A Verified Implementation of an Applicative Language with Dynamic Storage Allocation

Arthur D. Flatau

Technical Report 83 January 6, 1993

Computational Logic, Inc.
1717 West Sixth Street, Suite 290
Austin, Texas 78703-4776

TEL: +1 512 322 9951
FAX: +1 512 322 0656

EMAIL: flatau@cli.com

This work was supported in part at Computational Logic, Inc., by the Defense Advanced Research Projects Agency, ARPA Order 7406. 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., the Defense Advanced Research Projects Agency or the U.S. Government.

Abstract

A compiler for a subset of the Nqthm logic and a mechanically checked proof of its correctness is described. The Nqthm logic defines an applicative programming language very similar to McCarthy's pure Lisp[20]. The compiler compiles programs in the Nqthm logic into the Piton assembly level language [23]. The correctness of the compiler is proven by showing that the result of executing the Piton code is the same as produced by the Nqthm interpreter `V&C$`. The Nqthm logic defines several different abstract data types, or shells, as they are called in Nqthm. The user can also define additional shells. The definition of a shell includes the definition of a constructor function that returns new objects with the type of that shell. These objects can become garbage, so the run-time system of the compiler includes a garbage collector. The proof of the correctness of the compiler has not been entirely mechanically checked. A plan for completing the proof is described.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Achievements . . . 3
- 1.3 Related Work . . . 4
- 1.4 Project History . . . 6
- 1.5 The Boyer-Moore Logic . . . 7
- 1.6 Outline of the Presentation . . . 7

2 Piton **9**
- 2.1 An Informal Sketch of Piton . . . 9
  - 2.1.1 An Example Piton Program . . . 10
  - 2.1.2 Piton States . . . 11
  - 2.1.3 Type Checking . . . 12
  - 2.1.4 Data Types . . . 13
  - 2.1.5 The Data Segment . . . 15
  - 2.1.6 The Program Segment . . . 16
  - 2.1.7 Instructions . . . 18
  - 2.1.8 The Piton Interpreter . . . 27
  - 2.1.9 Erroneous States . . . 28

3 Informal Description of the Implementation **31**
- 3.1 An Example . . . 31
- 3.2 Representing Data in Piton . . . 32
- 3.3 The Garbage Collector . . . 37
- 3.4 Compiling from the Logic to Piton . . . 42
  - 3.4.1 Compiling Variable References . . . 42
  - 3.4.2 Compiling Constants . . . 43
  - 3.4.3 Compiling Function Calls . . . 44
  - 3.4.4 Compiling IF . . . 44
  - 3.4.5 The Postlude . . . 45
- 3.5 Optimizations . . . 46

4 The Correctness Theorem **47**
4.1 Interpreter Equivalence Proofs 47
4.2 The Statement of the Correctness Theorem 48
4.3 Proper Expressions and Programs 51
4.4 Mapping the Final Piton State Up 51
4.5 Using the Compiler 53

5 Formal Description of the Compiler **55**
5.1 The Common Sub-Expression Level 56
5.2 The Resource Representation Level 58
5.3 The Piton level 61

6 Proof of Correctness **63**
6.1 The S layer 66
6.2 The LR layer 68
6.2.1 Proper LR states 71
6.2.2 Proof of Correspondence 73
6.3 The Piton layer 75

7 Future work **77**
7.1 Finish compiler 77
7.2 Compiling Common Lisp 78
7.3 Optimizations 79
7.3.1 Common Sub-Expressions 79
7.3.2 Other optimizations 80
7.4 Other Garbage Collection Schemes 81
7.5 Examples 82

A Piton translation of example program **83**

B Function Definitions for the Implementation **95**

C Nqthm Libraries **127**
C.1 Bags Library 127
C.2 Naturals Library 127
C.3 Integers Library 127
C.4 Lists Library 127

D Piton Events **129**

E Listing of Events in the Proofs of Compiler With and Without Garbage Collector **131**

F Listing of Events in the Proof of the Compiler Without a Garbage Collector **133**

G Listing of Events in the Proof of the Compiler With a Garbage Collector **137**
G.1 Events with Mechanically Checked Proofs 137
G.2 Events without Mechanically Checked Proofs 137

Bibliography **139**

List of Figures

3.1 An example program 32
3.2 The structure of a *Node* 33
3.3 The effects of executing `I-ALLOC-NODE` 39
3.4 The effects of executing `CONS` and `CAR` 40

4.1 Interpreter Equivalence Diagram 48

5.1 An Example of Removing Common Subexpressions 58

6.1 Interpreter Equivalence Diagram 64

Chapter 1

Introduction

This dissertation describes a compiler for a subset of the Boyer-Moore logic, a formal statement of its correctness and the mechanically checked proof of that statement. The Boyer-Moore logic (hereafter referred to simply as the Logic) defines an applicative programming language similar to pure Lisp. The compiler produces code in the high-level assembly language Piton [23]. The proof has been broken down into steps in anticipation of its eventual extension to handle all of the Logic. The compiler for the full Logic is described and how the mechanically checked proof for the prototype can be extended is explained.

The Logic includes functions to construct new objects of various "types"; for instance, `CONS`. This requires an implementation to allocate storage dynamically and makes a garbage collector highly desirable.

The goal of this work is an implementation of a programming language that is shown to be correct via a machine-checked proof. Since the assembly language Piton, which is the output language of the compiler, has an assembler and linker that have been proven correct, and the gate-level description of FM8502, the machine that Piton is implemented on, has also been proven correct, we have a great deal of assurance that programs compiled by the compiler will be executed correctly. Moreover, the programming language is not a toy. The implementation has a substantial run-time support system, including dynamic memory allocation and automatic recovery of unused memory.

1.1 Motivation

More and more, computer systems are being used to control systems whose failure or unexpected behavior would be catastrophic, such as nuclear power plants and airplanes. These computer systems must behave correctly in order to minimize the chances of such a catastrophe. The correct behavior of a computer system depends on the correctness of the programs as well as the correct implementation of the

programming language in which the programs are written.

A correct implementation of a programming language must faithfully preserve the semantics of the programming language. Such an implementation typically includes a compiler, some run-time support routines for the compiler, an assembler, a linker, a loader, and some hardware. A correct implementation requires that all of these components function properly.

The most rigorous method of assuring that a program meets its specification is to produce a formal proof that it does and to have the proof checked by a machine. But even such assurance is of questionable value if similar rigor has not been applied to the implementation of the source programming language. An incorrect implementation could cause a program to behave differently from its specification. Ideally, verified programs should be run by verified implementations.

One might wonder how the compiler fits into the program development process. Suppose a programmer has written a program, `P`, in the Nqthm Logic and that it is desired to evaluate `P` on some concrete data, `D`. One way this can be done is to submit `P` and `D` to the compiler described here; the compiler will produce a Piton core image. That "initial" Piton core image can be executed via the Piton interpreter to produce a "final" Piton core image. Then, using an algorithm described in Chapter 4, a Nqthm object (e.g., a list) can be abstracted from the final Piton core image. That object is allegedly the value of the Nqthm program `P` on the data `D`.

The correctness theorem for the compiler (which is described in Chapter 4) says that under certain conditions the abstracted object is indeed the value of `P` on `D`. The conditions can be broken up into three categories. First, `P` on `D` must terminate (that is the Nqthm interpreter `V&C$` must return non-`F`). Second, `P` and `D` must be syntactically well-formed. Finally the Piton machine must have enough resources (stack and memory space) to correctly execute the program. All three of these things potentially can be proven about a particular program and data using the Nqthm theorem prover. Proving that the program terminates and is syntactically well-formed are relatively straightforward tasks. Unfortunately, proving that a program can be executed by Piton within a fixed set of resources requires much knowledge about the implementation.

At first glance, it seems unusual that the correctness of the compiler depends on the existence of sufficient resources in the target machine. The semantics (whether formally defined or not) of most programming languages allow one to determine the result of some computation. The semantics do not take into account resource limitations—the semantics of a program does not depend the amount of memory in the underlying machine. Certainly the semantics of the logic (as defined by `V&C$`) do not depend on any resource assumptions. However all programming language implementations have only finite resources. Many implementations will either compute the result of a program or report an error if there are not enough resources to do the computation. The compiler for the Logic uses a different approach: it does not check if resource errors occur. Rather, provided there are enough resources, the result of running the Piton machine on a Piton state produced by the compiler

is correct. This allows the implementation to blindly push arguments on the stack, allocate memory in the heap, etc. without ever checking whether this will cause an error. This results in a somewhat smaller implementation and the code produced runs faster, but there is a price. The user must ensure that there are sufficient resources for the computation.

The work described here includes a formal definition of what is meant by "sufficient resources" for a given `P` and `D`. This definition necessarily involves implementation details, e.g., how much stack space is used by the compiled code for a function call. The third set of hypotheses on our correctness result are phrased in terms of these concepts. As previously noted, it is thus possible, in principle, to use the theorem prover to establish that a given amount of resources are sufficient for a given `P` and `D`. But we do not give the programmer any assistance in proving this last set of hypotheses. It is conceivable that tools could be developed that would provide assistance with this task. It seems reasonable to believe that a tool could be constructed that could take a program in the Logic and produce formulae in the Logic, that if proven, could ensure that resource errors do not occur when executing the Piton code output by the compiler. This hypothetical tool could hide some of the details of the Piton implementation from the programmer. This research does not address the task of proving that a program does not cause resource errors.

The difficulty of doing the resource analysis is unclear. It would be nice if the amount of resources required could be computed by a simple function of the actual arguments to the program. For instance, to compute `(APPEND X Y)`, the amount of heap space required is proportional to `(LENGTH X)`. However for more complex programs computing the amount of resources required would likely be quite difficult. Analyzing the resource requirements of programs in the Logic is in principle no different from analyzing the resource requirements of programs in other languages. Other research in this area can be used to help with the resource analysis required by the correctness theorem.

1.2 Achievements

Among the significant achievements of this project are the following:

1. A mechanically verified implementation for a subset of the Logic has been produced. This subset includes variables, Boolean and list constants and (possibly recursive) function application. The primitive functions supported are: `CAR`, `CDR`, `CONS`, `FALSE`, `FALSEP`, `IF`, `LISTP`, `NLISTP`, `TRUE`, and `TRUEP`. User-defined functions are supported. Most of the problems raised in implementing the entire Logic, or indeed, any conventional applicative language are solved in the current implementation, because the implemented subset includes `CONS` (and thus requires dynamic storage allocation), function application, and several data types. The mechanically verified implementation lacks a garbage collector.

2. An extension to the above implementation — and an extension of its proof — to add a reference count garbage collector and additional data types, including user-defined ones is described.

3. Neither the source language of the compiler — the Logic — nor the target language of the compiler — Piton — have been changed to support a proof of correctness. Some proofs of compilers have used languages designed to make such proofs easier. The Logic is a small language, but it is hardly a toy language. It is very similar to a small but useful subset of Common Lisp [15]. Since Piton has been implemented on an actual machine, the compiler proof had to deal with the finite resources of that machine.

4. The abstract data-types of the Logic (e.g. `CONS`) are different from the concrete data-type of Piton (a large array). The compiler is proven to implement correctly these abstract data-types in Piton memory.

5. The run-time system, which includes dynamic storage allocation, has a mechanically checked proof of correctness. A proof of run-time system that includes a reference count garbage collector is in progress. There have been proofs of garbage collection algorithms in the past (e.g., [13] and [27]) but no proofs of actual code that include a garbage collector.

6. This work provides a solid base for a verified implementation of an important subset of Common Lisp. Common Lisp is a relatively widely used "real" programming language. There are many programs written in Common Lisp, in particular, the Boyer-Moore theorem prover. It would be quite useful to have a very reliable implementation of the Boyer-Moore theorem prover.

1.3 Related Work

The first attempt to prove the correctness of a compiler using an interpreter equivalence proof seems to be the work of McCarthy and Painter [21]. They prove by hand the correctness of a compiler for a simple expression language. Burstall [7] develops a theory for proving properties of programs with structural induction. He uses these techniques to prove the same expression compiler as McCarthy and Painter. Milner and Weyhrauch [22] give a machine checked proof of an expression compiler similar to that of McCarthy and Painter. This is part of a larger compiler that includes assignment, conditional, and while statements. Cartwright [8] and Aubin [1] also prove versions of the McCarthy-Painter compiler.

Boyer and Moore [4] prove mechanically the correctness of a compiler similar to that of McCarthy and Painter. Their target machine has an unbounded stack, infinite precision, an unbounded number of registers and the operations of the target machine are the same as the expression language. Their compiler implements a simple constant folding optimization.

London [18] proves by hand the correctness of two versions of a compiler for a subset of Lisp. The second version contains more optimizations than the first.

Morris [24] uses algebraic semantics to specify and partially prove by hand a simple compiler for an Algol like language. Cohn [11] use Edinburgh LCF to prove the correctness of a simple compiling algorithm.

Chirica and Martin [9] use axiomatic semantics to specify and prove a simple compiler. Lynn [19] uses Hoare-style axiomatic semantics to specify source and target languages of a slightly modified version of London's compiler. He also uses the same technique to prove a slightly modified version of McCarthy and Painter's compiler. The former proof was done mostly by hand, the latter with machine assistance.

Levy [17] gives an approach to proving the correctness of compilers by using interpretations between theories.

There have been many proofs of garbage collection algorithms. Topor [29] gives a proof of the Schorr-Waite list marking algorithm. Dijkstra et. al. [13] gives a proof of an algorithm that allows the garbage collector and the user's program to run concurrently. Russinoff [27] gives a machine checked proof of an improved version of that algorithm. Baker [2] gives an algorithm that allows the garbage collector to run in real-time. He also gives an informal proof of the correctness of the algorithm.

There have been three previous large compiler verification efforts. Polak [26] verified a compiler for a substantial subset of Pascal. The semantics of his source language and target language were specified using denotational semantics. Polak verified the lexical analyzer, parser, type checker, and code generator. The target machine of his compiler is a stack machine. However, the target machine does not impose resource limitations on the compiler or the correctness theorem. The memory of the target machine is infinite and each memory location can hold arbitrarily large integers. He does handle the Pascal predefined procedure `NEW` to allocate memory dynamically. However, his target machine has direct support for this feature in the form of an instruction that does the allocation. This is not a feature of most microprocessors.

Moore [23] verifies that an implementation of the language Piton is correctly implemented on the FM8502 microprocessor. The implementation consists of a compiler, an assembler and a linker. Piton is a stack-based, high-level assembly language (see Chapter 2 for more details about the Piton language). The data-types of Piton were designed to fit into one FM8502 word. Piton also provides global arrays that are implemented in the memory of FM8502, which is a large array. Piton provides two stacks which are also implemented in the FM8502 memory. One of the most difficult parts of the Piton proof is verifying the correct implementation of the stacks on FM8502.

Young [30] compiles a subset of Gypsy 2.05 [14], called Micro-Gypsy, into Piton. Micro-Gypsy is a von Neumann language similar to a subset of Pascal. The principal difference between Young's work and this one are:

1. Micro-Gypsy is a substantially smaller language than the Logic. There are only eight statements in Micro-Gypsy.

2. The data types of Micro-Gypsy are almost the same as the data types of the target language Piton. Micro-Gypsy allows the following simple types: integers (which have the same range as the target machine), Booleans and characters. These are directly representable in Piton. In addition Micro-Gypsy allows arrays containing elements of one of the simple types.

Subsequent to the Piton and Micro-Gypsy effort, Curzon [12] has proved the correctness of an assembly-level language implementation. However the target machine is an idealization of a real machine with infinite memory.

Oliva and Wand [25] give a denotational specification for a restricted subset of Scheme called PreScheme and derive a compiling algorithm for PreScheme. The output of the compiler is a byte-code for an abstract machine. The byte-code machine is implemented on a MC68020 (this last step has not been verified). PreScheme does not appear to do any dynamic memory allocation and thus the implementation does not have a garbage collector.

1.4 Project History

This project started in the spring of 1989 during a class taught by J Moore. Work began in earnest in October 1989. By December 1989, the proof of the correctness of the first phase of the compiler was complete. The first phase of the compiler is designed to eliminate common sub-expressions. The current version of the compiler does not eliminate common sub-expressions; however, the proof is general enough to be extended easily if a routine to eliminate common sub-expressions is added. A routine to eliminate common sub-expressions needs to maintain a syntactic invariant in order to be added to the compiler and have the majority of the proof remain unchanged.

The second part of the compiler represents the abstract objects of the Logic in the concrete memory of Piton. The proof of the correctness of this part was the most difficult part of the proof. The proof of a prototype without a garbage collector was completed in December 1991.

The final part of the compiler translates the programs into Piton. The proof of this (without the garbage collector) was completed in January 1992.

After completion of the prototype, the garbage collector was added. The proof of the first part of the compiler did not need to be changed. The proof of the second part is almost complete and continues.

The current version of the compiler only has 10 of the 61 predefined functions of the Boyer-Moore logic implemented. In addition, user-defined shells need to be added. Adding these predefined functions is straight-forward — most can be compiled as user-defined functions. User-defined shells can be handled the same as the

predefined shells. Only numbers and literal atoms are treated specially. Extending the compiler to handle all of the Logic is discussed more fully in section 7.1.

1.5 The Boyer-Moore Logic

The source language of the compiler is a subset of the Boyer-Moore Logic which has been discussed in several places, most notably in [6] where it is completely and precisely described. Readers not familiar with the Logic should refer to [6]. As in Pure Lisp, all the partial recursive functions can be defined in the Logic.

1.6 Outline of the Presentation

This dissertation describes a simple compiler for the Boyer-Moore logic and a proof that the compiler does its job correctly. Unless indicated by a footnote, any formula displayed in any of the chapters of this dissertation as a theorem has been proven and the proof has been mechanically checked. There are only four so footnoted formulas. Chapter 2 describes the target language of the compiler, Piton. Chapter 3 gives an informal description of the implementation of the Logic. Chapter 4 explains the compiler's correctness theorem. This chapter also describes the restrictions on programs that can be compiled. Chapter 5 gives a formal description of the compiler in the Boyer-Moore logic. Some of the interesting parts of the proof of the correctness theorem are described in Chapter 6. Finally, Chapter 7 describes how this work can be extended.

Chapter 2

Piton

Piton is a high-level assembly language developed by J Moore [23]. Piton was designed for verified applications and as the target for verified compilers for higher-level languages.

An unusual aspect of Piton is that it provides execute-only programs. It is impossible for a correct Piton program to overwrite itself. The subroutine call and return mechanism is more sophisticated than that of most assembly languages. Subroutines have named formal parameters, and parameter are passed to subroutines on a stack. There are seven abstract data types in Piton: integers, natural numbers, bit vectors, Booleans, data addresses, program addresses (labels), and subroutine names. The compiler described here does not use all of the data types, e.g., it does not use bit vectors and subroutine names. Piton is formally specified by an interpreter (P) in the Logic.

The rest of this chapter is an informal description of Piton, along with a listing of the Piton definition. It is taken with permission and minor editing from the Piton report by J Moore [23].

2.1 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
- 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

2.1.1 An Example Piton Program

We begin our presentation of Piton with a simple example. Below we exhibit a Piton program named `DEMO`. The program is a list constant in the computational logic of Boyer and Moore[6] 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 `DEMO` program has three formal parameters, `X`, `Y`, and `Z`, and two temporary variables, `A` and `I`.

```
(DEMO (X Y Z)                    ; Formals X, Y, and Z
      ((A (INT -1))              ; Temporary A, initial value -1
       (I (NAT 2)))              ; Temporary I, initial value 2
      (PUSH-LOCAL Y)             ; Push the value of Y
      (PUSH-CONSTANT (NAT 4))    ; Push the natural number 4
      (ADD-NAT)                  ; Add the top two items
      (RET))                     ; Return
```

When `DEMO` is called, the topmost three items from Piton's temporary stack are popped off and used as the actual values of the formals `X`, `Y`, and `Z`. In addition, `A` is initialized to the integer -1, and `I` is initialized to the natural number 2. The values of all five of these "local" variables are restored when `DEMO` returns to its caller.

The body of `DEMO` contains four Piton instructions. The first, `(PUSH-LOCAL Y)`, pushes the value of the local variable `Y` onto the temporary stack. The second, `(PUSH-CONSTANT (NAT 4))`, pushes the natural number 4 onto the temporary stack. The third, `(ADD-NAT)`, pops the topmost two items off the temporary stack, adds them together (expecting both to be naturals), and pushes the result onto the temporary stack. The last instruction returns control to the calling environment.

The sum just computed is on top of the stack and is considered the result. In summary, this silly program adds 4 to the value of its second argument and ignores the other arguments. Its two temporary variables are not used.

Now consider the following sequence of Piton instructions.

```
(PUSH-CONSTANT (ADDR (DELTA1 . 25)))
(PUSH-CONSTANT (NAT 17))
(PUSH-CONSTANT (BOOL T))
(CALL DEMO)
```

This sequence pushes three items onto the stack and then calls `DEMO`. The `CALL` pops the three objects off the stack and uses them as the actuals. `DEMO`'s first formal, `X`, is bound to the data address `(DELTA1 . 25)` — the address of the 25^{th} location of the global array named `DELTA1`. `DEMO`'s second argument, `Y`, is bound to the natural number 17. Its third argument, `Z`, is bound to the Boolean value `T`. The execution of `DEMO` pushes 21 (the sum of 17 and 4) and returns. Thus, the net effect of the four instructions above — barring a variety of run-time errors such as stack overflow — is to push a 21 onto the stack.

2.1.2 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 counter*, indicating which instruction in which subroutine is the next to be executed;
- a *control stack*, recording the hierarchy of subroutine invocations leading to the current state;
- a *temporary stack*, containing intermediate results as well as the arguments and results of subroutine calls;
- 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 *maximum control stack size*;
- a *maximum temporary stack size*;
- a *word size*, which governs the size of numeric constants and bit vectors; and

- a *program status word* (*psw*).

The formalization of this concept is embodied in the function P-STATE. P-STATE takes nine arguments and returns a p-state with the appropriate nine components. Thus, (P-STATE PC CSTK TSTK PROGS DATA MAXC MAXT W PSW) is a p-state.

We put a variety of additional restrictions on the components of a p-state. For example, we require that every instruction in every program be syntactically well formed and mention no variable other than the locals of the containing program or the globals declared in the data segment. We also require that every data object occurring in the state be compatible with the state, e.g., every object tagged "address" must be a legal address in that state, etc. We call such p-states *proper p-states*. The formalization of this concept is embodied in the function PROPER-P-STATEP.

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 is a stack of *frames*, the topmost frame describing the currently active subroutine invocation and the successive frames describing the hierarchy of suspended invocations. The topmost frame is the only frame directly accessible to Piton instructions. Each frame contains two fields. One contains the *bindings* of the local variables of the invoked program. The other contains the *return program counter*, which is the program counter to which control is to return when the 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.

2.1.3 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.

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 run-time type checking facilities.* The programmer must know what he is doing.

Such cavalier run-time treatment of types — i.e., no syntactic type checking and no run-time 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 run-time 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 run-time. 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.

2.1.4 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.[1] The formalization of the concept of "legal Piton data object" is embodied in the function `P-OBJECTP`.

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

Integers

Piton provides the integers, i, in the range $-2^{w-1} \leq i < 2^{w-1}$. We say such integers are *representable* in the given p-state. Observe that there is one more representable negative integer than representable positive integer. 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.

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.

Booleans

There are two Boolean objects, called `T` and `F`. They are written as `(BOOL T)` and `(BOOL F)`.[2] 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).

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, left and right shift, and equivalence.

Data Addresses

A Piton data address is a pair consisting of a name and a natural 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 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 $\mathtt{n}^{\mathtt{th}}$ element of the array associated with `name`, where

[2] Note to those readers familiar with our 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.

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.

Program Addresses

A Piton program address is a pair consisting of a name and a natural 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 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).

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))`.

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.

2.1.5 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`, in order to obtain an address into the area named `A`.

2.1.6 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

```
(name (v0 v1 ... vn-1)
      ((vn in) (vn+1 in+1)  ... (vn+k-1 in+k-1))
      ins0
      ins1
      ...
      insm),
```

where name is the *name* of the program and is some literal atom; $v_0, \ldots, v_{n-1}$ are the $n \geq 0$ *formal parameters* of the program and are literal atoms; $v_n, \ldots, v_{n+k-1}$ are the $k \geq 0$ *temporary variables* of the program and are literal atoms; $i_n, \ldots, i_{n+k-1}$ are the *initial values* of the corresponding temporary variables and are data objects in the state in which the program occurs; and $ins_0, \ldots ins_m$ 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 topmost 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.[3] Thus, if subroutine `MASTER` wants to jump to the 22nd 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, $\texttt{ins}_m$, must be a return or some form of unconditional jump. It is not permitted to "fall off" the end of a Piton program.

The formalization of the concept of a syntactically well-formed Piton program is embodied in the function `PROPER-P-PROGRAMP`. Part of the constraints on proper p-states is that they contain proper Piton programs.

2.1.7 Instructions

The instructions are organized informally into groups. The language as currently defined provides 65 instructions. 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. The instructions are listed in alphabetical order. 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 defintions of the corresponding functions and predicates are given in the Piton manual[23], the interested reader is refered there.

`(ADD-ADDR)` *Well Formedness*: No additional constraints. *Precondition*: There is a natural, n, on top of the stack and a data address, a, immediately below it. The result of incrementing a by n is a legal data address. *Effect*: Pop twice and then push the data address obtained by incrementing a by n.

`(ADD-INT)` *Well Formedness*: No additional constraints. *Precondition*: There is an integer, i, on top of the stack and an integer, j, immediately below it. $j + i$ is representable. *Effect*: Pop twice and then push the integer $j + i$.

`(ADD-INT-WITH-CARRY)` *Well Formedness*: No additional constraints. *Precondition*: There is an integer, i, on top of the stack; an integer, j, immediately below it; and a Boolean, c, below that. *Effect*: 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

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

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 $sum - 2^w$.

(`ADD-NAT`) *Well Formedness*: No additional constraints. *Precondition*: There is a natural, i, on top of the stack and a natural, j, immediately below it. $j + i$ is representable. *Effect*: Pop twice and then push the natural $j + i$.

(`ADD-NAT-WITH-CARRY`) *Well Formedness*: No additional constraints. *Precondition*: There is a natural, i, on top of the stack; a natural, j, immediately below it; and a Boolean, c, immediately below that. *Effect*: Pop three times. Let k be 1 if c is `T` and 0 otherwise. Let *sum* be the natural $i + j + k$. If *sum* is representable in the word size, w, of this p-state, push the Boolean `F` and then natural *sum*; if *sum* is not representable, push the Boolean `T` and the natural $sum - 2^w$.

(`ADD1-INT`) *Well Formedness*: No additional constraints. *Precondition*: There is an integer, i, on top of the stack; and $i + 1$ is representable. *Effect*: Pop once and then push the integer $i + 1$.

(`ADD1-NAT`) *Well Formedness*: No additional constraints. *Precondition*: There is a natural, i, on top of the stack and $i + 1$ is representable. *Effect*: Pop once and then push the natural $i + 1$.

(`AND-BITV`) *Well Formedness*: No additional constraints. *Precondition*: There is a bit vector, v1, on top of the stack and a bit vector, v2, immediately below it. *Effect*: Pop twice and then push the bit vector result of the componentwise conjunction of v1 and v2.

(`AND-BOOL`) *Well Formedness*: No additional constraints. *Precondition*: There is a Boolean, $b1$, on top of the stack and a Boolean, $b2$, immediately below it. *Effect*: Pop twice and then push the Boolean conjunction of $b1$ and $b2$.

(`CALL subr`) *Well Formedness*: `subr` is the name of a program in the program segment. *Precondition*: 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*: Transfer control to the first instruction in the body of `subr` after removing

the topmost 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`.

(`DEPOSIT`) *Well Formedness*: No additional constraints. *Precondition*: There is a data address, a, on top of the stack and an arbitrary object, *val*, immediately below it. *Effect*: Pop twice and then deposit *val* into the location addressed by a.

(`DEPOSIT-TEMP-STK`) *Well Formedness*: No additional constraints. *Precondition*: 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*: 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.

(`DIV2-NAT`) *Well Formedness*: No additional constraints. *Precondition*: There is a natural, i, on top of the stack and room to push at least one more item. *Effect*: Pop once and then push the natural floor of the quotient of i divided by 2 and then push the natural i mod 2.

(`EQ`) *Well Formedness*: No additional constraints. *Precondition*: The temporary stack contains at least two items and the top two are of the same type. *Effect*: Pop twice and then push the Boolean `T` if they are the same and the Boolean `F` if they are not.

(`FETCH`) *Well Formedness*: No additional constraints. *Precondition*: There is a data address, a, on top of the stack. *Effect*: Pop once and then push the contents of address a.

(`FETCH-TEMP-STK`) *Well Formedness*: No additional constraints. *Precondition*: There is a natural number, n, on top of the stack and n is less than the length of the stack. *Effect*: 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*.

(INT-TO-NAT) *Well Formedness*: No additional constraints. *Precondition*: There is a non-negative integer, i, on top of the stack. *Effect*: Pop and then push the natural i.

(JUMP lab) *Well Formedness*: lab is a label in the containing program. *Precondition*: None. *Effect*: Jump to lab.

(JUMP-CASE lab0 lab1 ... labn) *Well Formedness*: Each of the labi is a label in the containing program. *Precondition*: There is a natural, i, on top of the stack and $i \leq$ n. *Effect*: Pop once and then jump to lab$i$.

(JUMP-IF-TEMP-STK-EMPTY lab) *Well Formedness*: lab is a label in the containing program. *Precondition*: None. *Effect*: Jump to lab if the temporary stack is empty.

(JUMP-IF-TEMP-STK-FULL lab) *Well Formedness*: lab is a label in the containing program. *Precondition*: None. *Effect*: Jump to lab if the temporary stack is full.

(LOCN lvar) *Well Formedness*: lvar is a local variable of the containing program. *Precondition*: The value, n, of lvar is a natural number less than the number of locals of the current program. *Effect*: Push the value of the n^{th} local variable of the current program.

(LSH-BITV) *Well Formedness*: No additional constraints. *Precondition*: There is a bit vector, v, on top of the stack. *Effect*: Pop once and then push the bit vector result of left shifting each bit of v, bringing in a 0 on the right.

(LT-ADDR) *Well Formedness*: No additional constraints. *Precondition*: There is a data address, $a1$, on top of the stack and a data address, $a2$, immediately below it. $a1$ and $a2$ address the same data area. *Effect*: Pop twice and then push the Boolean T if $a2 < a1$ (i.e., the position addressed by $a2$ is to the left of that addressed by $a1$) and push the Boolean F otherwise.

(LT-INT) *Well Formedness*: No additional constraints. *Precondition*: There is an integer, i, on top of the stack and an integer, j, immediately below it. *Effect*: Pop twice and then push the Boolean T if $j < i$ and the Boolean F otherwise.

(LT-NAT) *Well Formedness*: No additional constraints. *Precondition*: There is a natural, i, on top of the stack and a natural, j, immediately below it. *Effect*: Pop twice and then push the Boolean T if $j < i$ and the Boolean F otherwise.

(MULT2-NAT) *Well Formedness*: No additional constraints. *Precondition*: There is a natural, i, on top of the stack and $2 * i$ is representable. *Effect*: Pop once and then push the natural $2 * i$.

(MULT2-NAT-WITH-CARRY-OUT) *Well Formedness*: No additional constraints. *Precondition*: There is a natural, i, on top of the stack and room to push at least one more item. *Effect*: Pop once. Then, if the natural $2 * i$ is representable, push the Boolean F and then the natural $2 * i$. Otherwise, push the Boolean T and the natural $2 * i - 2^w$, where w is the word size of this p-state.

(NEG-INT) *Well Formedness*: No additional constraints. *Precondition*: There is an integer, i, on top of the stack and $-i$ is representable. *Effect*: Pop once and then push the integer $-i$.

(NO-OP) *Well Formedness*: No additional constraints. *Precondition*: None. *Effect*: Do nothing; continue execution.

(NOT-BITV) *Well Formedness*: No additional constraints. *Precondition*: There is a bit vector, v, on top of the stack. *Effect*: Pop once and then push the bit vector result of the componentwise logical negation of v.

(NOT-BOOL) *Well Formedness*: No additional constraints. *Precondition*: There is a Boolean, b, on top of the stack. *Effect*: Pop once and then push the Boolean negation of b.

(OR-BITV) *Well Formedness*: No additional constraints. *Precondition*: There is a bit vector, v1, on top of the stack and a bit vector, v2, immediately below it. *Effect*: Pop twice and then push the bit vector result of the componentwise disjunction of v1 and v2.

(OR-BOOL) *Well Formedness*: No additional constraints. *Precondition*: There is a Boolean, b1, on top of the stack and a Boolean, b2, immediately below it. *Effect*: Pop twice and then push the Boolean disjunction of b1 and b2.

(POP) *Well Formedness*: No additional constraints. *Precondition*: There is at least one item on the stack. *Effect*: Pop and discard the top of the stack.

(POP* n) *Well Formedness*: n is a natural number. Note: (POP* 3) is well formed; (POP* (NAT 3)) is not. *Precondition*: there are at least n items on the stack. *Effect*: Pop and discard the topmost n items.

(POP-CALL) *Well Formedness*: No additional constraints. *Precondition*: A subroutine name, subr, is on top of the stack and, after removing that name, it is legal to CALL subr (i.e., sufficient arguments are on the temporary stack and the control stack has room for the new frame). *Effect*: Pop once and then execute (CALL subr).

(POP-GLOBAL gvar) *Well Formedness*: gvar is a global variable, i.e., the name of a data area in the data segment of the containing p-state. *Precondition*: There is an object, *val*, on top of the stack. *Effect*: Pop and assign *val* to the (0^{th} position of the array associated with the) global variable gvar.

(POP-LOCAL lvar) *Well Formedness*: lvar is a local variable of the containing program. *Precondition*: There is an object, *val*, on top of the stack. *Effect*: Pop and assign *val* to the local variable lvar.

(POP-LOCN lvar) *Well Formedness*: lvar is a local variable of the containing program. *Precondition*: The value, n, of lvar is less than the number of local variables of the current program and there is an object, *val*, on top of the stack. *Effect*: Pop and assign *val* to the n^{th} local variable.

(POPJ) *Well Formedness*: No additional constraints. *Precondition*: There is a program counter object, pc, on top of the stack and pc addresses the current program. *Effect*: Pop once and then transfer control to pc.

(POPN) *Well Formedness*: No additional constraints. *Precondition*: There is a natural, n, on top of the stack, and there are at least n items on the stack below it. *Effect*: Pop and discard $n + 1$ items. Thus, to pop n items off the stack, push n onto the stack and execute (POPN).

