE))))))
         N)
  (NUMBERP (UNTAG (GET-M-VALUE 'Y (MG-ALIST MG-STATE))))                              ; 6
  (SMALL-INTEGERP                                                                     ; 7
     (ITIMES (UNTAG (GET-M-VALUE 'Y (MG-ALIST MG-STATE)))
             (UNTAG (GET-M-VALUE 'X (MG-ALIST MG-STATE))))
     (MG-WORD-SIZE)))

 (EQUAL
  (MG-MEANING '(PROC-CALL-MG MG_MULTIPLY_BY_POSITIVE
                             (ANS X Y) NIL)
              PROC-LIST MG-STATE N)
  (MG-STATE
     'NORMAL
     (SET-ALIST-VALUE
      'ANS
      (TAG 'INT-MG
           (ITIMES (UNTAG (GET-M-VALUE 'Y (MG-ALIST MG-STATE)))
                   (UNTAG (GET-M-VALUE 'X (MG-ALIST MG-STATE)))))
      (MG-ALIST MG-STATE))
     (MG-PSW MG-STATE))))
```

Figure 8-3: MG-MULTIPLY-BY-POSITIVE-CORRECTNESS

MG-MULTIPLY-BY-POSITIVE-CORRECTNESS is a formalization of one of these program theorems in the Boyer-Moore framework. From the remarks above concerning the time required in the two versions of the proof, it should be obvious that a significant amount of intellectual effort is expended in translating the Gypsy program theorem into the Boyer-Moore framework and proving the resulting theorem. This process is as complicated as it is because much of the work of transforming a program theorem into a theorem in the Boyer-Moore logic is exposed. Much of the mechanism made transparent by the GVE is explicit in the Boyer-Moore version.

1. The *database* functions handled automatically by the GVE must be made explicit in the Boyer-Moore version. There is no need in the Gypsy program theorem to refer to fetching the procedure definition. There is no reason to make explicit the mechanism for retrieving and storing data values.
2. Assumptions derived from the parse are maintained by the system rather than inserted by the user into his theorem.
3. The process of vc generation is carried out by the GVE rather than by the user. In particular, the analysis of program paths is handled automatically.

The GVE proof of **MULTIPLY_BY_POSITIVE** shows *partial* correctness; the proof might still succeed in cases where the routine was non-terminating. Gypsy, in fact, has mechanisms designed for specifying non-terminating programs. Our proofs in the Boyer-Moore framework are easier if we require totality and perform the awkward computation of an adequate clock value.[43]

The **MULTIPLY_BY_POSITIVE** **exit** specification says (implicitly) that the programmer is not concerned with cases in which conditions are signaled. In the GVE proof, paths corresponding to such cases are handled by the vc generator and generate trivial vc's. In the Boyer-Moore formalism, it is necessary to characterize explicitly the situation in which conditions will not arise. Thus, the Boyer-Moore specification is more complete but also more complex.

Despite the apparent extra effort involved, proving the program with respect to the semantics provided by **MG-MEANING** has three very strong advantages over using the GVE to prove Micro-Gypsy programs.

1. The semantics assumed in the proof is exactly the semantics assumed in the compiler. Using the GVE involves the assumption that the Gypsy semantics embodied in the verification condition generator and the GVE prover's algebraic simplifier is the same as the semantics formalized in **MG-MEANING**. Care has been taken to assure that this is the case but there is no formal assurance.
2. The GVE accepts programs in standard Gypsy syntax. Compiling these programs with our verified translator means a error-prone hand translation to our abstract prefix form. Mechanizing this process helps but it will be some time before we have a verified preprocessor.
3. Soundness of the logic and the care with which it is implemented in the theorem prover are strong advantages of the Boyer-Moore proof system over the GVE. There is empirical evidence over 15 years for virtually bug-free performance of the Boyer-Moore prover that has not been matched by the Gypsy implementation.

Some of the benefits of a GVE-style proof could be gained while retaining these advantages by writing a verification condition generator for Micro-Gypsy within the Boyer-Moore framework. This a separate research topic which we have not investigated in detail, though some research has been aimed in this

[43]This is not strictly necessary; we could have merely given as a hypothesis that the final psw was not **'TIMED-OUT**. This is more troublesome because we need to continually draw the inference that if the final result isn't timed out, then there must be enough "clock" ticks left to perform the current step, *i.e.*, that the current step doesn't time out.

direction [Ragland 73].

8.3 Applying the Correctness Result to the Multiplication Routine

Suppose that we are interested in running our multiplication routine in our Piton implementation of Micro-Gypsy. What does the correctness result tell us?

Suppose that the **'MG-STATE** is bound to some particular **MG-STATE** in which **ANS**, **X**, and **Y** are integer variables. Suppose also that **'PROC-LIST** is bound to the list of procedures in figure 5-3,. We can compute an appropriate Piton state by evaluating (in the interpreter for the Boyer-Moore logic) the following function call

```
(MAP-DOWN1 MG-STATE PROC-LIST NIL 'MAIN
           '(PROC-CALL-MG MG_MULTIPLY_BY_POSITIVE
                          (ANS X Y) NIL)).
```

The resulting p-state, which is quite large since it contains the code for all of the predefined operations, is the one which we would like to now run on Piton to perform the Micro-Gypsy analog of the Gypsy statement **ANS := X * Y**. However, to say that this p-state can be "run on the Piton machine" can mean either of two things:

1. We can run the Piton interpreter on this p-state for an appropriate number of steps and then look at the result.
2. We can run our program using the Piton implementation on FM8502.

The proof that we have given really only guarantees that we can do the first.

