# A Compiler for NQTHM:
# A Progress Report

Arthur Flatau

Technical Report 74 January, 1992

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

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(s) 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

This report describes a compiler for the NQTHM logic and a mechanically checked proof of its correctness. The NQTHM logic defines an applicative programming language very similar to McCarthy's pure Lisp [McCarthy 62]. The compiler compiles programs in the NQTHM logic into the Piton assembly level language [Moore 88]. 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$**. A completed prototype for the compiler is discussed. The prototype includes 10 of the 61 predefined functions in the NQTHM logic. Design of a garbage collector is also discussed in this report.

# 1. Introduction

This report describes a compiler for an applicative programming language similar to pure Lisp and the mechanically checked proof of its correctness. The Boyer-Moore logic (hereafter referred to as the Logic) defines an applicative programming language similar to pure Lisp. The compiler produces code in the high-level assembly language Piton [Moore 88].

Since the Logic includes **`CONS`**, the implementation must dynamically allocate storage, and inclusion of a garbage collector is desirable.

Among the significant achievements of this project are the following: The compiler is proven to correctly implement one abstract data-type (e.g. CONS) on top of a different concrete data-type (a large array). The runtime system, including the dynamic storage allocation and reference count garbage collector, has mechnically checked proofs of correctness.

## 1.1 Motivation

The correctness of a computer system depends on many different things. Not only must the program be correct, but the compiler, the assembler, the run-time support and the hardware must work as specified.

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 that is output by the compiler has an assembler and linker that have been proven correct, and the hardware FM8502 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 runtime support system, including dynamic memory allocation and automatic recovery of unused memory.

## 1.2 Current Status

The construction and verification of the compiler is ongoing. Currently the compiler does not include the reference count garbage collector. In addition, user-defined shells have not been implemented, and only ten of the sixty one primitive (*Ground Zero*) functions of the Boyer Moore logic have been implemented. We indicate in this report which portions of the implementation are currently working and which are still left to be done. Production of the full scale compliler requires defining the functions of the Logic in Piton, proving they are correctly implemented and developing the garbage collector.

# 2. The Boyer-Moore Logic

The Boyer-Moore Logic [Boyer & Moore 88] defines an applicative programming language that bears a strong resemblance to pure Lisp [McCarthy 62]. In this section we briefly describe the Boyer-Moore logic as a programming language.

The syntax of Pure Lisp is used to write down terms in the Logic, e.g. we write **`(PLUS X Y)`** instead of **`PLUS (I, J)`** or **`I + J`**.

The *Ground Zero* Logic provides six built-in types. The user can defined additional types. All the types are distinct. The built in types include:

1. **`FALSE`** or **`F`**
2. **`TRUE`** or **`T`**.
3. **`NUMBERP`** the natural numbers

4. **LISTP** the ordered pairs
5. **LITATOM** the symbols or literal atoms
6. **NEGATIVEP** the negative integers

There are functions provided in the *Ground Zero* Logic that implement many of the usual operations on numbers, lists and symbols. The usual Lisp primitives for creating and accessing lists are available. **CONS** takes two objects and returns an ordered pair, the **CAR** of which is the first object and the **CDR** of which is the second object.

The function **SUBRP** determines if a literal atom names a function that is a ''primitive''. A primitive function is either one that is in the *Ground Zero* Logic or is the name of a shell constructor or shell accessor for a user defined shell. If a symbol is not a **SUBRP** we call it a ''non-**SUBRP**''.

The Logic also includes an interpreter that is capable of determining the value of terms in the Logic. The interpreter called ''**V&C$**'' (for value and cost) takes an expression **EXPR** and an ''environment'' **ALIST** which is an association list that gives bindings for the free variables in **EXPR**. **V&C$** returns a pair, the **CAR** of which is the result of evaluating the expression, and the **CDR** being the 'cost' of the evaluation. It can also return **FALSE** instead of a pair, which means that there does not exist a cost that allows the evaluation of the expression (i.e. attempting to evaluate the expression 'loops forever'). Note that the implementation of **V&C$** in the Boyer-Moore theorem prover, as well as the Piton translation provided in the compiler, never return **FALSE**. Instead they 'loop forever' on a expression that cannot be evaluated.

## 3. Piton

Piton is a high-level assembly language developed by J Moore [Moore 88]. 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 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 does not use all of the data types, e.g. it does not use bit vectors and subroutine names.

There are stack-based instructions for manipulating the various abstract objects, standard flow-of-control instructions and instructions for determining resource limitations. All the Piton instructions are shown in Figure 1. The compiler uses a fraction of the available Piton instructions.

The semantics of Piton are provided operationally by an interpreter in the Boyer-Moore Logic named **P**. **P** takes as arguments a Piton State or *p-state* and a natural number, which indicates how many instructions are to be executed.

A *p-state* has nine components:

- a *program counter*, which indicates the instruction to be executed next. The program counter is a Piton program address. Program addresses have two parts; the name of a subroutine and a natural number that specifies one of the instructions of that subroutine.
- a *control stack*, recording the hierarchy of subroutine invocations leading to the current state. 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 the *bindings* of the local variables of the invoked program

| Control | Integers | Natural Numbers |
|---|---|---|
| CALL | ADD-INT | ADD-NAT |
| JUMP | ADD1-INT | ADD-NAT-WITH-CARRY |
| JUMP-CASE | EQ | ADD1-NAT |
| NO-OP | INT-TO-NAT | EQ |
| RET | LT-INT | LT-NAT |
| TEST-BOOL-AND-JUMP | NEG-INT | SUB-NAT |
| TEST-INT-AND-JUMP | SUB-INT | SUB-NAT-WITH-CARRY |
| TEST-NAT-AND-JUMP | SUB-INT-WITH-CARRY | SUB1-NAT |
| | SUB1-INT | |

| Variables | Booleans | Data Addresses |
|---|---|---|
| POP-GLOBAL | AND-BOOL | DEPOSIT |
| POP-LOCAL | EQ | FETCH |
| PUSH-GLOBAL | NOT-BOOL | SUB-ADDR |
| PUSH-LOCAL | OR-BOOL | |

| Stack |
|---|
| DEPOSIT-TEMP-STK |
| FETCH-TEMP-STK |
| POP |
| POP* |
| POPN |
| PUSH-CONSTANT |
| PUSH-TEMP-STK-INDEX |

**Figure 1:** Piton Instructions

and *return program counter*.

- 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 disjoint collection of 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)*.

Piton is described fully in [Moore 88] and is summarized in [Moore 89].

## 3.1 Piton Deficiency

During the course of the construction and proof of the prototype compiler, a deficiency in Piton was discovered. Piton does not allow access to frames on the control stack other than the current one. This would make it difficult to build a garbage collector that stops the computation when the heap is full and reclaims any garbage. This deficiency has no effect on the current design of the garbage collector, since it only operates on the top frame of the control stack. Actual parameters could not be passed on the control stack with this kind of garbage collector. The easiest solution would be to build another stack in the Piton data segment for passing parameters.

## 4. Informal Description of the Compiler

The compiler is defined by the function **LOGIC->P**. **LOGIC->P** 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 bindings association list of the NQTHM interpreter **V&C$**. **PROGRAM-NAMES** is a list of symbols that represent functions in the Logic to be compiled. **HEAP-SIZE**, **MAX-CTRL**, **MAX-TEMP** and **WORD-SIZE** give the allocations for the resource limitations of Piton. **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.

In the description of the prototype compiler in this section we note how the compiler will change when the garbage collector is added.

### 4.1 An Example

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

*Variable Alist:* **(LIST (CONS ’X F) (CONS ’Y (CONS T (CONS T F))))**

```
(DEFN APP (X Y)
      (IF (LISTP X)
          (CONS (CAR X) (APP (CDR X) Y))
          Y))

(DEFN CHANGE-ELEMENTS (LIST)
      (IF (LISTP LIST)
          (IF (TRUEP (CAR LIST))
              (CONS F (CHANGE-ELEMENTS (CDR LIST)))
              (CONS T
                    (CHANGE-ELEMENTS (CDR LIST))))
         (IF (TRUEP LIST) F T)))
```

**Figure 2:** An example program

First we consider an example of compiling a simple expression. Note that the current prototype 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**, so we cannot easily write very interesting programs.

Consider the evaluation of the expression shown in Figure 2 under the substitution given by the variable alist in Figure 2, where the definitions of **APP** and **CHANGE-ELEMENTS** are also shown in Figure 2. The result of compiling with **LOGIC->P** is shown in Figure 3.