(PUSH-CONSTANT const) *Well Formedness*: const is either a legal Piton object in the containing p-state, the atom PC, or a label in the containing program. *Precondition*: There is room to push at least one item. *Effect*: 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.

(PUSH-CTRL-STK-FREE-SIZE) *Well Formedness*: No additional constraints. *Precondition*: There is room to push at least one item. *Effect*: Push the natural number indicating how many more cells can be created on the control stack before the maximum control stack size is exceeded.

(PUSH-GLOBAL gvar) *Well Formedness*: gvar is a global variable, i.e., the name of a data area in the data segment of the containing p-state. *Precondition*: There is room to push at least one item. *Effect*: Push the value of the global variable gvar, i.e., the contents of position 0 in the array associated with gvar.

(PUSH-LOCAL lvar) *Well Formedness*: lvar is a local variable of the containing program. *Precondition*: There is room to push at least one item. *Effect*: Push the value of the local variable lvar.

(PUSH-TEMP-STK-FREE-SIZE) *Well Formedness*: No additional constraints. *Precondition*: There is room to push at least one item. *Effect*: Push the natural number indicating how many more cells can be created on the temporary stack before the maximum temporary stack size is exceeded.

(PUSH-TEMP-STK-INDEX n) *Well Formedness*: n is a natural number. Note: n here must not be tagged; (PUSH-TEMP-STK-INDEX 3) is well formed and (PUSH-TEMP-STK-INDEX (NAT 3)) is not. *Precondition*: n is less than the length of the temporary stack, and there is room to push at least one item. *Effect*: 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 topmost-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.

(PUSHJ lab) *Well Formedness*: lab is a label in the containing program. *Precondition*: There is room to push at least one item. *Effect*: Push the program counter addressing the next instruction and then jump to lab.

(RET) *Well Formedness*: No additional constraints. *Precondition*: None. *Effect*: 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 topmost frame of the control stack and pop that frame off the control stack.

(RSH-BITV) *Well Formedness*: No additional constraints. *Precondition*: There is a bit vector, v, on top of the stack. *Effect*: Pop once and then push the bit vector result of right shifting each bit in v, bringing in a 0 on the left.

(SET-GLOBAL gvar) *Well Formedness*: gvar is a global variable, i.e., the name of a data area in the data segment of the containing p-state. *Precondition*: There is an object, *val*, on top of the stack. *Effect*: Assign *val* to the (0^{th} position of the array associated with the) global variable gvar. The stack is not popped.

(SET-LOCAL lvar) *Well Formedness*: lvar is a local variable in the containing program. *Precondition*: There is an object, *val*, on top of the stack. *Effect*: Assign *val* to the local variable lvar. The stack is not popped.

(SUB-ADDR) *Well Formedness*: No additional constraints. *Precondition*: There is a natural, n, on top of the stack and a data address, a, immediately below it. The result of decrementing a by n is a legal data address. *Effect*: Pop twice and then push the data address obtained by decrementing a by n.

(SUB-INT) *Well Formedness*: No additional constraints. *Precondition*: There is an integer, i, on top of the stack and an integer, j, immediately below it. $j - i$ is representable. *Effect*: Pop twice and then push the integer $j - i$.

(SUB-INT-WITH-CARRY) *Well Formedness*: No additional constraints. *Precondition*: There is an integer, i, on top of the stack; an integer, j, immediately below it; and a Boolean, c, below that. *Effect*: 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 $diff - 2^w$.

(SUB-NAT) *Well Formedness*: No additional constraints. *Precondition*: There is a natural, i, on top of the stack and natural, j, immediately below it. Furthermore, $j \geq i$. *Effect*: Pop twice and then push the natural $j - i$.

(SUB-NAT-WITH-CARRY) *Well Formedness*: No additional constraints. *Precondition*: There is a natural, i, on top of the stack; a natural, j, immediately below it; and a Boolean, c, immediately below that. *Effect*: Pop three times. Let k be 1 if c is T and 0 otherwise. If $j \geq i + k$, then push the Boolean F and the natural $j - (i + k)$. Otherwise, push the Boolean T and the natural $2^w - ((i + k) - j)$, where w is the word size of this p-state.

(SUB1-INT) *Well Formedness*: No additional constraints. *Precondition*: There is an integer, i, on top of the stack, and $i - 1$ is representable. *Effect*: Pop once and then push the integer $i - 1$.

(SUB1-NAT) *Well Formedness*: No additional constraints. *Precondition*: There is a non-zero natural, i, on top of the stack. *Effect*: Pop and then push the natural $i - 1$.

(TEST-BITV-AND-JUMP test lab) *Well Formedness*: test is either ALL-ZERO or NOT-ALL-ZERO, and lab is a label in the containing program. *Precondition*: There is a bit vector, v, on top of the stack. *Effect*: Pop once and then jump to lab if test is satisfied, as indicated below.

test	condition tested
ALL-ZERO	every component of v is 0
NOT-ALL-ZERO	some component of v is 1

(TEST-BOOL-AND-JUMP test lab) *Well Formedness*: test is either T or F and lab is a label in the containing program. *Precondition*: There is a Boolean, b, on top of the stack. *Effect*: Pop once and then jump to lab if test is satisfied, as indicated below.

test	condition tested
T	b = T
F	b = F

(TEST-INT-AND-JUMP test lab) *Well Formedness*: test is one of NEG, NOTNEG, ZERO, NOT-ZERO, POS or NOT-POS, and lab is a label in the containing program. *Precondition*: There is an integer, i, on top of the stack. *Effect*: 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-NAT-AND-JUMP test lab) *Well Formedness*: test is either ZERO or NOT-ZERO, and lab is a label in the containing program.[4] *Precondition*: There is a natural, n, on top of the stack. *Effect*: Pop once and then jump to lab if test is satisfied, as indicated below.

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

(XOR-BITV) *Well Formedness*: No additional constraints. *Precondition*: There is a bit vector, v1, on top of the stack and a bit vector, v2, immediately below it. *Effect*: Pop twice and then push the bit vector result of the componentwise exclusive-or of v1 and v2.

2.1.8 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.

[4] Technically, test may be anything whatsoever. If it is not ZERO, it is treated as though it were NOT-ZERO.

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.[5] 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`. `(P s n)` is the p-state obtained by executing `n` instructions starting in p-state `s`.

2.1.9 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 when we present our correctness theorem. 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 be 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.

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

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? 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.

Chapter 3

Informal Description of the Implementation

The compiler is defined by the function LOGIC->P, which takes seven arguments: EXPR, ALIST, PROGRAM-NAMES, HEAP-SIZE, MAX-CTRL, MAX-TEMP and WORD-SIZE. It produces a Piton state as its output. EXPR and ALIST are the expression and variable binding association list of the Nqthm interpreter V&C$. PROGRAM-NAMES is a list of symbols that represent functions in the Logic to be compiled. MAX-CTRL, MAX-TEMP and WORD-SIZE correspond to the Piton resource limitations. HEAP-SIZE is the maximum number of "CONS" cells that can be allocated.

Each user-defined Nqthm function is compiled into a Piton function. Each Nqthm predefined SUBR has a corresponding Piton function (except for IF, which is compiled in line). There are a few additional internal Piton functions called by the SUBRS.

3.1 An Example

The ultimate goal of this research is to produce a formally specified and mechanically verified implementation of the Logic. The implementation consists of a compiler for functions in the Logic and some run-time support functions. The specification requires the implementation preserve the semantics of the Logic as defined by the Nqthm function V&C$. V&C$ is described by Boyer and Moore in [5]. V&C$ is similar to the Lisp interpreter EVAL, except that instead of producing just a value for an expression, it produces a value and the cost of evaluating the expression. It takes as arguments an expression, EXPR and an association list, ALIST that gives values for the free variables in EXPR. It produces as a result either a value-cost pair or F. A value-cost pair is a CONS, the CAR of which is the value of the expression EXPR, and the CDR is the cost of obtaining it. The cost is the number of function applications involved in evaluating EXPR. V&C$ returns F if the

```
Expression: (APP (CONS X (CHANGE-ELEMENTS Y)) '(*1*TRUE . *1*FALSE))

Variable Alist: (LIST (CONS 'X F) (CONS 'Y (CONS T (CONS T F))))

DEFINITION
(APP X Y)
   =
(IF (LISTP X)
    (CONS (CAR X) (APP (CDR X) Y))
    Y)

DEFINITION
(CHANGE-ELEMENTS LIST)
   =
(IF (LISTP LIST)                                    ;[0]
    (IF (TRUEP (CAR LIST))                          ;[1]
        (CONS F (CHANGE-ELEMENTS (CDR LIST)))       ;[2]
        (CONS T                                     ;[3]
              (CHANGE-ELEMENTS (CDR LIST))))        ;[4]
    (IF (TRUEP LIST) F T))                          ;[5]
```

Figure 3.1: An example program

cost is infinite. A nonconstructive axiomatization of V&C$ is part of the standard Nqthm Logic.

Figure 3.1 shows a simple expression and a variable binding association list. The definitions of the functions APP and CHANGE-ELEMENTS are also shown in Figure 3.1. This example is used in this chapter to illustrate various aspects of the compiler.[1] The complete result of compiling the expression is shown in Appendix A.

3.2 Representing Data in Piton

The Logic has several different shells or "types", such as natural numbers, ordered pairs and literal atoms, as well as user-defined shells. Objects of all these different types are represented in the Piton data segment.

The Logic has six built-in shells: the natural numbers (NUMBERPs), the ordered pairs (LISTPs), the literal atoms or words (LITATOMs), the negative integers (NEGATIVEPs), TRUE or T and FALSE or F. In addition, the user can define more shells.

The Logic provides functions for determining the "type" of an object (e.g. LISTP and LITATOM). In order to implement these functions it must be possible to determine at run-time the type of an object from its representation. A garbage collector needs to determine the type of an object in order to find out if it is in

[1] The current version of the compiler only has the data-types TRUE, FALSE and CONS; the accessors CAR and CDR; the recognizers TRUEP, FALSEP and LISTP; and the abbreviation NLISTP.

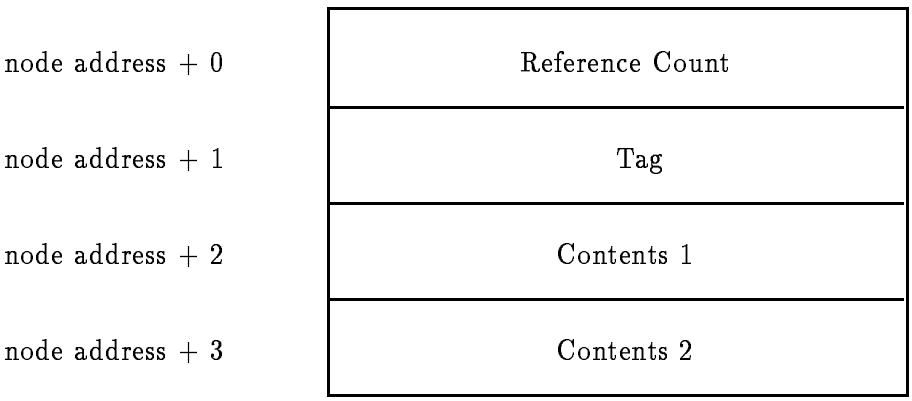

Figure 3.2: The structure of a *Node*

use. A tag encoding the type of an object is stored in the representation of each object.

All objects in the Logic are represented by Piton data addresses that point into the Piton data area `HEAP`. In this document, *Heap address n* is to mean the Piton data address `(ADDR (HEAP .n))`. Each data type is represented by one or more *nodes* in the heap. A *node* consists of four consecutive words in the heap. The address of a *node* is the address of the first word of the node and is always evenly divisible by four. The structure of a node is illustrated in Figure 3.2.

The first word in a *node* is a Piton natural number that is the reference count. The reference count is one less than the number of pointers to the node. A reference count of 0 means that there is exactly one pointer to the node. The second word contains a natural number that indicates the type of the node. Every different shell that is compiled has a different tag here. All shells of the same type have the same tag. The third and fourth words are used to represent the contents of the node. The contents of a node are Piton data addresses pointing to other nodes, or, when representing `NUMBERP`s, the first contents field contains a natural. Some data types require more than two content words. In these cases the last word of the node points to another node that contains up to three more words of content and (if necessary) a pointer to another node of a similar form.

The compiler currently allocates four areas in the data-segment:

1. `FREE-PTR`, an area of length one (i.e. a variable). The `FREE-PTR` contains the heap address of the beginning of the free list.

2. `ANSWER`, an area of length one (i.e. a variable). This is where the final answer is stored. The answer will be a heap address.

3. `HEAP`, a large area, the size of which is determined by `HEAP-SIZE`, a parameter to `LOGIC->P`. All objects are allocated in this area.

4. ALLOC-JUMP-TABLE, an array of program addresses. This is used to jump to the proper place in the routine I-ALLOC-NODE, depending on the type of the first node on the free list.

When user-defined types are added to the compiler another area will be needed.

5. SHELL-SIZES, an array of naturals that gives the number of accessors for different shells. This array is the same length as ALLOC-JUMP-TABLE and is indexed by the tag of a node. This is used by the Piton code for EQUAL and COUNT, which must be recursively called on the objects stored in each accessor.

The data segment that results from compiling the example of Figure 3.1 is:

```
'((FREE-PTR (ADDR (HEAP . 24)))
  (ANSWER (NAT 0))
  (ALLOC-NODE-JUMP-TABLE ((PC (I-ALLOC-NODE . 8))
                          (PC (I-ALLOC-NODE . 9))
                          (PC (I-ALLOC-NODE . 8))
                          (PC (I-ALLOC-NODE . 9))
                          (PC (I-ALLOC-NODE . 14))
                          (PC (I-ALLOC-NODE . 41))
                          (PC (I-ALLOC-NODE . 59))
                          (PC (I-ALLOC-NODE . 60))))
  (HEAP (NAT 0)              ; Undefined node [Heap address 0]
        (NAT 0)                ; Tag of 0 = Undefined tag
        (ADDR (HEAP . 0))
        (ADDR (HEAP . 0))
        (NAT 4)              ; FALSE  [Heap address 4]
        (NAT 2)                ; Tag of 2 = False
        (ADDR (HEAP . 0))
        (ADDR (HEAP . 0))
        (NAT 4)              ; TRUE  [Heap address 8]
        (NAT 3)                ; Tag of 3 = TRUE
        (ADDR (HEAP . 0))
        (ADDR (HEAP . 0))
        (NAT 0)              ; ZERO  [Heap address 12]
        (NAT 4)                ; Tag of 4 = NUMBERP
        (NAT 0)                ; Contents: 0
        (ADDR (HEAP . 0))
        (NAT 1)              ; (CONS T F)  [Heap address 16]
        (NAT 5)                ; Tag of 5 = CONS
        (ADDR (HEAP . 8))      ; CAR = Heap Address 8, represents T
        (ADDR (HEAP . 4))      ; CDR = Heap Address 4, represents F
        (NAT 0)              ; (CONS T (CONS T F))  [Heap address 20]
        (NAT 5)                ; Tag of 5 = CONS
        (ADDR (HEAP . 8))      ; CAR = Heap Address 8, represents T
        (ADDR (HEAP . 16))     ; CDR = Heap Address 16
        (ADDR (HEAP . 28))  ;  Unused node  [Heap address 24]
        (NAT 1)                ; Tag of 1 = node never used
        (NAT 0)
        (NAT 0)
        (ADDR (HEAP . 32))     ;  Unused node  [Heap address 28]
        (NAT 1)
```

```
(NAT 0)
(NAT 0)
(ADDR (HEAP . 36))     ;  Unused node  [Heap address 32]
(NAT 1)
(NAT 0)
(NAT 0)
(ADDR (HEAP . 40))     ;  Unused node  [Heap address 36]
(NAT 1)
(NAT 0)
(NAT 0)
(ADDR (HEAP . 44))     ;  Unused node  [Heap address 40]
(NAT 1)
(NAT 0)
(NAT 0)
(ADDR (HEAP . 48))     ;  Unused node  [Heap address 44]
(NAT 1)
(NAT 0)
(NAT 0)
(NAT 1)))              ;  last node on free list  [Heap address 48]
```

The comments above indicate what object in the Logic is represented at that address. There are six nodes on the free list. The variable FREE-PTR contains the address of the first node on the free list (heap address 24). The nodes on the free list are heap addresses 24, 28, 32, 36, 40, and 44. The final address in the heap is heap address 48. If an attempt is made to allocate this last node, the Piton interpreter will return a state with the error flag set. In order to use the correctness theorem about the compiler on a particular program, it is necessary to prove that enough heap space is allocated so that this error does not occur.

The first few addresses in HEAP are used to store a few constants. Heap address 0 contains the undefined node. This is a node that is guaranteed not to be the result of evaluating any form in the Logic. The object F is always represented by the heap address 4. To test whether a value is F, it is only necessary to test if it is equal to heap address 4.

The following lists the format of all data-types. Each of the types below is associated with a different tag.

1. *Undefined*: There is only one node with this type. The contents for nodes of this type are ignored, but have been arbitrarily set to heap address 0, i.e., the undefined node. The undefined node is used as an address that is known to be a node, but is *not* the address of the representation of any object in the Logic.

2. *Unused*: These are nodes on the free list that have never been used.

3. FALSEP: The content for nodes of type FALSEP are ignored, but have been arbitrarily set to heap address 0, i.e., the undefined node. There is only one representation of FALSE in HEAP.

4. TRUEP: The content for nodes of TRUEP type are ignored, but have been arbitrarily set to heap address 0, i.e., the undefined node. There is only one

representation of TRUE in the HEAP data area.

5. NUMBERP: The first content cell contains a natural number and the second contains an address pointing to another node. *FIXNUMs* are nodes where the second content cell points to the undefined node. *BIGNUMs* are represented with the least significant part in the first content. The second content cell points to another node in which the tag word, the reference count word, and first content word are used to represent the number, and the second content word contains a pointer to another node of this kind or the undefined node. The compiler only creates *FIXNUMs*. The format of *BIGNUM* is defined so that *BIGNUM* can be added and only the routines that manipulate them need to be changed.[2]

6. LISTP: Both the first and second content words contain addresses pointing to nodes. The first content word is the CAR and the second the CDR.

7. LITATOM:[3] The first content cell contains an address pointing to the UNPACK of the LITATOM. The second content cell points to another node. This node is used to store the SUBRP, BODY and FORMALS values for the literal atom. The actual format is that the second node uses the tag field to store the value that SUBRP should return. The reference count field and the first content field are used to form a doubly linked list of LITATOMs (see the next paragraph). The second content field points to yet another node. The third node stores the BODY and FORMALS values for the LITATOM in the first and second content field. A LITATOM requires three nodes. [4]

   LITATOMs are stored uniquely. They are stored in a very simple hash table. The LITATOM constructor PACK hashes on the first letter of the object it is given. The hash table has 27 buckets, one for each letter in the alphabet, plus another bucket for LITATOMs that do not go in the other 26 buckets. PACK takes the object it is given and if it is not a list puts it in the *other* bucket. If it is a list and the CAR is a natural number between 65 (the ASCII value for *A*) and 90 (the ASCII value for *Z*) then it is put in the bucket corresponding to that value. Otherwise, it is put in the *other* bucket. Once the bucket is determined, we search for a LITATOM that if UNPACKed is equal to the current object. If one is found, then the address of that LITATOM is returned; otherwise the new LITATOM is added at the end of the current bucket. Note that if there are no more pointers to a LITATOM it must be removed from the hash table before being reused.

[2]The current version of the compiler only compiles NUMBERPs inside constants and requires that they be representable as *FIXNUMs*. There are no functions to manipulate NUMBERPs in the compilable subset. However, 0 can be returned by CAR and CDR.

[3]Literal atoms are not implemented at all in the current version of the compiler.

[4]Note: Only LITATOMs corresponding to the names of programs compiled by the compiler will have BODY and FORMALS and SUBRP defined. This is not a significant drawback, as any names needed can be added to the list of program names passed as an argument to the compiler.

8. NEGATIVEP: The first content cell contains a pointer to another node (which should be a NUMBERP cell). The second content cell is not used but is initialized anyway to the address of the undefined node.[5]

9. User-defined shells:[6] If a type has more than two accessors, then more than one node is allocated for an object of that type. The first content field of the first node contains the representation of the first accessor. The second content field points to another node. The subsequent nodes, called continuation nodes, use the tag, reference count, and first content field to store accessors and the second content field to point to a continuation node.

3.3 The Garbage Collector

The implementation of the Logic includes a reference count garbage collector. Every *node* in the heap includes a reference count field. The reference count field for all nodes that are in use contains a Piton natural that indicates the number of pointers to the *node*. The number stored in the reference count field is actually one less than the number of pointers to the *node*. Pointers to nodes can be in the Piton program segment, data segment, temporary stack, and control stack. The program segment has pointers to the representation of quoted constants. The data segment contains pointers to various objects. For instance, a node in the data segment that represents (CONS x y) contains pointers to x and y. The temporary stack contains pointers to intermediate results. Finally, the control stack holds the formal parameters of functions. The control stack can also be used to store saved values of previously computed expressions. This can be used to eliminate common sub-expressions; see Section 3.5.

Every time the value of a variable or constant is pushed on the temporary stack, its reference count must be incremented. When values are removed from the temporary or control stack, the reference counts must be decreased.

Unused nodes are stored in a linked list. The Piton variable FREE-PTR points to the beginning of this free list. Each node is still tagged with its type, and the reference count field is used as a link to the next node. A proper free list is specified by the Nqthm function LR-PROPER-FREE-LISTP (see section 6.2.1). LR-PROPER-FREE-LISTP examines the reference count field of nodes to determine whether the node is on the free list (the reference count is NOT a Piton natural) or if it is not on the free list (the reference count is a Piton natural). Piton does not allow programs to determine the type of an object at run-time, so run-time programs cannot tell if the reference count field is a Piton natural or not. The compiler must ensure that all the nodes in the linked list do not have Piton naturals in their reference count fields. LR-PROPER-FREE-LISTP allows nodes that are not on the free list to be in use or garbage. However, the garbage collection code

[5]Negative numbers are not implemented in the current version of the compiler.
[6]User-defined shells are not implemented at all in the current version of the compiler.

ensures that all garbage is collected. This last property of the garbage collection code is not proven. The proof of the garbage collection code ensures that any nodes collected are not in use, but it does not verify that all nodes that are garbage are eventually collected.[7]

The garbage collector has two internal Piton functions.

1. I-ALLOC-NODE removes a node from the free list and returns the address of the node on the top of the temporary stack. I-ALLOC-NODE calls I-DECR-REF-COUNT, for instance, when removing an unused CONS cell from the free list, to decrement the reference counts of the pointers in CAR and CDR fields of the node.

2. I-DECR-REF-COUNT takes a pointer on top of the temporary stack and decrements the reference count of the object to which the pointer points. The pointer is returned on the top of the temporary stack. If the reference count is zero it returns the object to the free list. When a node is returned to the free list the reference count field of the node being returned is set to point to the first node on the free list and FREE-PTR is set to point to the node added.

The following example illustrates how the functions I-ALLOC-NODE and I-DECR-REF-COUNT work. Suppose Piton is executing a call to CONS with the temporary stack and data segment as shown in Figure 3.3 (only the top few entries on the temporary stack and the part of the data segment being manipulated are shown). The Piton code for CONS is:

```
;; CONS Takes two implicit args, the CDR is on top of
;; the stack and the CAR is just below it.
(CONS () ((TEMP (ADDR (HEAP . 0))))
      ;; First call I-ALLOC-NODE to get a new node.
      (CALL I-ALLOC-NODE)
      (SET-LOCAL TEMP)        ; Put CDR in node + CDR-OFFSET
      (PUSH-CONSTANT (NAT 3))
      (ADD-ADDR)
      (DEPOSIT)
      (PUSH-LOCAL TEMP)       ; Put CAR in node + CAR-OFFSET
      (PUSH-CONSTANT (NAT 2))
      (ADD-ADDR)
      (DEPOSIT)
      (PUSH-CONSTANT (NAT 5)) ; Put tag in node
      (PUSH-LOCAL TEMP)
      (PUSH-CONSTANT (NAT 1))
      (ADD-ADDR)
      (DEPOSIT)
      (PUSH-CONSTANT (NAT 0)) ; Set ref count to 0
      (PUSH-LOCAL TEMP)
      (DEPOSIT)
      (PUSH-LOCAL TEMP)
      (RET))
```

[7]The mechanically checked proof of this property of the garbage collector has not been completed.

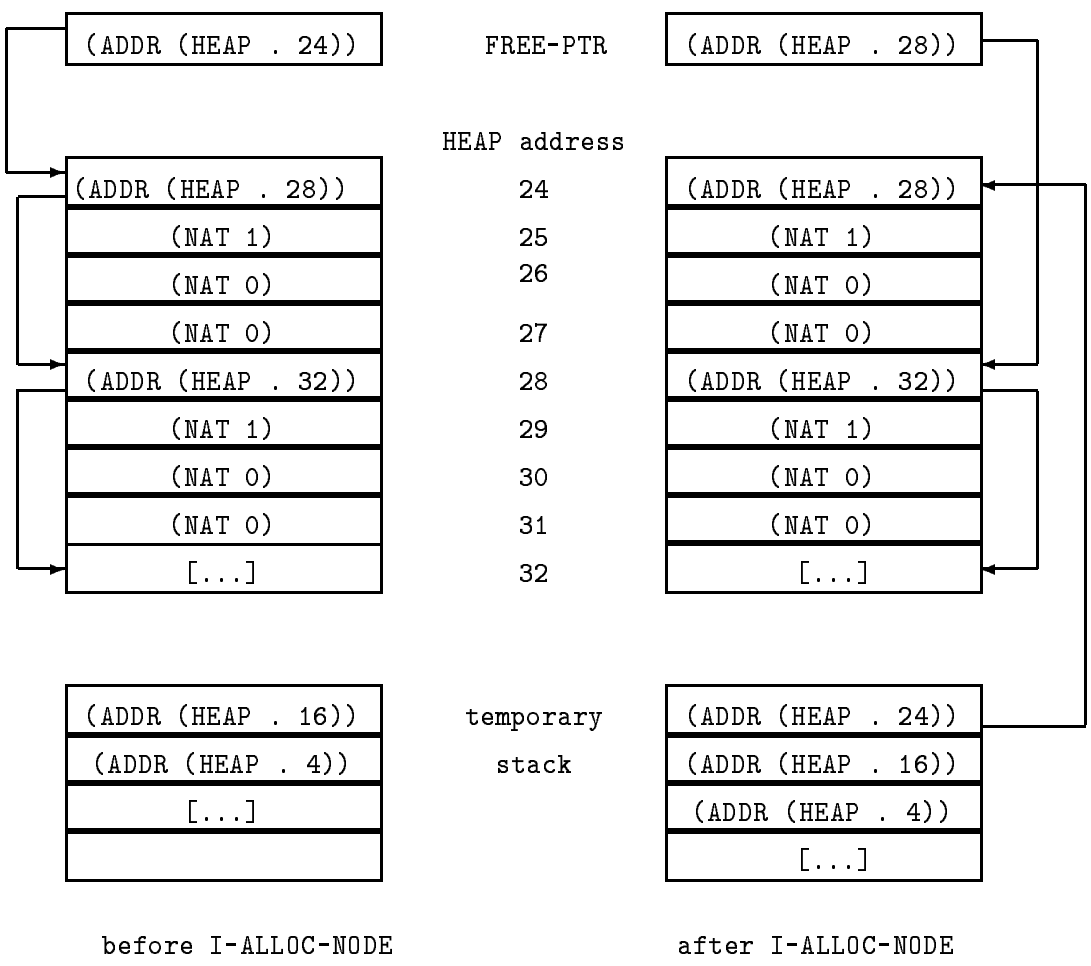

Figure 3.3: The effects of executing `I-ALLOC-NODE`

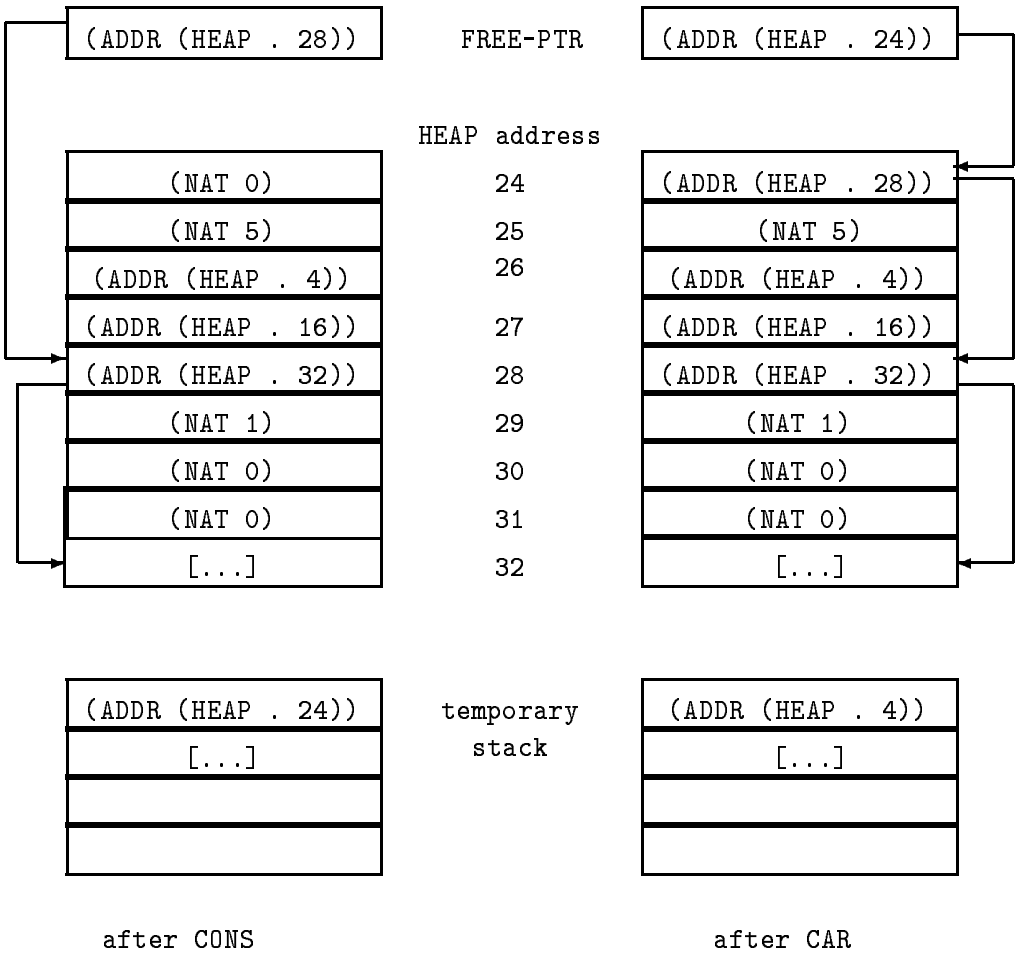

Figure 3.4: The effects of executing CONS and CAR

First CONS calls I-ALLOC-NODE. After control returns from the call to I-ALLOC-NODE, the data segment and temporary stack are as shown in Figure 3.3. The top of the temporary stack contains the address (ADDR (HEAP . 24)), which is a pointer to a new node that has been removed from the free list. The rest of the CONS code fills in the new node. The temporary stack is popped and the address that was on top of it is put in the CDR field of the node. The temporary stack is popped again and the CAR field filled. The tag field is set to (NAT 5) to indicate that this is a CONS node. Finally, (NAT 0) is put in the reference field to indicate there is one pointer to this node. The resulting data segment and temporary stack are shown (in part) in Figure 3.4.

Now suppose a call to CAR is executed on the Piton state resulting from the call to CONS. The code for CAR is:

```
(CAR (X) ((TEMP (NAT 0)))
         ;; First decrement ref count of argument, X
         (PUSH-LOCAL X)
         (CALL I-DECR-REF-COUNT)
         ;; Test if X is a CONS, i.e. has a tag of (NAT 5)
         (PUSH-CONSTANT (NAT 1))
         (ADD-ADDR)
         (FETCH)
         (PUSH-CONSTANT (NAT 5))
         (EQ)
         (TEST-BOOL-AND-JUMP T ARG1)
         ;; Not a CONS, return address representing 0 first
         ;; incrementing ref count.
         (PUSH-CONSTANT (ADDR (HEAP . 12)))
         (FETCH)
         (ADD1-NAT)
         (PUSH-CONSTANT (ADDR (HEAP . 12)))
         (DEPOSIT)
         (PUSH-CONSTANT (ADDR (HEAP . 12)))
         (RET)
         ;; Type is the same, return CAR field first
         ;; incrementing REF COUNT
         (DL ARG1 () (PUSH-LOCAL X))
         (PUSH-CONSTANT (NAT 2))
         (ADD-ADDR)
         (FETCH)
         (SET-LOCAL TEMP)
         (FETCH)
         (ADD1-NAT)
         (PUSH-LOCAL TEMP)
         (DEPOSIT)
         (PUSH-LOCAL TEMP)
         (RET))
```

First CAR pushes its argument X and calls I-DECR-REF-COUNT. Since the address (ADDR (HEAP . 24)) has exactly one pointer to it, I-DECR-REF-COUNT puts it back on the free list. When I-DECR-REF-COUNT returns, the address (ADDR (HEAP . 24)) is left on the top of the temporary stack. CAR then tests if

the address is a CONS node. Since it is, the address in the CAR field of the node is fetched. The address points to a node, the reference count of which is incremented. Finally the address is returned. The resulting data segment and temporary stack are shown (in part) in Figure 3.4. An important point is that after returning from the call to CAR the reference count of the address (ADDR (HEAP . 4)) is at least two. There is one pointer on the top of the temporary stack and a second pointer in the CAR field (heap address 26) of the node at heap address 24. This is true even though the node at heap address 24 is on the free list. Not until that node is allocated will the reference count of (ADDR (HEAP . 4)) be decremented.

3.4 Compiling from the Logic to Piton

Each user-defined function is independently compiled to Piton. Piton programs have four components: name, formals, temporary variable declarations, and body. The name of the Piton function has a U- prefixed to the name of the function in the Logic. The formal variables of the Piton function are the same as the Logic function FORMALS returns on the function name. The Piton temporary variable declarations are used for holding intermediate results. Originally the compiler was to have a routine to eliminate common sub-expressions. This has not been implemented but could be added without changing the rest of the compiler. See Section 3.5 for more details. The body is obtained by recursively compiling the form returned by applying BODY to the function name and appending a postlude.

The following sections demonstrate how to compile various constructs in the Logic into Piton. An example is the function CHANGE-ELEMENTS from figure 3.1, the Piton translation of which is in Appendix A.