To show that we can do the second requires a bit more care. There are restrictions on Piton p-states which are acceptable to the FM8502 implementation. These are discussed fully in the chapter 5 of the Piton report [Moore 88]. Briefly, they are as follows.

1. The initial p-state must satisfy a number of syntactic constraints checked with the function **PROPER-P-STATEP**.
2. The p-state must be *loadable*. This involves an allocation of FM8502 resources to the stacks, programs, etc. of the p-state.
3. The word-size must be 32.
4. The final p-state must be non-erroneous.
5. The type specification of the final p-state must be known. Else it is not possible to map-up the final results from the FM8502 into the Piton world.

These are the constraints that guarantee that the implementation of Micro-Gypsy on Piton really "stacks" on top of the implementation of Piton on the FM8502.

We have not provided a proof that we satisfy each of these constraints. In fact, we currently *do not* satisfy them because of the following.

1. The p-state created by our **MAP-DOWN1** function has labels which are numbers; **PROPER-P-STATEP** requires that all labels be literal atoms. This could be easily corrected by changing any label *n* to a label of the form *L-n*.
2. We do not check in any way whether our programs are loadable. We do not consider that this is a problem that the programmer or the verifier need consider. We recognize that only a finite number of programs will be acceptable given the finite resources of the machine. This

is necessarily true of any real computation.

The final three constraints are not a problem for us. The Micro-Gypsy word size is defined to be 32. A hypothesis of our proof is that no errors occur in our Micro-Gypsy program execution. We prove that if no resource errors occur in the Micro-Gypsy program, none occur in the execution of the Piton implementation. Finally, we know the types of all data in the Micro-Gypsy state. These are reflected in the Piton state and are never changed by the execution.

One problem not made obvious by this discussion is that the proof of the Micro-Gypsy implementation assumes that the final values of variables are accessible on the final Piton temp-stk. The FM8502 implementation of Piton, however, guarantees only the final values in the Piton data-segment. This could be remedied by wrapping the final Micro-Gypsy program in code which copies the very final results from the temp-stk into the data-segment.

Chapter 9
CONCLUSIONS

In this chapter we review the research related to our project, comment on the significance of the Micro-Gypsy code generator proof, and describe some ways in which the work could be extended and enhanced.

9.1 Related Work

We have followed a long tradition in formally defining our languages in an operational style. Dijkstra [Dijkstra 62] formulated the basic feature of the operational style as follows: "A machine defines (by its very structure) a language, viz. its input language; conversely, the semantic definition of a language specifies a machine that understands it." Wegner [Wegner 72a] credits McCarthy [McCarthy 62] as the founder of the operational approach to defining the semantics of programming languages with his definitions of Lisp (by the APPLY function) and of MicroAlgol [McCarthy 66]. McCarthy's contribution was noticing that the meanings of programs could be expressed in terms of their effect upon the computational state.

A generalization of McCarthy's work to include explicit relations among state components evolved into a well-developed formalism for expressing operational semantics, the Vienna Definition Language. [Wegner 72b, Ollongren 74] VDL has been used for describing PL/I [LucasWalk 69], Basic [Lee 72], and a subset of SNOBOL4 [Pagan 78]. An alternative direction was the work of Landin [Landin 64, Landin 65] showing that Algol-like languages can be viewed as syntactic variations of the lambda calculus. His formalism was defined in terms of the abstract SECD machine.

A special case of operational semantics is interpreting the meaning of a program with respect to an actual compiler or interpreter rather than an abstract interpreter [Garwick 66]. A variant is Pagan's claim [Pagan 76] that interpreters might better be constructed using existing programming languages such as Algol 68.

It was realized quite early that interpreter definitions provided a way of investigating a variety of implementations. The notion was that interpreters for a single language but with quite different properties could be proven equivalent. Lucas [Lucas 68] and Henhapl and Jones [HenhaplJones 70] provide "the first example of a group of interpreter equivalence proofs which establish an equivalence class of significantly different, practically important interpreters" [Wegner 72a]. McGowan [McGowan 71] proves the equivalence of three interpreters for lambda calculus and Berry [Berry 71] proves the equivalence of two models of block structure semantics. McGowan [McGowan 72] outlines a theory of interpreter equivalence theorems very similar to those we discussed in chapter 2. These writers also recognized the applicability of these proof techniques to compilers.

The first attempt to prove compilation via an interpreter equivalence proof seems to be the proof of

McCarthy and Painter [McCarthyPainter 67] of a compiler for expression evaluation. They prove, by hand, the correctness of an expression compiler for an idealized machine using recursion induction. Burstall [Burstall 69] proves the expression compiler again using structural induction. Versions of the McCarthy-Painter proof have been mechanically proof-checked by Milner and Weyhrauch [MilnerWeyhrauch 72], Cartwright [Cartwright 76], and Aubin [Aubin 76]. Boyer and Moore [Boyer 79] have mechanically checked the proof of an optimizing expression compiler. Painter [Painter 67] proves an extension of the expression compiler which included assignments, conditional gotos, and I/O statements. Kaplan [Kaplan 67] proves a compiler for a simple language involving assignment and loops, using the proof technique of McCarthy and Painter. These are again employed by London [London 71, London 72] to prove an existing compiler for Lisp. London's work was the first applied to any language not contrived explicitly for proof purposes. Newey [Newey 75] proves a Lisp interpreter but was unable to complete the mechanical proof of one of London's Lisp compilers.

Recently, interpreter equivalence proofs have been used to prove the Piton assembler/compiler [Moore 88], an operating system kernel [Bevier 87], and a microprocessor definition [Hunt 85]. Interpreter equivalence style proofs have also been used for proofs of the VIPER micro-processor [Cohn 87].