### 4.2 The Resources of 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.

```
(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)
 ==>    (P-STATE '(PC (MAIN . 0))
                 '((((X . (ADDR (HEAP . 4))) (Y . (ADDR (HEAP . 20)))
                     (PC (MAIN . 0))))
                   ()
                   '((MAIN (X Y) ()
                           (DL L-0 () (PUSH-LOCAL X))
                           (DL L-1 () (PUSH-LOCAL Y))
                           (DL L-2 () (CALL U-CHANGE-ELEMENTS))
                           (DL L-3 () (CALL CONS))
                           (DL L-4 () (PUSH-CONSTANT (ADDR (HEAP . 16))))
                           (DL L-5 () (CALL U-APP))
                           (DL L-6 () (SET-GLOBAL ANSWER))
                           (DL L-7 () (RET)))
                     (U-APP (X Y) ()
                            (DL L-0 () (PUSH-LOCAL X))
                            (DL L-1 () (CALL LISTP))
                            (DL L-2 () (PUSH-CONSTANT (ADDR (HEAP . 4))))
                            (DL L-3 () (EQ))
                            (DL L-4 () (TEST-BOOL-AND-JUMP T L-13))
                            (DL L-5 () (PUSH-LOCAL X))
                            (DL L-6 () (CALL CAR))
                            (DL L-7 () (PUSH-LOCAL X))
                            (DL L-8 () (CALL CDR))
                            (DL L-9 () (PUSH-LOCAL Y))
                            (DL L-10 () (CALL U-APP))
                            (DL L-11 () (CALL CONS))
                            (DL L-12 () (JUMP L-14))
                            (DL L-13 () (PUSH-LOCAL Y))
                            (DL L-14 () (RET)))
                     (U-CHANGE-ELEMENTS (LIST) ()
                                        (DL L-0 () (PUSH-LOCAL LIST))
                                        (DL L-1 () (CALL LISTP))
                                        (DL L-2 () (PUSH-CONSTANT (ADDR (HEAP . 4))))
                                        (DL L-3 () (EQ))
                                        (DL L-4 () (TEST-BOOL-AND-JUMP T L-23))
                                        (DL L-5 () (PUSH-LOCAL LIST))
                                        (DL L-6 () (CALL CAR))
                                        (DL L-7 () (CALL TRUEP))
                                        (DL L-8 () (PUSH-CONSTANT (ADDR (HEAP . 4))))
                                        (DL L-9 () (EQ))
                                        (DL L-10 () (TEST-BOOL-AND-JUMP T L-17))
                                        (DL L-11 () (PUSH-CONSTANT (ADDR (HEAP . 4))))
                                        (DL L-12 () (PUSH-LOCAL LIST))
                                        (DL L-13 () (CALL CDR))
                                        (DL L-14 () (CALL U-CHANGE-ELEMENTS))
                                        (DL L-15 () (CALL CONS))
                                        (DL L-16 () (JUMP L-22))
                                        (DL L-17 () (PUSH-CONSTANT (ADDR (HEAP . 8))))
                                        (DL L-18 () (PUSH-LOCAL LIST))
                                        (DL L-19 () (CALL CDR))
                                        (DL L-20 () (CALL U-CHANGE-ELEMENTS))
                                        (DL L-21 () (CALL CONS))
                                        (DL L-22 () (JUMP L-31))
                                        (DL L-23 () (PUSH-LOCAL LIST))
                                        (DL L-24 () (CALL TRUEP))
                                        (DL L-25 () (PUSH-CONSTANT (ADDR (HEAP . 4))))
                                        (DL L-26 () (EQ))
                                        (DL L-27 () (TEST-BOOL-AND-JUMP T L-30))
                                        (DL L-28 () (PUSH-CONSTANT (ADDR (HEAP . 4))))
                                        (DL L-29 () (JUMP L-31))
                                        (DL L-30 () (PUSH-CONSTANT (ADDR (HEAP . 8))))
                                        (DL L-31 () (RET)))
```

**Figure 3:** Piton translation of example program

```
(CAR (X) ()
     (PUSH-LOCAL X)
     (FETCH)
     (PUSH-CONSTANT (NAT 5))
     (EQ)
     (TEST-BOOL-AND-JUMP T ARG1)
     (PUSH-CONSTANT (ADDR (HEAP . 12)))
     (RET)
     (DL ARG1 () (PUSH-LOCAL X))
     (PUSH-CONSTANT (NAT 2))
     (ADD-ADDR)
     (FETCH)
     (RET))
(CDR (X) ()
     (PUSH-LOCAL X)
     (FETCH)
     (PUSH-CONSTANT (NAT 5))
     (EQ)
     (TEST-BOOL-AND-JUMP T ARG1)
     (PUSH-CONSTANT (ADDR (HEAP . 12)))
     (RET)
     (DL ARG1 () (PUSH-LOCAL X))
     (PUSH-CONSTANT (NAT 3))
     (ADD-ADDR)
     (FETCH)
     (RET))
(CONS () ((TEMP (NAT 0)))
      (PUSH-GLOBAL FREE-PTR)
      (PUSH-CONSTANT (NAT 3))
      (ADD-ADDR)
      (DEPOSIT)
      (PUSH-GLOBAL FREE-PTR)
      (PUSH-CONSTANT (NAT 2))
      (ADD-ADDR)
      (DEPOSIT)
      (PUSH-GLOBAL FREE-PTR)
      (PUSH-GLOBAL FREE-PTR)
      (PUSH-CONSTANT (NAT 1))
      (ADD-ADDR)
      (SET-LOCAL TEMP)
      (FETCH)
      (PUSH-CONSTANT (NAT 1))
      (PUSH-LOCAL TEMP)
      (DEPOSIT)
      (PUSH-CONSTANT (NAT 5))
      (PUSH-GLOBAL FREE-PTR)
      (DEPOSIT)
      (POP-GLOBAL FREE-PTR)
      (RET))
(FALSE () ()
       (PUSH-CONSTANT (ADDR (HEAP . 4)))
       (RET))
(FALSEP () ()
        (PUSH-CONSTANT (ADDR (HEAP . 4)))
        (EQ)
        (TEST-BOOL-AND-JUMP T TRUE)
        (PUSH-CONSTANT (ADDR (HEAP . 4)))
        (RET)
        (DL TRUE () (PUSH-CONSTANT (ADDR (HEAP . 8))))
        (RET))
(LISTP () ()
       (FETCH)
       (PUSH-CONSTANT (NAT 5))
       (EQ)
       (TEST-BOOL-AND-JUMP F FALSE)
```

**Figure 3, continued**

```
        (PUSH-CONSTANT (ADDR (HEAP . 8)))
        (RET)
        (DL FALSE () (PUSH-CONSTANT (ADDR (HEAP . 4))))
        (RET))
 (NLISTP () ()
         (FETCH)
         (PUSH-CONSTANT (NAT 5))
         (EQ)
         (TEST-BOOL-AND-JUMP F TRUE)
         (PUSH-CONSTANT (ADDR (HEAP . 4)))
         (RET)
         (DL TRUE () (PUSH-CONSTANT (ADDR (HEAP . 8))))
         (RET))
 (TRUE () ()
       (PUSH-CONSTANT (ADDR (HEAP . 8)))
       (RET))
 (TRUEP () ()
        (FETCH)
        (PUSH-CONSTANT (NAT 3))
        (EQ)
        (TEST-BOOL-AND-JUMP F FALSE)
        (PUSH-CONSTANT (ADDR (HEAP . 8)))
        (RET)
        (DL FALSE () (PUSH-CONSTANT (ADDR (HEAP . 4))))
        (RET)))
'((FREE-PTR (ADDR (HEAP . 24)))
  (ANSWER (NAT 0))
  (HEAP (NAT 0)            ; Undefined node [HEAP address 0]
        (NAT 1)
        (ADDR (HEAP . 0))
        (ADDR (HEAP . 0))
        (NAT 2)            ; FALSE [HEAP address 4]
        (NAT 1)
        (ADDR (HEAP . 0))
        (ADDR (HEAP . 0))
        (NAT 3)            ; TRUE [HEAP address 8]
        (NAT 1)
        (ADDR (HEAP . 0))
        (ADDR (HEAP . 0))
        (NAT 4)            ; ZERO [HEAP address 12]
        (NAT 1)
        (NAT 0)
        (ADDR (HEAP . 0))
        (NAT 5)            ; (CONS T F) [HEAP address 16]
        (NAT 1)
        (ADDR (HEAP . 8))
        (ADDR (HEAP . 4))
        (NAT 5)            ; (CONS T (CONS T F)) [HEAP address 20]
        (NAT 1)
        (ADDR (HEAP . 8))
        (ADDR (HEAP . 16))
```

**Figure 3, continued**

```
        ; unused node [HEAP address 24]
        (NAT 1)
        (ADDR (HEAP . 28))
        (NAT 0)
        (NAT 0)
        ; unused node [HEAP address 28]
        (NAT 1)
        (ADDR (HEAP . 32))
        (NAT 0)
        (NAT 0)
        ; unused node [HEAP address 32]
        (NAT 1)
        (ADDR (HEAP . 36))
        (NAT 0)
        (NAT 0)
        ; unused node [HEAP address 36]
        (NAT 1)
        (ADDR (HEAP . 40))
        (NAT 0)
        (NAT 0)
        ; unused node [HEAP address 40]
       (NAT 1)
       (ADDR (HEAP . 44))
       (NAT 0)
       (NAT 0)
        ; unused node [HEAP address 44]
       (NAT 1)
       (ADDR (HEAP . 48))
       (NAT 0)
       (NAT 0)
        ; last node in free list [HEAP address 48]
        (NAT 1)))
25 10 32 'RUN)
```

**Figure 3**, concluded

### 4.2.1 Representing Data in Piton

The compiler allocates three areas in the data-segment.

