1 Introduction

SECD has a long and honourable pedigree. It was designed as an abstract architecture by Landin ([Lan64], [Lan65a], [Lan65b], [Lan66]) in the 1960's when he was seeking a way to give an operational semantics for ALGOL 60. [Bur75], [Hen80], and [FH88] [1], explain the workings of eager and lazy SECD machines. [FH88] also explains how to make lazy evaluation efficient (call by need) and sketches Plotkin's proof ([Plo75]) of the correctness of an eager SECD machine.

We argued that SECD would fit into our long term plans rather well.

1. the architecture is straightforward with few instructions.
2. it was already proven that SECD executes λ correctly.
3. the transformation from a functional language into λ is both straightforward and amenable to proof.
4. purely functional languages are powerful and succinct. It is usually (always?) much easier to verify the same algorithm written in functional style than in imperative style.

At the time of our decision (1985), we were very much influenced by Henderson's text [Hen80] and by the work being done in functional style by Henderson's group at Oxford [HGAJ83a], [HGAJ83b]. The obvious route to us seemed to be use Lispkit code running on an SECD machine. (Both Lispkit and our SECD architecture are eager.) If we were to start now, our basic argument would remain the same but we would go for a better language with strong typing (Miranda or Hope) running on a better machine (a lazy combinator architecture).

The rest of this paper is devoted to sketching the Lispkit source language and its semantics explaining its translation schema through examples, and then commenting on the source listing of the compiler. The compiler comes with a workbench, a sort of mini–environment. The source for the compiler is given in the appendix of this paper. Examples of the workbench are in [HBGS89].

[1] Note that each describes a slightly different SECD machine.

2 S–expressions

Henderson uses S–expressions pervasively not only as the concrete representation of Lispkit source programs, but also to describe interpreters, compilers, and the SECD architecture. S–expressions are either *atoms* or *dotted pairs* of S–expressions:

S–expr ::= atom | (S–expr . S–expr)

The term *list* refers to an S–expression whose farthest right projection is the atom NIL.

- atoms are of two types: numeric and symbolic. A *numeric atom* is a (possibly) signed sequence of digits which we interpret as an integer. *Symbolic atoms* are further subdivided into constants and variables. They appear as a series of letters or digits (at least one letter is required to distinguish a symbolic atom from a numeric atom) or other characters. Three special symbolic atoms NIL, T, and F have particular meanings permanently associated with them. NIL is interpreted as the empty list; T as true; and F as false.
- a dotted pair results from carrying out a (Lisp) *cons* operation on two S–expressions. If *a* and *b* are S–expressions, then *(cons a b)* produces a dotted pair *(a . b)*. Henderson replaces the dot notation by the list form where possible in his text [Hen80]. The rules of transformation are simple
  - (a . NIL) may be written as (a)
  - (a . (b)) may be written as (a b)

  where a and b are S–expressions.

3 The Lispkit source

Lispkit source is introduced and well explained in [Hen80], pages 93–110. Our variant upon the SECD theme is covered in *op. cit.* pages 167–176, and a Lispkit compiler for SECD is developed in *op. cit.* pages 176–196. Several substantial programming efforts have been carried out in Lispkit so it should not be regarded as merely a toy language, see for example [HGAJ83b] which lists compilers for Lispkit and μFP written in Lispkit.

Lispkit is a purely functional variant of the Lisp language (it is a syntactically sugared λ calculus with a little typing). By 'purely functional' we mean that Lispkit has no destructive assignment operation, so that the value of an expression is uniquely determined by the value of its constituents and identical expressions always have the same value. No destructive assignments implies no side–effects such as assigments to global variables. Hence, with the addition of strong tying, Lispkit programs would be comparatively easy to verify.

Before detailing the concrete representation of Lispkit programs we give a flavour of its notation and power through some simple examples.

- Lispkit uses the prefix notation. The mathematical

  $$x * (y + 1)$$

  becomes

  ```
  (MUL x (ADD y (QUOTE 1))
  ```

  in Lispkit. All Lispkit constants are quoted. QUOTE, ADD, DIV, MUL, REM and SUB are built–in functions.