3.4.1 Compiling Variable References

A variable reference is a reference to one of the formal parameters of the function (e.g., the variable LIST is referenced on lines marked [0], [1], [2], [4] and [5] in CHANGE-ELEMENTS). When a function is called, its formals are bound to the results of evaluating the actual parameters. Thus, each formal is bound to a heap address. Variable references push the address that the variable is bound to on the temporary stack. This adds an additional reference to that heap address, so the reference count must be incremented.

The translation of the reference to LIST on line 0 of CHANGE-ELEMENTS is shown here:

```
(DL L-0 () (PUSH-LOCAL LIST))    ;  put reference count address on stack
(DL L-1 () (FETCH))              ;  fetch reference count
(DL L-2 () (ADD1-NAT))
(DL L-3 () (PUSH-LOCAL LIST))    ;  put reference count address on stack
(DL L-4 () (DEPOSIT))            ;  update reference count
(DL L-5 () (PUSH-LOCAL LIST))    ;  put node address on stack
```

The translations of additional references to LIST are the Piton instructions labeled L-11 to L-16, L-29 to L-34, L-45 to L-50 and L-55 to L-60 in the code in figure A.

3.4.2 Compiling Constants

A constant is translated into a Piton data address at which the constant is stored. The constant T is referenced on the lines labeled [2] and [5] of CHANGE-ELEMENTS. The constant F is referenced on the lines labeled [3] and [5]. To reference a constant, the address that represents the constant must be pushed on the temporary stack. Since this increases the references to that address, the reference count must be incremented as well. The translation of the constant F on line [2] is shown below:

```
(DL L-23 ()                      ;  Put reference count address on stack
   (PUSH-CONSTANT (ADDR (HEAP . 4))))
(DL L-24 () (FETCH))             ;  fetch reference count
(DL L-25 () (ADD1-NAT))          ;  add one to it
(DL L-26 ()                      ;  Put reference count address on stack
    (PUSH-CONSTANT (ADDR (HEAP . 4))))
(DL L-27 () (DEPOSIT))           ;  update reference count
(DL L-28 ()                      ;  Put T 's address on stack
    (PUSH-CONSTANT (ADDR (HEAP . 4))))
```

The translations of the references to the constant T on lines marked [3] and [5] of CHANGE-ELEMENTS are given by the instructions labeled L-39 to L-44, L-66 to L-71, respectively. The translation of F on the line marked [5] of CHANGE-ELEMENTS is given by the instructions labeled L-73 to L-78. In the example Piton state of Appendix A the constant '(*1*TRUE . *1*FALSE) is represented by the Piton address (ADDR (HEAP . 16)). The code to reference this constant is given by the instructions L-14 to L-19 of the program MAIN. The relevant parts of the Piton state are:

```
(P-STATE '(PC (MAIN . 0))
         '((((X ADDR (HEAP . 4))
             (Y ADDR (HEAP . 20)))
            (PC (MAIN . 0))))
         ()
         '((MAIN (X Y) ()
                 [...]
                 ;; Push address of constant: '(*1*TRUE . *1*FALSE)
                 ;; and increment ref count
                 (DL L-14 () (PUSH-CONSTANT (ADDR (HEAP . 16))))
                 (DL L-15 () (FETCH))
                 (DL L-16 () (ADD1-NAT))
                 (DL L-17 () (PUSH-CONSTANT (ADDR (HEAP . 16))))
                 (DL L-18 () (DEPOSIT))
                 (DL L-19 () (PUSH-CONSTANT (ADDR (HEAP . 16))))
                 [...])
           [...])
```

```
'((FREE-PTR (ADDR (HEAP . 24)))
  (ANSWER (NAT 0))
  (ALLOC-NODE-JUMP-TABLE [...])
  (HEAP [...]
        ;; (CONS T F)  [Heap address 16]
        (NAT 1)
        (NAT 5)
        (ADDR (HEAP . 8))
        (ADDR (HEAP . 4))
        [...]))
20 20 32 'RUN)
```

The compiler represents all the quoted constants in the functions being compiled in the initial Piton data segment. The bindings of variables in the initial variable association list are also represented in the initial Piton data segment. These constants share as much structure as possible. This is the only significant optimization performed by the compiler.

3.4.3 Compiling Function Calls

Consider the compilation of a function call, for example, (`f` $\texttt{arg}_0$ ...$\texttt{arg}_n$). First the arguments $\texttt{arg}_0$ through $\texttt{arg}_n$ are compiled, concatenating the code together. Then if `f` is a user-defined function, the Piton instruction (`CALL U-f`) is appended. If `f` is a predefined Nqthm functions, (`CALL f`) is appended. The compilation of (`CONS F (CHANGE-ELEMENTS (CDR LIST))`) on the line marked [2] of Figure A is given by the Piton instructions labeled `L-23` to `L-37`. First translate the constant `F`, which is given by lines labeled `L-23` to `L-28`. Next compile (`CHANGE-ELEMENTS (CDR LIST)`) by first compiling (`CDR LIST`) (instructions labeled `L-29` and `L-35`) and then add the call to `U-CHANGE-ELEMENTS` (instruction `L-36`). Finally, we add the call to `CONS` at instruction `L-37`. The names of all user-defined functions are prefixed with `U-`. This is so we can have internal functions (that are prefixed with `I-`) and not conflict with the names of user functions. In addition, this ensures that the name for the program `MAIN` is different from the names of user functions.

A Piton `CALL` instruction pops objects off of the Piton temporary stack and binds the formal parameters of the called functions to the objects. The reference counts of the objects on the temporary stack do not need to be changed, since after the `CALL` instruction is finished the objects are on the control stack.

Each predefined Logic `SUBRP` (with the exception of `IF`) is translated to a corresponding Piton function.

3.4.4 Compiling IF

`IF` is the most complex construct to compile. In the execution of an `IF`, (`IF test then else`) first the code for `test` is executed. If `test` returns `F`, then the code for `else` is executed; otherwise the code for `then` is executed. After `test`

is executed there is a heap address on top of the stack; this is popped off while testing whether it is F or not. This decreases the number of references to the address, so the reference count must also be decremented before executing then or else.

An example should clarify the compilation of IF. Consider the IF that starts on the line marked [0] of figure A. The translation is shown below:

```
(IF (LISTP LIST) [...] [...])

    (DL L-0 () (PUSH-LOCAL LIST))        ; Put LIST on stack
    (DL L-1 () (FETCH))                  ; Increment ref count of LIST
    (DL L-2 () (ADD1-NAT))
    (DL L-3 () (PUSH-LOCAL LIST))
    (DL L-4 () (DEPOSIT))
    (DL L-5 () (PUSH-LOCAL LIST))
    (DL L-6 () (CALL LISTP))             ; Call LISTP
    (DL L-7 () (CALL I-DECR-REF-COUNT)) ; Decrement ref. count of result
    (DL L-8 ()                           ; Put F address on stack
        (PUSH-CONSTANT (ADDR (HEAP . 4))))
    (DL L-9 () (EQ))                     ; test if result is F
    (DL L-10 ()                          ; jump to L-55 for else part
        (TEST-BOOL-AND-JUMP T L-55))
    [...]                                ; code for then part
    (DL L-54 () (JUMP L-79))             ; jump around else code
    (DL L-55 () [...])                   ; code for else part
    [...]
    (DL L-79 () [...])                   ; end of IF
```

The code for the test (LISTP LIST) is shown in the instructions labeled L-0 through L-6. The instructions L-7 through L-10 test if the result of (LISTP LIST) is F and then jump to the beginning of the else code, L-55 if so. The instructions labeled L-11 through L-54 are the compilation of the then part of the IF. The instruction (DL L-54 () (JUMP L-79)) jumps around the code for the else part of the IF. The instructions for the else part of the IF are labeled L-55 through L-78.

3.4.5 The Postlude

When a function returns, the control stack is popped, which removes the top call frame. A Piton call frame contains the values of the formal and temporary variables of the function. Popping the control stack removes pointers to the representation of objects, thus decreasing the reference counts of some *nodes*. The postlude of a function keeps the reference counts correct. The postlude of CHANGE-ELEMENTS from figure A is shown below.

```
(DL L-79 () (PUSH-LOCAL LIST))       ;  Put node address on stack
(DL L-80 () (CALL I-DECR-REF-COUNT)) ;  Call routine to decrement ref. count
(DL L-81 () (POP))                   ;  Pop result off stack
(DL L-82 () (RET))                   ;  return to caller
```

3.5 Optimizations

The compiler does not do much in the way of optimizations. There are two optimizations done by the compiler: constants share structure, and the constant `F` is uniquely represented, which makes the compilation `IF` a little better.

All the quoted constants in the compiled programs, as well as the expression and bindings in the variable alist, share as much structure as possible. The function `LR-COMPILE-QUOTE` (given in Appendix B on page 121) takes four arguments: a flag, an object, a data segment, and a table. It returns a pair: the new data segment with the object allocated in it and a new table in which the object is associated with the address to which it has been assigned. `LR-COMPILE-QUOTE` is called on every quoted object in the expression, in the bindings of the variable alist and in the programs being compiled. In order to make objects share as much structure as possible, `LR-COMPILE-QUOTE` first looks to see if the object is associated with an address in the table.

There is only one data address that is used to represent `F`. This allows the code generated for `(IF test then else)` to test if the result returned by `test` is `F`, by checking if the address returned by the code for `test` is the address for `F`. The alternative would be to fetch the tag of the result and see if it is the tag for `F`. Using the first approach saves a few Piton instructions when compiling `IF`.

There are provisions in the specification and in the compiler code for removing common sub-expressions. An algorithm for removing common sub-expressions could be added to the compiler without affecting most of the proof. The translation of the data structures of the Logic to Piton's finite memory and stacks would remain the same.

The translation of the programs to Piton is done in three stages. The first stage does nothing, but this is where common sub-expressions could be removed. The second stage assigns Piton addresses to quoted constants. The third stage translates the program to Piton. Only the proof of the first stage would need to be redone if a common sub-expression removal function were added.

Chapter 4

The Correctness Theorem

4.1 Interpreter Equivalence Proofs

This dissertation is about the proof of correctness of a compiler. The first part of understanding a proof is to understand what is being proved. This section discusses how the source and target languages of the compiler are specified and how to prove that the translation is correct.

Both the source language (the Logic) and the target language (Piton) have been formally specified operationally via interpreters. The interpreter for the Logic is the function `V&C$`. The interpreter for Piton is the function `P`. Informally, the compiler, `LOGIC->P`, is correct if the result of running `P` on the result of compiling from the Logic to Piton is the same as running `V&C$`. An illustration of what it means for `LOGIC->P` to be correct is shown in the diagram of Figure 4.1.

The commuting diagram of Figure 4.1 is a simplification of the situation for three reasons. First, the Piton interpreter `P` takes two arguments: a Piton state and a "clock". The clock is a number that tells `P` how many steps to run. The clock is missing from the diagram. Intuitively, the diagram is meant to suggest "there exists a clock sufficient to make P compute" an appropriate state. Since the Logic does not allow existential quantifiers, the usual solution is adopted: provide a witness expression for the clock.

Second, the compiler `LOGIC->P` also takes resource limits as arguments. This is ignored in Figure 4.1.

Third, the compiler so behaves only when the expression being compiled is syntactically well-formed, only uses the primitive functions in the compilable subset, and can be evaluated with the given resource limits.

Sometimes commuting diagrams are drawn with the arrow on the right pointing down instead of up. That is, instead of abstracting an answer from the final lower level state, the final lower level state is obtained by mapping the final upper level state down by the compiler. In general, it is not possible to use `LOGIC->P` and the

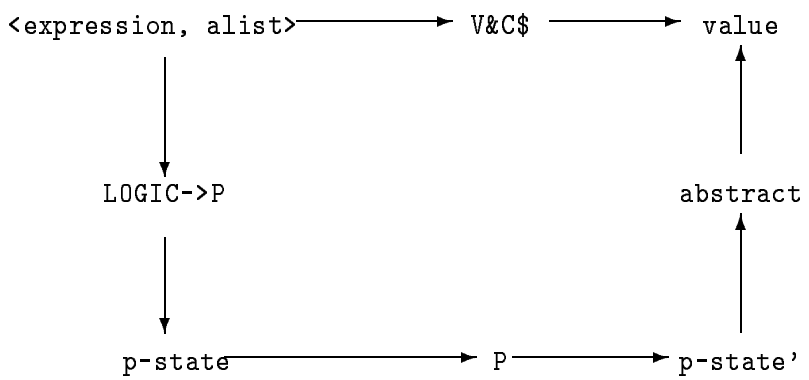

Figure 4.1: Interpreter Equivalence Diagram

final answer to obtain the same Piton state as obtained via first applying LOGIC->P and then running the Piton interpreter P. Running P causes the generation of intermediate results and garbage which are not generated when mapping down the final result. A major aspect of the formal specification is defining how this abstraction process works, i.e. how logical objects are recovered from a P-STATE.

4.2 The Statement of the Correctness Theorem

The following theorem has been proved about LOGIC->P:[1]

THEOREM 1 (LOGIC->P-OK-REALLY)

```
(IMPLIES (AND (L-PROPER-EXPRP T EXPR PNAMES (STRIP-CARS ALIST))        ;[1]
              (L-PROPER-PROGRAMSP PNAMES)                              ;[2]
              (ALL-LITATOMS (STRIP-CARS ALIST))                        ;[3]
              (L-DATA-SEG-BODY-RESTRICTEDP T EXPR)                     ;[4]
              (L-RESTRICTEDP PNAMES ALIST)                             ;[5]
              (V&C$ T EXPR ALIST)                                      ;[6]
              (L-RESTRICT-SUBRPS T EXPR)                               ;[7]
              (L-RESTRICT-SUBRPS-PROGS PNAMES)                         ;[8]
              (NOT (LESSP HEAP-SIZE                                    ;[9]
                          (TOTAL-HEAP-REQS EXPR ALIST PNAMES HEAP-SIZE)))
              (NOT (LESSP MAX-CTRL (MAX-CTRL-REQS EXPR ALIST PNAMES))) ;[10]
              (LESSP MAX-CTRL (EXP 2 WORD-SIZE))                       ;[11]
              (NUMBERP MAX-CTRL)                                       ;[12]
              (NOT (LESSP MAX-TEMP (MAX-TEMP-REQS EXPR ALIST PNAMES))) ;[13]
              (LESSP MAX-TEMP (EXP 2 WORD-SIZE))                       ;[14]
              (NUMBERP MAX-TEMP)                                       ;[15]
              (NOT (LESSP WORD-SIZE                                    ;[16]
```

[1] This theorem has been mechanically proven about a prototype of the implementation that did not have a garbage collector. Work continues on producing a mechanically checked proof for the compiler with a garbage collector added.

```
                  (MAX-WORD-SIZE-REQS EXPR ALIST PNAMES HEAP-SIZE)))
        (NUMBERP WORD-SIZE))                                        ;[17]
  (VALP (CAR (V&C$ T EXPR ALIST))
        (FETCH (LR-ANSWER-ADDR)
               (P-DATA-SEGMENT (P (LOGIC->P EXPR ALIST PNAMES
                                            HEAP-SIZE
                                            MAX-CTRL
                                            MAX-TEMP
                                            WORD-SIZE)
                                  (LOGIC->P-CLOCK EXPR ALIST PNAMES
                                                  HEAP-SIZE
                                                  MAX-CTRL
                                                  MAX-TEMP
                                                  WORD-SIZE))))
        (P-DATA-SEGMENT (P (LOGIC->P EXPR ALIST PNAMES
                                     HEAP-SIZE
                                     MAX-CTRL
                                     MAX-TEMP
                                     WORD-SIZE)
                           (LOGIC->P-CLOCK EXPR ALIST PNAMES
                                           HEAP-SIZE
                                           MAX-CTRL
                                           MAX-TEMP
                                           WORD-SIZE)))))
```

This daunting formula can be read as follows. Note that it is of the form `(IMPLIES hyps (VALP ...))`. Thus it says that under some hypothesis `hyps` the `VALP` expression is true. First focus on the `VALP` expression because it embodies the commutative diagram.

The upper left-hand corner of the commutative diagram, `<expression, alist>` is formally represented by the two variables `EXPR` and `ALIST`: the expression being evaluated and compiled, and the assignment of values to variables. The upper right-hand corner of the diagram, `value`, is `(CAR (V&C$ T EXPR ALIST))`, the object that is the value of `EXPR` under `ALIST` as "officially" defined by Boyer and Moore. The function `V&C$` takes three arguments: the first is called `FLAG`, the other two are `EXPR` and `ALIST`, as above. If `FLAG` has the value `'LIST`, then `EXPR` is treated as a list of terms. The result of `V&C$` with a `FLAG` equal `'LIST` is a list of value-cost pairs (some of which may be `F`). Note that `(CAR (V&C$ T EXPR ALIST))` is the first argument to the `VALP` expression being analyzed.

The lower left corner of the diagram, `p-state`, obtained by compiling `EXPR` is:

```
(LOGIC->P EXPR ALIST PNAMES
          HEAP-SIZE MAX-CTRL MAX-TEMP WORD-SIZE)
```

The lower right hand corner, `p-state'`, obtained by running the Piton machine, is:

```
(P (LOGIC->P EXPR ALIST PNAMES
             HEAP-SIZE MAX-CTRL MAX-TEMP WORD-SIZE)
   (LOGIC->P-CLOCK EXPR ALIST PNAMES
                   HEAP-SIZE MAX-CTRL MAX-TEMP WORD-SIZE))
```

LOGIC->P-CLOCK is the "witness function" which determines how long the Piton machine must run to evaluate the code for EXPR on ALIST.

Letting value denote:

```
(CAR (V&C$ T EXPR ALIST))
```

— the upper right-hand corner — and p-state' denote

```
(P (LOGIC->P EXPR ALIST PNAMES
             HEAP-SIZE MAX-CTRL MAX-TEMP WORD-SIZE)
   (LOGIC->P-CLOCK EXPR ALIST PNAMES
                   HEAP-SIZE MAX-CTRL MAX-TEMP WORD-SIZE))
```

— the lower right hand corner, then the VALP expression is:

```
(VALP value
      (FETCH (LR-ANSWER-ADDR) (P-DATA-SEGMENT p-state'))
      (P-DATA-SEGMENT p-state'))
```

This term expresses the abstraction drawn as the upward headed arrow. VALP is the formalization of that abstraction. The VALP expression can be read as follows. Consider the final Piton data segment (the third argument to the VALP expression). Fetch from that data segment the contents of the "answer address" (the second argument). The VALP expression says that the contents is a heap address into the final data segment and the logical object represented by that address in the final data segment is in fact the "officially" defined value of EXPR under ALIST.

The hypotheses of the theorem seem daunting at first, but they merely represent the hidden assumptions that are used when compiling and running programs. There are two kinds of hypotheses in the correctness theorem. First are syntactic restrictions on the programs that will be trivially satisfied when the compiler is used on actual examples. Second are hypotheses that deal with the resource limits of Piton.

The hypotheses labeled [1] and [2] state that the expression and functions are all proper (see section 4.3). Hypothesis [3] is another syntactic requirement, namely that all the variable names in the variable alist are literal atoms. Hypothesis [4] states that all the quoted constants in the expression only use shells that are defined (in the current version of the compiler all the shells are either TRUE, FALSE, a NUMBERP or a CONS). Hypothesis [5] states that the programs, as well as the bindings in the variable alist ALIST, only use defined shells. Hypothesis [6] states that V&C$ produces a non-FALSE result when applied to the expression EXPR., i.e., the expression has a defined value. Hypotheses [7] and [8] are only necessary in the prototype. They ensure that the expression and programs only use the SUBRPs that have been defined, namely CAR, CDR, CONS, FALSE, FALSEP, IF, LISTP, NLISTP, TRUE and TRUEP. Hypotheses [9-17] deal with the resource limitations of Piton. These hypotheses ensure that there are enough resources to run the program without errors. They also ensure that the resources are properly formed for use

in Piton (i.e., MAX-CTRL, MAX-TEMP and WORD-SIZE are numbers, and MAX-CTRL and MAX-TEMP will fit in one word of storage). The functions TOTAL-HEAP-REQS, MAX-CTRL-REQS and MAX-TEMP-REQS are witness functions which can be thought of as existential quantifications on the amount of heap space, control stack space, and temporary stack space, respectively.

4.3 Proper Expressions and Programs

The predicate L-PROPER-EXPRP takes four arguments: FLAG, EXPR, PROGRAM-NAMES, and FORMALS. It checks that expressions are proper, that is, all the subexpressions of EXPR are proper lists, and each function application has the correct number of arguments. In addition, any functions calls in EXPR must be either to predefined SUBRs or to functions named in PROGRAM-NAMES. Finally, it ensures that all references to variables are named in the list FORMALS. The predicate L-PROPER-PROGRAMSP takes one argument, PROGRAM-NAMES: a list of program names. It checks that for each member X of PROGRAM-NAMES, L-PROPER-EXPRP holds when applied to T, (BODY X), PROGRAM-NAMES and (FORMALS X).

The most important check done by L-PROPER-EXPR and L-PROPER-PROGRAMSP is that the only functions called are in PROGRAM-NAMES. An alternative to ensuring that all function applications are proper lists of the correct length would be to have the compiler extend argument lists as necessary and drop excess arguments. This alternative is probably a better choice for future versions of the compiler.

4.4 Mapping the Final Piton State Up

One of the most interesting parts of the compiler specification is the function, VALP, used to "map" the final Piton state to a value in the logic. Naively this function must take an address into the heap and extract a value from the heap. Imagine such a function called ABS1 which takes two arguments, an address and a data segment, and returns the object represented at that address. ABS1 is recursively defined to determine the type of the current node and then "chase" the pointers in that node in order to recover the components of the logical object represented. Note that this process does not terminate on all Piton data segments, e.g., consider a CONS node that points to itself. Such data segments are not constructed by executing the code output by the compiler (though they can be constructed with Piton). One solution would be to define the notion of a cycle-free data segment and arrange for ABS1 to be trivially defined on such arguments. This would require proving the P-STATEs generated by the compiler are cycle-free. But the Logic requires all function definitions to have an associated measure that decreases on any calls. Thus the imagined ABS1 is not admissible by Nqthm. A second solution to this would be to add a clock argument to ABS1, which would be decremented on every recursive call. A definition of ABS1 with a clock is:

DEFINITION

```
(ABS1 ADDR DATA-SEG CLOCK)
   =
(IF (ZEROP CLOCK)
    NIL
    (LET ((TAG (UNTAG (FETCH (ADD-ADDR ADDR (LR-TAG-OFFSET)) DATA-SEG))))
      (COND ((EQUAL TAG (LR-CONS-TAG))
             (CONS (ABS1 (FETCH (ADD-ADDR ADDR (LR-CAR-OFFSET)) DATA-SEG)
                         DATA-SEG
                         (SUB1 CLOCK))
                   (ABS1 (FETCH (ADD-ADDR ADDR (LR-CDR-OFFSET)) DATA-SEG)
                         DATA-SEG
                         (SUB1 CLOCK))))
            ((EQUAL TAG (LR-TRUE-TAG)) (TRUE))
            ((EQUAL TAG (LR-FALSE-TAG)) (FALSE))
            ((EQUAL TAG (LR-ADD1-TAG))
             ;; We should handle bignums here
             (UNTAG (FETCH (ADD-ADDR ADDR (LR-UNBOX-NAT-OFFSET)) DATA-SEG)))
            (T F))))
```

If the clock runs out, ABS1 just returns NIL. However, the clocks would likely be difficult to use in the proof.

A third solution would be a function (ABS2) that accumulates all the addresses that have been seen so far and returns NIL if it finds an address already seen. A possible definition of ABS2 is:

```
(ABS2 ADDR DATA-SEG ADDRESS-LIST)
   =
(IF (MEMBER-ADDR ADDR ADDRESS-LIST)
    NIL
      (IF (NOT (EQUAL (AREA-NAME ADDR) (LR-HEAP-NAME)))
          NIL
          (LET ((TAG (UNTAG (FETCH (ADD-ADDR ADDR (LR-TAG-OFFSET))
                                   DATA-SEG))))
            (COND ((EQUAL TAG (LR-CONS-TAG))
                   (CONS (ABS2 (FETCH (ADD-ADDR ADDR (LR-CAR-OFFSET))
                                      DATA-SEG)
                               DATA-SEG
                               (CONS (OFFSET ADDR) OFFSET-LIST))
                         (ABS2 (FETCH (ADD-ADDR ADDR (LR-CDR-OFFSET))
                                      DATA-SEG)
                               DATA-SEG
                               (CONS (OFFSET ADDR) OFFSET-LIST))))
                  ((EQUAL TAG (LR-TRUE-TAG)) (TRUE))
                  ((EQUAL TAG (LR-FALSE-TAG)) (FALSE))
                  ((EQUAL TAG (LR-ADD1-TAG))
                   (UNTAG (FETCH (ADD-ADDR ADDR (LR-UNBOX-NAT-OFFSET))
                                 DATA-SEG)))
                  (T F)))))
```

ABS2 is a hard function to admit to the Nqthm theorem prover; however, it could possibly be useful.

The solution used in the compiler proof is to define the predicate VALP that takes three arguments: an object, an address, and a data segment. VALP returns T

if the address in the data segment represents the object and returns F otherwise. The predicate VALP is somewhat unusual. VALP does not use a clock, but uses the object as both the expected result and the measure that decreases on recursive calls.

4.5 Using the Compiler

The compiler cannot correctly evaluate every expression in the Logic, or even every expression that may be evaluated in the execution environment of the theorem prover. All the finite resources of Piton are reflected in the correctness theorem. A programmer who wants to use the correctness theorem to state that his or her program runs correctly must be sure that there are enough resources for the program to run. This is an unavoidable fact of everyday life in programming. The common practice is not to worry about the resources until they are exceeded and (usually) an error message is printed. The difference here is that no error message is printed. Piton is implemented on FM8502 and it is impossible to tell from looking at an FM8502 state whether or not a Piton program (or a program in the Logic compiled into Piton) encountered an error. The way around this problem is to prove that the Logic program being compiled runs without resource errors.

All arithmetic is *fixnum* arithmetic. This is represented by hypothesis [16] of the LOGIC->P-OK-REALLY which ensures that the word size of Piton is large enough to represent the result of every arithmetic expression.

As with any programming language implementation, there is a limit on the number of objects that can be constructed. With the garbage collector running, this means there is a limit on the number of objects that can be "in use" at one time. The limitation of the size of the heap is represented by hypothesis [9] of LOGIC->P-OK-REALLY. The size of the stacks is also limited, as shown by hypotheses [10] and [13].

The compiler does not provide any input or output operations. Piton and FM8502 do not currently provide any facilities for doing input and output, and the Logic does not have any input and output.

Chapter 5

Formal Description of the Compiler

The compiler is defined by the function LOGIC->P.

```
DEFINITION
(LOGIC->P EXPR ALIST PNAMES HEAP-SIZE MAX-CTRL MAX-TEMP WORD-SIZE)
   =
(LR->P (S->LR (LOGIC->S EXPR ALIST PNAMES)

              HEAP-SIZE
              MAX-CTRL
              MAX-TEMP
              WORD-SIZE))
```

LOGIC->P is the composition of three functions. The innermost function, LOGIC->S, translates an expression, EXPR; a variable association list, ALIST; and a list of program names, PNAMES; and produces an "S-STATE". The next phase of compilation is representing the abstract data structures of the Logic in Piton. This is accomplished by the function S->LR, which takes an S-STATE and four resource limits — HEAP-SIZE, MAX-CTRL, MAX-TEMP and WORD-SIZE — and produces an LR-STATE. The final phase is the compilation of the functions into Piton. This is accomplished by LR->P, which maps LR-STATEs into P-STATEs.

This decomposition is important to the structure of the proof of the correctness result. For each intermediate level, e.g. S-STATEs and LR-STATEs, an appropriate operational semantics is defined. Each transformation is then proven to preserve the semantics in an appropriate sense. In essence the S level is a language very similar to the Logic but which provides a means of avoiding the repeated computation of given expressions. Programs at the LR level are very similar to the S level, but data objects are represented in a linear memory (the heap). Thus at the LR level programs and expression evaluation proceeds similarly to the higher level, but the *effect* of a CONS, say, is to change the addresses in the heap rather than to

produce an ordered pair. Ignoring the S level then, the transformation to the LR level preserves programs and changes the data representation. The transformation to the P level preserves data representation but changes the programming language and thus the proof is there concerned with showing that the pointer manipulation described abstractly and atomically at the LR level is mirrored by that done by the appropriate sequence of Piton instructions.

5.1 The Common Sub-Expression Level

The Logic appears to contain the special form LET. For instance, one can write the expression:

```
(LET ((X (FN Y))) (CONS X X))
```

If this expression were in Common Lisp, then (FN Y) would only be evaluated once and the result used as the value of X in the CONS expression. However, in the Logic LET is an abbreviation that allows users of Nqthm to more easily type formulas in the Logic (see [6]). The LET expression is just an abbreviation for:

```
(CONS (FN Y) (FN Y))
```

Since there is no method for the user to specify to the compiler that certain expressions should only be evaluated once, the compiler should remove common subexpressions. Any efficient implementation of the Logic must remove common subexpressions.

In the first step of the compiler this problem is considered. Imagine translating a Logic expression and association list into an "S-STATE". An S-STATE is a state of an abstract computing machine like V&C$ but which has provisions for storing and retrieving the values of previously encountered expressions. S-STATEs are represented by a new shell:

SHELL DEFINITION
add the shell S-STATE of 7 arguments, with
recognizer function symbol S-STATEP, and
accessors S-PNAME, S-POS, S-ANS, S-PARAMS,
S-TEMPS, S-PROGS and S-ERR-FLAG.

The S-STATE contains a "program counter" which consists of the S-PNAME and S-POS fields. The S-PNAME field contains the name of the current program, and the S-POS field determines the current expression. The S-ANS field contains the result of evaluating an S-STATE. The S-PARAMS field corresponds directly to the alist of V&C$. S-TEMPS contains an alist of triples: an expression, a boolean, and a value. It is used to store the result of a previously evaluated expression. If the boolean is not F, then the value is the value of evaluating the expression; otherwise the expression has not been evaluated yet. S-PROGS is a list of programs. Finally,

S-ERR-FLAG is an error flag; if it is 'RUN then an error has not been encountered. If it has any other value an error was found.

The translation from the Logic to the S level is done by LOGIC->S, which is defined as follows:

```
DEFINITION
(LOGIC->S EXPR ALIST FUN-NAMES)

   =
(S-STATE 'MAIN
         NIL
         NIL
         ALIST
         NIL
         (CONS (LIST 'MAIN (STRIP-CARS ALIST) NIL EXPR)
               (S-CONSTRUCT-PROGRAMS (REMOVE-DUPLICATES FUN-NAMES)))
         'RUN)
```

Programs at the S level are lists of the form:

```
(name (v0 v1 ... vn-1)
      (t0 t1 ... tk-1)
      expr),
```

where name is the name of the program and is some literal atom; $v_0, \ldots, v_{n-1}$ are the $n \geq 0$ *formal parameters* of the program and are literal atoms; $t_0, \ldots, t_{k-1}$ are the $k \geq 0$ *sub-expressions* of the program and are s-level expressions; and expr is the *body* of the program.

The S level introduces three additional forms that allow expressions to be evaluated and their results saved. These forms have the structure (*operator expression*), where *operator* is one of the following: (TEMP-FETCH), (TEMP-EVAL), or (TEMP-TEST). In a proper S-STATE, the *expression* must be defined in the S-TEMPS alist.

The meaning of the forms depends on the *operator*. In the description below, the *flag* and *value* are the CADR and CADDR, respectively, of the result of calling ASSOC on *expression* and the S-TEMPS field of the S-STATE. The forms have the following meaning:

(TEMP-FETCH) unconditionally fetches the *value* associated with *expression* from S-TEMPS. S-ERR-FLAG is set to an error value if *flag* is F.

(TEMP-EVAL) evaluates the CADR of the form. Return the result and set the *flag* field of S-TEMPS to T and the *value* field to the result.

(TEMP-TEST) if the *flag* field is not F, then the result is the *value* field associated with *expression*; otherwise the result is obtained by evaluating *expression*. The *flag* field of S-TEMPS is set to T and the *value* field is set to the result.

Consider the expression shown on lines [0] to [5] in Figure 5.1 and an equivalent S-level expression with common sub-expressions removed shown on lines [6]

```
(IF (CAR (CDR (APP X '(*1*TRUE . *1*FALSE))))                          ;[0]
    (CONS (IF (CDR (CDR (APP X '(*1*TRUE . *1*FALSE))))                ;[1]
              (CONS T X)                                               ;[2]
              (CONS F Y))                                              ;[3]
          (CONS T X))                                                  ;[4]
    (CDR (APP X '(*1*TRUE . *1*FALSE))))                               ;[5]

'(IF (CAR ((TEMP-EVAL) (CDR (APP X '(*1*TRUE . *1*FALSE)))))           ;[6]
    (CONS (IF (CDR ((TEMP-FETCH) (CDR (APP X '(*1*TRUE . *1*FALSE))))) ;[7]
              ((TEMP-EVAL) (CONS T X))                                 ;[8]
              (CONS F Y))                                              ;[9]
          ((TEMP-TEST) (CONS T X)))                                    ;[10]
    ((TEMP-FETCH) (CDR (APP X '(*1*TRUE . *1*FALSE)))))                ;[11]
```

Figure 5.1: An Example of Removing Common Subexpressions

to [11]. There have been two sub-expressions removed in the S-level expression. The first is (CDR (APP X '(*1*TRUE . *1*FALSE))) which occurs on lines [0], [1], and [5]. The second is (CONS T X) on lines [2] and [4]. When the expression is evaluated, the value for (CDR (APP X '(*1*TRUE . *1*FALSE))) will be computed and stored when the execution is at the first occurrence of the expression on line [6]. Then when the execution reaches the occurrences on lines [7] or [11], the stored value is used (note in any computation that only one of the occurrences on lines [7] or [11] will be executed).

If the test expression of the IF at line [7] is not F then we will evaluate (CONS T X) on line [8] and store the result. If the test is F then (CONS T X) will not be evaluated when execution reaches line [10]. Therefore, on line [10] we must test to see if (CONS T X) have been evaluated yet, so there is a (TEMP-TEST) form there.

5.2 The Resource Representation Level

The next step of compilation translates S-STATEs to an abstract machine that represents the objects in the Logic in terms of Piton's resources (i.e. addresses into a heap of Piton objects in Piton's data segment). In addition, this machine saves control information and intermediate results on the Piton stacks. This machine is called the LR machine. LR-STATEs are represented by the shell P-STATE, the same as Piton states. The translation from the S level to the LR level is done by the function S->LR. Roughly speaking, this translation scans the programs and parameter values of the S-STATE, finds each constant, represents it in the heap and replaces the constant by the heap address allocated.