1. **FREE-PTR**, an area of length one (i.e. a variable). The **FREE-PTR** contains a pointer (Piton data address) to 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 pointer into the Piton data segment.
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.

When the garbage collector is added an additional area will be needed:

4. **END-PTR**, an area of length one (i.e. a variable). The **END-PTR** contains a pointer (Piton data address) to the end of the free list. When unused cells are returned to the free list they are appended here.

The Logic has six built-in shells; the natural number (**NUMBERP**s), the ordered pairs (**LISTP**s), the literal atoms or words (**LITATOM**s), the negative integers (**NEGATIVEP**s), **TRUE** or **T** and **FALSE** or **F**. In addition, the user can define more shells.

```
                           +-----------------+
node address + 0           |       TAG       |
                           +-----------------+
node address + 1           | REFERENCE COUNT |
                           +-----------------+
node address + 2           |    CONTENTS 1   |
                           +-----------------+
node address + 3           |    CONTENTS 2   |
                           +-----------------+
```

**Figure 4:** The structure of a *Node*

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 the type of an object at run time. A garbage collector needs to determine the type of an object in order to find out if it is in use. A tag is stored with every object that encodes the type of the object.

All objects in the Logic are represented by Piton data addresses that point into the Piton data area **HEAP**. We will use *heap address n* to represent 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 4.

The first word in a *node* 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 second word is a Piton natural that is the reference count (in the current prototype, the reference counts are just set to an arbitrary Piton natural). 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 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 words of storage. 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.

As an example, consider the **P-STATE** shown in Figure 3. The comments 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 will be 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. The object **F** is always represented by the heap address **4**. **T** is always represented by the heap address **8**. To test whether a value is **F** or **T**, it is only necessary to test if it is equal to the appropriate address. The heap address **12** always contains the natural number **0**; however, it is possible that there will be multiple representations of **0**. The content fields for the nodes representing **T** and **F** are ignored but are initialized to the address of the undefined node (heap address **0**) so that the content fields of all nodes (except **NUMBERP** nodes) contain valid addresses.

The following lists the format of all data-types. Each of the below types 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 **not** to be 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. The reference count field is used to link with the next node on the free list.

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 **HEAP**.

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.

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**: 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 LITATOM. The actual format is that the second node uses the tag field to store the value that **SUBRP** should return. The reference count field points to the previous **LITATOM** in this bucket, and the first content field points to the next one (see below). 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.[1]

   **LITATOM**s are stored uniquely. They are stored in a very simple hash table. The **LITATOM** constructor **PACK** hashs 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 **LITATOM**s 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

[1]Note: **LITATOM**s that are created at runtime will have **BODY**, **FORMALS** and **SUBRP** equal **FALSE**. It is necessary for the **LITATOM** to be created at compile time to have the correct values for **BODY**, **FORMALS** and **SUBRP**. This is not a significant drawback. By adding an additional binding in the binding alist with the **LITATOM**s that need **SUBRP**, **FORMALS** and **BODY** will create them at compile time.

the *other* bucket. Otherwise, if it is a list it takes the **CAR**. If the **CAR** is a **NUMBERP** 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 **UNPACK**ed 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.

8. User-defined shells: 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 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.

Unused nodes are stored in a linked list, the Piton variable **FREE-PTR** points to the beginning of this free list. Each node is tagged with a natural indicating it is unused, and the reference count field is used as a link to the next node. By examining the reference count field it is possible to tell whether a 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 Piton programs to determine the type of an object at run-time. The runtime 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. Nodes that are not on the free list may be in use or could be garbage.

When the garbage counter is added, the free list will change. **FREE-PTR** will still point to the start of the free list. However, it will not be the case that all the nodes will be tagged as unused. When there are no more pointers to an object it will be returned to the free list. The Piton variable **END-PTR** points to the last node on the free list. When a node is returned to the free list the reference count field of the last node on the free list will be set to point to the node being added.

Consider what happens when a node that is being used as a **CONS** is returned to the free list, for instance, after executing a call to **CAR**. The reference count of the node pointed to by **END-PTR** is changed to point to the node being added, then **END-PTR** is set to point to the node added.

Consider now allocating a **CONS** cell that has been returned to the free list. The contents of the **CONS** cell are pointers to the nodes representing the **CAR** and **CDR**. First each of the reference counts of these nodes is checked. If they are **0** the nodes must be returned to the free list. Otherwise **1** is subtracted from the reference counts of these nodes.

### 4.3 Code generation

The last step in the compiler is the generation of the Piton code. Each function is compiled by itself. Piton programs have four components: name, formals, temporary variable declarations and body. The name of the Piton function has a **U-** prepended 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 6.1 for more details. The body is obtained by recursively compiling the form returned by applying **BODY** to the function name.

In the following sections we show how to compile various constructs in the Logic into Piton. An example is the function **CHANGE-ELEMENTS** from figure 2, which is shown with its Piton translation in figure 5.

```
(DEFN 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

(U-CHANGE-ELEMENTS (LIST) ()
                   (DL L-0 () (PUSH-LOCAL LIST))
                   (DL L-1 () (CALL LISTP))
                   (DL L-2 () (PUSH-CONSTANT (ADDR (HEAP . 4))))
                   (DL L-3 () (EQ))
                   (DL L-4 () (TEST-BOOL-AND-JUMP T L-23))
                   (DL L-5 () (PUSH-LOCAL LIST))
                   (DL L-6 () (CALL CAR))
                   (DL L-7 () (CALL TRUEP))
                   (DL L-8 () (PUSH-CONSTANT (ADDR (HEAP . 4))))
                   (DL L-9 () (EQ))
                   (DL L-10 () (TEST-BOOL-AND-JUMP T L-17))
                   (DL L-11 () (PUSH-CONSTANT (ADDR (HEAP . 4))))
                   (DL L-12 () (PUSH-LOCAL LIST))
                   (DL L-13 () (CALL CDR))
                   (DL L-14 () (CALL U-CHANGE-ELEMENTS))
                   (DL L-15 () (CALL CONS))
                   (DL L-16 () (JUMP L-22))
                   (DL L-17 () (PUSH-CONSTANT (ADDR (HEAP . 8))))
                   (DL L-18 () (PUSH-LOCAL LIST))
                   (DL L-19 () (CALL CDR))
                   (DL L-20 () (CALL U-CHANGE-ELEMENTS))
                   (DL L-21 () (CALL CONS))
                   (DL L-22 () (JUMP L-31))
                   (DL L-23 () (PUSH-LOCAL LIST))
                   (DL L-24 () (CALL TRUEP))
                   (DL L-25 () (PUSH-CONSTANT (ADDR (HEAP . 4))))
                   (DL L-26 () (EQ))
                   (DL L-27 () (TEST-BOOL-AND-JUMP T L-30))
                   (DL L-28 () (PUSH-CONSTANT (ADDR (HEAP . 4))))
                   (DL L-29 () (JUMP L-31))
                   (DL L-30 () (PUSH-CONSTANT (ADDR (HEAP . 8))))
                   (DL L-31 () (RET)))
```

**Figure 5:** The function CHANGE-ELEMENTS and its Piton translation

### 4.3.1 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 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 should be bound to a heap address. A variable reference is translated into the Piton instruction **PUSH-LOCAL**, with the variable as an argument. This is shown by the Piton instructions labeled **L-0**, **L-5**, **L-12**, **L-18** and **L-23** in the Piton code.