A number of compiler proofs have been undertaken using semantic styles other than the operational style. These tend to characterize the proof in terms of commutative diagrams which are superficially similar to those described in Chapter 2. However, they differ in that the arms of the diagram which represent the source and target language semantics are characterized with axiomatic (Hoare-style) semantics or by denotational/algebraic semantics.

Several attempts have been made to use axiomatic semantics as a basis for compiler proofs. Notable are the work of Chirica and Martin [ChiricaMartin 75] and Lynn [Lynn 78]. Chirica and Martin use a Hoare-style semantics for a simple language with expressions, assignments, and loops. Lynn considers the proof of a compiler for a subset of Lisp including user-defined functions and uses Hoare-style axioms to specify the semantics of the source and target languages.

Much work in compiler verification has followed the trend in formal semantics toward algebraic or denotational descriptions of programming languages. It is argued that the denotational approach is abstract without being out of touch with the implementation. [Thatcher 81] Burge [Burge 68] proves a compiler for expressions of the lambda calculus and Blum [Blum 69] proves a compiler which takes recursive functions to Turing machine code. The work of Burstall and Landin [BurstallLandin 69] formalizes the semantics algebraically and proves the equivalence of the results of the functions describing the source and target languages. This apparently inspired the similar work of Morris [Morris 72, Morris 73] and Chirica [Chirica 76]. Milne and Strachey [MilneStrachey 76] give a hand proof of a compiler for a language of approximately the complexity of Algol 68. Cohn [Cohn 79a, Cohn 79b] proves several components of a compiler using Edinburgh LCF.

The intricacies of denotational semantics have been used in a variety of novel ways for generating parts of compilers. Mosses [Mosses 79] uses an algebraic style semantics to demonstrate the correct *construction* of a compiler. His approach is to define the compiler as the composition of the semantics and *Map-Down* functions. Continuation semantics are a denotional approach to dealing with input/output and complicated control structures. Wand [Wand 82] uses clever representations of continuation-style semantics to construct concrete semantics that are very close to machine language. Rashovsky [Rashovsky 82] also shows how to translate continuations into code. An approach to compiler correctness using interpretation between theories is outlined by Levy [Levy 85]. This seems to be simply a variant of the algebraic approach.

The most ambitious previous mechanical compiler proof has been the work of Polak [Polak 81]. He uses denotational semantics for describing both the source and target language. The source language is a fairly substantial subset of Pascal; the target language is an abstract and rather high-level assembly language. Polak treats all phases of the compiler including the translation of program text into abstract syntax trees (analogous to our input abstract prefix form). These abstract syntax trees are fed to code generation functions which are proven to be partially correct with respect to pre and post-conditions.

Polak’s goal seems to have been as much to develop a theory of compiler verification as to carry out a rigorous proof of a particular compiler. He develops a significant amount of theory by hand, though he asserts that the proof of the compiler functionality is entirely machine checked with the Stanford Pascal Verifier. His work differs from ours mainly in the following respects.

1. His style of semantic specification is denotational and, we believe, much less accessible than our operational style semantics.
2. His proof has as a basis a large collection of unproved assumptions within the formal theory. Boyer (in private communication) reports finding several inconsistencies in these axioms.
3. Polak’s work does not have the broader context of the verified lower level support. Consequently, if his target language were implemented, we might still remain uncertain that his assembly language programs were correctly implemented.

Chirica and Martin [ChiricaMartin 86] revised their earlier work in a way that was very close to Polak’s. They diverge from Polak mainly in drawing a different boundary between compiler specification and implementation.

Of the various work we have mentioned, the following used mechanical proof support: the proofs by Milner and Weyhrauch [MilnerWeyhrauch 72], Cartwright [Cartwright 76], Aubin [Aubin 76] and Boyer and Moore [Boyer 79] of McCarthy-Painter style expression compilers; proofs or partial proofs of compilers by Newey [Newey 75], Lynn [Lynn 78], Cohn [Cohn 79a, Cohn 79b], Polak [Polak 81] and Moore [Moore 88]; proofs of microprocessors by Hunt [Hunt 85] and Cohn [Cohn 87]; and the proof of an operating system kernel by Bevier [Bevier 87].

9.2 Comments and Summary

The work described in this dissertation has several components.

1. We have described a subset of the Gypsy 2.05 programming language suitable for reliable compilation. The language is characterized syntactically by a recognizer and given an operational semantics with an interpreter.
2. We have included a description of the Piton language verified by J Moore to be correctly implemented on the FM8502 verified microprocessor. The semantics of this language is also defined operationally by an interpreter.
3. We have shown how Micro-Gypsy execution environments can be translated into Piton execution environments and formalized the notion that this translation is correct with an interpreter equivalence theorem. We have outlined the proof of this theorem in the text and provided the complete proof in the form of event lists in the various chapters and in the appendix.
4. We have illustrated how Micro-Gypsy programs may be proven correct using the semantics we provide or by using the Gypsy Verification Environment.

9.2.1 Significance of the Project

The principle contributions of this project are two-fold. We have demonstrated the feasibility of providing a rigorous mechanically-checked proof of a code generator. We have also contributed to providing a framework for constructing highly reliable application programs.

9.2.1-A A Rigorous Proof

Micro-Gypsy is a small language but contains enough functionality to illustrate the viability of our approach. We feel that there is no conceptual difficulty in extending our work in various ways, some of which we outline in Section 9.2.2. Because Micro-Gypsy is a subset of an existing language, Gypsy 2.05, we were not free to tailor our source language to make the compilation process trivial. Because our target language was also pre-existing, we were not free to choose assembly language features which would narrow the semantic gap between source and target language.

We believe that the implementation is realistic and fairly efficient. In particular, though our Micro-Gypsy interpreter uses *call by value-result* semantics, the implementation uses the more efficient *call by reference*. We believe that our project is the first instance of a mechanically checked proof involving a call by reference implementation of a call by value-result semantics.