DEFINITION

```
(S->LR S FHEAP-SIZE MAX-CTRL MAX-TEMP WORD-SIZE)
  =
```

```
;; Returns an P-STATE.
;; FREE-HEAP-SIZE is number of free nodes in resulting P-STATE
(LET ((TEMP-ALIST (LR-MAKE-TEMP-NAME-ALIST (STRIP-CARS (S-TEMPS S))
                                           (STRIP-CARS (S-PARAMS S))))
      (DATASEG-TABLE (LR-DATA-SEG-TABLE (S-PROGS S)
                                        (S-PARAMS S)
                                        FHEAP-SIZE)))
  (LET ((RETURN-PC
         (TAG 'PC
              (CONS (S-PNAME S)
                    (LR-P-PC-1 (LR-COMPILE-BODY T
                                                (S-BODY (S-PROG S))
                                                TEMP-ALIST
                                                (CDR DATASEG-TABLE))
                               (S-POS S))))))
    (S->LR1 S
            (P-STATE NIL
                     (LR-INITIAL-CSTK (S-PARAMS S)
                                      TEMP-ALIST
                                      (CDR DATASEG-TABLE)
                                      RETURN-PC)
                     NIL
                     NIL
                     (CAR DATASEG-TABLE)
                     MAX-CTRL
                     MAX-TEMP
                     WORD-SIZE
                     NIL)
            (CDR DATASEG-TABLE))))
```

S->LR calls the function S->LR1 which compiles the S level programs to produce LR level programs and creates an LR level program counter. This compilation step just replaces constants by heap addresses that represent them. The definition of S->LR1 is shown here:

DEFINITION

```
(S->LR1 S L TABLE)
   =
(P-STATE (TAG 'PC (CONS (S-PNAME S) (S-POS S)))
         (P-CTRL-STK L)
         (P-TEMP-STK L)
         (LR-COMPILE-PROGRAMS (S-PROGS S) TABLE)
         (P-DATA-SEGMENT L)
         (P-MAX-CTRL-STK-SIZE L)
         (P-MAX-TEMP-STK-SIZE L)
         (P-WORD-SIZE L)
         (S-ERR-FLAG S))
```

The function LR-DATA-SEG-TABLE (shown on page 123) gathers all the constants in the S-STATE and produces a data-segment and a table that gives the address of each constant. On each constant LR-DATA-SEG-TABLE calls LR-COMPILE-QUOTE. Constants can be in the programs of the S-STATE as well as the parameters of the S-STATE. The function LR-INITIAL-CSTK constructs the initial control stack for

the LR state. The initial temporary stack is empty. The only parts of the LR state that are different from the Piton P-STATE are the pc and the programs.

LR-COMPILE-QUOTE is shown here:

```
DEFINITION
(LR-COMPILE-QUOTE FLAG OBJECT HEAP TABLE)
   =
;; LR-COMPILE-QUOTE returns a pair, the new HEAP and the new TABLE.
(COND ((EQUAL FLAG 'LIST)
       (IF (LISTP OBJECT)
           (LET ((CAR-PAIR (LR-COMPILE-QUOTE T (CAR OBJECT) HEAP TABLE)))
             (LR-COMPILE-QUOTE 'LIST
                               (CDR OBJECT)
                               (CAR CAR-PAIR)
                               (CDR CAR-PAIR)))
           (CONS HEAP TABLE)))
      ((DEFINEDP OBJECT TABLE)
       (CONS (DEPOSIT (TAG 'NAT
                           (ADD1 (UNTAG (FETCH (CDR (ASSOC OBJECT TABLE))
                                               HEAP))))
                     (CDR (ASSOC OBJECT TABLE))
                     HEAP)
             TABLE))
      ((LISTP OBJECT)
       (LET ((PAIR (LR-COMPILE-QUOTE 'LIST
                                     (LIST (CAR OBJECT) (CDR OBJECT))
                                     HEAP
                                     TABLE)))
         (CONS (LR-ADD-TO-DATA-SEG (CAR PAIR)
                                   (LR-NEW-CONS (CDR (ASSOC
                                                       (CAR OBJECT)
                                                       (CDR PAIR)))
                                                (CDR (ASSOC
                                                       (CDR OBJECT)
                                                       (CDR PAIR)))))
               (CONS (CONS OBJECT (FETCH (LR-FP-ADDR) (CAR PAIR)))
                     (CDR PAIR)))))
      ((NUMBERP OBJECT)
       (CONS (LR-ADD-TO-DATA-SEG HEAP
                                 (LR-NEW-NODE (TAG 'NAT 0)
                                              (TAG 'NAT (LR-ADD1-TAG))
                                              (TAG 'NAT OBJECT)
                                              (LR-UNDEF-ADDR)))
             (CONS (CONS OBJECT (FETCH (LR-FP-ADDR) HEAP)) TABLE)))
      ((TRUEP OBJECT)
       (CONS (LR-ADD-TO-DATA-SEG HEAP (LR-NEW-NODE (TAG 'NAT 0)
                                                   (TAG 'NAT (LR-TRUE-TAG))
                                                   (LR-UNDEF-ADDR)
                                                   (LR-UNDEF-ADDR)))
             (CONS (CONS OBJECT (FETCH (LR-FP-ADDR) HEAP)) TABLE)))
      (T                              ; Assume it is undefined
       (CONS HEAP
             (CONS (CONS OBJECT (LR-UNDEF-ADDR)) TABLE))))
```

5.3 The Piton level

The final step in the compilation is to replace the LR programs and the LR program counter by Piton programs and a Piton program counter. The translation to the Piton level is done by the function LR->P shown here:

```
DEFINITION
(LR->P L)
   =
(P-STATE (LR-P-PC L)
         (P-CTRL-STK L)
         (P-TEMP-STK L)
         (COMP-PROGRAMS (P-PROG-SEGMENT L))
         (P-DATA-SEGMENT L)
         (P-MAX-CTRL-STK-SIZE L)
         (P-MAX-TEMP-STK-SIZE L)
         (P-WORD-SIZE L)
         (P-PSW L))
```

The translation involves translating the pc of the LR state, which is done by the function LR-P-PC, and translating the programs of the LR state, which is done by the function COMP-PROGRAMS.

```
DEFINITION
(COMP-PROGRAMS PROGRAMS)
   =
(APPEND (COMP-PROGRAMS-1 T PROGRAMS)
        (P-RUNTIME-SUPPORT-PROGRAMS))
```

COMP-PROGRAMS compiles the LR level programs by calling the function COMP-PROGRAMS-1 on the programs and then appends the code for the predefined SUBRs

```
DEFINITION
(COMP-PROGRAMS-1 FIRSTP PROGRAMS)
   =
(IF (LISTP PROGRAMS)
    (CONS (LR-MAKE-PROGRAM (NAME (CAR PROGRAMS))
                           (FORMAL-VARS (CAR PROGRAMS))
                           (TEMP-VAR-DCLS (CAR PROGRAMS))
                           (COMP-BODY FIRSTP
                                      (PROGRAM-BODY (CAR PROGRAMS))
                                      (FORMAL-VARS (CAR PROGRAMS))
                                      (TEMP-VAR-DCLS (CAR PROGRAMS))))
          (COMP-PROGRAMS-1 F (CDR PROGRAMS)))
    NIL)
```

Chapter 6

Proof of Correctness

The correctness proof is broken down into the proof of four commuting diagrams as shown in Figure 6.1. This separation makes for some elegance and modularity in the proof, but it is actually more important. Invariants on the heap which are crucial to the LR level, e.g. the unique representation of some data, and which are easily established in view of the very stylized pointer smashing allowed by the LR language, would be impossible to establish if the programming language allowed the no-holds barred manipulations permitted (but in fact never engaged in) at the Piton level. This phenomenon was described by Moore in [23] and the layering here was modeled on his. The first commuting sub-diagram shows the equivalence of `V&C$` and the interpreter `L-EVAL`, the definition of which is:

DEFINITION

```
(L-EVAL FLAG EXPR ALIST CLK)
   =
(COND ((EQUAL FLAG 'LIST)
       (IF (LISTP EXPR)
           (CONS (L-EVAL T (CAR EXPR) ALIST CLK)
                 (L-EVAL 'LIST (CDR EXPR) ALIST CLK))
           NIL))
      ((ZEROP CLK) F)
      ((LITATOM EXPR) (LIST (CDR (ASSOC EXPR ALIST))))
      ((NLISTP EXPR) (LIST EXPR))
      ((EQUAL (CAR EXPR) 'QUOTE) (LIST (CADR EXPR)))
      ((EQUAL (CAR EXPR) 'IF)
       (LET ((TEST (L-EVAL T (CADR EXPR) ALIST CLK)))
         (IF TEST
             (IF (CAR TEST)
                 (L-EVAL T (CADDR EXPR) ALIST CLK)
                 (L-EVAL T (CADDDR EXPR) ALIST CLK))
             F)))
      ((MEMBER F (L-EVAL 'LIST (CDR EXPR) ALIST CLK)) F)
      ((SUBRP (CAR EXPR))
       (LIST (APPLY-SUBR (CAR EXPR)
                         (STRIP-CARS (L-EVAL 'LIST (CDR EXPR) ALIST CLK))))))
```

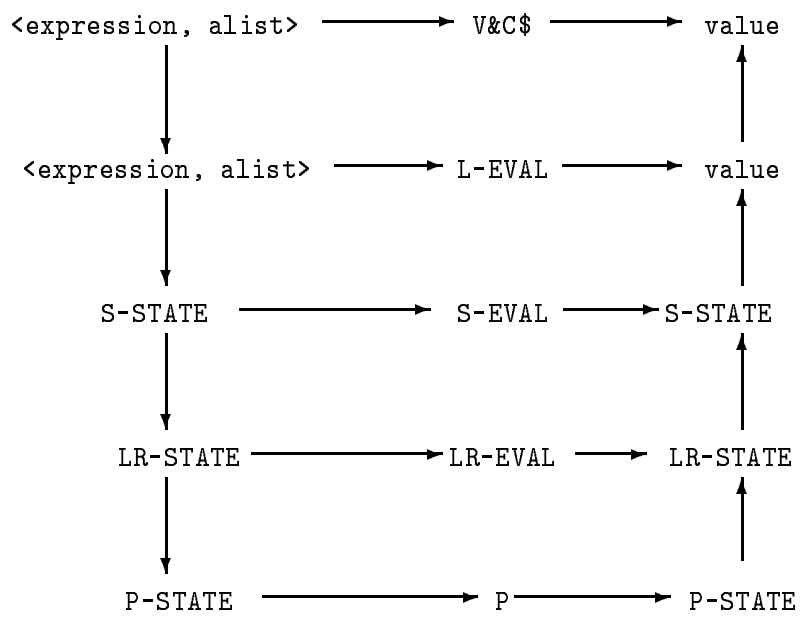

Figure 6.1: Interpreter Equivalence Diagram

```
(T (L-EVAL T
           (BODY (CAR EXPR))
           (PAIRLIST (FORMALS (CAR EXPR))
                     (STRIP-CARS (L-EVAL 'LIST (CDR EXPR) ALIST CLK)))
           (SUB1 CLK))))
```

The interpreter L-EVAL is very similar to the Logic interpreter V&C$. The interpreter V&C$ takes three arguments: FLAG, EXPR and ALIST. When it is given a FLAG that is not 'LIST it returns either a value-cost pair or F. The CAR of the value-cost pair is the value of the expression and the CDR is cost of evaluating the expression. V&C$ returns F if there is no cost large enough to evaluate EXPR.

L-EVAL takes four arguments. Three are the same as V&C$: FLAG, EXPR, and ALIST. The fourth argument, CLK, serves as a bound on the number of function applications that can be done. L-EVAL does not compute the costs that V&C$ does. However, it does produce the same value for the expression. This fact is embodied in the following theorem:

THEOREM 2 (V&C$-L-EVAL-EQUIVALENCE)

```
(IMPLIES (OR (AND (EQUAL FLAG 'LIST)
                  (NOT (MEMBER F
                               (L-EVAL FLAG EXPR ALIST CLOCK))))
             (AND (NOT (EQUAL FLAG 'LIST))
```

```
                (NOT (EQUAL (L-EVAL FLAG EXPR ALIST CLOCK)
                            F))))
        (AND (EQUAL (L-EVAL FLAG EXPR ALIST CLOCK)
                    (IF (EQUAL FLAG 'LIST)
                        (REMOVE-COSTS (V&C$ FLAG EXPR ALIST))
                        (LIST (CAR (V&C$ FLAG EXPR ALIST)))))
             (OR (AND (EQUAL FLAG 'LIST)
                      (NOT (MEMBER F (V&C$ FLAG EXPR ALIST))))
                 (AND (NOT (EQUAL FLAG 'LIST))
                      (NOT (EQUAL (V&C$ FLAG EXPR ALIST)
                                  F))))))
```

where the function REMOVE-COSTS is defined as follows:

DEFINITION

```
(REMOVE-COSTS LIST)
   =
(IF (LISTP LIST)
    (CONS (LIST (CAAR LIST)) (REMOVE-COSTS (CDR LIST)))
    NIL)
```

Theorem 2 states that if L-EVAL returns a non-F value then it computes the same result as V&C$ does. However, this theorem would be trivially satisfied if L-EVAL always returned F. The following theorem states that if the CLK argument to L-EVAL is at least one more that the cost computed by V&C$ (assuming V&C$ can compute the cost), then L-EVAL returns the same value as V&C$.

THEOREM 3 (L-EVAL-NOT-F-V&C$-EQUIVALENCE)

```
(IMPLIES (OR (AND (EQUAL FLAG 'LIST)
                  (NOT (MEMBER F (V&C$ FLAG EXPR ALIST)))
                  (LESSP (SUM-CDRS (V&C$ FLAG EXPR ALIST)) CLOCK))
             (AND (NOT (EQUAL FLAG 'LIST))
                  (NOT (EQUAL (V&C$ FLAG EXPR ALIST) F))
                  (LESSP (CDR (V&C$ FLAG EXPR ALIST)) CLOCK)))
         (AND (EQUAL (L-EVAL FLAG EXPR ALIST CLOCK)
                     (IF (EQUAL FLAG 'LIST)
                         (REMOVE-COSTS (V&C$ FLAG EXPR ALIST))
                         (LIST (CAR (V&C$ FLAG EXPR ALIST)))))
              (OR (AND (EQUAL FLAG 'LIST)
                       (NOT (MEMBER F (V&C$ FLAG EXPR ALIST))))
                  (AND (NOT (EQUAL FLAG 'LIST))
                       (NOT (EQUAL (V&C$ FLAG EXPR ALIST) F))))))
```

The commuting sub-diagrams of Figure 6.1 that map S-STATEs and LR-STATEs correspond to a different layer, each of which is formalized independently. This requires the formalization of two intermediate abstract machines. The first of these machines is called the "S" machine, which is defined by an interpreter S-EVAL which interprets S-STATEs. The second machine is called the "LR" machine and is defined by an interpreter LR-EVAL, which interprets LR-STATEs.

The most difficult part of developing the proof of each layer was the formalization of a "proper state" predicate and the proof that this predicate was maintained by the interpreter at that level. The proof of correctness requires states that are proper.

The proofs of the correspondence between the Logic and the S machine and the S machine and the LR machine both proceed in three parts. The definitions of LOGIC->S and S->LR are not general enough to allow an inductive proof of the correspondence between the upper and lower machines. Therefore, the theorem is generalized to look at "analogous" states (the state at the Logic level can be considered to be the expression, variable alist, and the programs) and show that the upper and lower machines produce the same result. The first theorem shows that the final states produce the same result given that the initial state is a proper one, and that the final state of the lower machine is not an error state. The second theorem adds some hypothesis under which the final state of the lower machine is guaranteed not to be an error state. Finally, the translators (LOGIC->S and S->LR) are shown to produce proper states when given proper states and to satisfy the added hypothesis to guarantee the final state is not an error state.

6.1 The S layer

The semantics of the S layer are defined operationally by the interpreter S-EVAL. S-EVAL takes three arguments: FLAG which is the same mutual recursion flag as V&C$ and L-EVAL; S an S-STATE; and C which is same as the clock argument, CLK of L-EVAL. S-EVAL returns an S-STATE; the S-ANS field of the S-STATE contains the result of evaluating S. The "proper state" predicate for the S level is S-GOOD-STATEP, which also ensures that the programs of the S level state are analogous to the corresponding programs in the Logic.

DEFINITION

```
(S-GOOD-STATEP S C)
   =
(AND (DEFINEDP (S-PNAME S) (S-PROGS S))
     (EQUAL (CAR (CAR (S-PROGS S))) 'MAIN)
     (S-PROGRAMS-PROPERP (S-PROGS S) (STRIP-LOGIC-FNAMES (CDR (S-PROGS S))))
     (S-PROGRAMS-OKP (CDR (S-PROGS S)))
     (EQUAL (STRIP-CARS (S-TEMPS S)) (PLIST (S-TEMP-LIST (S-PROG S))))
     (EQUAL (S-ERR-FLAG S) 'RUN)
     (TEMPS-OKP (S-TEMPS S) (S-PARAMS S) C))
```

The function S-PROGRAMS-OKP is the part of S-GOOD-STATEP that ensures that each of the programs in the S-STATE is analogous to the BODY of the function from which it is compiled. Recall that S level programs may contain three special forms for dealing with the storage and retrieval of values for expressions. S-PROGRAMS-OKP replaces these forms by the expressions they denote and then checks that the result expression is equal to the BODY of the corresponding Logic function.

The first theorem about L-EVAL and S-EVAL is:

THEOREM 4 (S-EVAL-L-EVAL-FLAG-T)

```
(IMPLIES (AND (S-GOOD-STATEP S C)                                    ;[1]
              (GOOD-POSP1 (S-POS S) (S-BODY (S-PROG S)))             ;[2]
              (EQUAL (S-ERR-FLAG (S-EVAL T S C)) 'RUN))              ;[3]
         (AND (L-EVAL T
                      (S-EXPAND-TEMPS T (S-EXPR S))
                      (S-PARAMS S)
                      C)
              (EQUAL (S-ANS (S-EVAL T S C))
                     (CAR (L-EVAL T
                                  (S-EXPAND-TEMPS T (S-EXPR S))
                                  (S-PARAMS S)
                                  C)))))
```

Hypothesis [1] ensures that the state S is proper and that the programs of S are analogous to the corresponding programs in the Logic. Hypothesis [2] states that the current expression specified in S is good, that it is not part of a quoted constant. Hypothesis [3] stipulates that the final state produced by S-EVAL is not an error state. Under these hypotheses, L-EVAL returns non-F and produces the same result as S-EVAL. The function S-EXPAND-TEMPS maps expressions at the S level into the Logic. (S-EXPR S) is the current expression of S being considered. (S-PARAMS S) is the current variable bindings alist.

The second theorem guarantees that the final S-STATE produced by S-EVAL is not an error state.

THEOREM 5 (S-EVAL-L-EVAL-FLAG-RUN-FLAG-T)

```
(IMPLIES (AND (S-GOOD-STATEP S C)                                    ;[1]
              (GOOD-POSP FLAG (S-POS S) (S-BODY (S-PROG S)))         ;[2]
              (S-ALL-TEMPS-SETP FLAG                                 ;[3]
                                (S-EXPR S)
                                (TEMP-ALIST-TO-SET (S-TEMPS S)))
              (S-ALL-PROGS-TEMPS-SETP (S-PROGS S))                   ;[4]
              (L-EVAL FLAG                                           ;[5]
                      (S-EXPAND-TEMPS FLAG (S-EXPR S))
                      (S-PARAMS S)
                      C)
              (S-CHECK-TEMPS-SETP (S-TEMPS S))                       ;[6]
              (NOT (EQUAL FLAG 'LIST)))                              ;[7]
         (AND (EQUAL (S-ERR-FLAG (S-EVAL FLAG S C)) 'RUN)
              ... ))
```

Finally, LOGIC->S is shown (under certain conditions) to produce a state S on which the hypotheses of Theorem 5 labeled [1], [3], [4] and [6] hold. Accordingly, the following theorem can be proved.

THEOREM 6 (LOGIC->S-OK)

```
(IMPLIES (AND (L-PROPER-EXPRP T EXPR PROGRAM-NAMES (STRIP-CARS ALIST)) ;[1]
              (L-PROPER-PROGRAMSP PROGRAM-NAMES)                       ;[2]
              (ALL-LITATOMS (STRIP-CARS ALIST))                        ;[3]
              (L-EVAL T EXPR ALIST CLOCK))                             ;[4]
         (EQUAL (S-ANS (S-EVAL T
                               (LOGIC->S EXPR ALIST PROGRAM-NAMES)
                               CLOCK))
                (CAR (L-EVAL T EXPR ALIST CLOCK))))
```

Theorem 6 states that if EXPR and PROGRAM-NAMES are proper and if L-EVAL returns a non-F result, then the result returned by S-EVAL on the result of compiling to the "S" level with LOGIC->S is the same as the result returned by LR-EVAL. Hypothesis [1] of Theorem 6 states that EXPR is proper; that is all sub-terms end in NIL, any user-defined functions called are named in PROGRAM-NAMES, and all references to variables are named in (STRIP-CARS ALIST). Hypothesis [2] states that BODY of the function named by PROGRAM-NAMES are proper expressions. Hypothesis [3] states that all the variables in ALIST are literal atoms.

6.2 The LR layer

The semantics of the LR layer are defined operationally by the interpreter LR-EVAL. LR-EVAL takes three arguments: FLAG which is the same mutual recursion flag as V&C$,L-EVAL and S-EVAL; L an LR-STATE, and C which is same as the clock arguments, CLK of L-EVAL and C of S-EVAL. LR-EVAL returns an LR-STATE; the P-DATA-SEGMENT field of the LR-STATE contains the representation of the result.

LR-EVAL was designed so that it reflects exactly the resource limitations of the Piton interpreter P. P takes two arguments: P a Piton state; and N a clock that indicates how many Piton instructions to execute. A P-STATE or an LR-STATE are considered erroneous if the P-PSW is something other than 'RUN or 'HALT. If in a given Piton state, P, there is not enough resources to execute the current instruction, then P will return an erroneous state. Assuming that LR-EVAL and P are given large enough clocks, LR-EVAL running on L (an LR-STATE) will return an erroneous state exactly when P running on (LR->P L) returns an erroneous state.

In the first formulation of LR-EVAL without the garbage collector, functions in the Logic were constructed that checked that there were enough resources to execute a given construct. These functions applied to an LR-STATE L, return LR-STATES with a P-PSW of 'RUN if there are enough resources; the stacks and data segment of the resulting LR-STATE are exactly the same as running P on (LR->P L).

Part of the definition of LR-EVAL is shown here:

DEFINITION

```
(LR-EVAL FLAG L C)
  =
(COND ((NOT (EQUAL (P-PSW L) 'RUN)) L)
```

```
      ((EQUAL FLAG 'LIST)
       [...]
       )
      ((ZEROP C) (LR-SET-ERROR L 'OUT-OF-TIME))
      ((LITATOM (LR-EXPR L))
       (LR-PUSH-TSTK L (LOCAL-VAR-VALUE (LR-EXPR L) (P-CTRL-STK L))))
      [...]
  )
```

The function LR-EXPR takes an LR-STATE and returns the current expression. The structure of LR-EVAL is quite similar to the structure of V&C$ and L-EVAL: a big conditional on the structure of the current expression. If LR-EVAL returns an LR-STATE with a P-PSW of 'RUN, then the TOP of P-TEMP-STK of the resulting LR-STATE is a heap address that represents the result. When the current expression is a LITATOM then LR-EVAL pushes the value of the current expression onto the temporary stack. This is accomplished by the function LR-PUSH-TSTK. The value of a LITATOM is found by calling the function LOCAL-VAR-VALUE on the LITATOM and the current control stack.

The function LR-PUSH-TSTK must check that there is enough stack space to push another object. The definition of LR-PUSH-TSTK (along with the function LR-SET-TSTK which it calls) are shown here:

DEFINITION

```
(LR-SET-TSTK S TEMP-STK)
   =
(P-STATE (P-PC S)
         (P-CTRL-STK S)
         TEMP-STK
         (P-PROG-SEGMENT S)
         (P-DATA-SEGMENT S)
         (P-MAX-CTRL-STK-SIZE S)
         (P-MAX-TEMP-STK-SIZE S)
         (P-WORD-SIZE S)
         (P-PSW S))
```

DEFINITION

```
(LR-PUSH-TSTK S VALUE)
   =
(IF (EQUAL (P-PSW S) 'RUN)
    (IF (LESSP (LENGTH (P-TEMP-STK S)) (P-MAX-TEMP-STK-SIZE S))
        (LR-SET-TSTK S (PUSH VALUE (P-TEMP-STK S)))
        (LR-SET-ERROR S 'LR-PUSH-TSTK-FULL-STACK))
    S)
```

In the current version of the compiler whenever something is put on the temporary stack, its reference count must be increased. This greatly complicates determining exactly when a resource error would occur in the corresponding Piton state. In this version of the proof, functions corresponding to each construct were defined. However instead of using the Logic to check if there are enough resources, an appropriate P-STATE is constructed and P is called with an appropriate clock.

Then an LR-STATE is constructed from the P-STATE by replacing the Piton programs with the LR programs and replacing the Piton program counter with an LR program counter.

Part of the new definition of LR-EVAL is:

DEFINITION

```
(LR-EVAL FLAG L C)
   =
(COND ((NOT (EQUAL (P-PSW L) 'RUN)) L)
      ((EQUAL FLAG 'LIST)
       [...]
       )
      ((ZEROP C) (P-HALT L 'OUT-OF-TIME))
      ((LITATOM (LR-EXPR L)) (LR-DO-LITATOM L))
      [...]
  )
```

The function LR-DO-LITATOM calls LR-DO-LITATOM-OKP to check that there enough resources. If so it executes P the appropriate number of steps (given by the function LR-DO-LITATOM-CLOCK) and then constructs, with P-SET-PC-AND-PROG-SEG, an LR-STATE to return.

The definitions of LR-DO-LITATOM, LR-DO-LITATOM-OKP and P-SET-PC-AND-PROG-SEG are:

DEFINITION

```
(P-SET-PC-AND-PROG-SEG P PC PROG-SEG)
   =
(P-STATE PC
         (P-CTRL-STK P)
         (P-TEMP-STK P)
         PROG-SEG
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         (P-PSW P))
```

DEFINITION

```
(LR-DO-LITATOM-OKP EXPR P)
(AND (LESSP (ADD1 (LENGTH (P-TEMP-STK P))) (P-MAX-TEMP-STK-SIZE P))
     (LESSP (ADD1 (UNTAG (FETCH (CDR (ASSOC EXPR
                                            (BINDINGS
                                              (TOP (P-CTRL-STK P)))))
                                (P-DATA-SEGMENT P))))
            (EXP 2 (P-WORD-SIZE P))))
```

DEFINITION

```
(LR-DO-LITATOM L)
(IF (LR-DO-LITATOM-OKP (LR-EXPR L) (LR->P L))
    (P-SET-PC-AND-PROG-SEG (P (LR->P L)
                              (P-LITATOM-CLOCK (P-DATA-SEGMENT L)))
                           (P-PC L)
                           (P-PROG-SEGMENT L))
    (LR-RESOURCE-ERROR L 'LR-DO-LITATOM))
```

6.2.1 Proper LR states

The "proper state" predicate for the LR level is called LR-PROPER-P-STATESP. The definition of LR-PROPER-P-STATESP is:

DEFINITION

```
(LR-PROPER-P-STATEP L)
   =
(AND (PROPER-P-STATEP (LR->P L))                                    ;[1]
     (DEFINEDP (AREA-NAME (P-PC L)) (P-PROG-SEGMENT L))             ;[2]
     (EQUAL (CAAR (P-PROG-SEGMENT L)) 'MAIN)                        ;[3]
     (ALL-USER-FNAMESP (CDR (STRIP-CARS (P-PROG-SEGMENT L))))       ;[4]
     (LR-PROGRAMS-PROPERP (P-PROG-SEGMENT L)                        ;[5]
                          (STRIP-LOGIC-FNAMES (CDR (P-PROG-SEGMENT L)))
                          (P-DATA-SEGMENT L))
     (LR-PROPER-P-CTRL-STKP (P-CTRL-STK L)                          ;[6]
                            (AREA-NAME (P-PC L))
                            (P-PROG-SEGMENT L)
                            (P-DATA-SEGMENT L))
     (LR-GOOD-PTRPS (P-TEMP-STK L) (P-DATA-SEGMENT L))              ;[7]
     (ALL-NOT-UNDEF-ADDRS (P-TEMP-STK L))                           ;[8]
     (LR-PROPER-HEAPP (P-DATA-SEGMENT L))                           ;[9]
     (LR-PROPER-REF-COUNTSP (LR-MAX-NODE (P-DATA-SEGMENT L))        ;[10]
                            (P-TEMP-STK L)
                            (P-CTRL-STK L)
                            (P-PROG-SEGMENT L)
                            (P-DATA-SEGMENT L))
     (EQUAL (VALUE (AREA-NAME (LR-ALLOC-NODE-JUMP-TABLE-ADDR))      ;[11]
                   (P-DATA-SEGMENT L))
            (LR-MAKE-ALLOC-NODE-JUMP-TABLE)))
```

The eleven conjuncts of this definition can be paraphrased as follows. [1] compiling L to Piton (via LR->P) produces a proper P-STATE; [2] The program named by the PC of L is defined in the L program segment; [3] the name of the first program is 'MAIN; [4] the remaining names of programs in the program segment are proper names for user-defined functions; [5] the program segment is well formed; [6] the control stack contains pointers to nodes that are not on the free list; [7] the temporary stack contains pointers to nodes that are not on the free list; [8] all the addresses on the temporary stack are not the *undefined* node; [9] the data segment forms a proper heap; [10] all the reference counts are proper; and [11] the value of the array in the global data area named by (AREA-NAME (LR-ALLOC-NODE-JUMP-TABLE-ADDR)) contains the proper table.[1]

Some of the interesting aspects of LR-PROPER-P-STATEP will be explained further here. The function LR-PROPER-HEAPP is defined as follows:

DEFINITION

```
(LR-PROPER-HEAPP DATA-SEG)
   =
(AND (LR-MINIMUM-HEAPP DATA-SEG)
```

[1] Currently the jump table used by the internal function I-ALLOC-NODE is a constant; however, when user-defined shells are added, it will need entries for each user-defined shell.

```
     (LR-NODEP (LR-MAX-NODE DATA-SEG) DATA-SEG)
     (LR-PROPER-FREE-LISTP DATA-SEG)
     (LR-PROPER-HEAPP1 (LR-MAX-NODE DATA-SEG) DATA-SEG))
```

The function LR-PROPER-FREE-LISTP is defined as:

DEFINITION

```
(LR-PROPER-FREE-LISTP DATA-SEG)
  =
(AND (ADPP (UNTAG (LR-FP-ADDR)) DATA-SEG)
     (EQUAL (FETCH (LR-MAX-NODE DATA-SEG) DATA-SEG) (LR-UNDEF-ADDR))
     (LR-CHECK-FREE-NODES (LR-FETCH-FP DATA-SEG)
                          (LR-FREE-LIST-NODES (LR-MAX-NODE DATA-SEG)
                                              DATA-SEG)
                          DATA-SEG
                          (LR-MAX-NODE DATA-SEG)))
```

LR-PROPER-FREE-LISTP needs to ensure that the free list is proper. The main property the free list needs is that changes to it should not affect objects that are still active. The free list must be scanned to ensure that none of the nodes on the free list are in use. However, since it is necessary to prove that the algorithm terminates, it is impossible to simply follow the free pointer. This is similar to the problems with defining the function VALP discussed in Section 4.4.

The solution used here consists of two steps. First, the entire heap is scanned and all nodes that could possibly be on the free list because the node's reference count field is not a Piton natural, are collected. The nodes are collected by the function LR-FREE-LIST-NODES. The second step is to start with the address stored in the Piton variable FREE-PTR and traverse the free list. The function LR-CHECK-FREE-NODES checks that each node on the free list is a member of the list of nodes, deleting each node checked from the list. This algorithm terminates since the list of nodes gets shorter on each recursion. If all nodes that are on the free list are members of the list of nodes, then since the list of nodes contains no nodes that are in use, changes to the free list cannot possibly affect nodes in use.

The function LR-PROPER-REF-COUNTSP is defined as:

DEFINITION

```
(LR-PROPER-REF-COUNTSP MAX-ADDR TEMP-STK CTRL-STK PROG-SEG DATA-SEG)
   =
(IF (LESSP (OFFSET MAX-ADDR) (LR-NODE-SIZE))
    T
    (LET ((SUB-ADDR (SUB-ADDR MAX-ADDR (LR-NODE-SIZE))))
      (AND (OR (EQUAL SUB-ADDR (LR-UNDEF-ADDR))
               (IF (EQUAL (TYPE (FETCH SUB-ADDR DATA-SEG)) 'NAT)
                   (NOT (LESSP (ADD1 (UNTAG (FETCH SUB-ADDR DATA-SEG)))
                               (TOTAL-OCCURRENCES SUB-ADDR
                                                  TEMP-STK
                                                  CTRL-STK
                                                  PROG-SEG
                                                  DATA-SEG)))
                   (EQUAL (TOTAL-OCCURRENCES SUB-ADDR
```

```
                                        TEMP-STK
                                        CTRL-STK
                                        PROG-SEG
                                        DATA-SEG)
                    0)))
        (LR-PROPER-REF-COUNTSP SUB-ADDR
                               TEMP-STK CTRL-STK PROG-SEG DATA-SEG))))
```

LR-PROPER-REF-COUNTSP ensures that the reference counts are correct. The function TOTAL-OCCURRENCES counts the number of times a node occurs in the temporary stack, control stack, the program segment, or the data segment. If the reference count field of a node is not a Piton natural then the TOTAL-OCCURRENCES of that node should be 0. If the reference count field is a Piton natural, then the TOTAL-OCCURRENCES of that node should be at least one more than the reference count that is stored. One should note that if the TOTAL-OCCURRENCES is greater than one more than the reference count, that node will not be garbage collected.