- Lispkit has an IF–expression built–in. The mathematical

  $$if\ a < b\ then\ a\ else\ b$$

  becomes

  ```
  (IF (LEQ a b) a b)
  ```

  in Lispkit. EQ and LEQ are built–in comparators. EQ compares atoms, whether numeric or not; LEQ only accepts numeric arguments.

- Lispkit models its function definitions and calls on the *LET* notation popularised by Landin in the 1960's writing

  let definiendum = expr **in** body

  instead of

  (λ definiendum . body) expr

  Lispkit follows this lead but permits several simultaneous local definitions may be made in *LET* and in *LETREC* definitions. Here is the general template

  (LET body (def_1.expr_1) ... (def_k.expr_k))

  Thus a definition of the function

  let t = x^2 + 3
  in t - (t / 2)

  in Landin's notation becomes the Lispkit

```
(LET (SUB t (DIV t (QUOTE 2)))
  ( t . (ADD (MUL x x) (QUOTE 3))))
```

  LETs are evaluated eagerly. Each *expr* is evaluated and assigned to its *definiendum* which is then added to the environment. *body* is evaluated in this augmented environment.

- Lispkit requires us to use *LETREC* when a definition is recursive. Here is a recursive definition of *append* which joins two lists together, followed by a call and its result:

```
(LETREC append
    (append LAMBDA (x y)
        (IF (EQ x (QUOTE NIL))
            y
            (CONS (CAR x) (append (CDR x) y)
            )
        )
    )
)
```

=> (append QUOTE((1 2 3) (3 4 (5))))

=> (1 2 3 3 4 (5))

append is a function requiring a pair of arguments *(x y)*. The body of append tests to see if the first argument is the null list *(EQ x (QUOTE NIL))* – and if so the second argument *y* is returned. If *x* is not null, we *CONS* the head of *x* (*CAR x*) on to the result of appending the tail of *x* (*CDR x*) to *y*. The definition works becuase the first argument to the inner call is 'shorter' each time.

The structure of the *LETREC* definition is the same as that of the *LET* construction

$$(\text{LET body } (\text{def}_1.\lambda\text{expr}_1) \ldots (\text{def}_k.\lambda\text{expr}_k))$$

but the local definitions are restricted to being functions with λ expressions as bodies. This simplifies the run time organisation of SECD.

Matching the append program to this template, the single local definition is interpreted as defin(iendum)ing a function *append* whose scope lies solely within the LETREC. *body* is the expression to be evaluated. In this case it is just the append function itself, but it could be more general (e.g. (CAR append)).

4 Concrete Lispkit

Concrete Lispkit programs are S–expressions. The basic types supported by Lispkit are the integers *int*, the booleans *bool*, and lists. The set of allowable S–expressions is given in table 1. For notational convenience, expressions are subscripted e's, atoms are subscripted x's. For extra clarity, we use f to denote an expression evaluating to a function and *body* (also an expression) to denote the body of lambda, let and letrec expressions.

x	bound variable*
(QUOTE s)	constant
(ADD e_1 e_2)	arithmetic expressions
(DIV e_1 e_2)	
(MUL e_1 e_2)	
(REM e_1 e_2)	
(SUB e_1 e_2)	
(EQ e_1 e_2)	relational expressions
(LEQ e_1 e_2)	
(ATOM e)	list expressions
(CAR e)	
(CDR e)	
(CONS e_1 e_2)	
(IF e_1 e_2 e_2)	conditional form
(f e_1 ...e_k)	function call
(LAMBDA (x_1...x_k) body)	lambda expression
(LET body (x_1.e_1)...(x_k.e_k))	simple block
(LETREC body (x_1.e_1)...(x_k.e_k))	recursive block

Table 1: Lispkit source

*A bound variable is the binding for an argument to a λ expression or a local definition bound on entry to a LET or a LETREC – it cannot be assigned to.

- **variables** occur bound in λ, *let*, and *letrec* expressions. When their definitions are called, their associated argument (λ) or local definition (either form *LET(REC)*) is evaluated eagerly and bound to the new variable (a slot is added to the environment). The variable and the binding are deleted from the environment when we exit from its defining λ, *let*, or *letrec* expression instance. It is not possible to alter this binding since assignments are not defined in Lispkit.