When the garbage collector is added it will be necessary to increment the reference count of the object. The following code should increment the reference count for the variable **LIST**. In the code in the following examples different labels are given to the Piton instructions than will actually be generated. The Piton code that the compiler generates labels the *Nth* instruction with **L-<N>**, where **<N>** is the ascii string representing the number **N**.

```
(DL L-0 () (PUSH-LOCAL LIST))
(DL L-1 () (PUSH-CONSTANT (NAT 1)))
(DL L-2 () (ADD-ADDR))
(DL L-3 () (FETCH))
(DL L-4 () (ADD1-NAT))
```

```
(DL L-5 () (PUSH-LOCAL LIST))
(DL L-6 () (PUSH-CONSTANT (NAT 1)))
(DL L-7 () (ADD-ADDR))
(DL L-8 () (DEPOSIT))
(DL L-9 () (PUSH-LOCAL LIST))
```

### 4.3.2 Compiling Constants

A constant is translated into a Piton data address at which the constant is stored. In order to put the address on the temporary stack, the Piton instruction **PUSH-CONSTANT** is used. On lines 2, 3, 5 and 6 of **CHANGE-ELEMENTS** the constants **F** and **T** are referenced. The translations of these references are the instructions labeled **L-11**, **L-17**, **L-28** and **L-30**, respectively. In figure 3 the constant **'(*1*TRUE . *1*FALSE)** is represented by the Piton address **(ADDR (HEAP . 16))** which is shown at line 4 of the program **MAIN**.

When the garbage collector is added the reference count of the constant will have to incremented. The push constant of **(ADDR (HEAP . 4))** at line **L-11** in figure 5 would have to be changed to:

```
(DL L-11 () (PUSH-CONSTANT (ADDR (HEAP . 5))))
(DL L-12 () (FETCH))
(DL L-14 () (ADD1-NAT))
(DL L-15 () (PUSH-CONSTANT (ADDR (HEAP . 5))))
(DL L-16 () (DEPOSIT))
(DL L-17 () (PUSH-CONSTANT (ADDR (HEAP . 4))))
```

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.

### 4.3.3 Compiling function calls

Consider the compilation of a function call, for example *(f arg0 ... argn)*. First the arguments *arg0* through *argn* 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 line 2 of Figure 5 is given by the Piton instructions labeled **L-11** to **L-15**. First copile **F** which is the **PUSH-LOCAL** labeled **L-11**. Next compile **(CHANGE-ELEMENTS (CDR LIST))** by first compiling **(CDR LIST)** (instructions labeled **L-12** and **L-13**) and then add the call to **U-CHANGE-ELEMENTS** (instruction **L-14**). Finally we add the call to **CONS** at instruction **L-15**. The names of all user-defined functions are prepended with **U-**. This is so we can have internal functions (that will be prepended with **I-**) and not conflict with the names of user functions. In addition, this ensures that the name for the main program **MAIN** is different from the names of user functions.

When the garbage collector is added the code for a function call will not change, although the code to evaluate the arguments in general will change.

Each predefined Logic **SUBRP** (with the exception of **IF**) is translated to a corresponding Piton function. The Piton functions for the **SUBRP**s defined in the prototype are part of the Piton state shown in Figure 3.

### 4.3.4 Compiling IF

**IF** is the most complex construct to compile. The execution of an **IF** proceeds by first executing the code for *test*. If *test* returns **F**, then execute the code for *else*; otherwise execute the code for *then*. An example should illustrate the compilation of **IF**. Consider the **IF** that starts on line 0 of figure 5. The code for the test **(LISTP LIST)** is the instructions labeled **L-0** and **L-1**. The instructions **L-2** through **L-4** test if the result of **(LISTP LIST)** is **F** and then jump to the beginning of the else code, **L-23** if so. The instructions labeled **L-5** through **L-21** are the compilation of the then part of the **IF**. The instruction **(DL L-22 () (JUMP L-31))** jumps around the code for the else part of the **IF**. The instructions for the else part of the **IF** are labeled **L-23** through **L-30**.

With the garbage collector it is necessary to decrement the reference count of the object return executing the code for *test* and if necessary it must be returned to the free list. When the garbage collector is added the code for the **IF** shown on line 0 will become:

```
(DL L-0 () (PUSH-LOCAL LIST))
(DL L-1 () (PUSH-CONSTANT (NAT 1)))
(DL L-2 () (ADD-ADDR))
(DL L-3 () (FETCH))
(DL L-4 () (ADD1-NAT))
(DL L-5 () (PUSH-LOCAL LIST))
(DL L-6 () (PUSH-CONSTANT (NAT 1)))
(DL L-7 () (ADD-ADDR))
(DL L-8 () (DEPOSIT))
(DL L-9 () (PUSH-LOCAL LIST))
(DL L-10 () (CALL LISTP))
(DL L-11 () (CALL I-DECREMENT-REF-COUNT))
(DL L-12 () (PUSH-CONSTANT (ADDR (HEAP . 4))))
(DL L-13 () (EQ))
(DL L-14 () (TEST-BOOL-AND-JUMP T L-23))
[...]
(DL L-22 () (JUMP L-31))
(DL L-23 () [...])
[...]
(DL L-31 () [...])
```

Note that the labels in the above will also change, but the old labels are shown as the target of jumps. The function **I-DECREMENT-REF-COUNT** that is called by the Piton instruction labeled **L-13** above is an internal function that decrements the reference count and returns the node to the free list if there are no more pointers to it, i.e., the reference count is zero. It always leaves the pointer that is on the top of the temporary stack on the top of the temporary stack.

### 4.4 Limitations

The compiler cannot correctly evaluate every expression in the Logic, or even every expression that can be evaluated in the execution environment of the theorem prover. In particular, all arithmetic will be *fixnum* arithmetic. In other words, the result of every arithmetic expression must be representable in one Piton word. 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 compiler does not provide any input and 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.

## 5. The Correctness Theorem

The theorem proved about our compiler is:

```
(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]
                 (max-word-size-reqs expr alist pnames heap-size)))
     (numberp word-size))                                    ; [17]
(lr-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)))))
```

The predicate **LR-VALP** takes three arguments: an object, an address and a data segment. **LR-VALP** returns **T** if the address in the data segment represents the object and returns **F** otherwise. The use of a predicate like **LR-VALP** at first seems odd. Ideally, one would like to extract the answer **V&C$** produces from the final **P-STATE**. Imagine such a function called **LR-ABS** which takes two arguments, an address and a data segment, and returns the object represented at that address. **LR-ABS** would have to be called recursively on another address and the data segment to extract the **CAR** of an object that is a **CONS**. There is no measure that decreases on such recursive calls. Thus **LR-ABS** could not be expected by the NQTHM theorem prover. **LR-VALP** solves this problem by using the object as both the expected result and the measure that decreases on recursive calls.

The conclusion of the theorem can be read as follows. If Piton is executed on the output of the compiler **LOGIC->P** for a certain number of steps, then the address stored in the variable **ANSWER** represents the same object as returned by **(CAR (V&C$ T EXPR ALIST))**. The function **LOGIC->P-CLOCK** is a ‘‘witness function’’, which can be thought of informally as an existential quantification on the number of steps Piton must run.

The hypotheses of the theorem seem daunting at first, but 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. That is, all the subexpressions are proper lists and each function application has the correct number of arguments. 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 (i.e., in the current prototype are either **TRUE**, **FALSE**, a **NUMBERP** or a **CONS**). Hypthosis **[5]** states that the programs, as well as the bindings in the variable alist **ALIST**, only use definedp shells. Hypothesis **[6]** states that **V&C$** produces a non **FALSE** result when applied to the expression **EXPR**. Hypotheses **[7]** and **[8]** are only necessary in the prototype. They ensure that the expression and programs only use the **SUBRP**s 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 existential quantifications on the amount of heap space, control stack space and temporary stack space, respectively.

## 6. Proof of the Correctness Theorem

The translation to Piton is accomplished in three steps. In the first step, the notion of a program counter and program segment are introduced. In addition, user programs are renamed by prefixing a **U-** to the name, so that they are distinguished from the predefined **SUBRP**s of Nqthm and the internal functions of the compiler. It would also be possible to eliminate common subexpressions at this step. The second step introduces the data structures and resource limitation of Piton. In this step the constants and data of the programs are assigned addresses in memory. The third step translates the programs into Piton. Each step involves the translation from one layer of abstraction to another and requires the proof of the correspondence between the two layers. Each layer has its semantics defined by an interpreter.