Our translator was entirely specified in the Boyer-Moore logic and the proof carried out using the Kaufmann-enhanced Boyer-Moore theorem prover. The proof is fully mechanically checked. The only explicit axioms assumed in the proof insure that the two declared constants representing the maximum sizes for the Piton temporary stack and control stack are numbers less than 2^{32}. The only other assumptions of the proof are the axioms provided by the Boyer-Moore implementation and the axioms added as a result of function and shell definitions. We have been extremely careful in formulating definitions, but there is always a possibility that our definitions do not reflect the desired intuition. This, unfortunately, is a hazard of program verification which can be minimized but never entirely eliminated.

9.2.1-B Trusted Systems

A frequent criticism of program verification is that there is a large semantic gap between verified high-level language programs and their ultimate representation in machine language running on real hardware. The Micro-Gypsy project is a component of a larger project designed to partially bridge this gap. In Chapter 8 we illustrated how Micro-Gypsy programs could be proved. These programs and accompanying Micro-Gypsy execution environments can then be compiled using the verified Micro-Gypsy code generator into Piton execution environments in such a way that the program semantics are provably preserved. Piton has been proven to be correctly implemented on the FM8502 microprocessor which, in turn, has been verified down to the gate level. As explained in Section 8.3, we still have some work to do to show that the output of the Micro-Gypsy code generator is acceptable to the implementation of Piton on the FM8502. Our proof, however, takes a large step toward closing the semantic gap.

Once completed, the "stack" of verified components--the Micro-Gypsy code generator, Piton assembler, and FM8502--will provide an environment for building verified applications with a much higher degree of reliability than any methodology previously available. This environment could be extended in various ways discussed in Section 9.2.2, but we believe the Micro-Gypsy code generator to be a significant tool for furthering our ultimate goal of building highly reliable programs.

9.2.2 Future Work

There are a number of potentially useful directions for future research building upon the current work.

9.2.2-A Finishing the Stack

Our proof has shown that Micro-Gypsy programs are correctly compiled into Piton programs. However, the proof of the implementation of Piton on the FM8502 imposes more restrictions on Piton programs than does the semantic definition of Piton. We cannot prove that the output of our code generator satisfies *all* of these constraints. It is not reasonable to prove that our programs are never too big to load into the memory of the FM8502, for example. Some of the constraints are purely syntactic and we can prove that all of the Piton programs we generate satisfy them. This proof is necessary to show that the Micro-Gypsy and Piton implementations "stack" as we have indicated. The particular constraints of the Piton implementation are discussed in Section 8.3.

9.2.2-B Building Verified Applications

The Micro-Gypsy code generator, sitting as it does on the Piton and FM8502 work, provides the capability of building extremely reliable applications. However, the usability of the language in its current form is questionable. Building some small applications in which correctness is critical would both show the viability of the methodology and point out deficiencies of the language.

9.2.2-C Extending the Language

Some deficiencies in the current subset are apparent. The language was pared down to illustrate the feasibility of constructing a rigorously verified code generator while keeping the project manageable. The result is a language with limited functionality but one which we believe can be easily extended in a number of ways.

1. We could add more of the type structure of Gypsy 2.05, including records and multi-dimensional arrays. The dynamic data type of Gypsy--sets, sequences, and mappings--are more problematic since in their full generality they require run-time storage management. We could envision adding Gypsy’s bounded dynamic types and allocating the maximum storage on procedure entry.
2. Adding some subset of the Gypsy expression language would eliminate the need for the current raft of predefined procedures at the cost of complicating the parameter passing mechanism. In particular, the semantics and translator would have to take cognizance of the distinction between *var* and *const* parameters as they do not now do. Array elements could become first-class citizens with respect to procedure parameters.
3. Some input and output facilities would have to be added to the language.

Some of these extensions are easy; some, such as I/O, would be conceptually challenging. Any of them would require a significant amount of proof effort.

9.2.2-D The Preprocessor

There is currently a gap between the syntax of the Micro-Gypsy programs which can be handled in the Gypsy Verification Environment and the abstract prefix syntax acceptable to the code generator. We envision this gap being bridged by a preprocessor. This should be defined and verified. Extending the language with Gypsy expressions as suggested in Section 9.2.2-C would simplify the preprocessor by eliminating the need to "flatten" expressions into a sequence of calls to predefined routines.

9.2.2-E Optimization

The code generator does not currently have an optimization phase. It would be relatively easy to define and prove a peephole optimizer that operated on the code generator output. This would be an interpreter equivalence proof where both the source and target language interpreters were the Piton interpreter. The *Map-Down* function would be the optimization routines and the *Map-Up* function the identity function on the data space. A related proof is described by Samet [Samet 75].

9.2.2-F Verified Proof Support

We noted in Chapter 8 that proving Micro-Gypsy programs against the interpreter semantics is difficult relative to proofs of the analogous programs in the GVE. One reason is because many of the facilities provided by the GVE must be handled "manually" when proving outside the GVE. It might be possible to build a highly reliable verification system around Micro-Gypsy. A first step would be a verification condition generator which used the interpreter semantics for generating verification conditions. The proof of this system component would yield a valuable addition to our suite of verified system components. The proof support, in the form of the Kaufmann-enhanced Boyer-Moore prover is already highly reliable, but we could envision formalizing and proving key components.

Appendix A
The Kaufmann-Enhanced Boyer-Moore System

Sections A1 and A2 of this appendix were written by Boyer and Moore and taken with permission from [Boyer 87]. Section A3 was written by Matt Kaufmann.