- **constants** are always quoted – there is no exception to this rule. Hence

  127, NIL, (Error message***)

  are represented in Lispkit by

  (QUOTE 127), (QUOTE NIL), (QUOTE (Error message***))

  respectively.

- **operators** are defined as prefix not infix. For example, we write (5 + 3) as

  (ADD (QUOTE 5) (QUOTE 3))

  Each operator has its own specific keyword. Keywords have fixed meanings in Lispkit programs and may NOT be redefined. Here is a table of the 12 built–in operators in alphabetical order:

Lispkit notation	mathematical notation	argument(s)	result
(ADD e_1 e_2)	$e_1 + e_2$	int × int	int
(ATOM e)	is_atom(e)	s–exp	bool
(CAR e)	car(e)	dotted pair	s–exp
(CDR e)	cdr(e)	dotted pair	s–exp
(CONS e_1 e_2)	cons(e_1, e_2)	s–exp × s–exp	s–exp
(DIV e_1 e_2)	e_1 / e_2	int × int	int
(EQ e_1 e_2)	$e_1 = e_2$	s–exp × s–exp	bool
(LEQ e_1 e_2)	$e_1 \leq e_2$	s–exp × s–exp	bool
(MUL e_1 e_2)	e_1 * e_2	int × int	int
(QUOTE s)	s	S–exp	constant
(REM e_1 e_2)	e_1 **rem** e_2	int × int	int
(SUB e_1 e_2)	e_1 - e_2	int × int	int

Table 2: Built–in Lispkit functions

- **IF–expressions** take the form

  (IF e_1 e_2 e_3)

  in Lispkit, i.e. a 4–list whose first item is the keyword *IF*. The IF–expression is introduced as a separate construct because we want to execute e_1 and only one of e_2 and e_3 – a call by value interpretation would compute all three arguments. The mathematical

  if $x \leq y$ *then -1 else if* $x = y$ *then 0 else 1*

  would be written

```
(IF (LEQ x y) (QUOTE -1)
   (IF (EQ x y) (QUOTE 0) (QUOTE 1)))
```

  in Lispkit.

- **LAMBDA expressions** of the form

  $\lambda\ x_1\ x_2\ \ldots\ x_k\ .$ body

  have the concrete syntax

  (LAMBDA (x_1 x_2 ... x_k) body)

  i.e. a 3–list consisting of the keyword *LAMBDA* followed by the expression used to compute the value of the function followed by a list of the argument names. The mathematical

  $\lambda\ L\ M$. *if null L then 0 else if null M then 0 else 1*

  is written

```
(LAMBDA (L M)
  (IF (EQ (QUOTE NIL) L) (QUOTE 0)
    (IF (EQ (QUOTE NIL) M) (QUOTE 0) (QUOTE 1))))
```

  in Lispkit.

- **function calls** form a list consisting of the function name followed by the arguments. For example

  (append (CDR x) y)

  Notice that functions are defined in *LET*s and in *LETREC*s by binding their names to lambda definitions, so that given the (informal) definition

  f = λ (args) . body

  we store away the translated code for

  λ (args) . body

  at the look–up position for *f* in the environment. A call *(f x y)* on such a two–argument function can thus be treated as though it were

  (λ (args) . body) x y

- **LET expressions** are used to give non–recursive local definitions – they are *not* assignments. A LET definition enables us to do a computation once, save the result in an augmented environment, and use the saved value several times. For example, instead of writing

  $$x*x*x + x*x*x + x*x*x$$

  we may write

  let *trip* = $x*x*x$ **in** *trip* + *trip* + *trip*

  Several let definitions may be made at once. So the mathematical

  let $x_1 = e_1$ **and** $x_2 = e_2$ **and** ... **and** $x_k = e_k$ **in** *body*

  with interpretation

  (λ x_1 x_2 ... x_k . body) e_1 e_k ... e_k

is rendered

$$(\text{LET body } (x_1.e_1)...(x_k.e_k))$$

in Lispkit – a list of arbitary length (at least 2) consisting of the keyword *LET* followed by the computation rule followed by a list of local definitions. Each definition appears as a *dotted pair*.