6.2.2 Proof of Correspondence

The first theorem proved about the correspondence between the S level and the LR level is:[2]

THEOREM 7 (LR-EVAL-S-EVAL-EQUIVALENCE)

```
(IMPLIES (AND (LR-PROPER-P-STATEP (S->LR1 S L TABLE))                    ;[1]
              (GOOD-POSP FLAG (S-POS S) (S-BODY (S-PROG S)))             ;[2]
              (LR-S-SIMILAR-STATESP (S-PARAMS S)                         ;[3]
                                    (S-TEMPS S)
                                    (S->LR1 S L TABLE)
                                    TABLE)
              (S-GOOD-STATEP S C)                                        ;[4]
              (EQUAL (P-PSW (LR-EVAL FLAG (S->LR1 S L TABLE) C)) 'RUN)) ;[5]
         (AND (LR-S-SIMILAR-STATESP (S-PARAMS S)
                                    (S-TEMPS (S-EVAL FLAG S C))
                                    (LR-EVAL FLAG (S->LR1 S L TABLE) C)
                                    TABLE)
              (LR-CHECK-RESULT (IF (EQUAL FLAG 'LIST) 'LIST T)
                               (S-ANS (S-EVAL FLAG S C))
                               (P-TEMP-STK (LR-EVAL FLAG
                                                    (S->LR1 S L TABLE)
                                                    C))
                               (P-DATA-SEGMENT (LR-EVAL FLAG
                                                        (S->LR1 S L TABLE)
                                                        C)))
              (EQUAL (S-ERR-FLAG (S-EVAL FLAG S C)) 'RUN)))
```

[2]This theorem is similar to a theorem with the same name that has been proven mechanically about a prototype of the implementation that did not have a garbage collector. This theorem is shown in Appendix F on page 133. Work continues on producing a mechanically checked proof for the compiler with a garbage collector added.

The function LR-S-SIMILAR-STATESP ensures that S (an S-STATE) corresponds to L (an LR-STATE). The function S->LR1 returns an LR-STATE that is the same as L except with the program segment equal to the compilation of the programs of S and the program counter corresponding to the program counter of S. The function LR-CHECK-RESULT ensures that the value returned by S-EVAL is the same as the result represented by the top of the temporary stack of the LR-STATE returned by LR-EVAL. When the first argument to LR-CHECK-RESULT, S-EVAL, and LR-EVAL is 'LIST, a list of arguments is being evaluated. LR-CHECK-RESULT, in this case, ensures the top entries on the temporary stack represent the same values the result returned by S-EVAL (which is given by the S-ANS field of the result).

The definition of LR-CHECK-RESULT is:

DEFINITION.

```
(LR-CHECK-RESULT FLAG VALUE TEMP-STK DATA-SEG)
   =
(IF (EQUAL FLAG 'LIST)
   (LR-CHECK-RESULT1 (REVERSE VALUE) TEMP-STK DATA-SEG)
   (LR-VALP VALUE (TOP TEMP-STK) DATA-SEG))
```

The proof of Theorem 7 is by a complicated induction. The induction is similar to that suggested by the definition of S-EVAL, however L must be modified similarly to the way LR-EVAL modifies it.

The following theorem is used to relieve the hypothesis labeled [5] of Theorem 7.[3]

THEOREM 8 (LR-EVAL-S-EVAL-FLAG-RUN)

```
(IMPLIES (AND (LR-PROPER-P-STATEP (LR->P (S->LR1 S L TABLE)))
              (GOOD-POSP FLAG (S-POS S) (S-BODY (S-PROG S)))
              (LR-S-SIMILAR-STATESP (S-PARAMS S)
                                    (S-TEMPS S)
                                    (S->LR1 S L TABLE)
                                    TABLE)
              (S-GOOD-STATEP S C)
              (S-ALL-TEMPS-SETP FLAG
                                (IF (EQUAL FLAG 'LIST)
                                    (S-EXPR-LIST S)
                                    (S-EXPR S))
                                (TEMP-ALIST-TO-SET (S-TEMPS S)))
              (S-ALL-PROGS-TEMPS-SETP (S-PROGS S))
              (S-CHECK-TEMPS-SETP (S-TEMPS S))
              (EQUAL (S-ERR-FLAG (S-EVAL FLAG S C)) 'RUN)
              (LR-CHECK-RESOURCESP FLAG S L C)
              (NOT (LESSP (P-WORD-SIZE L) (S-MAX-SUBR-REQS))))
         (EQUAL (P-PSW (LR-EVAL FLAG (S->LR1 S L TABLE) C)) 'RUN))
```

[3] This theorem is similar to a theorem with the same name, that has been proven mechanically about a prototype of the implementation that did not have a garbage collector. This theorem is shown in Appendix F on page 134. Work continues on producing a mechanically checked proof for the compiler with a garbage collector added.

6.3 The Piton layer

The proof of the correspondence between the LR level and Piton can be made without resorting to having similar states. The function, LR->P, which maps LR-STATEs to P-STATEs, is used in the main proof at this level. The main theorem of this level is:[4]

THEOREM 9 (LR-EVAL-P-PC-EQUIVALENCE)

```
(IMPLIES (AND (LR-PROPER-P-STATEP L)
              (GOOD-POSP FLAG
                         (OFFSET (P-PC L))
                         (PROGRAM-BODY (P-CURRENT-PROGRAM L)))
              (OR (NOT (EQUAL FLAG 'LIST))
                  (NOT (EQUAL (CAR (LR-EXPR-LIST L)) 'IF)))
              (LR-PROPER-FORMALSP (CDR (P-PROG-SEGMENT L)))
              (EQUAL (P-PSW (LR-EVAL FLAG L C)) 'RUN))
         (EQUAL (P (LR->P L) (P-CLOCK1 FLAG L C))
                (P-SET-PC (LR->P (LR-EVAL FLAG L C))
                          (P-FINAL-PC FLAG L 0))))
```

The conclusion of this theorem equates two Piton states. The left hand side of the equality is the result of running Piton on the (LR->P L), which is the compilation of L. The right hand side evaluates L (an LR-STATE) using LR-EVAL and then compiles with LR->P the result. The function LR->P produces a P-STATE that is the same as the state it is given except for the P-PC and P-PROG-SEGMENT fields. LR-EVAL does not change the P-PROG-SEGMENT field of the argument L. A state returned by LR-EVAL has a valid LR program counter in the P-PC field. However the P-PC of the result of mapping this state with LR->P is not necessarily the same as the P-PC of the left hand side. Therefore we use the function P-SET-PC that takes a P-STATE and a Piton program counter and returns a new P-STATE, which is the same as the argument but with the P-PC field set to the program counter argument.

The proof of this theorem involves showing that for each big step done by LR-EVAL, the Piton interpreter P run the proper number of steps produces the same result. In the completed proof of the compiler without the garbage collector, LR-EVAL is defined with functions that use the Logic to construct a correct LR-STATE. The proof involves showing that these manipulations are the same as done by the compiled Piton code.

In the current version of the compiler with the garbage collector, the manipulations at the LR level are done by constructing an appropriate P-STATE and applying P with an appropriate clock. Then an LR-STATE is constructed from the resulting P-STATE by putting the LR programs back and a proper LR program

[4] This theorem is similar to a theorem with the same name, that has been proven mechanically about a prototype of the implementation that did not have a garbage collector. This theorem is shown in Appendix F on page 134. Work continues on producing a mechnically checked proof for the compiler with a garbage collector added.

counter. Thus the mechanical proof of correspondence should be straightforward. That the two levels are so close seems to suggest that the LR level is not necessary and instead the compiler should just go straight to Piton. This is not so. At the LR level the function LR-PROPER-P-STATEP is the "good state" predicate on LR-STATEs. LR-PROPER-P-STATEP an invariant on LR-STATEs under LR-EVAL. Imagine a corresponding predicate P-PROPER-P-STATEP defined on P-STATE such that the following is true:

```
(IFF (LR-PROPER-P-STATEP L) (P-PROPER-P-STATEP (LR->P L)))
```

This proposed predicate would not be preserved by running P arbitrarily, because it does not hold of intermediate states.

From the theorems 2, 3, 4, 5, 6, 7, 8 and 9 the final correctness theorem, Theorem 1, named LOGIC->P-OK-REALLY (shown on page 48), can be proven.

Chapter 7

Future work

7.1 Finish compiler

The compiler and the mechanical proof are not complete at this time. The current version of the compiler (which includes the garbage collector) does not include user-defined shells and includes only the built-in SUBRS CAR, CDR, CONS, FALSE, FALSEP, LISTP, NLISTP, TRUE and TRUEP (as well as IF). The proof of the correctness of this version is not quite complete. The proof that the proper state function LR-PROPER-P-STATEP is preserved by the interpreter LR-EVAL has been completed. The proof of the theorems LR-EVAL-S-EVAL-EQUIVALENCE (on page 73) and LR-EVAL-S-EVAL-FLAG-RUN (on page 74) should be straightforward. The lemmas needed for the proof should be very similar to the lemmas need for the first prototype compiler (without a garbage collector). It is estimated that completing the proof of the prototype compiler should take another two man-months. Completion of the rest of the compiler is estimated to take two to three additional man-months. The completion of this part will involve mostly the proofs of each of the pre-defined SUBRS.

There are two possible ways to add the additional SUBRS to the compiler. They could each be coded by hand and correctness proofs could be done for each. Most of the functions could be compiled by the compiler and then hand optimized. Alternatively, most of the additional SUBRS can be compiled by the compiler, making SUBRS virtually the same as user-defined functions. L-EVAL could be modified to interpret the SUBRS compiled in this fashion. From that level on down, the SUBRS could be treated the same as user-defined functions. The functions that would have to be coded by hand are ones whose implementations are not the same as their definitions. These include ADD1, SUB1, PLUS, DIFFERENCE, QUOTIENT, REMAINDER and LESSP. In addition, PACK would have to be done by hand to implement the hash table for storing literal atoms (see section 6). Also, the following functions related to the interpreter for the Logic would have to be compiled

by hand: SUBRP, FORMALS, BODY, and APPLY-SUBR. Finally, the "definitions" of the functions COUNT and EQUAL are extended each time a new shell is added to the Logic. These two functions need to be coded by hand. All the other SUBRS can be compiled with the compiler.

The Logic is somewhat incomplete in that it does not provide functions similar to BODY and FORMALS that apply to SUBRs. It is possible to list the SUBRs that are pre-defined in the Logic; however, the user can create new SUBRs by using ADD-SHELL. There is no way in the Logic to determine how many arguments a shell constructor takes, to tell what constructor is associated with each accessor or to find the type restrictions or default functions for a shell accessor. For this reason, it is necessary to give the entire ADD-SHELL form to the compiler LOGIC->P. LOGIC->P will have to take an additional argument SHELLS that is a list of ADD-SHELL forms for the compiler. The specification of the compiler will require that the ADD-SHELL be the same as an ADD-SHELL in the current history; otherwise, the interpreter V&C$ might produce a different result than Piton running on the result produced by the compiler. This specification will require adding axioms to define a function that can extract various parts of a shell definition from its name.

Constructing a proper call to the compiler is somewhat complex. A user needs to provide all the names of the functions that need to be compiled and all the ADD-SHELL forms required. However, it would be a straightforward task to write a Common Lisp function to construct the proper call to the compiler given an expression and a variable bindings alist.

7.2 Compiling Common Lisp

Common Lisp [15] is used by a much larger community than the Logic. The Logic is similar to an applicative subset of Common Lisp. Modifying the compiler to compile this subset of Common Lisp should be straightforward. Modifying the correctness proof should not be hard either. The major difference between the Logic and Common Lisp are the different types. Integers and lists in Common Lisp are similar to naturals and lists in the Logic. Functions that operate on naturals in the Logic (e.g., PLUS) would have to be modified to Common Lisp functions that operate on integers (e.g. +). Functions that operate on lists would be quite similar. Extending the compiler to handle rational numbers would involve extensive additional work. Floating point numbers are not supported by Piton or FM8502. Theoretically, floating point numbers could be implemented in software, but this would entail much work. A proof of correctness of a compiler with floating point numbers would be quite a challenge.

Common Lisp has other types that are not in the Logic, specifically arrays of many different types. Adding arrays to a Common Lisp compiler would require a more sophisticated allocation scheme. The best solution would require different areas in the Piton data segment for allocating objects of different types. Objects of different sizes would have to be allocated in the same data area. This would require

the garbage collector to compact such data areas or risk running out of space due to fragmentation. The correctness proof of a compacting garbage collector seems challenging. Using the correctness proof on Common Lisp programs would put a large burden on the Common Lisp programmer to prove that the various areas for objects of different sizes are all large enough.

A common type of array in Common Lisp is the string. Each array element in a string (and some other types of array as well) requires fewer bits (typically 8) than a Piton word (which is 32 bits on FM8502). The usual implementation of strings packs several bytes into one word. There is no support in either Piton or FM8502 for doing byte operations. FM8502 (and subsequently Piton) could be extended to have byte operations. If Piton does not support byte operations, there are two alternatives for handling strings. The compiler could allocate a full word for each character in a string. This is quite wasteful of memory use but is efficient in terms of execution speed. The second alternative is to pack bytes together and extract the byte by using logical *and* operations with appropriate bitmasks and shifting the result. This is economical in memory use, but would execute slowly.

The verification of a "real" Common Lisp compiler would require solving several hard problems in addition to the ones mentioned above. First, Common Lisp would have to be formalized — that is the Common Lisp manual [15] would have to be described in a formal language. Second, the output language of the compiler would have to be described formally — as had been noted, Piton, as it stands now, would not be an adequate target for Common Lisp. Third, some method for verifying that the input and output of Common Lisp are correctly implemented would have to be developed. Finally to have a Common Lisp implementation that has been verified down to a gate level description, as the Nqthm compiler is, requires that the implementation of the target language be verified as well. This challenges mean that many years of research are required to produce a verified "real" compiler for a "real" language like Common Lisp.

7.3 Optimizations

The code generated by the compiler is very simple. There are many optimizations that could be added. In general, useful optimizations would be ones that eliminate updating reference counts and additional allocation.

7.3.1 Common Sub-Expressions

The first phase of the compiler was intended to eliminate common sub-expressions. The major proofs of the top level are general enough that adding a function to eliminate common sub-expressions should require only a proof that the function maintains an invariant. Of course, this does not guarantee that the function eliminates all (or even any) sub-expressions. The rest of the proof (which is the largest part) would remain the same. In a compiler for Common Lisp that includes `LET`,

a function that eliminates common sub-expressions would be less important, since the programmer has a way of saving the values of previously computed expressions anyway.

7.3.2 Other optimizations

There are many other optimizations that could be made to the code. Since the Logic has an associated theorem prover Nqthm, it might be possible to prove the validity of certain optimizations. This seems like a promising research direction.

An example of a useful optimization that might lend itself to proof is the following. Consider the function APP of Figure 3.1, which is shown here again, along with of its Piton translation.

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

(U-APP (X Y) ()
       (DL L-0 () (PUSH-LOCAL X))
       (DL L-1 () (FETCH))
       (DL L-2 () (ADD1-NAT))
       (DL L-3 () (PUSH-LOCAL X))
       (DL L-4 () (DEPOSIT))
       (DL L-5 () (PUSH-LOCAL X))
       (DL L-6 () (CALL LISTP))
       (DL L-7 () (CALL I-DECR-REF-COUNT))
       (DL L-8 () (PUSH-CONSTANT (ADDR (HEAP . 4))))
       (DL L-9 () (EQ))
       (DL L-10 () (TEST-BOOL-AND-JUMP T L-34))
       ...
       (DL L-34 () (PUSH-LOCAL Y))
       ... )
```

The Piton code for the (LISTP X), along with the code for the IF, is shown. The following steps occur when this code is executed:

1. X is pushed on the temporary stack and its reference count is incremented.
2. the function LISTP is called.
3. LISTP decrements the reference count of X.
4. LISTP pushes the address representing F or T on the stack depending on whether X points to a node with a tag of CONS.
5. The function LISTP returns.

6. The reference count of the result returned by LISTP is decremented.

7. If F was returned by the call to LISTP, jump to the instruction labeled L-24, otherwise fall through to the instruction labeled L-11.

The reference counts for X and the address returned by LISTP have both been incremented and then decremented. However, neither of these reference counts could possibly become zero. The reference count of X cannot become zero because it is always on the control stack. The reference count of the address returned by LISTP cannot become zero as it is an address representing T or F and these addresses always have a reference count that is positive. So much time is spent incrementing and decrementing reference counts with no net effect, when the only desired effect is to branch depending on whether X points to a node representing a CONS or not. The Piton code fragment for APP should be:

```
(U-APP (X Y) ()
       (DL L-0 () (PUSH-LOCAL X))               ; Push address of node
       (DL L-1 () (ADD1-ADDR))                  ; Add to get address of tag
       (DL L-2 () (FETCH))                      ; Fetch tag address
       (DL L-3 () (PUSH-CONSTANT (NAT 5)))      ; Compare to tag for CONS
       (DL L-4 () (EQ))
       (DL L-5 () (TEST-BOOL-AND-JUMP F L-34)) ; Jump if tag not CONS
       ...
       (DL L-34 () (PUSH-LOCAL Y))
       ... )
```

The resulting code is more efficient in terms of both program memory and execution time. There are several optimizations going on at once in this example. First the function LISTP is coded inline, this is not always better in terms of space. Second, the incrementing and decrementing of the reference count for X can now be taken out. Third, the incrementing and decrementing of the result of (LISTP X) can be removed and the branching can be done on the result of the comparisons of the tags.

It should be possible to generalize this somewhat. The argument X to the function LISTP need not be a parameter to the function, it need only be an expression that is guaranteed to have a reference count of at least one before it is pushed for the call. For instance, the expression (CAR X) would also have a reference count of at least one.

7.4 Other Garbage Collection Schemes

The reference count garbage collection scheme used in the Logic implementation is quite simple. Since the Logic does not contain circular structures, a reference count scheme is adequate. Common Lisp programs can create circular structures; the Common Lisp destructive routines RPLACA and RPLACD can modify the CAR and CDR fields of a CONS cell. If RPLACA and RPLACD are included in the subset

of Common Lisp that is compiled, a reference count garbage collection scheme would become very complicated. It is unlikely that `RPLACA` and `RPLACD` would become part of a source language of the compiler. This is not because it would be hard to prove the implementations of them correct, but rather that reasoning about programs that use destructive operations is generally difficult. Using the correctness theorem requires reasoning about any programs compiled.

Other garbage collection schemes can handle circular data structures.Mark and sweep garbage collection schemes can handle circular data structures[10]. They do not lend themselves well to any real time systems. Most mark and sweep garbage collection algorithms are not run until the heap is exhausted. Then the computation stops while free nodes are found. There have been adaptions to mark and sweep schemes that do a little bit of work each time an allocation is done[2]. These algorithms are more suitable to real time systems. Many garbage collection algorithms split memory in half. Allocation is done in one half until it has been filled up. Then all nodes that are in use are copied to the other half, and the first half becomes the new heap.[28] These algorithms have the advantage that memory is compacted after each garbage collection. However, they pose some problems for a formal proof. The proof would have to deal with addresses of objects changing, while the value of functions applied to those addresses should remain exactly the same. The current dynamic allocation and compiler algorithms used in the compiler only used nodes of a fixed size. The benefit of this is that the garbage collector is much simpler; the drawback is that both time and space are wasted. Future versions of the compiler should be able to handle nodes of different sizes. Finally, a garbage collector that runs on a processor separate from the processor doing the computation would be desirable. As with all proofs of programs involving parallelism, the proof of such a garbage collector would be very difficult.

7.5 Examples

The compiler has not been tested with hard examples. It is still not known how difficult it would be to apply the correctness theorem to real programs. Using the correctness theorem requires proving that the program, under some conditions on its input, does not exceed the resource limitations imposed by the Piton implementation. How difficult a task this proof would be for interesting programs is not known.

The code generated by the compiler is quite simple and straightforward. One of the goals of this project was to have the implementation of the Logic be reasonably efficient and usable. It would be useful to try the compiler on moderately large programs and compare the performance to other implementations (e.g., the Nqthm theorem prover's implementation in Common Lisp).

Appendix A

Piton translation of example program

This chapter shows the Piton state resulting from the following call to the compiler:

```
(LOGIC->P '(APP (CONS X (CHANGE-ELEMENTS Y)) '(*1*TRUE . *1*FALSE))
          (LIST (CONS 'X F) (CONS 'Y (CONS T (CONS T F))))
          '(APP CHANGE-ELEMENTS) 12 20 20 32)
```

The resulting Piton state is:

```
(P-STATE '(PC (MAIN . 0))
         '((((X ADDR (HEAP . 4))
             (Y ADDR (HEAP . 20)))
            (PC (MAIN . 0))))
         ()
         '((MAIN (X Y) ()
                 ;; Push X and increment ref count
                 (DL L-0 () (PUSH-LOCAL X))
                 (DL L-1 () (FETCH))
                 (DL L-2 () (ADD1-NAT))
                 (DL L-3 () (PUSH-LOCAL X))
                 (DL L-4 () (DEPOSIT))
                 (DL L-5 () (PUSH-LOCAL X))
                 ;; Push Y and increment ref count
                 (DL L-6 () (PUSH-LOCAL Y))
                 (DL L-7 () (FETCH))
                 (DL L-8 () (ADD1-NAT))
                 (DL L-9 () (PUSH-LOCAL Y))
                 (DL L-10 () (DEPOSIT))
                 (DL L-11 () (PUSH-LOCAL Y))
                 (DL L-12 () (CALL U-CHANGE-ELEMENTS))
                 (DL L-13 () (CALL CONS))
                 ;; Push address of constant: '(*1*TRUE . *1*FALSE)
                 ;; and increment ref count
                 (DL L-14 () (PUSH-CONSTANT (ADDR (HEAP . 16))))
                 (DL L-15 () (FETCH))
```

```
      (DL L-16 () (ADD1-NAT))
      (DL L-17 () (PUSH-CONSTANT (ADDR (HEAP . 16))))
      (DL L-18 () (DEPOSIT))
      (DL L-19 () (PUSH-CONSTANT (ADDR (HEAP . 16))))
      (DL L-20 () (CALL U-APP))
      ;; Store answer in data-segment
      (DL L-21 () (SET-GLOBAL ANSWER))
      (DL L-22 () (RET)))
(U-APP (X Y) ()
       ;; Push X and increment ref count
       (DL L-0 () (PUSH-LOCAL X))
       (DL L-1 () (FETCH))
       (DL L-2 () (ADD1-NAT))
       (DL L-3 () (PUSH-LOCAL X))
       (DL L-4 () (DEPOSIT))
       (DL L-5 () (PUSH-LOCAL X))
       (DL L-6 () (CALL LISTP))
       ;; (LISTP X) is on top of stack, decrement its ref
       ;; count and branch if it was the address of F
       (DL L-7 () (CALL I-DECR-REF-COUNT))
       (DL L-8 () (PUSH-CONSTANT (ADDR (HEAP . 4))))
       (DL L-9 () (EQ))
       (DL L-10 () (TEST-BOOL-AND-JUMP T L-34))
       ;; Push X and increment ref count
       (DL L-11 () (PUSH-LOCAL X))
       (DL L-12 () (FETCH))
       (DL L-13 () (ADD1-NAT))
       (DL L-14 () (PUSH-LOCAL X))
       (DL L-15 () (DEPOSIT))
       (DL L-16 () (PUSH-LOCAL X))
       (DL L-17 () (CALL CAR))
       ;; Push X and increments its reference count
       (DL L-18 () (PUSH-LOCAL X))
       (DL L-19 () (FETCH))
       (DL L-20 () (ADD1-NAT))
       (DL L-21 () (PUSH-LOCAL X))
       (DL L-22 () (DEPOSIT))
       (DL L-23 () (PUSH-LOCAL X))
       (DL L-24 () (CALL CDR))
       ;; Push Y and increment ref count
       (DL L-25 () (PUSH-LOCAL Y))
       (DL L-26 () (FETCH))
       (DL L-27 () (ADD1-NAT))
       (DL L-28 () (PUSH-LOCAL Y))
       (DL L-29 () (DEPOSIT))
       (DL L-30 () (PUSH-LOCAL Y))
       (DL L-31 () (CALL U-APP))
       (DL L-32 () (CALL CONS))
       (DL L-33 () (JUMP L-40))
       ;; Push Y and increment ref count
       (DL L-34 () (PUSH-LOCAL Y))
       (DL L-35 () (FETCH))
       (DL L-36 () (ADD1-NAT))
       (DL L-37 () (PUSH-LOCAL Y))
       (DL L-38 () (DEPOSIT))
```

```
        (DL L-39 () (PUSH-LOCAL Y))
        ;; Decrement reference counts of arguments
        ;; before returning
        (DL L-40 () (PUSH-LOCAL X))
        (DL L-41 () (CALL I-DECR-REF-COUNT))
        (DL L-42 () (POP))
        (DL L-43 () (PUSH-LOCAL Y))
        (DL L-44 () (CALL I-DECR-REF-COUNT))
        (DL L-45 () (POP))
        (DL L-46 () (RET)))
(U-CHANGE-ELEMENTS (LIST) ()
                   ;; Push LIST and increment ref count
                   (DL L-0 () (PUSH-LOCAL LIST))
                   (DL L-1 () (FETCH))
                   (DL L-2 () (ADD1-NAT))
                   (DL L-3 () (PUSH-LOCAL LIST))
                   (DL L-4 () (DEPOSIT))
                   (DL L-5 () (PUSH-LOCAL LIST))
                   (DL L-6 () (CALL LISTP))
                   ;; (LISTP LIST) is on top of stack,
                   ;; decrement its ref count and branch if
                   ;; it was the address of F
                   (DL L-7 () (CALL I-DECR-REF-COUNT))
                   (DL L-8 () (PUSH-CONSTANT (ADDR (HEAP . 4))))
                   (DL L-9 () (EQ))
                   (DL L-10 () (TEST-BOOL-AND-JUMP T L-55))
                   ;; Push LIST and increment ref count
                   (DL L-11 () (PUSH-LOCAL LIST))
                   (DL L-12 () (FETCH))
                   (DL L-13 () (ADD1-NAT))
                   (DL L-14 () (PUSH-LOCAL LIST))
                   (DL L-15 () (DEPOSIT))
                   (DL L-16 () (PUSH-LOCAL LIST))
                   (DL L-17 () (CALL CAR))
                   (DL L-18 () (CALL TRUEP))
                   ;; (CAR (TRUEP LIST)) is on top of stack,
                   ;; decrement its ref count and branch if
                   ;; it was the address of F
                   (DL L-19 () (CALL I-DECR-REF-COUNT))
                   (DL L-20 () (PUSH-CONSTANT (ADDR (HEAP . 4))))
                   (DL L-21 () (EQ))
                   (DL L-22 () (TEST-BOOL-AND-JUMP T L-39))
                   ;; Push address of constant F and
                   ;; increment ref count
                   (DL L-23 () (PUSH-CONSTANT (ADDR (HEAP . 4))))
                   (DL L-24 () (FETCH))
                   (DL L-25 () (ADD1-NAT))
                   (DL L-26 () (PUSH-CONSTANT (ADDR (HEAP . 4))))
                   (DL L-27 () (DEPOSIT))
                   (DL L-28 () (PUSH-CONSTANT (ADDR (HEAP . 4))))
                   ;; Push LIST and increment ref count
                   (DL L-29 () (PUSH-LOCAL LIST))
                   (DL L-30 () (FETCH))
                   (DL L-31 () (ADD1-NAT))
                   (DL L-32 () (PUSH-LOCAL LIST))
```

```
(DL L-33 () (DEPOSIT))
(DL L-34 () (PUSH-LOCAL LIST))
(DL L-35 () (CALL CDR))
(DL L-36 () (CALL U-CHANGE-ELEMENTS))
(DL L-37 () (CALL CONS))
;; Jump around else code
(DL L-38 () (JUMP L-54))
;; Push address of constant T and
;; increment ref count
(DL L-39 () (PUSH-CONSTANT (ADDR (HEAP . 8))))
(DL L-40 () (FETCH))
(DL L-41 () (ADD1-NAT))
(DL L-42 () (PUSH-CONSTANT (ADDR (HEAP . 8))))
(DL L-43 () (DEPOSIT))
(DL L-44 () (PUSH-CONSTANT (ADDR (HEAP . 8))))
;; Push LIST and increment ref count
(DL L-45 () (PUSH-LOCAL LIST))
(DL L-46 () (FETCH))
(DL L-47 () (ADD1-NAT))
(DL L-48 () (PUSH-LOCAL LIST))
(DL L-49 () (DEPOSIT))
(DL L-50 () (PUSH-LOCAL LIST))
(DL L-51 () (CALL CDR))
(DL L-52 () (CALL U-CHANGE-ELEMENTS))
(DL L-53 () (CALL CONS))
;; Jump around else code
(DL L-54 () (JUMP L-79))
(DL L-55 () (PUSH-LOCAL LIST))
(DL L-56 () (FETCH))
(DL L-57 () (ADD1-NAT))
(DL L-58 () (PUSH-LOCAL LIST))
(DL L-59 () (DEPOSIT))
(DL L-60 () (PUSH-LOCAL LIST))
(DL L-61 () (CALL TRUEP))
;; (TRUEP LIST)  is on top of stack,
;; decrement its ref count and branch if
;; it was the address of F
(DL L-62 () (CALL I-DECR-REF-COUNT))
(DL L-63 () (PUSH-CONSTANT (ADDR (HEAP . 4))))
(DL L-64 () (EQ))
(DL L-65 () (TEST-BOOL-AND-JUMP T L-73))
;; Push address of constant F and
;; increment ref count
(DL L-66 () (PUSH-CONSTANT (ADDR (HEAP . 4))))
(DL L-67 () (FETCH))
(DL L-68 () (ADD1-NAT))
(DL L-69 () (PUSH-CONSTANT (ADDR (HEAP . 4))))
(DL L-70 () (DEPOSIT))
(DL L-71 () (PUSH-CONSTANT (ADDR (HEAP . 4))))
;; Jump around else code
(DL L-72 () (JUMP L-79))
;; Push address of constant T and
;; increment ref count
(DL L-73 () (PUSH-CONSTANT (ADDR (HEAP . 8))))
(DL L-74 () (FETCH))
```

```
                    (DL L-75 () (ADD1-NAT))
                    (DL L-76 () (PUSH-CONSTANT (ADDR (HEAP . 8))))
                    (DL L-77 () (DEPOSIT))
                    (DL L-78 () (PUSH-CONSTANT (ADDR (HEAP . 8))))
                    ;; Decrement reference counts of
                    ;; arguments before returning
                    (DL L-79 () (PUSH-LOCAL LIST))
                    (DL L-80 () (CALL I-DECR-REF-COUNT))
                    (DL L-81 () (POP))
                    (DL L-82 () (RET)))
(CAR (X) ((TEMP (NAT 0)))
     ;; First decrement ref count of argument, X
     (PUSH-LOCAL X)
     (CALL I-DECR-REF-COUNT)
     ;; Test if X is a CONS, i.e. has a tag of (NAT 5)
     (PUSH-CONSTANT (NAT 1))
     (ADD-ADDR)
     (FETCH)
     (PUSH-CONSTANT (NAT 5))
     (EQ)
     (TEST-BOOL-AND-JUMP T ARG1)
     ;; Not a CONS, return address representing 0 first
     ;; incrementing ref count.
     (PUSH-CONSTANT (ADDR (HEAP . 12)))
     (FETCH)
     (ADD1-NAT)
     (PUSH-CONSTANT (ADDR (HEAP . 12)))
     (DEPOSIT)
     (PUSH-CONSTANT (ADDR (HEAP . 12)))
     (RET)
     ;; Type is the same, return CAR field first
     ;; incrementing REF COUNT
     (DL ARG1 () (PUSH-LOCAL X))
     (PUSH-CONSTANT (NAT 2))
     (ADD-ADDR)
     (FETCH)
     (SET-LOCAL TEMP)
     (FETCH)
     (ADD1-NAT)
     (PUSH-LOCAL TEMP)
     (DEPOSIT)
     (PUSH-LOCAL TEMP)
     (RET))
(CDR (X) ((TEMP (NAT 0)))
     ;; First decrement ref count of argument, X
     (PUSH-LOCAL X)
     (CALL I-DECR-REF-COUNT)
     ;; Test if X is a CONS, i.e. has a tag of (NAT 5)
     (PUSH-CONSTANT (NAT 1))
     (ADD-ADDR)
     (FETCH)
     (PUSH-CONSTANT (NAT 5))
     (EQ)
     (TEST-BOOL-AND-JUMP T ARG1)
     ;; Not a CONS, return address representing 0 first
```

```
      ;; incrementing ref count.
      (PUSH-CONSTANT (ADDR (HEAP . 12)))
      (FETCH)
      (ADD1-NAT)
      (PUSH-CONSTANT (ADDR (HEAP . 12)))
      (DEPOSIT)
      (PUSH-CONSTANT (ADDR (HEAP . 12)))
      (RET)
      ;; Type is the same, return CDR field first
      ;; incrementing REF COUNT
      (DL ARG1 () (PUSH-LOCAL X))
      (PUSH-CONSTANT (NAT 3))
      (ADD-ADDR)
      (FETCH)
      (SET-LOCAL TEMP)
      (FETCH)
      (ADD1-NAT)
      (PUSH-LOCAL TEMP)
      (DEPOSIT)
      (PUSH-LOCAL TEMP)
      (RET))
;; CONS Takes two implicit args, the CDR is on top of
;; the stack and the CAR is just below it.
(CONS () ((TEMP (ADDR (HEAP . 0))))
      ;; First call I-ALLOC-NODE to get a new node.
      (CALL I-ALLOC-NODE)
      (SET-LOCAL TEMP)        ; Put CDR in node + CDR-OFFSET
      (PUSH-CONSTANT (NAT 3))
      (ADD-ADDR)
      (DEPOSIT)
      (PUSH-LOCAL TEMP)       ; Put CAR in node + CAR-OFFSET
      (PUSH-CONSTANT (NAT 2))
      (ADD-ADDR)
      (DEPOSIT)
      (PUSH-CONSTANT (NAT 5)) ; Put tag in node
      (PUSH-LOCAL TEMP)
      (PUSH-CONSTANT (NAT 1))
      (ADD-ADDR)
      (DEPOSIT)
      (PUSH-CONSTANT (NAT 0)) ; Set ref count to 0
      (PUSH-LOCAL TEMP)
      (DEPOSIT)
      (PUSH-LOCAL TEMP)
      (RET))
(FALSE () ()
       ;; Push address of F incrementing REF COUNT
       (PUSH-CONSTANT (ADDR (HEAP . 4)))
       (FETCH)
       (ADD1-NAT)
       (PUSH-CONSTANT (ADDR (HEAP . 4)))
       (DEPOSIT)
       (PUSH-CONSTANT (ADDR (HEAP . 4)))
       (RET))
;; FALSEP takes one implicit arg on stack.
(FALSEP () ()
```