### 6.1 Eliminating Common Subexpressions -- the S Layer

The first layer is called the S level. Common subexpressions can be eliminated at this level. The semantics of the S level are defined by an interpreter called **S-EVAL**. **S-EVAL** takes three arguments: a flag for mutual recursion (the same as **V&C$**), an **S-STATE** and a clock. It returns an **S-STATE**. An **S-STATE** has seven fields. **S-PNAME** holds the name of the current program. **S-POS** holds the position of the current expression inside the current program. **S-ANS** contains the answer when **S-EVAL** returns. **S-PARAMS** holds the variable bindings alist (which is the same as the **ALIST** argument to **V&C$**). **S-TEMPS** holds an alist whose entries are triple, the **CAR** is an expression, the **CADR** is a flag which is **F** if the value of the expression has not been stored yet, (otherwise it is non-**F**) and the **CADDR** contains the value of the expression of the **CADR** is non-**F**. **S-PROGS** holds the programs. **S-ERR-FLAG** is an error flag that is **'run** unless an error has been encountered.

Originally the compiler was to have an algorithm to remove common sub-expressions. This requirement has been eliminated. However, the mechanism to add such algorithm remains. It is possible to add an algorithm to remove common sub-expressions and prove it maintains a syntactic property with only minor changes to the rest of the proof.

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: **(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 fetch the **<value>** associated with **<expression>** from **S-TEMPS**. **S-ERR-FLAG** is set to an error value if **<flag>** is **F**.

**(TEMP-EVAL)** evaluate 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 calling S-EVAL on **<expression>**. The **<flag>** field of S-TEMPS is set to T and the **<value>** field is set to the result.

```
(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 6:** An Example of Removing Common Subexpressions

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

## 6.2 Piton Resources -- The LR Layer

The translation from S-level programs to LR-level is accomplished by the function **S->LR**. **S->LR** takes an **S-STATE** and resource limitations **HEAP-SIZE**, **MAX-CTRL**, **MAX-TEMP** and **WORD-SIZE** and produces a **P-STATE**. The interpreter at the LR-level is **LR-EVAL**. **LR-EVAL** takes three arguments: a flag for mutual recursion, a **P-STATE** and a clock. The **P-STATE** is the same as the Piton state that results from compiling the LR state, except for the programs and the program counter. The programs are almost the same as the higher level (the S level), and the PC is formed from the S states **S-PNAME** (which names the current program) and the **S-POS** which specifies the current expression that is being evaluated. **LR-EVAL** returns error states in exactly the same cases as the Piton interpreter **P** returns error states when executed on translation of the LR state to Piton.

Programs at the LR level are almost the same as at the S level. There are two differences. First the three additional forms introduced at the S level now have two arguments instead of one. The second argument is the name of a local variable in which to save the previously computed value of the expression. This variable is a Piton temporary variable, which is allocated on the control stack. The second difference between programs at the LR level and the S level is that **QUOTE** forms have their guts replaced with the address in the data segment that represents that form.

**S->LR** must assign constant objects addresses in the **P-STATE** data segment and store the representations of the objects in the data segment. All the constant objects share the maximum possible structure in the resulting **P-STATE**.

### 6.3 Piton Code

The last level is Piton. The translation from the LR-level to Piton is done by the function **LR->P** which maps **P-STATE**s to **P-STATE**s. **LR->P** compiles the programs and computes the proper PC. The translations of the additional constructs for common sub-expression elimination introduced at the S-level are described in this section. In Section 4.3 the Piton translation of all other constructs in the Logic are described.

In the description below **<expression>** stands for the expression whose value is stored. Each different expression is associated with a local variable of the form **T***$n$, where $n$ is a number. The compiler guarantees that the variable is not one of the formal parameters to the function being translated and is uniquely determined by **<expression>**. The local temporary variables are initialized to the undefined node (heap address 0). In order to test whether **<expression>** has been evaluated and the result stored, the value of the correspondings local temporary variable is compared to heap address 0.

In the following description the assumption is made that the variable associated with the expression being compiled is **T*1**. Also assume that the code starts at the 27th instruction and the code for expression is 30 instructions long.

1. The compilation of a **(TEMP-FETCH)** form is:

```
(DL L-26 () (PUSH-LOCAL T*1))
```

2. The compilation of a **(TEMP-EVAL)** form is:

```
code to evaluate <expression>
(DL L-56 () (SET-LOCAL T*1))
```

3. The compilation of a **(TEMP-TEST)** form is:

```
(DL L-26 () (PUSH-LOCAL T*1))
(DL L-27 () (PUSH-CONSTANT (ADDR (HEAP . 0))))
(DL L-28 () (EQ))
(DL L-29 () (TEST-BOOL-AND-JUMP F L-61))
code to evaluate <expression>
(DL L-59 () (SET-LOCAL T*1))
(DL L-60 () (JUMP L-62))
(DL L-61 () (PUSH-LOCAL T*1))
```

When the garbage collector is added, the translation of the forms will change as shown below.

1. The compilation of a **(TEMP-FETCH)** form is:

```
(DL L-26 () (PUSH-LOCAL T*1))
(DL L-27 () (PUSH-CONSTANT (NAT 1)))
(DL L-28 () (ADD-ADDR))
(DL L-29 () (FETCH))
(DL L-30 () (ADD1-NAT))
(DL L-31 () (PUSH-LOCAL T*1))
(DL L-32 () (PUSH-CONSTANT (NAT 1)))
(DL L-33 () (ADD-ADDR))
(DL L-34 () (DEPOSIT))
(DL L-35 () (PUSH-LOCAL T*1))
```

2. The compilation of a **(TEMP-EVAL)** form is:

```
code to evaluate <expression>
(DL L-56 () (SET-LOCAL T*1))
(DL L-57 () (PUSH-CONSTANT (NAT 1)))
(DL L-58 () (ADD-ADDR))
(DL L-59 () (FETCH))
```

```
(DL L-60 () (ADD1-NAT))
(DL L-61 () (PUSH-LOCAL T*1))
(DL L-62 () (PUSH-CONSTANT (NAT 1)))
(DL L-63 () (ADD-ADDR))
(DL L-64 () (DEPOSIT))
(DL L-65 () (PUSH-LOCAL T*1))
```

3. The compilation of a **(TEMP-TEST)** form is:

```
(DL L-26 () (PUSH-LOCAL T*1))
(DL L-27 () (PUSH-CONSTANT (ADDR (HEAP . 0))))
(DL L-28 () (EQ))
(DL L-29 () (TEST-BOOL-AND-JUMP F L-61))
code to evaluate <expression>
(DL L-59 () (SET-LOCAL T*1))
(DL L-60 () (JUMP L-62))
(DL L-61 () (PUSH-LOCAL T*1))
(DL L-62 () (PUSH-CONSTANT (NAT 1)))
(DL L-63 () (ADD-ADDR))
(DL L-64 () (FETCH))
(DL L-65 () (ADD1-NAT))
(DL L-66 () (PUSH-LOCAL T*1))
(DL L-67 () (PUSH-CONSTANT (NAT 1)))
(DL L-68 () (ADD-ADDR))
(DL L-69 () (DEPOSIT))
(DL L-70 () (PUSH-LOCAL T*1))
```

## 6.4 Statistics

Table 1 provides some statistics concerning the number and kinds of events in the proof.

| | DEFN | PROVE LEMMA | TOGGLE | ADD-SHELL | ADD-AXIOM | DEF THEORY | Total |
|---|---|---|---|---|---|---|---|
| Library | 90 | 579 | 189 | 0 | 0 | 13 | 871 |
| Piton | 417 | 45 | 166 | 1 | 4 | 0 | 633 |
| Compiler | 324 | 1772 | 402 | 1 | 4 | 0 | 2503 |
| Total | 831 | 2396 | 757 | 2 | 8 | 13 | 4007 |

**Table 1:** Summary of Proof Statistics

The line Libraries in Table 1 are events from the naturals, integers and lists libraries of Bevier and Wilding [Bevier 88]. The four **ADD-AXIOM**s in the events from Piton were all proved by Moore during the course of the Piton correctness proof [Moore 88]. The four **ADD-AXIOM**s of the rest of the proof are all lemmas of the *Ground Zero* logic but are implemented in the current release of the Boyer-Moore theorem prover Nqthm. They will be included in a soon to be released version of Nqthm.

# Appendix A
# References


[Bevier 88] Bill Bevier.
*A Library for Hardware Verification*.
Internal Note 57, Computational Logic, Inc., June, 1988.

[Boyer & Moore 88]
R. S. Boyer and J S. Moore.
*A Computational Logic Handbook.*
Academic Press, Boston, 1988.

[McCarthy 62] McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P., Levin, M.I.
*LISP 1.5 Programmer's Manual*
MIT, 1962.

[Moore 88] J Strother Moore.
*PITON: A Verified Assembly Level Language*.
Technical Report 22, Computational Logic, Inc., 1988.

[Moore 89] J Strother Moore.
A Mechanically Verified Language Implemenation.
*Journal of Automated Reasoning* 5(4):493-518, December, 1989.
Also published as CLI Technical Report 30.

# Appendix B
# Code for the Compiler

The following are all the Nqthm functions used in the compiler.

**Definition**
```
(S-TEMP-EVAL) = '(TEMP-EVAL)
```

**Definition**
```
(S-TEMP-FETCH) = '(TEMP-FETCH)
```

**Definition**
```
(S-TEMP-TEST) = '(TEMP-TEST)
```

**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-)))
```