- **LETREC expressions** are used to give recursive local definitions – they are *not* assignments. They have the same structure as LET

$$(\text{LETREC body } (x_1.e_1)...(x_k.e_k))$$

but the expressions in their local definitions are restricted to being LAMBDA expressions. We have already seen a definition of *append*. Here we illustrate the structure of *LETREC* definitions with a more complicated example, *even*, which uses recursion via the auxiliary definition of *odd*.

```
(LETREC even
   (even LAMBDA (n)
     (IF (EQ n (QUOTE 0)) (QUOTE T) (odd (SUB n (QUOTE 1)))))
   (odd LAMBDA (n)
     (IF (EQ n (QUOTE 0)) (QUOTE F) (even (SUB n (QUOTE 1)))))
)
```

The body consists of a single call on *even*. When we enter the LETREC, we add two definitions to the environment of the *LETREC* expression. These are the definition of *even* (corresponding to $(x_1.e_1)$ in the definition template) and the definition of *odd* (corresponding to $(x_2.e_2)$ in the definition template). The body is to be evaluated in an extended environment which holds these local definitions.

NB. As remarked above, Lispkit restricts local definitions in LETREC blocks to to definitions of functions with the form

$$(\text{name (LAMBDA definition)})$$

This enables a simple approach to implementing recursion effectively. If a computation requires local definitions of f, g, h and j, and f and j are neither functions nor recursive, then they can be lifted from the LETREC structure into an embedded LET as shown below. Smart optimising compilers would look for and effect this transformation, but ours does not.

(LETREC
  (LET body (f . $expr_f$) (j . $expr_j$))
    (h . $expr_h$)
    (g . $expr_g$)
))