```
          ;; We are going to pop implicit arg off stack so call
          ;; I-DECR-REF-COUNT  which leaves arg on temp-stk
          (CALL I-DECR-REF-COUNT)
          ;; Test if address is address of F
          (PUSH-CONSTANT (ADDR (HEAP . 4)))
          (EQ)
          (TEST-BOOL-AND-JUMP T TRUE)
          ;; Arg was not F push F and increment ref count
          (FETCH)
          (ADD1-NAT)
          (PUSH-CONSTANT (ADDR (HEAP . 4)))
          (DEPOSIT)
          (PUSH-CONSTANT (ADDR (HEAP . 4)))
          (RET)
          ;; Arg was F push T and increment ref count
          (DL TRUE () (PUSH-CONSTANT (ADDR (HEAP . 8))))
          (FETCH)
          (ADD1-NAT)
          (PUSH-CONSTANT (ADDR (HEAP . 8)))
          (DEPOSIT)
          (PUSH-CONSTANT (ADDR (HEAP . 8)))
          (RET))
 (LISTP () ()
        ;; We are going to pop implicit arg off stack so call
        ;; I-DECR-REF-COUNT  which leaves arg on temp-stk
        (CALL I-DECR-REF-COUNT)
        ;; Now get tag of node and test if it is a CONS.
        (PUSH-CONSTANT (NAT 1))
        (ADD-ADDR)
        (FETCH)
        (PUSH-CONSTANT (NAT 5))
        (EQ)
        (TEST-BOOL-AND-JUMP F FALSE)
        ;; Node is a CONS, push T incrementing ref count
        (PUSH-CONSTANT (ADDR (HEAP . 8)))
        (FETCH)
        (ADD1-NAT)
        (PUSH-CONSTANT (ADDR (HEAP . 8)))
        (DEPOSIT)
        (PUSH-CONSTANT (ADDR (HEAP . 8)))
        (RET)
        ;; Node is NOT a CONS, push F incrementing ref count
        (DL FALSE () (PUSH-CONSTANT (ADDR (HEAP . 4))))
        (FETCH)
        (ADD1-NAT)
        (PUSH-CONSTANT (ADDR (HEAP . 4)))
        (DEPOSIT)
        (PUSH-CONSTANT (ADDR (HEAP . 4)))
        (RET))
 ;; Code for NLISTP is quite similar to code for LISTP.
(NLISTP () ()
         (CALL I-DECR-REF-COUNT)
         (PUSH-CONSTANT (NAT 1))
         (ADD-ADDR)
         (FETCH)
```

```
        (PUSH-CONSTANT (NAT 5))
        (EQ)
        (TEST-BOOL-AND-JUMP F TRUE)
        (PUSH-CONSTANT (ADDR (HEAP . 4)))
        (FETCH)
        (ADD1-NAT)
        (PUSH-CONSTANT (ADDR (HEAP . 4)))
        (DEPOSIT)
        (PUSH-CONSTANT (ADDR (HEAP . 4)))
        (RET)
        (DL TRUE () (PUSH-CONSTANT (ADDR (HEAP . 8))))
        (FETCH)
        (ADD1-NAT)
        (PUSH-CONSTANT (ADDR (HEAP . 8)))
        (DEPOSIT)
        (PUSH-CONSTANT (ADDR (HEAP . 8)))
        (RET))
(TRUE () ()
      ;; Push address of T incrementing ref count
      (PUSH-CONSTANT (ADDR (HEAP . 8)))
      (FETCH)
      (ADD1-NAT)
      (PUSH-CONSTANT (ADDR (HEAP . 8)))
      (DEPOSIT)
      (PUSH-CONSTANT (ADDR (HEAP . 8)))
      (RET))
;; Code for TRUEP is similar to code for LISTP, except
;; it tests if object on top of stack represents T,
;; that is has a tag of (NAT 3)
(TRUEP () ()
       (CALL I-DECR-REF-COUNT)
       (PUSH-CONSTANT (NAT 1))
       (ADD-ADDR)
       (FETCH)
       (PUSH-CONSTANT (NAT 3))
       (EQ)
       (TEST-BOOL-AND-JUMP F FALSE)
       (PUSH-CONSTANT (ADDR (HEAP . 8)))
       (FETCH)
       (ADD1-NAT)
       (PUSH-CONSTANT (ADDR (HEAP . 8)))
       (DEPOSIT)
       (PUSH-CONSTANT (ADDR (HEAP . 8)))
       (RET)
       (DL FALSE () (PUSH-CONSTANT (ADDR (HEAP . 4))))
       (FETCH)
       (ADD1-NAT)
       (PUSH-CONSTANT (ADDR (HEAP . 4)))
       (DEPOSIT)
       (PUSH-CONSTANT (ADDR (HEAP . 4)))
       (RET))
(I-DECR-REF-COUNT (X) ()
                  ;; See if ref count is 0, then we need to
                  ;; put it on FREE-LIST
                  (PUSH-LOCAL X)
```

```
                    (FETCH)
                    (TEST-NAT-AND-JUMP ZERO RETURN-IT)
                    (PUSH-LOCAL X)
                    (FETCH)
                    (SUB1-NAT)
                    (PUSH-LOCAL X)
                    (DEPOSIT)
                    (PUSH-LOCAL X)
                    (RET)
                    ;; First make X point to first node
                    ;; on FREE-LIST
                    (DL RETURN-IT () (PUSH-GLOBAL FREE-PTR))
                    (PUSH-LOCAL X)
                    (DEPOSIT)
                    ;; Now make FREE-PTR point to X
                    (PUSH-LOCAL X)
                    (SET-GLOBAL FREE-PTR)
                    (RET))
(I-ALLOC-NODE () ((X (NAT 0)))
                (PUSH-CONSTANT (ADDR (ALLOC-NODE-JUMP-TABLE . 0)))
                (PUSH-GLOBAL FREE-PTR)
                (PUSH-CONSTANT (NAT 1))
                (ADD-ADDR)
                (FETCH)
                (ADD-ADDR)
                ;; Tag has to be one in [0..5] if the data-segment is
                ;; LR-PROPER-HEAPP, i.e. tag is one of
                ;; (LR-UNDEFINED-TAG) (LR-INIT-TAG) (LR-FALSE-TAG)
                ;; (LR-TRUE-TAG) (LR-ADD1-TAG)  or (LR-CONS-TAG).
                ;; However there is code for PACK and MINUS and
                ;; user-defined shells.  Also UNDEFINED, FALSE
                ;; and TRUE can't really happen.  For the
                ;; ones that can't happen we loop forever.
                (FETCH)
                (POPJ)

                ;; For UNDEFINED and F
                (DL CANT-HAPPEN () (JUMP CANT-HAPPEN))

                ;; We can just use nodes with tag INIT, we only
                ;; need take them off the free list.  We are
                ;; here if tag was TRUE as well.
                (DL INIT () (PUSH-GLOBAL FREE-PTR))
                (PUSH-GLOBAL FREE-PTR)
                (FETCH)
                (POP-GLOBAL FREE-PTR)
                (RET)

                ;; We need to follow the LR-BIGNUM-OFFSET
                ;; until we get to LR-UNDEFINED-TAG (which
                ;; is found immediately for fixnums).
                (DL ADD1 () (PUSH-GLOBAL FREE-PTR))
                (PUSH-GLOBAL FREE-PTR)
                (FETCH)
                (POP-GLOBAL FREE-PTR)
```

```
(SET-LOCAL X)
(DL ADD1-LOOP () (PUSH-CONSTANT (NAT 3)))
(ADD-ADDR)
(FETCH)
(PUSH-CONSTANT (ADDR (HEAP . 0)))
(EQ)
(TEST-BOOL-AND-JUMP T ADD1-DONE)
;; We have a bignum, put INIT-TAG as tag
;; and put at front of free-list
(PUSH-CONSTANT (NAT 1))
(PUSH-LOCAL X)
(PUSH-CONSTANT (NAT 1))
(ADD-ADDR)
(DEPOSIT)
;; Set ref count to point to first element of FREE-LIST
(PUSH-GLOBAL FREE-PTR)
(PUSH-LOCAL X)
(DEPOSIT)
(PUSH-LOCAL X)
(SET-GLOBAL FREE-PTR)
(PUSH-CONSTANT (NAT 3))
(ADD-ADDR)
(SET-LOCAL X)
(JUMP ADD1-LOOP)
(DL ADD1-DONE () (PUSH-LOCAL X))
(RET)

;; First remove first node from free list, saving
;; in local X.
(DL CONS () (PUSH-GLOBAL FREE-PTR))
(SET-LOCAL X)
(FETCH)
(POP-GLOBAL FREE-PTR)
(PUSH-LOCAL X)
(PUSH-CONSTANT (NAT 2))
(ADD-ADDR)
(FETCH)
(CALL I-DECR-REF-COUNT)
(POP)
(PUSH-LOCAL X)
(PUSH-CONSTANT (NAT 3))
(ADD-ADDR)
(FETCH)
(CALL I-DECR-REF-COUNT)
(POP)
(PUSH-LOCAL X)
(RET)

;; Can't happen, yet.
(DL PACK () (JUMP PACK))

;; Can't happen, yet.
(DL MINUS () (JUMP MINUS))
;; Can't happen, yet.
(DL USER-DEFINED () (JUMP USER-DEFINED))))
```

```
'((FREE-PTR (ADDR (HEAP . 24)))
  (ANSWER (NAT 0))
  (ALLOC-NODE-JUMP-TABLE ((PC (I-ALLOC-NODE . 8))
                          (PC (I-ALLOC-NODE . 9))
                          (PC (I-ALLOC-NODE . 8))
                          (PC (I-ALLOC-NODE . 9))
                          (PC (I-ALLOC-NODE . 14))
                          (PC (I-ALLOC-NODE . 41))
                          (PC (I-ALLOC-NODE . 59))
                          (PC (I-ALLOC-NODE . 60))))
  (HEAP ;;  Undefined node [Heap address 0]
        (NAT 0) (NAT 0) (ADDR (HEAP . 0)) (ADDR (HEAP . 0))
        ;; FALSE  [Heap address 4]
        (NAT 4) (NAT 2) (ADDR (HEAP . 0)) (ADDR (HEAP . 0))
        ;; TRUE  [Heap address 8]
        (NAT 4) (NAT 3) (ADDR (HEAP . 0)) (ADDR (HEAP . 0))
        ;; ZERO  [Heap address 12]
        (NAT 0) (NAT 4) (NAT 0) (ADDR (HEAP . 0))
        ;; (CONS T F)  [Heap address 16]
        (NAT 1) (NAT 5) (ADDR (HEAP . 8)) (ADDR (HEAP . 4))
        ;; (CONS T (CONS T F))  [Heap address 20]
        (NAT 0) (NAT 5) (ADDR (HEAP . 8)) (ADDR (HEAP . 16))
        ;;  Unused node  [Heap address 24]
        (ADDR (HEAP . 28)) (NAT 1) (NAT 0) (NAT 0)
        ;;  Unused node  [Heap address 28]
        (ADDR (HEAP . 32)) (NAT 1) (NAT 0) (NAT 0)
        ;;  Unused node  [Heap address 32]
        (ADDR (HEAP . 36)) (NAT 1) (NAT 0) (NAT 0)
        ;;  Unused node  [Heap address 36]
        (ADDR (HEAP . 40)) (NAT 1) (NAT 0) (NAT 0)
        ;;  Unused node  [Heap address 40]
        (ADDR (HEAP . 44)) (NAT 1) (NAT 0) (NAT 0)
        ;;  Unused node  [Heap address 44]
        (ADDR (HEAP . 48)) (NAT 1) (NAT 0) (NAT 0)
        ;;  last node on free list  [Heap address 48]
        (NAT 1)))
20 20 32 'RUN)
```

Appendix B

Function Definitions for the Implementation

This section lists in all the functions used in the compiler. The following functions are used from the definition of Piton (in Appendix D) and are not listed ADD-ADDR, ADD1-ADDR, AREA-NAME, DEFINEDP, DEFINITION, DEPOSIT, DL, FORMAL-VARS, GET, NAME, OFFSET, PC, P-CURRENT-PROGRAM, P-FRAME, P-STATE, PROGRAM-BODY, STRIP-CDRS, TAG, TEMP-VAR-DCLS. TOP and VALUE. The functions FIRST, LENGTH, PLIST and RESTN are defined in the Lists library (in Appendix Section C.4).

DEFINITION
```
(S-TEMP-EVAL) = '(TEMP-EVAL)
```

DEFINITION
```
(S-TEMP-FETCH) = '(TEMP-FETCH)
```

DEFINITION
```
(S-TEMP-TEST) = '(TEMP-TEST)
```

```
;; LOGIC->S returns an S-STATE.  The interpreter S-EVAL
;; interprets S-STATEs
```

SHELL DEFINITION
add the shell S-STATE of 7 arguments, with
recognizer function symbol S-STATEP, and
accessors S-PNAME, S-POS, S-ANS, S-PARAMS,
S-TEMPS, S-PROGS and S-ERR-FLAG.

DEFINITION

```
(USER-FNAME-PREFIX)
   =
(LIST (CAR (UNPACK 'U-)) (CADR (UNPACK 'U-)))

;; U- is prefixed to user-definedp functions to distinguish them
;; from MAIN and internal functions
```

DEFINITION

```
(USER-FNAME NAME)
   =
(PACK (APPEND (USER-FNAME-PREFIX) (UNPACK NAME)))
```

DEFINITION

```
(S-CONSTRUCT-PROGRAMS FUN-LIST)
   =
;; S-CONSTRUCT-PROGRAMS constructs programs at the S level.  The
;; body of the programs is the same as the logic.  This function
;; would need to be modified if commons sub-expressions are
;; eliminated.
(IF (LISTP FUN-LIST)
    (CONS (LIST (USER-FNAME (CAR FUN-LIST))   ; name
                (FORMALS (CAR FUN-LIST))      ; formals
                NIL                           ; temps
                (BODY (CAR FUN-LIST)))
          (S-CONSTRUCT-PROGRAMS (CDR FUN-LIST)))
    NIL)
```

DEFINITION

```
(DELETE-ALL X L)
   =
(IF (LISTP L)
    (IF (EQUAL X (CAR L))
        (DELETE-ALL X (CDR L))
        (CONS (CAR L) (DELETE-ALL X (CDR L))))
    L)
```

DEFINITION

```
(REMOVE-DUPLICATES L)
   =
(IF (LISTP L)
    (CONS (CAR L) (REMOVE-DUPLICATES (DELETE-ALL (CAR L) (CDR L))))
    L)
```

DEFINITION

```
(LOGIC->S EXPR ALIST FUN-NAMES)
   =
(S-STATE 'MAIN
         NIL
```

```
        NIL
        ALIST
        NIL
        (CONS (LIST 'MAIN (STRIP-CARS ALIST) NIL EXPR)
              (S-CONSTRUCT-PROGRAMS (REMOVE-DUPLICATES FUN-NAMES)))
        'RUN)

;; S-FORMALS, S-TEMP-LIST and S-BODY are accessors for
;; programs constructed by S-CONSTRUCT-PROGRAMS.
```

DEFINITION

```
(S-FORMALS S-PROGRAM)
   =
(CADR S-PROGRAM)
```

DEFINITION

```
(S-TEMP-LIST S-PROGRAM)
   =
(CADDR S-PROGRAM)
```

DEFINITION

```
(S-BODY S-PROGRAM)
   =
(CADDDR S-PROGRAM)
```

DEFINITION

```
(S-PROG S)
   =
(DEFINITION (S-PNAME S) (S-PROGS S))
```

```
;; We used to have an LR-STATE shell.  Now we just use a
;;  P-STATE  shell.  However we refer to LR-STATES which are
;; P-STATEs with LR level programs.  The function
;;  LR->P compiles an LR-STATE to a Piton state, by
;; compiling the programs and converting the P-PC to a Piton
;; PC.  We use P-STATE shells instead of LR-STATE shells
;; because we used to have define functions analogous to
;;  P-OBJECTP (and functions that called P-OBJECTP) that
;; took LR-STATES or parts thereof.  We use the Piton notion
;; of a PROPER state.  It should be the case that all the
;; LR-STATEs we are interested in are
;; PROPER-P-STATEPs after we apply LR->P to them.

;; An LR PC object is a combination of a Piton PC object and an S level
;;  S-PNAME and S-POS.  The translation of (S-PNAME S)
;; and (S-POS S) from the S level is:
;;(TAG 'PC (CONS (S-PNAME S) (S-POS S)))

;; Each element of P-PROG-SEGMENT is a program.  A program is a list
;; of the form:
;;
;; (name (formal1 formal2 ... formaln)
;;       ((temp1 init1)
;;        ...
;;        (tempk initk))
;;       body)
;;
;; The name and each formal and temp is a symbol.  The initial values
;; of the temps are tagged values.  Body is a form similar to that for
;; the S level, but temporary expressions have been replaced the name of
;; a temporary variable added to them
;; e.g. ((S-TEMP-EVAL) <expr>) -> ((S-TEMP-EVAL) <expr> <var>).
;; In the case of (S-TEMP-FETCH) <expr> is never used but we put it
;; in for consistency and so it is easier to convert back to S-STATES.
;; Also (QUOTE <expr>) has been replaced by (QUOTE <addr>)
;; where <addr> is a heap address that represents <expr>.


;; Defined values to use as tags for nodes in the heap.

DEFINITION
(LR-UNDEFINED-TAG) = 0                  ; Tag of UNDEFINED node.


DEFINITION
(LR-INIT-TAG) = 1                       ; Used in nodes on FREE-LIST
                                          ; that have not been used


DEFINITION
(LR-FALSE-TAG) = 2


DEFINITION
```

```
(LR-TRUE-TAG) = 3
```

DEFINITION
```
(LR-ADD1-TAG) = 4
```

DEFINITION
```
(LR-CONS-TAG) = 5
```

DEFINITION
```
(LR-PACK-TAG) = 6
```

DEFINITION
```
(LR-MINUS-TAG) = 7
```

DEFINITION
```
;; LR-HEAP-NAME is the name of the data area used to store the heap.
(LR-HEAP-NAME) = 'HEAP

;; Defined functions to represent constant data addresses.
```

DEFINITION
```
;; LR-NODE-SIZE is the size of a node.
(LR-NODE-SIZE) = 4
```

DEFINITION
```
(LR-UNDEF-ADDR) = (TAG 'ADDR '(HEAP . 0))
```

DEFINITION
```
(LR-F-ADDR) = (ADD-ADDR (LR-UNDEF-ADDR) (LR-NODE-SIZE))
```

DEFINITION
```
(LR-T-ADDR) = (ADD-ADDR (LR-F-ADDR) (LR-NODE-SIZE))
```

DEFINITION
```
(LR-0-ADDR) = (ADD-ADDR (LR-T-ADDR) (LR-NODE-SIZE))
```

DEFINITION
```
(LR-FP-ADDR) = (TAG 'ADDR '(FREE-PTR . 0))
```

DEFINITION
```
(LR-ANSWER-ADDR) = (TAG 'ADDR '(ANSWER . 0))
```

DEFINITION

```
(LR-ALLOC-NODE-JUMP-TABLE-ADDR)
   =
(TAG 'ADDR '(ALLOC-NODE-JUMP-TABLE . 0))
```

DEFINITION

```
(LR-FETCH-FP DATA-SEG)
   =
(FETCH (LR-FP-ADDR) DATA-SEG)
```

DEFINITION

```
(LR-MINIMUM-HEAP-SIZE)
   =
(OFFSET (ADD-ADDR (LR-0-ADDR) (LR-NODE-SIZE)))
```

```
;; The heap is a (presumably large) Piton data area.  It contains Nodes which
;; are four words.  One word is for the reference count, one for the tag,
;; and two for the contents.  Some data-types only require one word for the
;; contents (e.g. FIXNUMS) in that case one word is wasted.  Some
;; (user-defined) data-types require more than two words.  In this case the
;; second word is a pointer to another node.  This contains up to three
;; words of data, the fourth word (if the data type needs more that four
;; words) is used to link another node with the same format.
```

DEFINITION

```
;; LR-NEW-NODE returns another node to be stuck in memory
(LR-NEW-NODE REF-COUNT TAG VALUE1 VALUE2)
   =
(LIST REF-COUNT TAG VALUE1 VALUE2)
```

```
;; Defined offsets for various fields of a node.
```

DEFINITION

```
(LR-TAG-OFFSET) = 1
```

DEFINITION

```
(LR-CAR-OFFSET) = 2
```

DEFINITION

```
(LR-CDR-OFFSET) = 3
```

DEFINITION

```
(LR-UNBOX-NAT-OFFSET) = 2
```

DEFINITION

```
;; LR-BIGNUM-OFFSET is where continuation nodes are stored for bignums
(LR-BIGNUM-OFFSET) = 3
```

DEFINITION
```
(LR-UNPACK-OFFSET) = 2
```

DEFINITION
```
(LR-NEGATIVE-GUTS-OFFSET) = 2
```

DEFINITION
```
;; LR-MAKE-PROGRAM is accessed by the Piton accessors: NAME,
;; FORMAL-VARS, TEMP-VAR-DCLS  and PROGRAM-BODY.  Also
;; LOCAL-VARS.
(LR-MAKE-PROGRAM NAME FORMALS TEMPS BODY)
   =
(CONS NAME (CONS FORMALS (CONS TEMPS BODY)))
```

DEFINITION
```
(ASCII-0) = 48
```

DEFINITION
```
(ASCII-1) = 49
```

DEFINITION
```
(ASCII-9) = 57
```

DEFINITION
```
(ASCII-DASH) = 45
```

DEFINITION
```
(LIST-ASCII-0) = (LIST (ASCII-0))
```

DEFINITION
```
(LIST-ASCII-1) = (LIST (ASCII-1))
```

;; The following functions are used in the definitions of GENSYM
;; which returns a literal atom guaranteed to be different.

DEFINITION
```
(INCREMENT-NUMLIST NUMLIST)
   =
;; NUMLIST is a list of Ascii codes of digits, units digit first
(IF (LISTP NUMLIST)
    (IF (EQUAL (CAR NUMLIST) (ASCII-9))
        (CONS (ASCII-0) (INCREMENT-NUMLIST (CDR NUMLIST)))
        (CONS (ADD1 (CAR NUMLIST)) (CDR NUMLIST)))
    (LIST-ASCII-1))
```

DEFINITION

```
(MAKE-SYMBOL INITIAL DIGIT-LIST)
   =
(PACK (APPEND (APPEND INITIAL DIGIT-LIST) 0))
```

DEFINITION

```
(COUNT-CODELIST1 NUMLIST)
   =
;; NUMLIST is a list of Ascii codes, units digit first if we
;; think of them as being codes of digits of a number
(IF (LISTP NUMLIST)
    (PLUS (CAR NUMLIST)
          (TIMES 10 (COUNT-CODELIST1 (CDR NUMLIST))))
    0)
```

DEFINITION

```
(SUBSEQP LIST1 LIST2)
   =
(AND (NOT (LESSP (LENGTH LIST2) (LENGTH LIST1)))
     (EQUAL (FIRSTN (LENGTH LIST1) LIST2) LIST1))
```

DEFINITION

```
(COUNT-CODELIST INITIAL ASCII-list)
   =
(IF (SUBSEQP INITIAL ASCII-LIST)
    (COUNT-CODELIST1 (RESTN (LENGTH INITIAL) ASCII-LIST))
    0)
```

DEFINITION

```
(MAX-COUNT-CODELIST INITIAL LIST)
   =
(IF (LISTP LIST)
    (MAX (COUNT-CODELIST INITIAL (UNPACK (CAR LIST)))
         (MAX-COUNT-CODELIST INITIAL (CDR LIST)))
    0)
```

DEFINITION

```
;; GENSYM returns a pair, the new symbol and the next number to use
(GENSYM INITIAL NUM-LIST ATOM-LIST)
   =
(IF (MEMBER (MAKE-SYMBOL INITIAL NUM-LIST) ATOM-LIST)
    (GENSYM INITIAL (INCREMENT-NUMLIST NUM-LIST) ATOM-LIST)
    (CONS (MAKE-SYMBOL INITIAL NUM-LIST)
          (INCREMENT-NUMLIST NUM-LIST)))

;; The following lemma can be proven about GENSYM:
;; (prove-lemma GENSYM-IS-NEW (rewrite)
;;  (not (member (car (gensym initial num-list atom-list)) atom-list)))
```

DEFINITION

```
;; MAKE-TEMP-NAME-ALIST takes a temps-alist triple a la S-TEMPS and
;; returns an alist with entries of the form:
;; (<temp expression> . <variable>) where <variable> is guaranteed to
;; occur only once in the resulting alist and is guaranteed not to occur
;; in FORMALS.
(LR-MAKE-TEMP-NAME-ALIST-1 INITIAL NUM-LIST TEMP-LIST FORMALS)
   =
(IF (LISTP TEMP-LIST)
    (LET ((GENSYM (GENSYM INITIAL NUM-LIST FORMALS)))
      (CONS (CONS (CAR TEMP-LIST) (CAR GENSYM))
            (LR-MAKE-TEMP-NAME-ALIST-1 INITIAL
                                       (CDR GENSYM)
                                       (CDR TEMP-LIST)
                                       FORMALS)))
    NIL)
```

DEFINITION

```
(LR-MAKE-TEMP-NAME-ALIST TEMP-LIST FORMALS)
   =
(LR-MAKE-TEMP-NAME-ALIST-1 (UNPACK 'T*) (LIST-ASCII-0) TEMP-LIST FORMALS)
```

DEFINITION

```
(LR-NEW-CONS CAR CDR)
   =
(LR-NEW-NODE (TAG 'NAT (LR-CONS-TAG)) (TAG 'NAT 1) CAR CDR)
```

DEFINITION

```
;;DEPOSIT-A-LIST  deposits LIST of objects at ADDR, ADDR+1,
;;ADDR+2, ... in DATA-SEG.
(DEPOSIT-A-LIST LIST ADDR DATA-SEG)
   =
(IF (LISTP LIST)
    (DEPOSIT (CAR LIST)
             ADDR
             (DEPOSIT-A-LIST (CDR LIST)
                             (ADD1-ADDR ADDR)
                             DATA-SEG))
    DATA-SEG)
```

DEFINITION

```
(PAIR-FORMALS-WITH-ADDRESSES FORMALS TABLE)
   =
(IF (LISTP FORMALS)
    (CONS (CONS (CAAR FORMALS) (CDR (ASSOC (CDAR FORMALS) TABLE)))
          (PAIR-FORMALS-WITH-ADDRESSES (CDR FORMALS) TABLE))
    NIL)
```

DEFINITION

```
(LR-MAKE-INITIAL-TEMPS TEMP-VARS)
```

```
   =
(IF (LISTP TEMP-VARS)
    (CONS (CONS (CAR TEMP-VARS) (LR-UNDEF-ADDR))
          (LR-MAKE-INITIAL-TEMPS (CDR TEMP-VARS)))
    NIL)
```

DEFINITION

```
;; LR-INITIAL-CSTK constructs the initial control stack.
(LR-INITIAL-CSTK PARAMS TEMP-ALIST TABLE PC)
   =
(LIST (P-FRAME (APPEND (PAIR-FORMALS-WITH-ADDRESSES PARAMS TABLE)
                       (LR-MAKE-INITIAL-TEMPS (STRIP-CDRS TEMP-ALIST)))
               PC))
```

DEFINITION

```
;; Compiles BODY.  TEMP-ALIST is an alist mapping temporary
;; expressions to variables.  CONST-TABLE is an alist mapping
;; constants to addresses.
(LR-COMPILE-BODY FLAG BODY TEMP-ALIST CONST-TABLE)
   =
(IF (EQUAL FLAG 'LIST)
    (IF (LISTP BODY)
        (CONS (LR-COMPILE-BODY T (CAR BODY) TEMP-ALIST CONST-TABLE)
              (LR-COMPILE-BODY 'LIST (CDR BODY) TEMP-ALIST CONST-TABLE))
        NIL)
    (IF (LISTP BODY)
        (COND ((OR (EQUAL (CAR BODY) (S-TEMP-FETCH))
                   (EQUAL (CAR BODY) (S-TEMP-EVAL))
                   (EQUAL (CAR BODY) (S-TEMP-TEST)))
               (LIST (CAR BODY)
                     (LR-COMPILE-BODY T (CADR BODY) TEMP-ALIST CONST-TABLE)
                     (VALUE (CADR BODY) TEMP-ALIST)))
              ((EQUAL (CAR BODY) 'QUOTE)
               (LIST 'QUOTE (VALUE (CADR BODY) CONST-TABLE)))
              (T (CONS (CAR BODY)
                       (LR-COMPILE-BODY 'LIST
                                        (CDR BODY)
                                        TEMP-ALIST
                                        CONST-TABLE))))
        BODY))
```

DEFINITION

```
;; LR-MAKE-TEMP-VAR-DCLS takes TEMP-ALIST as above and
;; constructs a new alist suitable for using as a Piton
;; TEMP-VAR-DECLS alist.  Each temporary is bound to the address
;; of the UNDEFINED node.
(LR-MAKE-TEMP-VAR-DCLS TEMP-ALIST)
   =
(IF (LISTP TEMP-ALIST)
    (CONS (LIST (CDAR TEMP-ALIST) (LR-UNDEF-ADDR))
          (LR-MAKE-TEMP-VAR-DCLS (CDR TEMP-ALIST)))
    NIL)
```

DEFINITION

```
;; LR-COMPILE-PROGRAMS constructs LR programs from S programs.
(LR-COMPILE-PROGRAMS PROGRAMS CONST-TABLE)
   =
(IF (LISTP PROGRAMS)
    (LET ((PROG (CAR PROGRAMS)))
      (LET ((TEMP-ALIST (LR-MAKE-TEMP-NAME-ALIST (S-TEMP-LIST PROG)
                                                 (S-FORMALS PROG))))
        (CONS (LR-MAKE-PROGRAM (CAR PROG)
                               (S-FORMALS PROG)
                               (LR-MAKE-TEMP-VAR-DCLS TEMP-ALIST)
                               (LR-COMPILE-BODY T
                                                (S-BODY PROG)
                                                TEMP-ALIST
                                                CONST-TABLE))
              (LR-COMPILE-PROGRAMS (CDR PROGRAMS) CONST-TABLE))))
    NIL)

;; The following functions are used to generate the Piton code for
;; the run-time support functions.
```

DEFINITION

```
(P-I-DECR-REF-COUNT-CODE)
   =
(LIST 'I-DECR-REF-COUNT '(X) ()
      ;; See if ref count is 0, then we need to put it on FREE-LIST
      '(PUSH-LOCAL X)
      '(FETCH)
      '(TEST-NAT-AND-JUMP ZERO RETURN-IT)
      '(PUSH-LOCAL X)
      '(FETCH)
      '(SUB1-NAT)
      '(PUSH-LOCAL X)
      '(DEPOSIT)
      '(PUSH-LOCAL X)
      '(RET)
      ;; First make X point to first node on FREE-LIST
      (LIST 'DL 'RETURN-IT () (LIST 'PUSH-GLOBAL (AREA-NAME (LR-FP-ADDR))))
      '(PUSH-LOCAL X)
      '(DEPOSIT)
      ;; Now make FREE-PTR point to X
      '(PUSH-LOCAL X)
      (LIST 'SET-GLOBAL (AREA-NAME (LR-FP-ADDR)))
      '(RET))
```

DEFINITION

```
(P-I-ALLOC-NODE-CODE)
   =
(LIST 'I-ALLOC-NODE () '((X (NAT 0)))
      (LIST 'PUSH-CONSTANT (LR-ALLOC-NODE-JUMP-TABLE-ADDR))
      '(PUSH-GLOBAL FREE-PTR)
```

```
(LIST 'PUSH-CONSTANT (TAG 'NAT (LR-TAG-OFFSET)))
'(ADD-ADDR)
'(FETCH)
'(ADD-ADDR)
;; Tag has to be one in [0..5] if the data-segment is
;; LR-PROPER-HEAPP, i.e. tag is one of
;; (LR-UNDEFINED-TAG) (LR-INIT-TAG) (LR-FALSE-TAG)
;; (LR-TRUE-TAG) (LR-ADD1-TAG)  or (LR-CONS-TAG).   However
;; there is code for PACK and MINUS and user-defined shells.
;; Also UNDEFINED, FALSE and TRUE can't really
;; happen.  For the ones that can't happen we loop forever.
'(FETCH)
'(POPJ)

'(DL CANT-HAPPEN () (JUMP CANT-HAPPEN)) ; For UNDEFINED and F

;; We can just use nodes with tag INIT, we only need take them off
;; the free list.
;; We are here if tag was TRUE as well.
;; LR-PROPER-HEAPP allows multiple instances of TRUE, however
;; there really are not more than one.
'(DL INIT () (PUSH-GLOBAL FREE-PTR)) ; This is the node to return
'(PUSH-GLOBAL FREE-PTR)
'(FETCH)
'(POP-GLOBAL FREE-PTR)
'(RET)

;; We need to follow the LR-BIGNUM-OFFSET until we get to
;; LR-UNDEFINED-TAG  (which is found immediately for fixnums).
'(DL ADD1 ()
     (PUSH-GLOBAL FREE-PTR))          ; X will be set to this
'(PUSH-GLOBAL FREE-PTR)               ; take first node off free list
'(FETCH)
'(POP-GLOBAL FREE-PTR)
'(SET-LOCAL X)
(LIST 'DL 'ADD1-LOOP ()
      (LIST 'PUSH-CONSTANT (TAG 'NAT (LR-BIGNUM-OFFSET))))
'(ADD-ADDR)
'(FETCH)
(LIST 'PUSH-CONSTANT (LR-UNDEF-ADDR))
'(EQ)
'(TEST-BOOL-AND-JUMP T ADD1-DONE)
;; We have a bignum, put INIT-TAG as tag and put at front of
;; free-list
(LIST 'PUSH-CONSTANT (TAG 'NAT (LR-INIT-TAG)))
'(PUSH-LOCAL X)
(LIST 'PUSH-CONSTANT (TAG 'NAT (LR-TAG-OFFSET)))
'(ADD-ADDR)
'(DEPOSIT)
'(PUSH-GLOBAL FREE-PTR)               ; Set ref count to point
'(PUSH-LOCAL X)                       ; to first element of FREE-LIST
'(DEPOSIT)
'(PUSH-LOCAL X)                       ; Make FREE-PTR point to this node
'(SET-GLOBAL FREE-PTR)
(LIST 'PUSH-CONSTANT (TAG 'NAT (LR-BIGNUM-OFFSET)))
```

```
      '(ADD-ADDR)
      '(SET-LOCAL X)
      '(JUMP ADD1-LOOP)
      '(DL ADD1-DONE () (PUSH-LOCAL X))
      '(RET)

      ;; First remove first node from free list, saving in local X.
      '(DL CONS () (PUSH-GLOBAL FREE-PTR))
      '(SET-LOCAL X)
      '(FETCH)                              ; FREE-PTR <- FREE-PTR.NEXT
      '(POP-GLOBAL FREE-PTR)

      ;; Now decrement CAR and CDR ref counts
      '(PUSH-LOCAL X)
      (LIST 'PUSH-CONSTANT (TAG 'NAT (LR-CAR-OFFSET)))
      '(ADD-ADDR)
      '(FETCH)
      '(CALL I-DECR-REF-COUNT)              ; I-DECR-REF-COUNT leaves
      '(POP)                                ; address on top of stack.
      '(PUSH-LOCAL X)
      (LIST 'PUSH-CONSTANT (TAG 'NAT (LR-CDR-OFFSET)))
      '(ADD-ADDR)
      '(FETCH)
      '(CALL I-DECR-REF-COUNT)              ; I-DECR-REF-COUNT leaves
      '(POP)                                ; address on top of stack.
      '(PUSH-LOCAL X)
      '(RET)

      '(DL PACK () (JUMP PACK))             ; Can't happen, yet.