In [Boyer 79] is described a quantifier free first-order logic and a large and complicated computer program that proves theorems in that logic. The major application of the logic and theorem prover is the formal verification of properties of computer programs, algorithms, system designs, etc. In this section is described the logic and the theorem prover.

A.1 The Logic

A complete and precise definition of the logic can be found in Chapter III of [Boyer 79] together with the minor revisions detailed in section 3.1 of [Boyer 81].

The prefix syntax of Pure Lisp is used to write down terms. For example, we write **`(PLUS I J)`** where others might write **`PLUS(I,J)`** or **`I+J`**.

The logic is first-order, quantifier free, and constructive.[44] It is formally defined as an extension of propositional calculus with variables, function symbols, and the equality relation. Axioms are added defining the following:

- the Boolean objects **`(TRUE)`** and **`(FALSE)`**, abbreviated **`T`** and **`F`**;
- The if-then-else function, **`IF`**, with the property that **`(IF X Y Z)`** is **`Z`** if **`X`** is **`F`** and **`Y`** otherwise;
- the Boolean "connector functions" **`AND`**, **`OR`**, **`NOT`**, and **`IMPLIES`**; for example, **`(NOT P)`** is **`T`** if **`P`** is **`F`** and **`F`** otherwise;
- the equality function **`EQUAL`**, with the property that **`(EQUAL X Y)`** is **`T`** or **`F`** according to whether **`X`** is **`Y`**;
- inductively constructed objects, including:
    - Natural Numbers. Natural numbers are built from the constant **`(ZERO)`** by successive applications of the constructor function **`ADD1`**. The function **`NUMBERP`** recognizes natural numbers, e.g., is **`T`** or **`F`** according to whether its argument is a natural number or not. The function **`SUB1`** returns the predecessor of a non-**`0`** natural number.
    - Ordered Pairs. Given two arbitrary objects, the function **`CONS`** returns an ordered pair containing them. The function **`LISTP`** recognizes such pairs. The functions **`CAR`** and **`CDR`** return the two components of such a pair.
    - Literal Atoms. Given an arbitrary object, the function **`PACK`** constructs an atomic symbol with the given object as its "print name." **`LITATOM`** recognizes such objects and **`UNPACK`** returns the print name.
- Each of the classes above is called a "shell." **`T`** and **`F`** are each considered the elements of two singleton shells. Axioms insure that all shell classes are disjoint;

[44] The latest versions of the logic contain partial functions and bounded quantification and are not strictly constructive. The work described in this thesis makes no use of these features and is purely constructive.

- the definitions of several useful functions, including:
    - **LESSP** which, when applied to two natural numbers, returns **T** or **F** according to whether the first is smaller than the second;
    - **LEX2**, which, when applied to two pairs of naturals, returns **T** or **F** according as whether the first is lexicographically smaller than the second; and
    - **COUNT** which, when applied to an inductively constructed object, returns its "size;" for example, the **COUNT** of an ordered pair is one greater than the sum of the **COUNT**s of the components.

The logic provides a principle under which the user can extend it by the addition of new shells. By instantiating a set of axiom schemas the user can obtain a set of axioms describing a new class of inductively constructed **N**-tuples with type-restrictions on each component. For each shell there is a recognizer (e.g., **LISTP** for the ordered pair shell), a constructor (e.g., **CONS**), an optional empty object (e.g., there is none for the ordered pairs but **(ZERO)** is the empty natural number), and **N** accessors (e.g., **CAR** and **CDR**).

The logic provides a principle of recursive definition under which new function symbols may be introduced. Consider the definition of the list concatenation function:

```
DEFINITION.
(APPEND X Y)
   =
(IF (LISTP X)
    (CONS (CAR X) (APPEND (CDR X) Y))
    Y).
```

The equations submitted as definitions are accepted as new axioms under certain conditions that guarantee that one and only one function satisfies the equation. One of the conditions is that certain derived formulas be theorems. Intuitively, these formulas insure that the recursion "terminates" by exhibiting a "measure" of the arguments that decreases, in a well-founded sense, in each recursion. A suitable derived formula for **APPEND** is:

```
(IMPLIES (LISTP X)
         (LESSP (COUNT (CDR X))
                (COUNT X))).
```

However, in general the user of the logic is permitted to choose an arbitrary measure function (**COUNT** was chosen above) and one of several relations (**LESSP** above).

The rules of inference of the logic, in addition to those of propositional calculus and equality, include mathematical induction. The formulation of the induction principle is similar to that of the definitional principle. To justify an induction schema it is necessary to prove certain theorems that establish that, under a given measure, the inductive hypotheses are about "smaller" objects than the conclusion.

Using induction it is possible to prove such theorems as the associativity of **APPEND**:

```
THEOREM.
(EQUAL (APPEND (APPEND A B) C)
       (APPEND A (APPEND B C))).
```

A.2 The Mechanization of the Logic

The theorem prover for the logic, as it stood in 1979, is described completely in [Boyer 79]. Many improvements have been added since. In [Boyer 81] is described a "metafunction" facility which permits the user to define new proof procedures in the logic, prove them correct mechanically, and have them used efficiently in subsequent proof attempts. During the period 1980-1985 a linear arithmetic decision procedure was integrated into the rule-driven simplifier. The problems of integrating a decision procedure into a heuristic theorem prover for a richer theory are discussed in [Boyer 85]. The theorem prover is briefly sketched here.

The theorem prover is a computer program that takes as input a term in the logic and repeatedly transforms it in an effort to reduce it to non-**F**. The theorem prover employs eight basic transformations:

- decision procedures for propositional calculus, equality, and linear arithmetic;
- term rewriting based on axioms, definitions and previously proved lemmas;
- application of verified user-supplied simplifiers called "metafunctions;"
- renaming of variables to eliminate "destructive" functions in favor of "constructive" ones;
- heuristic use of equality hypotheses;
- generalization by the replacement of terms by type-restricted variables;
- elimination of apparently irrelevant hypotheses; and
- mathematical induction.