5 The denotational semantics of Lispkit

$$
\begin{array}{lcl}
\mathcal{E}[\![\ \mathrm{x}\ ]\!]\sigma & = & \sigma(\mathrm{x}) \\
\mathcal{E}[\![\ (\mathrm{QUOTE\ s})\ ]\!]\sigma & = & \mathrm{s} \\
\mathcal{E}[\![\ (\mathrm{ADD\ e_1\ e_2})\ ]\!]\sigma & = & \neg\ \mathrm{is\_num}\ (\mathcal{E}[\![\ \mathrm{e_1}\ ]\!]\sigma) \rightarrow \mathrm{error}\ | \\
 & & \neg\ \mathrm{is\_num}\ (\mathcal{E}[\![\ \mathrm{e_2}\ ]\!]\sigma) \rightarrow \mathrm{error}\ | \\
 & & \mathcal{E}[\![\ \mathrm{e_1}\ ]\!]\sigma + \mathcal{E}[\![\ \mathrm{e_2}\ ]\!]\sigma \\
\mathcal{E}[\![\ (\mathrm{ATOM\ e})\ ]\!]\sigma & = & \mathbf{is\_atom}\ (\mathcal{E}[\![\ \mathrm{e}\ ]\!]\sigma) \\
\mathcal{E}[\![\ (\mathrm{CAR\ e})\ ]\!]\sigma & = & \neg\ \mathrm{is\_cons}\ (\mathcal{E}[\![\ \mathrm{e}\ ]\!]\sigma) \rightarrow \mathrm{error}\ | \\
 & & \mathrm{hd}(\mathcal{E}[\![\ \mathrm{e}\ ]\!]\sigma) \\
\mathcal{E}[\![\ (\mathrm{CDR\ e})\ ]\!]\sigma & = & \neg\ \mathrm{is\_cons}\ (\mathcal{E}[\![\ \mathrm{e}\ ]\!]\sigma) \rightarrow \mathrm{error}\ | \\
 & & \mathrm{tl}\ (\mathcal{E}[\![\ \mathrm{e}\ ]\!]\sigma) \\
\mathcal{E}[\![\ (\mathrm{CONS\ e_1\ e_2})\ ]\!]\sigma & = & \neg\ \mathrm{is\_atom}\ (\mathcal{E}[\![\ \mathrm{e_1}\ ]\!]\sigma) \rightarrow \mathrm{error}\ | \\
 & & \neg\ \mathrm{is\_cons}\ (\mathcal{E}[\![\ \mathrm{e_2}\ ]\!]\sigma) \rightarrow \mathrm{error}\ | \\
 & & \mathbf{cons}\ (\mathcal{E}[\![\ \mathrm{e_1}\ ]\!]\sigma, \mathcal{E}[\![\ \mathrm{e_2}\ ]\!]\sigma) \\
\mathcal{E}[\![\ (\mathrm{DIV\ e_1\ e_2})\ ]\!]\sigma & = & \neg\ \mathrm{is\_num}\ (\mathcal{E}[\![\ \mathrm{e_1}\ ]\!]\sigma) \rightarrow \mathrm{error}\ | \\
 & & \neg\ \mathrm{is\_num}\ (\mathcal{E}[\![\ \mathrm{e_2}\ ]\!]\sigma) \rightarrow \mathrm{error}\ | \\
 & & (\mathcal{E}[\![\ \mathrm{e_2}\ ]\!]\sigma = 0) \rightarrow \mathrm{error}\ | \\
 & & \mathcal{E}[\![\ \mathrm{e_1}\ ]\!]\sigma\ /\ \mathcal{E}[\![\ \mathrm{e_2}\ ]\!]\sigma) \\
\mathcal{E}[\![\ (\mathrm{EQ\ e_1\ e_2})\ ]\!]\sigma & = & \mathrm{is\_atom}\ (\mathcal{E}[\![\ \mathrm{e_1}\ ]\!]\sigma) \wedge \mathrm{is\_atom}\ (\mathcal{E}[\![\ \mathrm{e_2}\ ]\!]\sigma) \\
 & & \wedge\ (\mathcal{E}[\![\ \mathrm{e_1}\ ]\!]\sigma = \mathcal{E}[\![\ \mathrm{e_2}\ ]\!]\sigma) \\
\mathcal{E}[\![\ (\mathrm{LEQ\ e_1\ e_2})\ ]\!]\sigma & = & \neg\ \mathrm{is\_num}\ (\mathcal{E}[\![\ \mathrm{e_1}\ ]\!]\sigma) \rightarrow \mathrm{error}\ | \\
 & & \neg\ \mathrm{is\_num}\ (\mathcal{E}[\![\ \mathrm{e_2}\ ]\!]\sigma) \rightarrow \mathrm{error}\ | \\
 & & (\mathcal{E}[\![\ \mathrm{e_1}\ ]\!]\sigma \leq \mathcal{E}[\![\ \mathrm{e_2}\ ]\!]\sigma) \\
\mathcal{E}[\![\ (\mathrm{MUL\ e_1\ e_2})\ ]\!]\sigma & = & \neg\ \mathrm{is\_num}\ (\mathcal{E}[\![\ \mathrm{e_1}\ ]\!]\sigma) \rightarrow \mathrm{error}\ | \\
 & & \neg\ \mathrm{is\_num}\ (\mathcal{E}[\![\ \mathrm{e_2}\ ]\!]\sigma) \rightarrow \mathrm{error}\ | \\
 & & \mathcal{E}[\![\ \mathrm{e_1}\ ]\!]\sigma * \mathcal{E}[\![\ \mathrm{e_2}\ ]\!]\sigma \\
\mathcal{E}[\![\ (\mathrm{REM\ e_1\ e_2})\ ]\!]\sigma & = & \neg\ \mathrm{is\_num}\ (\mathcal{E}[\![\ \mathrm{e_1}\ ]\!]\sigma) \rightarrow \mathrm{error}\ | \\
 & & \neg\ \mathrm{is\_num}\ (\mathcal{E}[\![\ \mathrm{e_2}\ ]\!]\sigma) \rightarrow \mathrm{error}\ | \\
 & & \mathcal{E}[\![\ \mathrm{e_1}\ ]\!]\sigma\ \mathbf{rem}\ \mathcal{E}[\![\ \mathrm{e_2}\ ]\!]\sigma \\
\mathcal{E}[\![\ (\mathrm{SUB\ e_1\ e_2})\ ]\!]\sigma & = & \neg\ \mathrm{is\_num}\ (\mathcal{E}[\![\ \mathrm{e_1}\ ]\!]\sigma) \rightarrow \mathrm{error}\ | \\
 & & \neg\ \mathrm{is\_num}\ (\mathcal{E}[\![\ \mathrm{e_2}\ ]\!]\sigma) \rightarrow \mathrm{error}\ | \\
 & & \mathcal{E}[\![\ \mathrm{e_1}\ ]\!]\sigma - \mathcal{E}[\![\ \mathrm{e_2}\ ]\!]\sigma \\
\mathcal{E}[\![\ (\mathrm{IF\ e_1\ e_2\ e_3}\ )\ ]\!]\sigma & = & \neg\ \mathrm{is\_bool}\ (\mathcal{E}[\![\ \mathrm{e_1}\ ]\!]\sigma) \rightarrow \mathrm{error}\ | \\
 & & \neg\ \mathcal{E}[\![\ \mathrm{e_1}\ ]\!]\sigma \rightarrow \mathcal{E}[\![\ \mathrm{e_3}\ ]\!]\sigma\ | \\
 & & \mathcal{E}[\![\ \mathrm{e_2}\ ]\!]\sigma
\end{array}
$$