      '(DL MINUS () (JUMP MINUS))           ; Can't happen, yet.

      '(DL USER-DEFINED () (JUMP USER-DEFINED)) ; Can't happen, yet.
      )
```

DEFINITION

```
(P-RECOGNIZER-CODE NAME TAG)
   =
;; P-RECOGNIZER-CODE generates code for shell recognizer, e.g.
;; LISTP.  The argument is passed on temporary stack implicitly.
(LIST NAME '() '()
      ;; We are going to pop implicit arg off stack so call
      ;; I-DECR-REF-COUNT  which leaves arg on temp-stk
      '(CALL I-DECR-REF-COUNT)
      ;; Now get tag of node and test against desired node.
      (LIST 'PUSH-CONSTANT (TAG 'NAT (LR-TAG-OFFSET)))
      '(ADD-ADDR)
      '(FETCH)
      (LIST 'PUSH-CONSTANT (TAG 'NAT TAG))
      '(EQ)
      '(TEST-BOOL-AND-JUMP F FALSE)
      ;; Node is of desired type, push T incrementing ref count
      (LIST 'PUSH-CONSTANT (LR-T-ADDR))
      '(FETCH)
```

```
      '(ADD1-NAT)
      (LIST 'PUSH-CONSTANT (LR-T-ADDR))
      '(DEPOSIT)
      (LIST 'PUSH-CONSTANT (LR-T-ADDR))
      '(RET)
      ;; Node is NOT of desired type, push F incrementing ref count
      (LIST 'DL 'FALSE '() (LIST 'PUSH-CONSTANT (LR-F-ADDR)))
      '(FETCH)
      '(ADD1-NAT)
      (LIST 'PUSH-CONSTANT (LR-F-ADDR))
      '(DEPOSIT)
      (LIST 'PUSH-CONSTANT (LR-F-ADDR))
      '(RET))
```

DEFINITION

```
;; P-ACCESSOR-CODE generates code for shell recognizer, e.g.
;; CAR.  Currently only works for shells that fit in one node,
;; i.e. have one or two fields.  TAG is the tag of shells of the
;; appropriate type, e.g. (LR-CONS-TAG), DEFAULT is the
;; address to use if the node is not of the type specified by TAG,
;; e.g. (LR-0-ADDR) the address of a node representing 0,
;; OFFSET is the offset in the node, e.g. (LR-CAR-OFFSET).
(P-ACCESSOR-CODE NAME TAG DEFAULT OFFSET)
   =
(LIST NAME '(X)
      '((TEMP (NAT 0)))
      ;; First decrement ref count of argument, X
      '(PUSH-LOCAL X)
      '(CALL I-DECR-REF-COUNT)
      ;; Test if X has type specified by TAG.
      (LIST 'PUSH-CONSTANT (TAG 'NAT (LR-TAG-OFFSET)))
      '(ADD-ADDR)
      '(FETCH)
      (LIST 'PUSH-CONSTANT (TAG 'NAT TAG))
      '(EQ)
      '(TEST-BOOL-AND-JUMP T ARG1)
      ;; Type is different, use DEFAULT first incrementing
      ;; ref count.
      (LIST 'PUSH-CONSTANT DEFAULT)
      '(FETCH)
      '(ADD1-NAT)
      (LIST 'PUSH-CONSTANT DEFAULT)
      '(DEPOSIT)
      (LIST 'PUSH-CONSTANT DEFAULT)
      '(RET)
      ;; Type is the same, return appropriate field as specified
      ;; by OFFSET first incrementing ref count
      '(DL ARG1 () (PUSH-LOCAL X))
      (LIST 'PUSH-CONSTANT (TAG 'NAT OFFSET))
      '(ADD-ADDR)
      '(FETCH)
      '(SET-LOCAL TEMP)
      '(FETCH)
      '(ADD1-NAT)
```

```
      '(PUSH-LOCAL TEMP)
      '(DEPOSIT)
      '(PUSH-LOCAL TEMP)
      '(RET))
```

DEFINITION

```
(P-CAR-CODE)
   =
(P-ACCESSOR-CODE 'CAR (LR-CONS-TAG) (LR-0-ADDR) (LR-CAR-OFFSET))
```

DEFINITION

```
(P-CDR-CODE)
   =
(P-ACCESSOR-CODE 'CDR (LR-CONS-TAG) (LR-0-ADDR) (LR-CDR-OFFSET))
```

DEFINITION

```
(P-CONS-CODE)
  =
;; Takes two implicit args
;; Note that we assume that the CDR is on top of the stack
;; and the CAR is just below it.  This is what would normally be
;; the case if you evaluate left to right.
(LIST 'CONS '()
      (LIST (LIST 'TEMP (LR-UNDEF-ADDR)))
      ;; First call I-ALLOC-NODE to get a new node.
      '(CALL I-ALLOC-NODE)
      '(SET-LOCAL TEMP)                        ; Put CDR in node + CDR-OFFSET
      (LIST 'PUSH-CONSTANT (TAG 'NAT (LR-CDR-OFFSET)))
      '(ADD-ADDR)
      '(DEPOSIT)
      '(PUSH-LOCAL TEMP)                       ; Put CAR in node + CAR-OFFSET
      (LIST 'PUSH-CONSTANT (TAG 'NAT (LR-CAR-OFFSET)))
      '(ADD-ADDR)
      '(DEPOSIT)
      (LIST 'PUSH-CONSTANT (TAG 'NAT (LR-CONS-TAG))) ; Put tag in node
      '(PUSH-LOCAL TEMP)
      (LIST 'PUSH-CONSTANT (TAG 'NAT (LR-TAG-OFFSET)))
      '(ADD-ADDR)
      '(DEPOSIT)
      '(PUSH-CONSTANT (NAT 0))                 ; Set ref count to 0
      '(PUSH-LOCAL TEMP)
      '(DEPOSIT)
      '(PUSH-LOCAL TEMP)
      '(RET)))
```

DEFINITION

```
(P-FALSE-CODE)
   =
(LIST 'FALSE '() '()
      ;; Push address of F incrementing ref count
      (LIST 'PUSH-CONSTANT (LR-F-ADDR))
```

```
      '(FETCH)
      '(ADD1-NAT)
      (LIST 'PUSH-CONSTANT (LR-F-ADDR))
      '(DEPOSIT)
      (LIST 'PUSH-CONSTANT (LR-F-ADDR))
      '(RET))

;; FALSEP takes one implicit arg on stack.
```

DEFINITION

```
(P-FALSEP-CODE)
   =
(LIST 'FALSEP '() '()
      ;; We are going to pop implicit arg off stack so call
      ;; I-DECR-REF-COUNT  which leaves arg on temp-stk
      '(CALL I-DECR-REF-COUNT)
      ;; Test if address is address of F
      (LIST 'PUSH-CONSTANT (LR-F-ADDR))
      '(EQ)
      '(TEST-BOOL-AND-JUMP T TRUE)
      ;; Arg was not F push F and increment ref count
      (LIST 'PUSH-CONSTANT (LR-F-ADDR))
      '(FETCH)
      '(ADD1-NAT)
      (LIST 'PUSH-CONSTANT (LR-F-ADDR))
      '(DEPOSIT)
      (LIST 'PUSH-CONSTANT (LR-F-ADDR))
      '(RET)
      ;; Arg was F push T and increment ref count
      (LIST 'DL 'TRUE '() (LIST 'PUSH-CONSTANT (LR-T-ADDR)))
      '(FETCH)
      '(ADD1-NAT)
      (LIST 'PUSH-CONSTANT (LR-T-ADDR))
      '(DEPOSIT)
      (LIST 'PUSH-CONSTANT (LR-T-ADDR))
      '(RET))
```

DEFINITION

```
(P-LISTP-CODE)
   =
(P-RECOGNIZER-CODE 'LISTP (LR-CONS-TAG))
```

DEFINITION

```
(P-NLISTP-CODE)
   =
;; Code for NLISTP is almost the same as for LISTP.
(LIST 'NLISTP '() '()
      '(CALL I-DECR-REF-COUNT)
      ;; Now get Tag.
      (LIST 'PUSH-CONSTANT (TAG 'NAT (LR-TAG-OFFSET)))
      '(ADD-ADDR)
      '(FETCH)
      (LIST 'PUSH-CONSTANT (TAG 'NAT (LR-CONS-TAG)))
```

```
      '(EQ)
      '(TEST-BOOL-AND-JUMP F TRUE)
      (LIST 'PUSH-CONSTANT (LR-F-ADDR))
      '(FETCH)
      '(ADD1-NAT)
      (LIST 'PUSH-CONSTANT (LR-F-ADDR))
      '(DEPOSIT)
      (LIST 'PUSH-CONSTANT (LR-F-ADDR))
      '(RET)
      (LIST 'DL 'TRUE '() (LIST 'PUSH-CONSTANT (LR-T-ADDR)))
      '(FETCH)
      '(ADD1-NAT)
      (LIST 'PUSH-CONSTANT (LR-T-ADDR))
      '(DEPOSIT)
      (LIST 'PUSH-CONSTANT (LR-T-ADDR))
      '(RET))
```

DEFINITION

```
(P-TRUE-CODE)
   =
(LIST 'TRUE '() '()
      ;; Push address of T incrementing ref count
      (LIST 'PUSH-CONSTANT (LR-T-ADDR))
      '(FETCH)
      '(ADD1-NAT)
      (LIST 'PUSH-CONSTANT (LR-T-ADDR))
      '(DEPOSIT)
      (LIST 'PUSH-CONSTANT (LR-T-ADDR))
      '(RET))
```

DEFINITION

```
(P-TRUEP-CODE)
   =
(P-RECOGNIZER-CODE 'TRUEP (LR-TRUE-TAG))
```

DEFINITION

```
(P-RUNTIME-SUPPORT-PROGRAMS)
   =
(LIST (P-CAR-CODE)
      (P-CDR-CODE)
      (P-CONS-CODE)
      (P-FALSE-CODE)
      (P-FALSEP-CODE)
      (P-LISTP-CODE)
      (P-NLISTP-CODE)
      (P-TRUE-CODE)
      (P-TRUEP-CODE)
      (P-I-DECR-REF-COUNT-CODE)
      (P-I-ALLOC-NODE-CODE))

;; The function LR-MAKE-LABEL takes a number, e.g. 5 and
;; makes a label, e.g. L-5.
```

DEFINITION

```
(LR-CONVERT-DIGIT-TO-ASCII DIGIT)
   =
(PLUS (ASCII-0) DIGIT)
```

DEFINITION

```
(LR-CONVERT-NUM-TO-ASCII NUMBER LIST)
   =
(IF (LESSP NUMBER 10)
    (CONS (LR-CONVERT-DIGIT-TO-ASCII NUMBER) LIST)
    (LR-CONVERT-NUM-TO-ASCII (QUOTIENT NUMBER 10)
                             (CONS (LR-CONVERT-DIGIT-TO-ASCII
                                    (REMAINDER NUMBER 10))
                                   LIST)))
```

DEFINITION

```
(LR-MAKE-LABEL N)
   =
(PACK (CONS (CAR (UNPACK 'L))
            (CONS (ASCII-DASH)
                  (APPEND (LR-CONVERT-NUM-TO-ASCII N NIL) 0))))
```

DEFINITION

```
;; LABEL-INSTRS takes a list of instructions and labels them
;; with (LR-MAKE-LABEL N), (LR-MAKE-LABEL N+1), ...
(LABEL-INSTRS INSTRS N)
   =
(IF (LISTP INSTRS)
    (CONS (DL (LR-MAKE-LABEL N) () (CAR INSTRS))
          (LABEL-INSTRS (CDR INSTRS) (ADD1 N)))
    NIL)
```

DEFINITION

```
;; COMP-IF compiles an IF.  TEST-INSTRS are the instructions
;; generated for the test, THEN-INSTRS are the instructions
;; for the then and ELSE-INSTRS are the instructions for the
;; else.  N is the number used to construct the label of the
;; first instruction (i.e. the first instruction in TEST-INSTRS.
(COMP-IF TEST-INSTRS THEN-INSTRS ELSE-INSTRS N)
   =
(APPEND TEST-INSTRS
        (APPEND (LIST '(CALL I-DECR-REF-COUNT)
                      (LIST 'PUSH-CONSTANT (LR-F-ADDR))
                      '(EQ)
                      (LIST 'TEST-BOOL-AND-JUMP 'T
                            (LR-MAKE-LABEL (PLUS N 5
                                                 (LENGTH TEST-INSTRS)
                                                 (LENGTH THEN-INSTRS)))))
                (APPEND THEN-INSTRS
                        (CONS (LIST 'JUMP
```

```
                                    (LR-MAKE-LABEL
                                     (PLUS N 5
                                           (LENGTH TEST-INSTRS)
                                           (LENGTH THEN-INSTRS)
                                           (LENGTH ELSE-INSTRS))))
                                ELSE-INSTRS))))
```

DEFINITION

```
;; LR-P-C-SIZE-IF returns the size (number of Piton instructions) in
;; the compilation of an IF.  TEST-SIZE, THEN-SIZE and
;; ELSE-SIZE are the sizes of the test, then and else instructions.
(LR-P-C-SIZE-IF TEST-SIZE THEN-SIZE ELSE-SIZE)
   =
(PLUS TEST-SIZE
      4
      THEN-SIZE
      1
      ELSE-SIZE)
```

DEFINITION

```
;; COMP-TEMP-FETCH generates code for a (S-TEMP-FETCH) form,
;; which unconditionally uses the already computed value.
;; EXPR is the expression being compiled, so (CADDR EXPR) is
;; the variable used to store this form.
(COMP-TEMP-FETCH EXPR N)
   =
(LIST (LIST 'PUSH-LOCAL (CADDR EXPR))
      '(FETCH)
      '(ADD1-NAT)
      (LIST 'PUSH-LOCAL (CADDR EXPR))
      '(DEPOSIT)
      (LIST 'PUSH-LOCAL (CADDR EXPR)))
```

DEFINITION

```
;; LR-P-C-SIZE-TEMP-FETCH returns size of a (S-TEMP-FETCH) form.
(LR-P-C-SIZE-TEMP-FETCH) = 6
```

DEFINITION

```
;; COMP-TEMP-EVAL generates code for a (S-TEMP-EVAL) form,
;; which unconditionally always computes the value and stores it.
;; EXPR is the expression being compiled, so (CADDR EXPR) is
;; the variable used to store this form.  INSTRS are the
;; instructions to evaluate EXPR.
(COMP-TEMP-EVAL EXPR INSTRS N)
   =
(APPEND INSTRS
        (LIST (LIST 'SET-LOCAL (CADDR EXPR))
              '(FETCH)
              '(ADD1-NAT)
              (LIST 'PUSH-LOCAL (CADDR EXPR))
              '(DEPOSIT)
```

```
            (LIST 'PUSH-LOCAL (CADDR EXPR))))
```

DEFINITION

```
;; LR-P-C-SIZE-TEMP-EVAL returns size of a (S-TEMP-EVAL) form.
(LR-P-C-SIZE-TEMP-EVAL SIZE)
   =
(PLUS SIZE 6)
```

DEFINITION

```
;; COMP-TEMP-TEST generates code for a (S-TEMP-TEST) form,
;; which tests if the expression has been computed already, if so
;; it uses the already computed value, otherwise the value is
;; computed and stored.  EXPR is the expression being
;; compiled, so (CADDR EXPR) is the variable used to store
;; this form.INSTRS are the instructions to evaluate EXPR.
;;N will the label on the first instruction of generated.
(COMP-TEMP-TEST EXPR INSTRS N)
   =
(APPEND (LIST (LIST 'PUSH-LOCAL (CADDR EXPR))
              (LIST 'PUSH-CONSTANT (LR-UNDEF-ADDR))
              '(EQ)
              ;;  If the value is already computed jump to
              ;; PUSH-LOCAL below.
              (LIST 'TEST-BOOL-AND-JUMP 'F
                    (LR-MAKE-LABEL (PLUS N 5
                                         (LENGTH INSTRS)))))
        (APPEND INSTRS
                ;; Pop the value into appropriate temp.
                (LIST (LIST 'POP-LOCAL (CADDR EXPR))
                      ;; The reference count must be incremented.
                      ;; If the stored value is used, then an
                      ;; additional pointer is added since the
                      ;; value is put on the temp stack.  If the
                      ;; value is computed, we store it for use
                      ;; later on the control stack.
                      (LIST 'PUSH-LOCAL (CADDR EXPR))
                      '(FETCH)
                      '(ADD1-NAT)
                      (LIST 'PUSH-LOCAL (CADDR EXPR))
                      '(DEPOSIT)
                      (LIST 'PUSH-LOCAL (CADDR EXPR))))))
```

DEFINITION

```
;; LR-P-C-SIZE-TEMP-TEST returns size of a (S-TEMP-TEST) form.
(LR-P-C-SIZE-TEMP-TEST SIZE)
   =
(PLUS 4 SIZE 7)
```

DEFINITION

```
;; COMP-QUOTE generates code for a QUOTE form, just put
;; the address on the stack and increment the ref count.
```

```
(COMP-QUOTE EXPR N)
   =
(LIST (LIST 'PUSH-CONSTANT (CADR EXPR))
      '(FETCH)
      '(ADD1-NAT)
      (LIST 'PUSH-CONSTANT (CADR EXPR))
      '(DEPOSIT)
      (LIST 'PUSH-CONSTANT (CADR EXPR)))
```

DEFINITION

```
;; LR-P-C-SIZE-QUOTE returns the size of a quote form
(LR-P-C-SIZE-QUOTE) = 6
```

DEFINITION

```
;; COMP-LITATOM generates code for referencing a variable, just push
;; the variable on the stack and increment the ref count.
(COMP-LITATOM EXPR N)
   =
(LIST (LIST 'PUSH-LOCAL EXPR)
      '(FETCH)
      '(ADD1-NAT)
      (LIST 'PUSH-LOCAL EXPR)
      '(DEPOSIT)
      (LIST 'PUSH-LOCAL EXPR))
```

DEFINITION

```
;; LR-P-C-SIZE-LITATOM returns the size of a variable reference
(LR-P-C-SIZE-LITATOM) = 6
```

DEFINITION

```
;; COMP-FUNCALL generates the code for a function call, just do
;; the appropriate call.
(COMP-FUNCALL EXPR INSTRS)
   =
(APPEND INSTRS
        (IF (DEFINEDP (CAR EXPR) (P-RUNTIME-SUPPORT-PROGRAMS))
            (LIST (LIST 'CALL (CAR EXPR)))
            (LIST (LIST 'CALL (USER-FNAME (CAR EXPR))))))
```

DEFINITION

```
;; LR-P-C-SIZE-FUNCALL returns the size of a function call
(LR-P-C-SIZE-FUNCALL SIZE)
   =
(PLUS SIZE 1)
```

DEFINITION

```
;; COMP-BODY-1 returns a list of Piton instructions to compile EXPR.
;; N is the number of Piton instructions previously generated, it is used
;; to generate unique labels.
```

```
(COMP-BODY-1 FLAG EXPR N)
   =
(COND ((EQUAL FLAG 'LIST)
       (IF (LISTP EXPR)
           (APPEND (COMP-BODY-1 T (CAR EXPR) N)
                   (COMP-BODY-1 'LIST
                                (CDR EXPR)
                                (PLUS N (LENGTH (COMP-BODY-1 T
                                                             (CAR EXPR)
                                                             N)))))
           NIL))
      ((LISTP EXPR)
       (COND ((EQUAL (CAR EXPR) 'IF)
              (LET ((TEST (COMP-BODY-1 T (CADR EXPR) N)))
                (LET ((THEN (COMP-BODY-1 T
                                         (CADDR EXPR)
                                         (PLUS N (LENGTH TEST) 4))))
                  (COMP-IF TEST
                           THEN
                           (COMP-BODY-1 T
                                        (CADDDR EXPR)
                                        (PLUS N (LENGTH TEST)
                                              4 (LENGTH THEN) 1))
                           N))))
             ((EQUAL (CAR EXPR) (S-TEMP-FETCH)) (COMP-TEMP-FETCH EXPR N))
             ((EQUAL (CAR EXPR) (S-TEMP-EVAL))
              (COMP-TEMP-EVAL EXPR (COMP-BODY-1 T (CADR EXPR) N) N))
             ((EQUAL (CAR EXPR) (S-TEMP-TEST))
              (COMP-TEMP-TEST EXPR
                              (COMP-BODY-1 T (CADR EXPR) (PLUS N 4))
                              N))
             ((EQUAL (CAR EXPR) 'QUOTE) (COMP-QUOTE EXPR N))
             (T (COMP-FUNCALL EXPR (COMP-BODY-1 'LIST (CDR EXPR) N)))))
      (T (COMP-LITATOM EXPR N)))
```

DEFINITION

```
;; COMP-POP-FORMALS generates code to decrement the reference
;; count of all actual parameters before returning from a function call.
(COMP-POP-FORMALS FORMALS)
   =
(IF (LISTP FORMALS)
    (APPEND (LIST (LIST 'PUSH-LOCAL (CAR FORMALS))
                  '(CALL I-DECR-REF-COUNT)
                  '(POP))
            (COMP-POP-FORMALS (CDR FORMALS)))
    NIL)
```

DEFINITION

```
(LR-P-C-SIZE-FORMALS FORMALS)
   =
(TIMES 3 (LENGTH FORMALS))
```

DEFINITION

```
;; COMP-POP-TEMPS generates code to decrement the reference count
;; of all temporary variables before returning from a function call.
(COMP-POP-TEMPS TEMP-VAR-DCLS N)
   =
(IF (LISTP TEMP-VAR-DCLS)
    (APPEND (LIST (LIST 'PUSH-LOCAL (CAAR TEMP-VAR-DCLS))
                  (LIST 'PUSH-CONSTANT (LR-UNDEF-ADDR))
                  '(EQ)
                  (LIST 'TEST-BOOL-AND-JUMP T (LR-MAKE-LABEL (PLUS N 5)))
                  '(CALL I-DECR-REF-COUNT)
                  '(POP))
            (COMP-POP-TEMPS (CDR TEMP-VAR-DCLS) (PLUS N 5)))
    NIL)
```

DEFINITION

```
(LR-P-C-SIZE-TEMPS TEMPS)
   =
(TIMES 6 (LENGTH TEMPS))
```

DEFINITION

```
;; LR-P-C-SIZE returns the size of (the number of Piton instructions
;; in the compilation of) EXPR.
(LR-P-C-SIZE FLAG EXPR)
   =
(COND ((EQUAL FLAG 'LIST)
       (IF (LISTP EXPR)
           (PLUS (LR-P-C-SIZE T (CAR EXPR))
                 (LR-P-C-SIZE 'LIST (CDR EXPR)))
           0))
      ((LISTP EXPR)
       (COND ((EQUAL (CAR EXPR) 'IF)
              (LR-P-C-SIZE-IF (LR-P-C-SIZE T (CADR EXPR))
                              (LR-P-C-SIZE T (CADDR EXPR))
                              (LR-P-C-SIZE T (CADDDR EXPR))))
             ((EQUAL (CAR EXPR) (S-TEMP-FETCH))
              (LR-P-C-SIZE-TEMP-FETCH))
             ((EQUAL (CAR EXPR) (S-TEMP-EVAL))
              (LR-P-C-SIZE-TEMP-EVAL (LR-P-C-SIZE T (CADR EXPR))))
             ((EQUAL (CAR EXPR) (S-TEMP-TEST))
              (LR-P-C-SIZE-TEMP-TEST (LR-P-C-SIZE T (CADR EXPR))))
             ((EQUAL (CAR EXPR) 'QUOTE) (LR-P-C-SIZE-QUOTE))
             (T (LR-P-C-SIZE-FUNCALL (LR-P-C-SIZE 'LIST (CDR EXPR))))))
      (T 6))
```

DEFINITION

```
;; LR-P-C-SIZE-LIST generates the size of the first N expressions
;; in EXPR-LIST.
(LR-P-C-SIZE-LIST N EXPR-LIST)
   =
(IF (ZEROP N)
    0
```

```
    (IF (LESSP N (LENGTH EXPR-LIST))
        (PLUS (LR-P-C-SIZE T (GET N EXPR-LIST))
              (LR-P-C-SIZE-LIST (SUB1 N) EXPR-LIST))
        (LR-P-C-SIZE-LIST (SUB1 (LENGTH EXPR-LIST)) EXPR-LIST)))

;; LR-P-PC-1 returns the number of Piton instructions before the start of
;; the expression denoted by POS in the compilation of EXPR.
```

DEFINITION

```
(LR-P-PC-1 EXPR POS)
   =
(COND ((NOT (LISTP POS)) 0)
      ((NOT (LISTP EXPR)) 0)
      ((ZEROP (CAR POS)) 0)
      ((EQUAL (CAR EXPR) 'IF)
       (COND ((ZEROP (CAR POS)) 0)
             ((EQUAL (CAR POS) 1)
              (LR-P-PC-1 (CADR EXPR) (CDR POS)))
             ((EQUAL (CAR POS) 2)
              (PLUS (LR-P-C-SIZE T (CADR EXPR))
                    4
                    (LR-P-PC-1 (CADDR EXPR) (CDR POS))))
             (T (PLUS (LR-P-C-SIZE T (CADR EXPR))
                      4
                      (LR-P-C-SIZE T (CADDR EXPR))
                      1
                      (LR-P-PC-1 (CADDDR EXPR) (CDR POS))))))
      ((EQUAL (CAR EXPR) (S-TEMP-FETCH)) 0)
      ((EQUAL (CAR EXPR) (S-TEMP-EVAL))
       (LR-P-PC-1 (CADR EXPR) (CDR POS)))
      ((EQUAL (CAR EXPR) (S-TEMP-TEST))
       (PLUS 4 (LR-P-PC-1 (CADR EXPR) (CDR POS))))
      ((EQUAL (CAR EXPR) 'QUOTE) 0)
      (T (PLUS (LR-P-C-SIZE-LIST (SUB1 (CAR POS)) EXPR)
               (LR-P-PC-1 (GET (CAR POS) EXPR) (CDR POS)))))
```

DEFINITION

```
;; COMP-BODY compiles the LR function body BODY, with formal
;; parameters FORMALS and temporary variable definitions
;; TEMP-VAR-DCLS.  FIRSTP indicates whether the program is
;; the main one or not.  The main program does not need to
;; decrement the reference counts of the actuals and temporary
;; variables, since Piton halts.
;; Need the REMOVE-DUPLICATES of FORMAL-VARS.  Otherwise if there is a
;; duplicate in the formals, we will decrement the reference count
;; more than once when returning from a user defined function call.
(COMP-BODY FIRSTP BODY FORMALS TEMP-VAR-DCLS)
   =
(LABEL-INSTRS
 (APPEND (COMP-BODY-1 T BODY 0)
         (IF FIRSTP
             (LIST (LIST 'SET-GLOBAL (AREA-NAME (LR-ANSWER-ADDR)))
                   '(RET))
             (APPEND (COMP-POP-FORMALS (REMOVE-DUPLICATES FORMALS))
```

```
                    (APPEND (COMP-POP-TEMPS
                             TEMP-VAR-DCLS
                             (PLUS (LR-P-C-SIZE T BODY)
                                   (TIMES 2
                                          (LENGTH FORMALS))))
                            '((RET))))))
 0)
```

DEFINITION

```
;; COMP-PROGRAMS-1 compiles programs.
(COMP-PROGRAMS-1 FIRSTP PROGRAMS)
   =
(IF (LISTP PROGRAMS)
    (CONS (LR-MAKE-PROGRAM (NAME (CAR PROGRAMS))
                           (FORMAL-VARS (CAR PROGRAMS))
                           (TEMP-VAR-DCLS (CAR PROGRAMS))
                           (COMP-BODY FIRSTP
                                      (PROGRAM-BODY (CAR PROGRAMS))
                                      (FORMAL-VARS (CAR PROGRAMS))
                                      (TEMP-VAR-DCLS (CAR PROGRAMS))))
          (COMP-PROGRAMS-1 F (CDR PROGRAMS)))
    NIL)
```

DEFINITION

```
;; COMP-PROGRAMS produces the program segment by compiling
;; PROGRAMS and appending the run-time support function.
(COMP-PROGRAMS PROGRAMS)
   =
(APPEND (COMP-PROGRAMS-1 T PROGRAMS)
        (P-RUNTIME-SUPPORT-PROGRAMS))
```

DEFINITION

```
;; LR-P-PC produces the Piton PC from the LR PC
(LR-P-PC L)
   =
(TAG 'PC (CONS (AREA-NAME (P-PC L))
               (LR-P-PC-1 (PROGRAM-BODY (P-CURRENT-PROGRAM L))
                          (OFFSET (P-PC L)))))
```

DEFINITION

```
;; LR->P compiles an LR state.
(LR->P P)
   =
(P-STATE (LR-P-PC P)
         (P-CTRL-STK P)
         (P-TEMP-STK P)
         (COMP-PROGRAMS (P-PROG-SEGMENT P))
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
```

```
        (P-PSW P))
```

DEFINITION

```
;; This table provides program addresses in the subroutine I-ALLOCATE-NODE
;; to jump to based on type of the node.  When user-defined subrs are
;; added it will need an argument to tell the tags for the user-defined
;; subrs.
(LR-MAKE-ALLOC-NODE-JUMP-TABLE)
   =
(LIST (PC 'CANT-HAPPEN (P-I-ALLOC-NODE-CODE)) ; UNDEFINED node
      (PC 'INIT (P-I-ALLOC-NODE-CODE))
      (PC 'CANT-HAPPEN (P-I-ALLOC-NODE-CODE)) ; F
      (PC 'INIT (P-I-ALLOC-NODE-CODE))        ; T
      (PC 'ADD1 (P-I-ALLOC-NODE-CODE))
      (PC 'CONS (P-I-ALLOC-NODE-CODE))
      (PC 'PACK (P-I-ALLOC-NODE-CODE))
      (PC 'MINUS (P-I-ALLOC-NODE-CODE)))
```

DEFINITION

```
;; LR-INIT-HEAP-CONTENTS constructs the contents of the heap data
;; area.  SIZE is the number nodes in the heap.  The contents
;; are initially linked together through the reference count field.
(LR-INIT-HEAP-CONTENTS ADDR SIZE)
   =
(IF (ZEROP SIZE)
    (LIST (TAG 'NAT (LR-INIT-TAG)))
    (APPEND (LR-NEW-NODE (ADD-ADDR ADDR (LR-NODE-SIZE))
                         (TAG 'NAT (LR-INIT-TAG))
                         (TAG 'NAT 0)
                         (TAG 'NAT 0))
            (LR-INIT-HEAP-CONTENTS (ADD-ADDR ADDR (LR-NODE-SIZE))
                                   (SUB1 SIZE))))
```

DEFINITION

```
(LR-ADD-TO-DATA-SEG DATA-SEG NEW-NODE)
   =
(IF (NOT (LESSP (SUB1 (LENGTH (VALUE (LR-HEAP-NAME) DATA-SEG)))
                (PLUS (OFFSET (LR-FETCH-FP DATA-SEG))
                      (LENGTH NEW-NODE))))
    (DEPOSIT (FETCH (LR-FETCH-FP DATA-SEG) DATA-SEG)
             (LR-FP-ADDR)
             (DEPOSIT-A-LIST NEW-NODE
                             (LR-FETCH-FP DATA-SEG)
                             DATA-SEG))
    DATA-SEG)

;; We put a 0 for the reference count of F (this means that there is
;; exactly one pointer to it, i.e. we add an implicit pointer).
;; This is because we need to make sure it is never GC'd.
```

DEFINITION

```
;; LR-INIT-DATA-SEG constructs the initial data-segment.
```

```
;; HEAP-SIZE is the number of nodes in the data-segment.
(LR-INIT-DATA-SEG HEAP-SIZE)
   =
(DEPOSIT-A-LIST (LIST (TAG 'NAT 0)
                      (TAG 'NAT (LR-FALSE-TAG))
                      (LR-UNDEF-ADDR)
                      (LR-UNDEF-ADDR))
                (LR-F-ADDR)
                (DEPOSIT-A-LIST
                 (LIST (TAG 'NAT 0)
                       (TAG 'NAT (LR-UNDEFINED-TAG))
                       (LR-UNDEF-ADDR)
                       (LR-UNDEF-ADDR))
                 (LR-UNDEF-ADDR)
                 (LIST (LIST (AREA-NAME (LR-FP-ADDR))
                             (ADD-ADDR (LR-F-ADDR) (LR-NODE-SIZE)))
                       (LIST (AREA-NAME (LR-ANSWER-ADDR))
                             (TAG 'NAT 0))
                       (LIST (AREA-NAME (LR-ALLOC-NODE-JUMP-TABLE-ADDR))
                             (LR-MAKE-ALLOC-NODE-JUMP-TABLE))
                       (CONS (LR-HEAP-NAME)
                             (LR-INIT-HEAP-CONTENTS
                              (TAG 'ADDR (CONS (LR-HEAP-NAME) 0))
                              HEAP-SIZE)))))
```

DEFINITION

```
;; COUNT-LIST is used to admit LR-COMPILE-QUOTE.
(COUNT-LIST FLAG OBJECT)
   =
(COND ((EQUAL FLAG 'LIST)
       (IF (LISTP OBJECT)
           (PLUS (COUNT-LIST T (CAR OBJECT))
                 (COUNT-LIST 'LIST (CDR OBJECT)))
           1))
      ((LISTP OBJECT)
       (ADD1 (ADD1 (PLUS (COUNT-LIST T (CAR OBJECT))
                         (COUNT-LIST T (CDR OBJECT))))))
      ((NUMBERP OBJECT)
       (ADD1 (COUNT OBJECT)))
      (T 1))
```

DEFINITION

```
;; LR-COMPILE-QUOTE is the main function that compiles a quoted
;; constant.  HEAP is a data-segment.  TABLE is an
;; association list that associates objects with addresses into
;; HEAP.  LR-COMPILE-QUOTE returns a pair, the new HEAP
;; and the new TABLE.
(LR-COMPILE-QUOTE FLAG OBJECT HEAP TABLE)
   =
(COND ((EQUAL FLAG 'LIST)
       (IF (LISTP OBJECT)
           (LET ((CAR-PAIR (LR-COMPILE-QUOTE T (CAR OBJECT) HEAP TABLE)))
             (LR-COMPILE-QUOTE 'LIST
```