The theorem prover contains many heuristics to control the orchestration of these basic techniques.

In a shallow sense, the theorem prover is fully automatic: the system accepts no advice or directives from the user once a proof attempt has started. The only way the user can alter the behavior of the system during a proof attempt is to abort the proof attempt. However, in a deeper sense, the theorem prover is interactive: the system's behavior is influenced by the data base of lemmas which have already been formulated by the user and proved by the system. Each conjecture, once proved, is converted into one or more "rules" which guide the theorem prover's actions in subsequent proof attempts.

A data base is thus more than a logical theory: it is a set of rules for proving theorems in the given theory. The user leads the theorem prover to "difficult" proofs by "programming" its rule base. Given a goal theorem, the user generally discovers a proof himself, identifies the key steps in the proof, and then formulates them as lemmas, paying particular attention to their interpretation as rules.

The key role of the user in our system is guiding the theorem prover to proofs by the strategic selection of the sequence of theorems to prove and the proper formulation of those theorems. Successful users of the system must know how to prove theorems in the logic and must understand how the theorem prover interprets them as rules.

A.3 An Interactive Enhancement to the Prover

This section describes a system [Kaufmann 88] for checking the provability of terms in the Boyer-Moore logic. This system is loaded on top of the Boyer-Moore Theorem Prover, as explained below, and is integrated with that prover[45]. Thus, the user can give commands at a low level (such as deleting a

---

[45]This system uses however a slight extension of the syntax for terms, written by J Moore, which allows **COND**, **CASE**, and **LET**.

hypothesis) or at a high level (such as calling the Boyer-Moore Theorem Prover). As with a variety of proof-checking systems, this system is goal-directed: a proof is completed when the main goal and all subgoals have been proved. A notion of *macro commands* lets the user create compound commands, in the spirit of the *tactics* and *tacticals* of LCF [Gordon 79]. Upon completion of an interactive proof, the lemma with its proof may be stored as a Boyer-Moore *event* which can be added to the user's current library of events (i.e. definitions and lemmas).

During the course of a session, the user will submit a number of commands which will alter the state of the system. Some of these commands will create new goals to be proved. The session is complete when all goals have been proved, in the sense that their conclusions have been reduced to **T**.

The history of an interactive session is stored as a *state stack*, which is a list of *proof states* (or, *"states"* for short). A *state* contains a collection of *goals*, where each *goal* has a list of *hypotheses* and a *conclusion*. Each of the goal's hypotheses can either be active or hidden; hidden hypotheses are generally ignored by proof commands unless (and until) they are made active (again). Dependencies are recorded between goals: the goals are stored in a directed acyclic graph, where an arc joins one goal to another if the former depends on the latter. At the start of an interactive session, only one state is on the state stack, namely the one corresponding to the user's input.

When the interactive proof is complete, i.e. when all goals have been proved, the user may create an event by submitting an appropriate **EXIT** command. For example, an interactive session for proving the associativity of the **APPEND** function might culminate, by way of the command **(EXIT ASSOCIATIVITY-OF-APPEND (REWRITE))**, with the printing of:

```
(PROVE-LEMMA ASSOCIATIVITY-OF-APPEND
             (REWRITE)
             (EQUAL (APPEND (APPEND X Y) Z)
                    (APPEND X (APPEND Y Z)))
             ((INSTRUCTIONS (INDUCT (APPEND X Y))
                            PROMOTE
                            (DIVE 1 1)
                            X UP X NX X TOP
                            (DIVE 1 2)
                            = TOP S S)))
```

along with a query asking the user if he wants to submit this as a top-level event for the evolving Boyer-Moore "history" (i.e. database). The **INSTRUCTIONS** hint is a record of all the (successful) proof commands given during the session. This event may be submitted like any other Boyer-Moore event when running events in *batch mode*, i.e. when submitting events to Lisp rather than creating them through the interactive proof checker.

Appendix B
The Events in the Proof

The real product of the work described in this dissertation is the formal specification and proof of correctness of a code generator for Micro-Gypsy. The proof is embodied in a list of events[46] which can be submitted to the Kaufmann-enhanced Boyer-Moore theorem prover described in Appendix A. Many of these events have already been given in the alphabetically arranged lists in sections 3.5, 4.2, 5.7, and 6.3.

The other events, include all of the lemmas which comprise the proof, are included in Appendix B of the original dissertation. This list has been omitted from the current version.

[46]The form in which we have presented events in this dissertation is a minor syntactic variant of the form in which they are accepted by the prover.

References

[Aubin 76] R. Aubin.
Mechanizing Structural Induction.
PhD thesis, University of Edinburgh, Edinburgh, Scotland, 1976.

[Berry 71] D.M. Berry.
Block Structure: Retention or Deletion?
In *3rd SIGACT Symposium on the Theory of Computing*. 1971.

[Bevier 87] W. Bevier.
A Verified Operating System Kernel.
PhD thesis, The University of Texas at Austin, October, 1987.

[Blum 69] E.K. Blum.
Toward a Theory of Semantics and Compilers for Programming Languages.
Journal of Computer and System Sciences 3(3):248-275, August, 1969.

[Boebert 85] W.E. Boebert, W.D. Young, R.Y. Kain, S.A. Hansohn.
Secure ADA Target: Issues, System Design, and Verification.
In *Proceedings of the IEEE Symposium on Security and Privacy*. 1985.

[Boyer 79] R.S. Boyer, J S. Moore.