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

**Definition**
```
(S-CONSTRUCT-PROGRAMS FUN-LIST)
   =
(IF (LISTP FUN-LIST)
    (CONS (LIST (USER-FNAME (CAR FUN-LIST))
                (FORMALS (CAR FUN-LIST))
                NIL
                (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)
```

**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))
```

**Definition**
```
(LR-UNDEFINED-TAG) = 0
```

**Definition**
```
(LR-INIT-TAG) = 1
```

**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) = 'HEAP
```

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

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

**Definition**
```
(LR-NEW-NODE TAG REF-COUNT VALUE1 VALUE2)
   =
(LIST TAG REF-COUNT VALUE1 VALUE2)
```

**Definition**
```
(LR-REF-COUNT-OFFSET) = 1
```

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

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

**Definition**
```
(LR-UNPACK-OFFSET) = 2
```

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

**Definition**
```
(LR-NEGATIVE-GUTS-OFFSET) = 2
```

**Definition**
```
(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))
```

**Definition**
```
(INCREMENT-NUMLIST NUMLIST)
   =
(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)
   =
(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 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)))
```

Definition

```
(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 LIST ADDR DATA-SEG)
    =
(IF (LISTP LIST)
    (DEPOSIT (CAR LIST)
             ADDR
             (DEPOSIT-A-LIST (CDR LIST)
                             (ADD1-ADDR ADDR)
                             DATA-SEG))
    DATA-SEG)
```

Definition

```
(LR-INIT-HEAP-CONTENTS ADDR SIZE)
    =
(IF (ZEROP SIZE)
    (LIST (TAG 'NAT (LR-INIT-TAG)))
    (APPEND (LR-NEW-NODE (TAG 'NAT (LR-INIT-TAG))
                         (ADD-ADDR ADDR (LR-NODE-SIZE))
                         (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 (FETCH (LR-FP-ADDR) DATA-SEG))
```

```
                      (LENGTH NEW-NODE))))
    (DEPOSIT (FETCH (ADD-ADDR (FETCH (LR-FP-ADDR) DATA-SEG)
                              (LR-REF-COUNT-OFFSET))
                    DATA-SEG)
             (LR-FP-ADDR)
             (DEPOSIT-A-LIST NEW-NODE
                             (FETCH (LR-FP-ADDR) DATA-SEG)
                             DATA-SEG))
    DATA-SEG)
```

**Definition**

```
(LR-INIT-DATA-SEG HEAP-SIZE)
   =
(DEPOSIT-A-LIST (LIST (TAG 'NAT (LR-FALSE-TAG))
                      (TAG 'NAT 1)
                      (LR-UNDEF-ADDR)
                      (LR-UNDEF-ADDR))
                (LR-F-ADDR)
                (DEPOSIT-A-LIST (LIST (TAG 'NAT (LR-UNDEFINED-TAG))
                                      (TAG 'NAT 1)
                                      (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))
                                      (CONS (LR-HEAP-NAME)
                                            (LR-INIT-HEAP-CONTENTS
                                             (TAG 'ADDR
                                                  (CONS (LR-HEAP-NAME) 0))
                                             HEAP-SIZE)))))
```

**Definition**

```
(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 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 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 (LR-ADD1-TAG))
                                              (TAG ’NAT 1)
                                              (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 (LR-TRUE-TAG))
                                                   (TAG ’NAT 1)
                                                   (LR-UNDEF-ADDR)
                                                   (LR-UNDEF-ADDR)))
             (CONS (CONS OBJECT (FETCH (LR-FP-ADDR) HEAP)) TABLE)))
      (T (CONS HEAP (CONS (CONS OBJECT (LR-UNDEF-ADDR)) TABLE))))
```

**Definition**

```
(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 (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 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**

```
(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 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**

```
(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 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 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)
```

**Definition**

```
(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)
              (PLUS (LR-P-C-SIZE T (CADR EXPR))
                    (LR-P-C-SIZE T (CADDR EXPR))
                    (LR-P-C-SIZE T (CADDDR EXPR)) 4))
             ((EQUAL (CAR EXPR) (S-TEMP-FETCH)) 1)
             ((EQUAL (CAR EXPR) (S-TEMP-EVAL))
              (PLUS (LR-P-C-SIZE T (CADR EXPR)) 1))
             ((EQUAL (CAR EXPR) (S-TEMP-TEST))
              (PLUS (LR-P-C-SIZE T (CADR EXPR)) 7))
             ((EQUAL (CAR EXPR) 'QUOTE) 1)
             (T (PLUS (LR-P-C-SIZE 'LIST (CDR EXPR)) 1))))
      (T 1))
```

**Definition**

```
(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)))
```

**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 3
                    (LR-P-C-SIZE T (CADR EXPR))
                    (LR-P-PC-1 (CADDR EXPR) (CDR POS))))
             (T (PLUS (LR-P-C-SIZE T (CADR EXPR))
                      (LR-P-C-SIZE T (CADDR EXPR))
                      (LR-P-PC-1 (CADDDR EXPR) (CDR POS))
                      4))))
      ((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 (LR-P-PC-1 (CADR EXPR) (CDR POS)) 4))
     ((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**

```
(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**

```
(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**

```
(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**

```
(P-RECOGNIZER-CODE NAME TAG)
   =
(LIST NAME '() '()
      '(FETCH)
      (LIST 'PUSH-CONSTANT (TAG 'NAT TAG))
      '(EQ)
      '(TEST-BOOL-AND-JUMP F FALSE)
      (LIST 'PUSH-CONSTANT (LR-T-ADDR))
      '(RET)
```

```
      (LIST 'DL 'FALSE '() (LIST 'PUSH-CONSTANT (LR-F-ADDR)))
      '(RET))
```

**Definition**
```
(P-ACCESSOR-CODE NAME TAG DEFAULT OFFSET)
   =
(LIST NAME '(X) '()
      '(PUSH-LOCAL X)
      '(FETCH)
      (LIST 'PUSH-CONSTANT (TAG 'NAT TAG))
      '(EQ)
      '(TEST-BOOL-AND-JUMP T ARG1)
      (LIST 'PUSH-CONSTANT DEFAULT)
      '(RET)
      '(DL ARG1 () (PUSH-LOCAL X))
      (LIST 'PUSH-CONSTANT (TAG 'NAT OFFSET))
      '(ADD-ADDR)
      '(FETCH)
      '(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 )
   =
(LIST 'CONS '()
      '((TEMP (NAT 0)))
      '(PUSH-GLOBAL FREE-PTR)
      (LIST 'PUSH-CONSTANT (TAG 'NAT (LR-CDR-OFFSET)))
      '(ADD-ADDR)
      '(DEPOSIT)
      '(PUSH-GLOBAL FREE-PTR)
      (LIST 'PUSH-CONSTANT (TAG 'NAT (LR-CAR-OFFSET)))
      '(ADD-ADDR)
      '(DEPOSIT)
      '(PUSH-GLOBAL FREE-PTR)
      '(PUSH-GLOBAL FREE-PTR)
      (LIST 'PUSH-CONSTANT (TAG 'NAT (LR-REF-COUNT-OFFSET)))
      '(ADD-ADDR)
      '(SET-LOCAL TEMP)
      '(FETCH)
      '(PUSH-CONSTANT (NAT 1))
      '(PUSH-LOCAL TEMP)
      '(DEPOSIT)
      (LIST 'PUSH-CONSTANT (TAG 'NAT (LR-CONS-TAG)))
      '(PUSH-GLOBAL FREE-PTR)
      '(DEPOSIT)
      '(POP-GLOBAL FREE-PTR)
      '(RET))
```

**Definition**
```
(P-FALSE-CODE )
   =
(LIST 'FALSE '() '()
      (LIST 'PUSH-CONSTANT (LR-F-ADDR))
      '(RET))
```

**Definition**
```
(P-FALSEP-CODE )
   =
(LIST 'FALSEP '() '()
      (LIST 'PUSH-CONSTANT (LR-F-ADDR))
      '(EQ)
```