```
                          (CDR OBJECT)
                          (CAR CAR-PAIR)
                          (CDR CAR-PAIR)))
        (CONS HEAP TABLE)))
     ((DEFINEDP OBJECT TABLE)
      (CONS (DEPOSIT (TAG 'NAT
                          (ADD1 (UNTAG (FETCH (CDR (ASSOC OBJECT TABLE))
                                              HEAP))))
                     (CDR (ASSOC OBJECT TABLE))
                     HEAP)
            TABLE))
     ((LISTP OBJECT)
      (LET ((PAIR (LR-COMPILE-QUOTE 'LIST
                                    (LIST (CAR OBJECT) (CDR OBJECT))
                                    HEAP
                                    TABLE)))
        (CONS (LR-ADD-TO-DATA-SEG (CAR PAIR)
                                  (LR-NEW-CONS (CDR (ASSOC
                                                      (CAR OBJECT)
                                                      (CDR PAIR)))
                                               (CDR (ASSOC
                                                      (CDR OBJECT)
                                                      (CDR PAIR)))))
              (CONS (CONS OBJECT (FETCH (LR-FP-ADDR) (CAR PAIR)))
                    (CDR PAIR)))))
     ((NUMBERP OBJECT)
      (CONS (LR-ADD-TO-DATA-SEG HEAP
                                (LR-NEW-NODE (TAG 'NAT 0)
                                             (TAG 'NAT (LR-ADD1-TAG))
                                             (TAG 'NAT OBJECT)
                                             (LR-UNDEF-ADDR)))
            (CONS (CONS OBJECT (FETCH (LR-FP-ADDR) HEAP)) TABLE)))
     ((TRUEP OBJECT)
      (CONS (LR-ADD-TO-DATA-SEG HEAP (LR-NEW-NODE (TAG 'NAT 0)
                                                  (TAG 'NAT (LR-TRUE-TAG))
                                                  (LR-UNDEF-ADDR)
                                                  (LR-UNDEF-ADDR)))
            (CONS (CONS OBJECT (FETCH (LR-FP-ADDR) HEAP)) TABLE)))
     (T                              ; Assume it is undefined
       (CONS HEAP
             (CONS (CONS OBJECT (LR-UNDEF-ADDR)) TABLE))))
```

DEFINITION

```
;; LR-DATA-SEG-TABLE-BODY returns a pair, the CAR is the extension of
;; DATA-SEG with any constants laid down in it, the CDR is an alist
;; mapping objects in the logic to addresses in the new DATA-SEG
;; where they are represented.  The initial TABLE is such an alist
(LR-DATA-SEG-TABLE-BODY FLAG EXPR DATA-SEG TABLE)
   =
(COND ((EQUAL FLAG 'LIST)
       (IF (LISTP EXPR)
           (LET ((DST1 (LR-DATA-SEG-TABLE-BODY T
                                               (CAR EXPR)
                                               DATA-SEG
```

```
                                              TABLE)))
            (LR-DATA-SEG-TABLE-BODY 'LIST
                                    (CDR EXPR)
                                    (CAR DST1)
                                    (CDR DST1)))
          (CONS DATA-SEG TABLE)))
      ((LISTP EXPR)
       (COND ((OR (EQUAL (CAR EXPR) (S-TEMP-FETCH))
                  (EQUAL (CAR EXPR) (S-TEMP-EVAL))
                  (EQUAL (CAR EXPR) (S-TEMP-TEST)))
              (LR-DATA-SEG-TABLE-BODY T (CADR EXPR) DATA-SEG TABLE))
             ((EQUAL (CAR EXPR) 'QUOTE)
              (LR-COMPILE-QUOTE T (CADR EXPR) DATA-SEG TABLE))
             (T (LR-DATA-SEG-TABLE-BODY 'LIST
                                        (CDR EXPR)
                                        DATA-SEG
                                        TABLE))))
      (T                               ; should be a LITATOM
        (CONS DATA-SEG TABLE)))
```

DEFINITION

```
(LR-DATA-SEG-TABLE-LIST PROGS DATA-SEG TABLE)
   =
(IF (LISTP PROGS)
    (LR-DATA-SEG-TABLE-LIST (CDR PROGS)
                            (CAR (LR-DATA-SEG-TABLE-BODY
                                  T
                                  (S-BODY (CAR PROGS))
                                  DATA-SEG
                                  TABLE))
                            (CDR (LR-DATA-SEG-TABLE-BODY
                                  T
                                  (S-BODY (CAR PROGS))
                                  DATA-SEG
                                  TABLE)))
    (CONS DATA-SEG TABLE))
```

DEFINITION

```
(LR-INIT-DATA-SEG-TABLE PARAMS DATA-SEG TABLE)
   =
(IF (LISTP PARAMS)
    (LET ((DS-TAB (LR-COMPILE-QUOTE T (CDAR PARAMS) DATA-SEG TABLE)))
      (LR-INIT-DATA-SEG-TABLE (CDR PARAMS) (CAR DS-TAB) (CDR DS-TAB)))
    (CONS DATA-SEG TABLE))
```

DEFINITION

```
;; LR-DATA-SEG-TABLE constructs the initial data-segment used by
;; LR->P.  It firsts puts down T and 0 in the correct
;; place.  Then it calls LR-INIT-DATA-SEG-TABLE to compile
;; the constants in the variable alist PARAMS and calls
;; LR-DATA-SEG-TABLE-LIST to compile the constants in the programs
;;PROGS.
```

```
(LR-DATA-SEG-TABLE PROGS PARAMS HEAP-SIZE)
   =
(LET ((INIT-DS-TABLE1 (LR-COMPILE-QUOTE 'LIST
                                        (LIST T 0)
                                        (LR-INIT-DATA-SEG HEAP-SIZE)
                                        (LIST (CONS F (LR-F-ADDR))))))
  (LET ((INIT-DS-TABLE2 (LR-INIT-DATA-SEG-TABLE PARAMS
                                                (CAR INIT-DS-TABLE1)
                                                (CDR INIT-DS-TABLE1))))
    (LR-DATA-SEG-TABLE-LIST PROGS
                            (CAR INIT-DS-TABLE2)
                            (CDR INIT-DS-TABLE2))))
```

DEFINITION

```
;; S->LR1 compiles the programs and the PC in S, leaving the
;; other fields the sames as L.
(S->LR1 S L TABLE)
   =
(P-STATE (TAG 'PC (CONS (S-PNAME S) (S-POS S)))
         (P-CTRL-STK L)
         (P-TEMP-STK L)
         (LR-COMPILE-PROGRAMS (S-PROGS S) TABLE)
         (P-DATA-SEGMENT L)
         (P-MAX-CTRL-STK-SIZE L)
         (P-MAX-TEMP-STK-SIZE L)
         (P-WORD-SIZE L)
         (S-ERR-FLAG S))
```

DEFINITION

```
;; Returns an LR-STATE, FHEAP-SIZE is number of free
;; nodes in the resulting LR-STATE.  MAX-CTRL, MAX-TEMP and
;; WORD-SIZE give the resource limitations.
(S->LR S FHEAP-SIZE MAX-CTRL MAX-TEMP WORD-SIZE)
   =
(LET ((TEMP-ALIST (LR-MAKE-TEMP-NAME-ALIST (STRIP-CARS (S-TEMPS S))
                                           (STRIP-CARS (S-PARAMS S))))
      (DATASEG-TABLE (LR-DATA-SEG-TABLE (S-PROGS S)
                                        (S-PARAMS S)
                                        FHEAP-SIZE)))
  (LET ((RETURN-PC
         (TAG 'PC
              (CONS (S-PNAME S)
                    (LR-P-PC-1 (LR-COMPILE-BODY T
                                                (S-BODY (S-PROG S))
                                                TEMP-ALIST
                                                (CDR DATASEG-TABLE))
                               (S-POS S))))))
    (S->LR1 S
            (P-STATE NIL
                     (LR-INITIAL-CSTK (S-PARAMS S)
                                      TEMP-ALIST
                                      (CDR DATASEG-TABLE)
                                      RETURN-PC)
```

```
                    NIL
                    NIL
                    (CAR DATASEG-TABLE)
                    MAX-CTRL
                    MAX-TEMP
                    WORD-SIZE
                    NIL)
            (CDR DATASEG-TABLE))))
```

DEFINITION

```
;; LOGIC->P puts it all together.
(LOGIC->P EXPR ALIST PNAMES HEAP-SIZE MAX-CTRL MAX-TEMP WORD-SIZE)
   =
(LR->P (S->LR (LOGIC->S EXPR ALIST PNAMES)
              HEAP-SIZE
              MAX-CTRL
              MAX-TEMP
              WORD-SIZE))
```

Appendix C

Nqthm Libraries

This chapter lists the events in lists, integers, naturals, and bags libraries of Bevier, Kaufmann and Wilding (see [16] and [3]).

C.1 Bags Library

The list of events in the Bags Library has been omitted in the current version of this report.

C.2 Naturals Library

The list of events in the Natural Library has been omitted in the current version of this report.

C.3 Integers Library

The list of events in the Integers Library has been omitted in the current version of this report.

C.4 Lists Library

The list of events in the Lists Library has been omitted in the current version of this report.

Appendix D

Piton Events

This chapter lists the events from Moore's Piton proof ([23]) that are necessary for the compiler proof. There are four axioms added at the end of this section. These are named P-STEP-PRESERVES-PROPER-P-STATEP, ONCE-ERRORP-ALWAYS-ERRORP-STEP, ONCE-ERRORP-ALWAYS-ERRORP and P-PRESERVES-PROPER-P-STATEP. These are the same as PROVE-LEMMA events with the same name that were proven by Moore.

The events listed in this chapter depend on the events in Appendix C.

The list of events used in the compiler proof, that were taken from the Piton proof has been omitted in the current version of this report.

Appendix E

Listing of Events in the Proofs of Compiler With and Without Garbage Collector

This chapter lists the events in the compiler proofs. The events in this chapter are used in the proofs of the compiler with and without the garbage collector. The events here will also be used unchanged in the proof of the full compiler. Every PROVE-LEMMA event in this section has been mechanically checked.

The events listed in this chapter depend on the events in Appendices C and D.

The list of events used in both versions of the compiler proof has been omitted in the current version of this report.

Appendix F

Listing of Events in the Proof of the Compiler Without a Garbage Collector

This chapter lists all the events in the proof of the compiler without a garbage collector. Note that this version of the compiler contains only the following SUBRs: CAR, CDR, CONS, FALSE, FALSEP, IF, LISTP, NLISTP, TRUE and TRUEP. Also user-defined shells are not implemented. All of the lemmas in this chapter have mechanically checked proofs.

The events listed in this chapter depend on the events in Appendices C, D, and E.

The list of events that make up the compiler proof has been omitted in the current version of this report, with the exception of those referenced in Chapter 6

```
(PROVE-LEMMA LR-EVAL-S-EVAL-EQUIVALENCE ()
  (IMPLIES (AND (PROPER-P-STATEP (LR->P (S->LR1 S L TABLE)))
                (LR-PROPER-HEAPP (P-DATA-SEGMENT L))
                (GOOD-POSP FLAG (S-POS S) (S-BODY (S-PROG S)))
                (LR-PROGRAMS-PROPERP (S->LR1 S L TABLE) TABLE)
                (LR-S-SIMILAR-STATESP (S-PARAMS S)
                                      (S-TEMPS S)
                                      (S->LR1 S L TABLE)
                                      TABLE)
                (S-GOOD-STATEP S C)
                (EQUAL (P-PSW (LR-EVAL FLAG (S->LR1 S L TABLE) C)) 'RUN))
           (AND (LR-S-SIMILAR-STATESP (S-PARAMS S)
                                      (S-TEMPS (S-EVAL FLAG S C))
                                      (LR-EVAL FLAG (S->LR1 S L TABLE) C)
                                      TABLE)
                (LR-CHECK-RESULT (IF (EQUAL FLAG 'LIST) 'LIST T)
                                 (S-ANS (S-EVAL FLAG S C))
                                 (P-TEMP-STK (LR-EVAL FLAG
                                                      (S->LR1 S L TABLE)
```

```
                                                    C))
                                (P-DATA-SEGMENT (LR-EVAL FLAG
                                                         (S->LR1 S L TABLE)
                                                         C))
                                (P-TEMP-STK L))
                (EQUAL (S-ERR-FLAG (S-EVAL FLAG S C)) 'RUN)))
  ((INDUCT (IHINT-2 FLAG S L TABLE C))
   (ENABLE LR-EVAL-IF-P-PSW-1)
   (EXPAND (S-EVAL FLAG S C) (S-EVAL 'LIST S C) (S-EVAL FLAG S 0))
   (DISABLE DEFINEDP LR-COMPILE-PROGRAMS LR-EVAL LR-MAKE-TEMP-NAME-ALIST
            S-EVAL
            GOOD-POSP-LIST-NX-T-SIMPLE
            GOOD-POSP1-CONS-LESSP-4-IF-LR-PROPER-EXPRP
            FORMAL-VARS-LR-COMPILE-PROGRAMS
            PROPER-P-STATEP-LR->P-STRIP-CARS-BINDINGS-CTRL-STK
            S-EVAL-L-EVAL-FLAG-RUN-FLAG-T
            S-EVAL-L-EVAL-FLAG-RUN-HELPER-5
            S-EVAL-L-EVAL-FLAG-T
            TEMP-VAR-DCLS-LR-COMPILE-PROGRAMS)))

(PROVE-LEMMA LR-EVAL-S-EVAL-FLAG-RUN (REWRITE)
  (IMPLIES (AND (PROPER-P-STATEP (LR->P (S->LR1 S L TABLE)))
                (LR-PROPER-HEAPP (P-DATA-SEGMENT L))
                (GOOD-POSP FLAG (S-POS S) (S-BODY (S-PROG S)))
                (LR-PROGRAMS-PROPERP (S->LR1 S L TABLE) TABLE)
                (LR-S-SIMILAR-STATESP (S-PARAMS S)
                                      (S-TEMPS S)
                                      (S->LR1 S L TABLE)
                                      TABLE)
                (S-GOOD-STATEP S C)
                (S-ALL-TEMPS-SETP FLAG
                                  (IF (EQUAL FLAG 'LIST)
                                      (S-EXPR-LIST S)
                                      (S-EXPR S))
                                  (TEMP-ALIST-TO-SET (S-TEMPS S)))
                (S-ALL-PROGS-TEMPS-SETP (S-PROGS S))
                (S-CHECK-TEMPS-SETP (S-TEMPS S))
                (EQUAL (S-ERR-FLAG (S-EVAL FLAG S C)) 'RUN)
                (LR-CHECK-RESOURCESP FLAG S L C)
                (NOT (LESSP (P-WORD-SIZE L) (S-MAX-SUBR-REQS))))
           (EQUAL (P-PSW (LR-EVAL FLAG (S->LR1 S L TABLE) C)) 'RUN))
  ((INDUCT (IHINT-2 FLAG S L TABLE C))
   (DISABLE-THEORY ADDITION)
   (ENABLE P-PSW-RUN-LR-IF-OK-P-PSW-RUN)
   (EXPAND (S-EVAL FLAG S C) (S-EVAL 'LIST S C) (S-EVAL FLAG S 0)
           (S-ALL-TEMPS-SETP FLAG
                             (S-EXPR S)
                             (TEMP-ALIST-TO-SET (S-TEMPS S)))
           (S-ALL-TEMPS-SETP 'LIST
                             (S-EXPR-LIST S)
                             (TEMP-ALIST-TO-SET (S-TEMPS S))))
   (DISABLE LR-COMPILE-BODY LR-EVAL LR-COMPILE-PROGRAMS
            LR-MAKE-TEMP-NAME-ALIST MAX S-COLLECT-ALL-TEMPS S-ALL-TEMPS-SETP
            S-EVAL
            GOOD-POSP-LIST-NX-T-SIMPLE
            L-PROPER-EXPR-S-ALL-TEMPS-SETP
            LR-PROGRAMS-PROPERP-LR->P-S->LR1-DEFINEDP-S-PNAME
            PROPER-P-STATEP-LR->P-STRIP-CARS-BINDINGS-CTRL-STK
            S-EVAL-L-EVAL-FLAG-T
            S-EVAL-L-EVAL-FLAG-RUN-FLAG-T)))

(PROVE-LEMMA LR-EVAL-P-PC-EQUIVALENCE ()
  (IMPLIES (AND (PROPER-P-STATEP (LR->P L))
                (GOOD-POSP FLAG
                           (OFFSET (P-PC L))
                           (PROGRAM-BODY (P-CURRENT-PROGRAM L)))
                (OR (NOT (EQUAL FLAG 'LIST))
                    (NOT (EQUAL (CAR (CUR-EXPR (BUTLAST (OFFSET (P-PC L)))
                                               (PROGRAM-BODY
                                                (P-CURRENT-PROGRAM L))))
                                'IF)))
                (LR-PROGRAMS-PROPERP L TABLE)
                (LR-P-PROPER-STATEP (P-TEMP-STK L)
```

```
                                  (P-CTRL-STK L)
                                  (P-DATA-SEGMENT L)
                                  TABLE)
                (LR-PROPER-FORMALSP (CDR (P-PROG-SEGMENT L)))
                (EQUAL (P-PSW (LR-EVAL FLAG L C)) 'RUN))
           (EQUAL (P (LR->P L) (P-CLOCK1 FLAG L C))
                  (P-SET-PC (LR->P (LR-EVAL FLAG L C))
                            (P-FINAL-PC FLAG L 0))))
  ((INDUCT (LR-EVAL FLAG L C))
   (ENABLE ADD-ADDR ERRORP LR-P-PC LR-PUSH-TSTK
           ASSOCIATIVITY-OF-PLUS PLUS-ZERO-ARG2
           LR-EVAL-IF-P-PSW-1
           P-SET-PC-LR->P-LR-SET-EXPR)
   (EXPAND (LR-EVAL FLAG L C) (LR-EVAL 'LIST L C)
           (P-CLOCK1 FLAG L C)
           (P-CLOCK1 'LIST L C)
           (P-FINAL-PC 'LIST L 0) (P-FINAL-PC T L 0)
           (LR-P-C-SIZE 'LIST (LR-EXPR-LIST L)))
   (DISABLE-THEORY ADDITION)
   (DISABLE LR-EVAL LR-P-PC-1 LR-P-C-SIZE P-CLOCK1
            GOOD-POSP1-CONS-LESSP-4-IF-S-PROPER-EXPRP
            GOOD-POSP1-CONS-LESSP-4-IF-LR-PROPER-EXPRP
            GOOD-POSP1-DV-1-TEMP-EVAL-TEST
            GOOD-POSP1-LIST-GOOD-POSP-LIST-T
            GOOD-POSP-LIST-NX-T-SIMPLE
            GOOD-POSP1-LR-PROPER-EXPRP-GET-CADDDR
            P-PLUS
            PROGRAM-BODY-ASSOC-COMP-PROGRAMS)))
```

Appendix G

Listing of Events in the Proof of the Compiler With a Garbage Collector

This chapter lists the events in the proof of the compiler with a garbage collector. Note that this version of the compiler contains only the following SUBRs: CAR, CDR, CONS, FALSE, FALSEP, IF, LISTP, NLISTP, TRUE and TRUEP. Also user-defined shells are not implemented.

The events listed in this chapter depend on the events in Appendices C, D, and E.

G.1 Events with Mechanically Checked Proofs

This section is a listing of all the events in the chronology of the proof of the compiler with a garbage collector. Every PROVE-LEMMA event in this section has been mechanically checked.

The list of events that make up the compiler proof has been omitted in the current version of this report.

G.2 Events without Mechanically Checked Proofs

This section lists major lemmas used in the proof of the compiler with a garbage collector. The PROVE-LEMMA events in this section have not been mechanically checked.

The list of events that make up the compiler proof has been omitted in the current version of this report.

Bibliography

[1] Raymond Aubin. Strategies for mechanizing structural induction. In *International Joint Conference on Artificial Intelligence*, 1977.

[2] H.G. Baker. List-processing in real time on a serial computer. *Communications of the ACM*, 21(4):185–214, April 1978.

[3] Bill Bevier. A library for hardware verification. Internal Note 57, Computational Logic, Inc., 1988.

[4] R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, New York, 1979.

[5] R. S. Boyer and J S. Moore. The addition of bounded quantification and partial functions to a computational logic and its theorem prover. *Journal of Automated Reasoning*, 4(2):117–172, 1988.

[6] R. S. Boyer and J S. Moore. *A Computational Logic Handbook.* Academic Press, Boston, 1988.

[7] R.M. Burstall. Proving properties of programs by structural induction. *Computer Journal*, 12(1):41–48, February 1969.

[8] R. Cartwright. *A Practical Formal Semantic Definition and Verification System for Typed LISP.* PhD thesis, Stanford University, 1976.

[9] 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 1977.

[10] Jacques Cohen. Garbage collection of linked data structures. *Computing Surveys*, pages 341–367, 1981.

[11] A. Cohn. High level proof in lcf. In *Proceedings of the Fifth Symposium on Automated Deduction*, 1979.

[12] Paul Corzon. Deriving correctness properties of compiled code. In *Higher Order Logic Theorem Proving and Its Applications*, 1992.

[13] E.W. Dijkstra, L. Lamport, A.J. Martin, C.S. Schloten, and E.F.M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *CACM*, 21(11), November 1978.

[14] Donald I. Good, Robert L. Akers, Lawrence M. Smith, and William D. Young. Combined report on gypsy 2.05, middle-gypsy and micro-gypsy – draft. Technical Report 1-d, Computational Logic, Inc., March 1992. supercedes August 2, 1989 edition.

[15] Guy L. Steele Jr. *Common LISP: The Language.* Digital Press, 1984.

[16] Matthew Kaufmann. An integer library for nqthm. Internal Note 182, Computational Logic, Inc., 1990.

[17] B.H. Levy. An approach to compiler correctness using interpretation between theories. Technical Report 85(8354)-1, Aerospace Corporation, April 1985.

[18] R.L. London. Correctness of two compilers for a lisp subset. Technical Report AIM-151, Stanford Univerity Artificial Intelligence Laboratory, October 1971.

[19] Donald Scott Lynn. Interactive compiler proving using hoare proof rules. Technical Report ISI/RR-78-70, Information Sciences Institute, January 1978.

[20] J. McCarthy, P.W. Abrahams, D.J. Edwards, T.P. Hart, and M.I. Levin. *LISP 1.5 Programmer's Manual.* MIT, 1962.

[21] John McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In *Proceeding of Symposium on Applied Mathematics*, volume 19. American Mathematical Society, 1967.

[22] R. Milner and R. Weyhrauch. *Proving Compiler Correctness in a Mechanized Logic*, pages 51–70. Edinburgh University Press, Edinburgh, Scotland, 1972.

[23] J Strother Moore. Piton: A verified assembly level language. Technical Report 22, Computational Logic, Inc., 1988.

[24] 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.

[25] Dino P. Oliva and Mitchell Wand. A verified compiler for pure prescheme. Technical Report NU-CCS-92-5, Northeastern University College of Computer Science, 1992.

[26] W. Polak. *Compiler Specification and Verification.* Springer-Verlag, Berlin, 1981.

[27] D. M. Russinoff. A mechnically verified incremental garbage collector. *Formal Aspects of Computing*, to appear.

[28] Thomas A. Standish. *Data Structure Techniques.* Addison-Wesley, Reading, Mass., 1980.

[29] R. Topor. The correctness of the schorr-waite list marking algorithm. Technical Report MIP-R-104, University of Edinburgh, July 1974.

[30] William D. Young. A verified code generator for a subset of gypsy. Technical Report 33, Computational Logic, Inc., 1988. Ph.D. Thesis, University of Texas at Austin.

Index

ABS1, 54
ABS2, 55
actuals, 21
ADD-ADDR instruction, summary, 22
ADD-INT instruction, summary, 22
ADD-INT-WITH-CARRY instruction, summary, 22
ADD-NAT instruction, summary, 23
ADD-NAT-WITH-CARRY instruction, summary, 23
ADD1-INT instruction, summary, 23
ADD1-NAT instruction, summary, 23
ALLOC-JUMP-TABLE, 37
AND-BITV instruction, summary, 23
AND-BOOL instruction, summary, 23
ANSWER, 36
APP, 35, 81
array, 19
ASCII-0, 101
ASCII-1, 101
ASCII-9, 101
ASCII-DASH, 101

BIGNUMs, 39
bindings, 16
BODY, 39, 45
body, 20

CALL instruction, summary, 23
CAR, 44
CHANGE-ELEMENTS, 35, 45–48
Common Lisp, 79
COMP-BODY, 118
COMP-BODY-1, 115
COMP-FUNCALL, 115
COMP-IF, 112
COMP-LITATOM, 115
COMP-POP-FORMALS, 116
COMP-POP-TEMPS, 117
COMP-PROGRAMS, 63, 119
COMP-PROGRAMS-1, 63, 119
COMP-QUOTE, 114
COMP-TEMP-EVAL, 113

COMP-TEMP-FETCH, 113
COMP-TEMP-TEST, 114
CONS, 41
control stack, 15
COUNT-CODELIST, 102
COUNT-CODELIST1, 102
COUNT-LIST, 121
current program, 16

data area, 19
data segment, 15
def-label form, 21
DELETE-ALL, 96
DEPOSIT instruction, summary, 24
DEPOSIT-A-LIST, 103
DEPOSIT-TEMP-STK instruction, summary, 24
DIV2-NAT instruction, summary, 24

effects function, 31
EQ instruction, summary, 24
erroneous, 32
error conditions, 32

FALSE, 38
FALSEP, 38
FETCH instruction, summary, 24
FETCH-TEMP-STK instruction, summary, 24
FIXNUMs, 39
formal parameters, 20
FORMALS, 39, 45
frame, 16
FREE-PTR, 36, 38

GENSYM, 102
global variable, 19

HEAP, 36, 38, 39
HEAP-SIZE, 36

I-ALLOC-NODE, 37, 41
I-DECR-REF-COUNT, 41
INCREMENT-NUMLIST, 101
initial value, 20
INT-TO-NAT instruction, summary, 24

JUMP instruction, summary, 25
JUMP-CASE instruction summary, 25
JUMP-IF-TEMP-STK-EMPTY instruction, summary, 25
JUMP-IF-TEMP-STK-FULL instruction, summary, 25

L-EVAL, 64, 65
L-EVAL-NOT-F-V&C$-EQUIVALENCE, 66
L-PROPER-EXPRP, 54
L-PROPER-PROGRAMSP, 54
label, 21

LABEL-INSTRS, 112
LIST-ASCII-0, 101
LIST-ASCII-1, 101
LISTP, 39
LITATOM, 39
local variables, 21
LOCN instruction, summary, 25
LOGIC->P, 34, 37, 50, 51, 57, 125
LOGIC->P-OK-REALLY, 51
LOGIC->S, 57, 59, 67, 68, 96
LOGIC->S-OK, 68
LR-0-ADDR, 99
LR->P, 57, 63, 119
LR-ADD-TO-DATA-SEG, 120
LR-ADD1-TAG, 99
LR-ALLOC-NODE-JUMP-TABLE-ADDR, 99
LR-ANSWER-ADDR, 99
LR-BIGNUM-OFFSET, 100
LR-CAR-OFFSET, 100
LR-CDR-OFFSET, 100
LR-CHECK-FREE-NODES, 73
LR-CHECK-RESULT, 75
LR-COMPILE-BODY, 60, 104
LR-COMPILE-PROGRAMS, 105
LR-COMPILE-QUOTE, 49, 62, 121
LR-CONS-TAG, 99
LR-CONVERT-DIGIT-TO-ASCII, 112
LR-CONVERT-NUM-TO-ASCII, 112
LR-DATA-SEG-TABLE, 60, 123
LR-DATA-SEG-TABLE-BODY, 122
LR-DATA-SEG-TABLE-LIST, 123
LR-EVAL, 65, 66
LR-EVAL-P-PC-EQUIVALENCE, 76, 130
LR-EVAL-S-EVAL-EQUIVALENCE, 74, 129
LR-EVAL-S-EVAL-FLAG-RUN, 75, 130
LR-F-ADDR, 99
LR-FALSE-TAG, 98
LR-FETCH-FP, 100
LR-FP-ADDR, 99
LR-FREE-LIST-NODES, 73
LR-HEAP-NAME, 99
LR-INIT-DATA-SEG, 120
LR-INIT-DATA-SEG-TABLE, 123
LR-INIT-HEAP-CONTENTS, 120
LR-INIT-TAG, 98
LR-INITIAL-CSTK, 60, 104
LR-MAKE-ALLOC-NODE-JUMP-TABLE, 120
LR-MAKE-INITIAL-TEMPS, 103
LR-MAKE-LABEL, 112
LR-MAKE-PROGRAM, 101
LR-MAKE-TEMP-NAME-ALIST, 60, 103
LR-MAKE-TEMP-NAME-ALIST-1, 103
LR-MAKE-TEMP-VAR-DCLS, 104
LR-MINIMUM-HEAP-SIZE, 100
LR-MINUS-TAG, 99

LR-NEGATIVE-GUTS-OFFSET, 101
LR-NEW-CONS, 103
LR-NEW-NODE, 100
LR-NODE-SIZE, 99
LR-P-C-SIZE, 117
LR-P-C-SIZE-FORMALS, 116
LR-P-C-SIZE-FUNCALL, 115
LR-P-C-SIZE-IF, 113
LR-P-C-SIZE-LIST, 117
LR-P-C-SIZE-LITATOM, 115
LR-P-C-SIZE-QUOTE, 115
LR-P-C-SIZE-TEMP-EVAL, 114
LR-P-C-SIZE-TEMP-FETCH, 113
LR-P-C-SIZE-TEMP-TEST, 114
LR-P-C-SIZE-TEMPS, 117
LR-P-PC, 119
LR-P-PC-1, 60, 118
LR-PACK-TAG, 99
LR-PROPER-FREE-LISTP, 40, 73
LR-PROPER-HEAPP, 72
LR-PROPER-P-STATESP, 72
LR-PROPER-REF-COUNTSP, 73
LR-S-SIMILAR-STATESP, 75
LR-STATE, 65
LR-T-ADDR, 99
LR-TAG-OFFSET, 100
LR-TRUE-TAG, 98
LR-UNBOX-NAT-OFFSET, 100
LR-UNDEF-ADDR, 99
LR-UNDEFINED-TAG, 98
LR-UNPACK-OFFSET, 101
LSH-BITV instruction, summary, 25
LT-ADDR instruction, summary, 25
LT-INT instruction, summary, 25
LT-NAT instruction, summary, 25

MAKE-SYMBOL, 101
MAX-COUNT-CODELIST, 102
maximum control stack size, 15
maximum temporary stack size, 15
MULT2-NAT instruction, summary, 26
MULT2-NAT-WITH-CARRY-OUT instruction, summary, 26

name (of a program), 20
NEG-INT instruction, summary, 26
NEGATIVEP, 40
NO-OP instruction, summary, 26
NOT-BITV instruction, summary, 26
NOT-BOOL instruction, summary, 26
NUMBERP, 39

ok predicate, 31
OR-BITV instruction, summary, 26
OR-BOOL instruction, summary, 26

P, 31, 32, 50, 51, 65
P-ACCESSOR-CODE, 108
P-CAR-CODE, 109
P-CDR-CODE, 109
P-CONS-CODE, 109
P-FALSE-CODE, 109
P-FALSEP-CODE, 110
P-I-ALLOC-NODE-CODE, 105
P-I-DECR-REF-COUNT-CODE, 105
P-LISTP-CODE, 110
P-NLISTP-CODE, 110
P-OBJECTP, 17
P-RECOGNIZER-CODE, 107
P-RUNTIME-SUPPORT-PROGRAMS, 111
P-STATE, 15, 16, 65
P-TRUE-CODE, 111
P-TRUEP-CODE, 111
PAIR-FORMALS-WITH-ADDRESSES, 103
POP instruction, summary, 26
POP* instruction, summary, 27
POP-CALL instruction, summary, 27
POP-GLOBAL instruction, summary, 27
POP-LOCAL instruction, summary, 27
POP-LOCN instruction, summary, 27
POPJ instruction, summary, 27
POPN instruction, summary, 27
precondition, 31
program counter, 15
program definition, 20
program segment, 15
program status word, 16
proper p-state, 16
proper state, 67
proper states, 72
PROPER-P-PROGRAMP, 22
PROPER-P-STATEP, 16
psw, 16
PUSH-CONSTANT instruction, summary, 27
PUSH-CTRL-STK-FREE-SIZE instruction, summary, 28
PUSH-GLOBAL instruction, summary, 28
PUSH-LOCAL instruction, summary, 28
PUSH-TEMP-STK-FREE-SIZE instruction, summary, 28
PUSH-TEMP-STK-INDEX instruction, summary, 28
PUSHJ instruction, summary, 28

REMOVE-COSTS, 66
REMOVE-DUPLICATES, 96
representable, 18
RET instruction, summary, 29
return program counter, 16
RSH-BITV instruction, summary, 29

S->LR, 57, 60, 67, 124
S->LR1, 60, 61, 124
S-ANS, 95

S-BODY, 97
S-CONSTRUCT-PROGRAMS, 96
S-ERR-FLAG, 95
S-EVAL, 65, 66
S-EVAL-L-EVAL-FLAG-RUN-FLAG-T, 68
S-EVAL-L-EVAL-FLAG-T, 68
S-FORMALS, 97
S-GOOD-STATEP, 67
S-PARAMS, 95
S-PNAME, 95
S-POS, 95
S-PROG, 97
S-PROGRAMS-OKP, 67
S-PROGS, 95
S-STATE, 58, 65, 95
S-STATEP, 95
S-TEMP-EVAL, 95
S-TEMP-FETCH, 95
S-TEMP-LIST, 97
S-TEMP-TEST, 95
S-TEMPS, 95
SET-GLOBAL instruction, summary, 29
SET-LOCAL instruction, summary, 29
SHELL-SIZES, 37
step function, 31
SUB-ADDR instruction, summary, 29
SUB-INT instruction, summary, 29
SUB-INT-WITH-CARRY instruction, summary, 29
SUB-NAT instruction, summary, 29
SUB-NAT-WITH-CARRY instruction, summary, 30
SUB1-INT instruction, summary, 30
SUB1-NAT instruction, summary, 30
subroutine definition, 20
SUBRP, 39
SUBSEQP, 102

temporary stack, 15
temporary variables, 20
TEST-BITV-AND-JUMP instruction, summary, 30
TEST-BOOL-AND-JUMP instruction, summary, 30
TEST-INT-AND-JUMP instruction, summary, 30
TEST-NAT-AND-JUMP instruction, summary, 31
TOTAL-OCCURRENCES, 74
TRUE, 39
TRUEP, 39

Undefined, 38
Unused, 38
USER-FNAME, 96
USER-FNAME-PREFIX, 95

V&C$, 34, 35, 50, 52, 53, 65
V&C$-L-EVAL-EQUIVALENCE, 65
VALP, 56
value, 20

word size, 15

`XOR-BITV` instruction, summary, 31