A Computational Logic.
Academic Press, New York, 1979.

[Boyer 81] R.S. Boyer, J S. Moore.
Metafunctions: Proving Them Correct and Using them Efficiently as New Proof Procedures.
The Correctness Problem in Computer Science.
Academic Press, London, 1981.

[Boyer 85] R.S. Boyer, J S. Moore.
Integrating Decision Procedures into Heuristic Theorem Provers: A Case Study of Linear Arithmetic.
Technical Report ICSCA-CMP-44, Institute for Computing Science, University of Texas at Austin, January, 1985.

[Boyer 87] R.S. Boyer, J S. Moore.
The Addition of Bounded Quantification and Partial Functions to A Computational Logic and its Theorem Prover.
Technical Report ICSCA-CMP-52, Institute for Computing Science, University of Texas at Austin, January, 1987.

[BoyerMoore 88] R. S. Boyer and J S. Moore.
A User's Manual for a Computational Logic.
Technical Report CLI-18, CLInc, June, 1988.

[Burge 68] W.H. Burge.
Proving the Correctness of a Compiler.
Technical Report, IBM Research Division, June, 1968.

[Burstall 69] R.M. Burstall.
Proving Properties of Programs by Structural Induction.
Computer Journal 12(1):41-48, February, 1969.

[BurstallLandin 69]
R.M. Burstall and P. Landin.
Programs and their Proofs: An Algebraic Approach.
Machine Intelligence 7.
American Elsevier, New York, 1969, pages 17-43.

[Cartwright 76] R. Cartwright.
A Practical Formal Semantic Definition and Verification System for Typed LISP.
PhD thesis, Stanford University, 1976.

[Chirica 76] L.M. Chirica.
An Approach to Compiler Correctness.
PhD thesis, University of California at Los Angeles, October, 1976.

[ChiricaMartin 75]
L.M. Chirica and D.F. Martin.
An Approach to Compiler Correctness.
In *Proceedings of the International Conference on Reliable Software*, pages 96-103. April, 1975.

[ChiricaMartin 86]
L.M. Chirica and D.F. Martin.
Toward Compiler Implementation Correctness Proofs.
ACM Transaction on Programming Languages and Systems 8(2):185-214, 1986.

[Cohen 86] Richard Cohen.
Proving Gypsy Programs.
Technical Report CLI-4, CLInc, May, 1986.

[Cohn 79a] A. Cohn.
High Level Proof in LCF.
In *Proceedings of the Fifth Symposium on Automated Deduction*. 1979.

[Cohn 79b] A. Cohn.
Machine Assisted Proofs of Recursion Implementation.
PhD thesis, University of Edinburgh, Edinburgh, Scotland, 1979.

[Cohn 87] A. Cohn.
A Proof of the Correctness of the Viper Microprocessor: The First Level.
Technical Report 104, University of Cambridge Computer Laboratory, January, 1987.

[Dijkstra 62] E.W. Dijkstra.
An Attempt to Unify the Constituent Concepts of Serial Program Execution.
Symbolic Languages in Data Processing: Proceedings ICC Symposium.
Gordon & Breach, New York, 1962, pages 237-251.

[Garwick 66] J.V. Garwick.
The Definition of Programming Languages by Their Compilers.
Formal Language Description Languages for Computer Programming.
North Holland, Amsterdam, 1966, pages 139-147.

[Good 84] D.I. Good.
SCOMP Trusted Processes.
ICSCA Internal Note 138, The University of Texas at Austin.
1984

[GoodAkersSmith 86]
D.I. Good, R.L. Akers, L.M. Smith.
Report on Gypsy 2.05.
Technical Report CLI-1, CLInc, October, 1986.

[GoodDivitoSmith 88]
D.I. Good, B.L. Divito, M.K. Smith.
Using The Gypsy Methodology.
Technical Report Draft CLI-2, CLInc, January, 1988.

[Gordon 79] M. J. Gordon, A. J. Milner, and C. P. Wadsworth.
Edinburgh LCF.
Springer-Verlag, New York, 1979.

[HenhaplJones 70]
W. Henhapl and C.B. Jones.
The Block Structure Concept and Some Possible Implementations.
Technical Report 25.104, IBM Laboratories, Vienna, 1970.

[Hunt 85] Warren A. Hunt.
FM8501: A Verified Microprocessor.
Technical Report ICSCA-CMP-47, Institute for Computing Science, University of Texas at Austin, December, 1985.

[Kaplan 67] D.M. Kaplan.
Correctness of a Compiler for Algol-like Programs.
Technical Report 48, Stanford University Artificial Intelligence Laboratory, July, 1967.

[Kaufmann 88] Matt Kaufmann.
A User's Manual for an Interactive Enhancement to the Boyer-Moore Theorem Prover.
Technical Report CLI-19, CLInc, May, 1988.

[KaufmannYoung 87]
Matt Kaufmann and William Young.
Comparing Specification Paradigms for Secure Systems: Gypsy and the Boyer-Moore Logic.
In *Proceeding of the 10th National Computer Security Conference*. 1987.

[Keeton-Williams 82]
J. Keeton-Williams, S.R. Ames, B.A. Hartman, and R.C. Tyler.
Verification of the ACCAT-Guard Downgrade Trusted Process.
Technical Report NTR-8463, The Mitre Corporation, Bedford, MA., 1982.

[Landin 64] P. Landin.
The Mechanical Evaluation of Expressions.
Computer Journal 6(4):308-320, 1964.

[Landin 65] P. Landin.
A Correspondence Between ALGOL 60 and Church's Lambda Notation.
Communications of the ACM 8(2,3):89-101, 158-165, Feb.-March, 1965.