```
      '(TEST-BOOL-AND-JUMP T TRUE)
      (LIST 'PUSH-CONSTANT (LR-F-ADDR))
      '(RET)
      (LIST 'DL 'TRUE '() (LIST 'PUSH-CONSTANT (LR-T-ADDR)))
      '(RET))
```

**Definition**

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

**Definition**

```
(P-NLISTP-CODE )
   =
(LIST 'NLISTP '() '()
      '(FETCH)
      (LIST 'PUSH-CONSTANT (TAG 'NAT (LR-CONS-TAG)))
      '(EQ)
      '(TEST-BOOL-AND-JUMP F TRUE)
      (LIST 'PUSH-CONSTANT (LR-F-ADDR))
      '(RET)
      (LIST 'DL 'TRUE '() (LIST 'PUSH-CONSTANT (LR-T-ADDR)))
      '(RET))
```

**Definition**

```
(P-TRUE-CODE )
   =
(LIST 'TRUE '() '()
      (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))
```

**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 INSTRS N)
   =
(IF (LISTP INSTRS)
    (CONS (DL (LR-MAKE-LABEL N) () (CAR INSTRS))
          (LABEL-INSTRS (CDR INSTRS) (ADD1 N)))
    NIL)
```

**Definition**

```
(COMP-TEMP-TEST EXPR INSTRS N)
   =
(APPEND (LIST (LIST 'PUSH-LOCAL (CADDR EXPR))
              (LIST 'PUSH-CONSTANT (LR-UNDEF-ADDR))
              '(EQ)
              (LIST 'TEST-BOOL-AND-JUMP 'F
                    (LR-MAKE-LABEL (PLUS N 6
                                         (LENGTH INSTRS)))))
        (APPEND INSTRS
                (LIST (LIST 'SET-LOCAL (CADDR EXPR))
                      (LIST 'JUMP
                            (LR-MAKE-LABEL
                             (PLUS N 7 (LENGTH INSTRS))))
                      (LIST 'PUSH-LOCAL (CADDR EXPR)))))
```

**Definition**

```
(COMP-IF TEST-INSTRS THEN-INSTRS ELSE-INSTRS N)
   =
(APPEND TEST-INSTRS
        (APPEND (LIST (LIST 'PUSH-CONSTANT (LR-F-ADDR))
                      '(EQ)
                      (LIST 'TEST-BOOL-AND-JUMP 'T
                            (LR-MAKE-LABEL (PLUS N 4
                                                 (LENGTH TEST-INSTRS)
                                                 (LENGTH THEN-INSTRS)))))
                (APPEND THEN-INSTRS
                        (CONS (LIST 'JUMP
                                    (LR-MAKE-LABEL
                                     (PLUS N 4
                                           (LENGTH TEST-INSTRS)
                                           (LENGTH THEN-INSTRS)
                                           (LENGTH ELSE-INSTRS))))
                              ELSE-INSTRS))))
```

**Definition**

```
(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 (LR-P-C-SIZE T (CAR EXPR)))))
           NIL))
      ((LISTP EXPR)
       (COND ((EQUAL (CAR EXPR) 'IF)
              (COMP-IF (COMP-BODY-1 T (CADR EXPR) N)
                       (COMP-BODY-1 T
                                    (CADDR EXPR)
                                    (PLUS N 3 (LR-P-C-SIZE T (CADR EXPR))))
                       (COMP-BODY-1 T
                                    (CADDDR EXPR)
                                    (PLUS N 4
                                          (LR-P-C-SIZE T (CADR EXPR))
                                          (LR-P-C-SIZE T (CADDR EXPR))))
                       N))
             ((EQUAL (CAR EXPR) (S-TEMP-FETCH))
              (LIST (LIST 'PUSH-LOCAL (CADDR EXPR))))
             ((EQUAL (CAR EXPR) (S-TEMP-EVAL))
              (APPEND (COMP-BODY-1 T (CADR EXPR) N)
```

```
                      (LIST (LIST 'SET-LOCAL (CADDR EXPR)))))
             ((EQUAL (CAR EXPR) (S-TEMP-TEST))
              (COMP-TEMP-TEST EXPR
                              (COMP-BODY-1 T (CADR EXPR) (PLUS N 4))
                              N))
             ((EQUAL (CAR EXPR) 'QUOTE)
              (LIST (LIST 'PUSH-CONSTANT (CADR EXPR))))
             (T (APPEND (COMP-BODY-1 'LIST (CDR EXPR) N)
                        (IF (DEFINEDP (CAR EXPR)
                              (P-RUNTIME-SUPPORT-PROGRAMS))
                            (LIST (LIST 'CALL (CAR EXPR)))
                            (LIST (LIST 'CALL (USER-FNAME (CAR EXPR))))))))
      (T (LIST (LIST 'PUSH-LOCAL EXPR))))
```

**Definition**

```
(COMP-BODY BODY)
   =
(LABEL-INSTRS (APPEND (COMP-BODY-1 T BODY 0) '((RET))) 0)
```

**Definition**

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

**Definition**

```
(COMP-PROGRAMS PROGRAMS)
   =
(CONS (LR-MAKE-PROGRAM (NAME (CAR PROGRAMS))
                       (FORMAL-VARS (CAR PROGRAMS))
                       (TEMP-VAR-DCLS (CAR PROGRAMS))
                       (LABEL-INSTRS
                        (APPEND (COMP-BODY-1 T
                                             (PROGRAM-BODY (CAR PROGRAMS))
                                             0)
                                (LIST (LIST 'SET-GLOBAL
                                            (AREA-NAME (LR-ANSWER-ADDR)))
                                      '(RET)))
                        0))
      (APPEND (COMP-PROGRAMS-1 (CDR PROGRAMS))
              (P-RUNTIME-SUPPORT-PROGRAMS)))
```

**Definition**

```
(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**

```
(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
# Code for the Specifictions

The following are all the functions (except witness functions) used in the statement of the correctness theorem.

**Definition**
```
(SUBR-ARITY-ALIST )
   =
'((ADD1                     1)  (ADD-TO-SET            2)
  (AND                      2)  (APPEND                2)
  (APPLY-SUBR               2)  (APPLY$                2)
  (ASSOC                    2)  (BODY                  1)
  (CAR                      1)  (CDR                   1)
  (CONS                     2)  (COUNT                 1)
  (DIFFERENCE               2)  (EQUAL                 2)
  (EVAL$                    3)  (FALSE                 0)
  (FALSEP                   1)  (FIX                   1)
  (FIX-COST                 2)  (FOR                   6)
  (FORMALS                  1)  (GEQ                   2)
  (GREATERP                 2)  (IF                    3)
  (IFF                      2)  (IMPLIES               2)
  (LEQ                      2)  (LESSP                 2)
  (LISTP                    1)  (LITATOM               1)
  (MAX                      2)  (MEMBER                2)
  (MINUS                    1)  (NEGATIVEP             1)
  (NEGATIVE-GUTS            1)  (NLISTP                1)
  (NOT                      1)  (NUMBERP               1)
  (OR                       2)  (ORDINALP              1)
  (ORD-LESSP                2)  (PACK                  1)
  (PAIRLIST                 2)  (PLUS                  2)
  (QUANTIFIER-INITIAL-VALUE 1)  (QUANTIFIER-OPERATION  3)
  (QUOTIENT                 2)  (REMAINDER             2)
  (STRIP-CARS               1)  (SUB1                  1)
  (SUBRP                    1)  (SUM-CDRS              1)
  (TIMES                    2)  (TRUE                  0)
  (TRUEP                    1)  (UNION                 2)
  (UNPACK                   1)  (V&C$                  3)
  (V&C-APPLY$               2)  (ZERO                  0)
  (ZEROP                    1))
```

**Definition**
```
(ARITY FUNCTION)
   =
(IF (SUBRP FUNCTION)
    (CADR (ASSOC FUNCTION (SUBR-ARITY-ALIST)))
    (IF (EQUAL FUNCTION 'QUOTE)
        1
        (LENGTH (FORMALS FUNCTION))))
```