[Lee 72] J.A.N. Lee.
Computer Semantics.
Van Nostrand Reinhold, New York, 1972.

[Levy 85] B.H. Levy.
An Approach to Compiler Correctness Using Interpretation Between Theories.
Technical Report 85(8354)-1, Aerospace Corporation, April, 1985.

[London 71] R.L. London.
Correctness of Two Compilers for a Lisp Subset.
Technical Report AIM-151, Stanford Univerity Artificial Intelligence Laboratory, October, 1971.

[London 72] R.L. London.
Correctness of a Compiler for a LISP Subset.
In *Proceedings of an ACM Conference on Proving Assertions about Programs*. 1972.

[Lucas 68] P. Lucas.
Two Constructive Realizations of the Block Concept and Their Realization.
Technical Report 25.082, IBM Laboratories, Vienna, 1968.

[LucasWalk 69] P. Lucas and K Walk.
On the Formal Description of PL/I.
Annual Reviews in Automatic Programming 6(3), 1969.

[Lynn 78] D.S. Lynn.
Interactive Compiler Proving Using Hoare Proof Rules.
Technical Report ISI/RR-78-70, Information Sciences Institute, January, 1978.

[McCarthy 62] John McCarthy.
Towards a Mathematical Science of Computation.
In *Proceedings of the IFIP Congress*. North Holland, Amsterdam, 1962.

[McCarthy 66] John McCarthy.
A Formal Description of a Subset of ALGOL.
Formal Language Description Languages.
North Holland, Amsterdam, 1966.

[McCarthyPainter 67]
John McCarthy and J. Painter.
Correctness of a Compiler for Arithmetic Expressions.
In *Proceeding of Symposium on Applied Mathematics*. American Mathematical Society, 1967.

[McGowan 71] C. McGowan.
Correctness Results for Lambda Calculus Interpreters.
PhD thesis, Cornell University, 1971.

[McGowan 72] C. McGowan.
An Inductive Proof Technique for Interpreter Equivalence.
Formal Semantics of Programming Languages.
Prentice-Hall, Englewood Cliffs, N.J., 1972, pages 139-147.

[MilnerWeyhrauch 72]
R. Milner and R. Weyhrauch.
Proving Compiler Correctness in a Mechanized Logic.
Machine Intelligence 7.
Edinburgh University Press, Edinburgh, Scotland, 1972, pages 51-70.

[MilneStrachey 76]
R. Milne and C. Strachey.
A Theory of Programming Language Semantics.
Chapman and Hall, London, 1976.

[Moore 88] J S. Moore.
PITON: A Verified Assembly Level Language.
Technical Report CLI-22, CLInc, June, 1988.

[Morris 72] F.L. Morris.
Correctness of Translations of Programming Languages--An Algebraic Approach.
Technical Report AIM-174, Stanford Univerity Artificial Intelligence Laboratory, August, 1972.

[Morris 73] F.L. Morris.
Advice of Structuring Compilers and Proving Them Correct.
In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 144-152. October, 1973.

[Mosses 79] P. Mosses.
A Constructive Approach to Compiler Correctness.
Technical Report DAIMI IR-16, Aarhus University, December, 1979.

[Newey 75] M.C. Newey.
Formal Semantics of LISP with Applications to Program Correctness.
Technical Report AIM-257, Stanford Univerity Artificial Intelligence Laboratory, January, 1975.

[Ollongren 74] A. Ollengren.
Definition of Programming Languages by Interpreting Automata.
Academic Press, London, 1974.

[Pagan 76] F.G. Pagan.
On Interpreter-oriented Definitions of Programming Languages.
Computer Journal 19(2):151-155, 1976.

[Pagan 78] F.G. Pagan.
Formal Semantics of a SNOBOL4 Subset.
Computer Languages 3:13-30, 1978.

[Painter 67] J.A. Painter.
Semantic Correctness of a Compiler for an Algol-like Language.
PhD thesis, Stanford University, 1967.

[Polak 81] W. Polak.
Compiler Specification and Verification.
Springer-Verlag, Berlin, 1981.

[Ragland 73] L.C. Ragland.
A Verified Program Verifier.
PhD thesis, University of Texas at Austin, 1973.

[Rashovsky 82] M. Rashovsky.
Denotational Semantics as a Specification of Code Generators.
In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, pages 230-244. June, 1982.

[Samet 75] H. Samet.
Automatically Proving the Correctness of Translations involving Optimized Code.
Technical Report AIM-259, Stanford Univerity Artificial Intelligence Laboratory, May, 1975.

[Siebert 84] A.E. Siebert and D.I. Good.
General Message Flow Modulator.
Technical Report ICSCA-CMP-42, Institute for Computing Science, University of Texas at Austin, March, 1984.

[Smith 81] M.K. Smith, A. Siebert, B. Divito, and D. Good.
A Verified Encrypted Packet Interface.
Software Engineering Notes 6(3), July, 1981.

[Thatcher 81] J.W. Thatcher, E.G. Wagner, J.B. Wright.
More on Advice on Structuring Compilers and Proving Them Correct.
Theoretical Computer Science 15:223-249, 1981.

[Wand 82] M. Wand.
Deriving Target Code as a Representation of Continuation Semantics.
ACM Transaction on Programming Languages and Systems 4(3):496-517, 1982.

[Wegner 72a] Peter Wegner.
Programming Language Semantics.
Formal Semantics of Programming Languages.
Prentice-Hall, Englewood Cliffs, N.J., 1972, pages 149-248.

[Wegner 72b] Peter Wegner.
The Vienna Definition Language.
Computing Surveys 4(1), 1972.

Index

Table of Contents

List of Figures

List of Tables