**Definition**
```
(L-PROPER-EXPRP FLAG BODY PROGRAM-NAMES FORMALS)
   =
(COND ((EQUAL FLAG 'LIST)
       (IF (LISTP BODY)
           (AND (L-PROPER-EXPRP T (CAR BODY) PROGRAM-NAMES FORMALS)
                (L-PROPER-EXPRP 'LIST (CDR BODY) PROGRAM-NAMES FORMALS))
           (EQUAL BODY NIL)))
      ((LISTP BODY)
       (COND ((NOT (PLISTP BODY)) F)
             ((EQUAL (CAR BODY) 'QUOTE)
              (EQUAL (LENGTH (CDR BODY)) (ARITY (CAR BODY))))
             ((SUBRP (CAR BODY))
              (AND (EQUAL (LENGTH (CDR BODY)) (ARITY (CAR BODY)))
                   (NOT (MEMBER (CAR BODY) PROGRAM-NAMES))
                   (L-PROPER-EXPRP 'LIST
```

```
                                   (CDR BODY)
                                   PROGRAM-NAMES
                                   FORMALS)))
             ((BODY (CAR BODY))
              (AND (EQUAL (LENGTH (CDR BODY)) (ARITY (CAR BODY)))
                   (MEMBER (CAR BODY) PROGRAM-NAMES)
                   (L-PROPER-EXPRP 'LIST
                                   (CDR BODY)
                                   PROGRAM-NAMES
                                   FORMALS)))
             (T F)))
      ((LITATOM BODY) (MEMBER BODY FORMALS))
      (T F))
```

**Definition**

```
(L-PROPER-PROGRAMSP-1 PROGRAM-NAMES ALL-PROGRAM-NAMES)
   =
(IF (LISTP PROGRAM-NAMES)
    (AND (FORMALS (CAR PROGRAM-NAMES))
         (ALL-LITATOMS (FORMALS (CAR PROGRAM-NAMES)))
         (L-PROPER-EXPRP T
                         (BODY (CAR PROGRAM-NAMES))
                         ALL-PROGRAM-NAMES
                         (FORMALS (CAR PROGRAM-NAMES)))
         (L-PROPER-PROGRAMSP-1 (CDR PROGRAM-NAMES) ALL-PROGRAM-NAMES))
    T)
```

**Definition**

```
(L-PROPER-PROGRAMSP PROGRAM-NAMES)
   =
(L-PROPER-PROGRAMSP-1 PROGRAM-NAMES PROGRAM-NAMES)
```

**Definition**

```
(S-RESTRICTED-OBJECTP FLAG OBJECT)
   =
(COND ((EQUAL FLAG 'LIST)
       (IF (LISTP OBJECT)
           (AND (S-RESTRICTED-OBJECTP T (CAR OBJECT))
                (S-RESTRICTED-OBJECTP 'LIST (CDR OBJECT)))
           T))
      ((EQUAL OBJECT T) T)
      ((EQUAL OBJECT F) T)
      ((LISTP OBJECT)
       (S-RESTRICTED-OBJECTP 'LIST (LIST (CAR OBJECT) (CDR OBJECT))))
      ((NUMBERP OBJECT) T)
      (T F))
```

**Definition**

```
(L-DATA-SEG-BODY-RESTRICTEDP FLAG EXPR)
   =
(COND ((EQUAL FLAG 'LIST)
       (IF (LISTP EXPR)
           (AND (L-DATA-SEG-BODY-RESTRICTEDP T (CAR EXPR))
                (L-DATA-SEG-BODY-RESTRICTEDP 'LIST (CDR EXPR)))
           T))
      ((LISTP EXPR)
       (COND ((EQUAL (CAR EXPR) 'QUOTE)
              (S-RESTRICTED-OBJECTP T (CADR EXPR)))
             (T (L-DATA-SEG-BODY-RESTRICTEDP 'LIST (CDR EXPR)))))
      (T T))
```

**Definition**

```
(L-DATA-SEG-LIST-RESTRICTEDP FUN-NAMES)
   =
(IF (LISTP FUN-NAMES)
    (AND (L-DATA-SEG-BODY-RESTRICTEDP T (BODY (CAR FUN-NAMES)))
         (L-DATA-SEG-LIST-RESTRICTEDP (CDR FUN-NAMES)))
    T)
```

**Definition**

```
(S-INIT-DATA-SEG-RESTRICTEDP PARAMS)
```

=
```
(IF (LISTP PARAMS)
    (AND (S-RESTRICTED-OBJECTP T (CDAR PARAMS))
         (S-INIT-DATA-SEG-RESTRICTEDP (CDR PARAMS)))
    T)
```

**Definition**
```
(L-RESTRICTEDP FUN-NAMES ALIST)
```
=
```
(AND (S-INIT-DATA-SEG-RESTRICTEDP ALIST)
     (L-DATA-SEG-LIST-RESTRICTEDP FUN-NAMES))
```

**Definition**

```
(L-RESTRICT-SUBRPS FLAG EXPR)
```
=
```
(COND ((EQUAL FLAG 'LIST)
       (IF (LISTP EXPR)
           (AND (L-RESTRICT-SUBRPS T (CAR EXPR))
                (L-RESTRICT-SUBRPS 'LIST (CDR EXPR)))
           T))
      ((LISTP EXPR)
       (COND ((EQUAL (CAR EXPR) 'QUOTE) T)
             ((EQUAL (CAR EXPR) 'IF)
              (L-RESTRICT-SUBRPS 'LIST (CDR EXPR)))
             ((SUBRP (CAR EXPR))
              (AND (DEFINEDP (CAR EXPR) (P-RUNTIME-SUPPORT-PROGRAMS))
                   (L-RESTRICT-SUBRPS 'LIST (CDR EXPR))))
             ((BODY (CAR EXPR))
              (L-RESTRICT-SUBRPS 'LIST (CDR EXPR)))
             (T T)))
      (T T))
```

**Definition**
```
(L-RESTRICT-SUBRPS-PROGS PNAMES)
```
=
```
(IF (LISTP PNAMES)
    (AND (L-RESTRICT-SUBRPS T (BODY (CAR PNAMES)))
         (L-RESTRICT-SUBRPS-PROGS (CDR PNAMES)))
    T)
```

**Definition**
```
(LR-BOUNDARY-OFFSETP OFFSET)
```
=
```
(EQUAL (REMAINDER OFFSET (LR-NODE-SIZE)) 0)
```

**Definition**
```
(LR-BOUNDARY-NODEP NODE)
```
=
```
(LR-BOUNDARY-OFFSETP (OFFSET NODE))
```

**Definition**
```
(LR-NODEP ADDR DATA-SEG)
```
=
```
(AND (EQUAL (TYPE ADDR) 'ADDR)
     (EQUAL (CDDR ADDR) NIL)
     (LISTP ADDR)
     (ADPP (UNTAG ADDR) DATA-SEG)
     (LR-BOUNDARY-NODEP ADDR)
     (EQUAL (AREA-NAME ADDR) (LR-HEAP-NAME)))
```

**Definition**
```
(LR-GOOD-POINTERP ADDR DATA-SEG)
```
=
```
(AND (LR-NODEP ADDR DATA-SEG)
     (EQUAL (TYPE (FETCH (ADD-ADDR ADDR (LR-REF-COUNT-OFFSET))
                         DATA-SEG))
            'NAT))
```

**Definition**
```
(LR-VALP VALUE ADDR DATA-SEG)
```
=

```
(IF (LR-GOOD-POINTERP ADDR DATA-SEG)
    (LET ((TAG (UNTAG (FETCH ADDR DATA-SEG))))
      (COND ((LISTP VALUE)
             (AND (EQUAL TAG (LR-CONS-TAG))
                  (LR-VALP (CAR VALUE)
                           (FETCH (ADD-ADDR ADDR (LR-CAR-OFFSET)) DATA-SEG)
                           DATA-SEG)
                  (LR-VALP (CDR VALUE)
                           (FETCH (ADD-ADDR ADDR (LR-CDR-OFFSET)) DATA-SEG)
                           DATA-SEG)))
            ((TRUEP VALUE) (EQUAL TAG (LR-TRUE-TAG)))
            ((FALSEP VALUE) (EQUAL TAG (LR-FALSE-TAG)))
            ((NUMBERP VALUE)
             (AND (EQUAL TAG (LR-ADD1-TAG))
                  (EQUAL VALUE
                         (UNTAG (FETCH (ADD-ADDR ADDR
                                                 (LR-UNBOX-NAT-OFFSET))
                                       DATA-SEG)))))
            (T F)))
    F)
```

# Table of Contents

List of Figures

List of Tables