Table 3: Denotational semantics for Lispkit: 1

$\mathcal{E}$ ⟦ (f e_1 ... e_k) ⟧σ	=	¬ arity f = k → error | $\mathcal{E}$ ⟦ f ⟧σ $\mathcal{E}$ ⟦ e_1 ⟧σ ... $\mathcal{E}$ ⟦ e_k ⟧σ
$\mathcal{E}$ ⟦ (LAMBDA (x_1...x_k) body) ⟧σ	=	λ (v_1...v_k) . $\mathcal{E}$ ⟦ body) ⟧(σ[v_1/x_1 ... v_k/x_k])
$\mathcal{E}$ ⟦ (LET body (x_1.e_1)... (x_k.e_k)) ⟧σ	=	$\mathcal{E}$ ⟦ (LAMBDA (x_1...x_k) body) ⟧σ $\mathcal{E}$ ⟦ e_1 ⟧σ ... $\mathcal{E}$ ⟦ e_k ⟧σ
$\mathcal{E}$ ⟦ (LETREC body (x_1.e_1)... (x_k.e_k)) ⟧σ	=	$\mathcal{E}$ ⟦ LET body (x_1,...,x_k) = Y (λ x_1.e_1,..., λ x_k.e_k) ⟧σ

Table 4: Denotational semantics for Lispkit: 2

The semantics describe the transitions that take place when we carry out a specific code segment. We refer to the starting state for each transition as the initial state and denote it by σ.

- **variables.**

  An identifier is look'ed–up in the current environment σ. As a program is compiled, the translator builds up a name environment in exactly the same way it will build up a value environment at run time, i.e. as a list of incremental environments, each of which is also a list. The look–up table at compile time may contain such properties as the name of the variable, its type, the number of arguments (if a function), and its line of declaration (for good error messages). The look–up rule used to see if a variable x belongs to a specific level l of the environment is

  ```
  member x l = if NULL l then false else
               if x = (CAR l) then true else member x (CDR l)
  ```

  and the look–up rule applied to see if a variable x belongs to the whole environment L is

  ```
  MEMBER x L = if NULL L then false else
               member x (CAR L) ∨ MEMBER x (CDR L)
  ```

  The first occurrence of x that is located is the one taken. This will always be the most current since additions to the operating environment are always made at the front. MEMBER returns false if x is not defined, in which case the program will not compile.

  We can extend *member* to return a pair *(bool* × *int)* letting the second value returned denote the depth of the match within l when a match for x is found. We can also extend *MEMBER* to return a pair *(bool* × *(int* × *int))* letting the second value returned denote the index pair *(m.n)* of the match within L when a match for x is found.

- **constants.**

  The value of a QUOTEd item is the value of the item itself.

- **binary operators.**

  To evaluate a binary expression in environment σ, we first evaluate e_1 in σ, then we evaluate e_2 in σ, then we *op* them together. The binary operators are ADD, DIV, MUL, REM, SUB, and CONS. Notice that each of these operations may fail to type check. The compiler makes sure that each of these functions is called with two arguments. We ensure that the arguments to the 5 arithmetic operations (ADD, DIV, MUL, REM, and SUB) are each of type *int* and that the arguments to CONS are first an atom and then a list. We do NOT check that the atom and the list elements are of the same type.

- **unary operators.**

  To evaluate a unary expression in environment σ, we first evaluate e in σ, then we apply *op* to it. The unary operators are ATOM, CAR, and CDR. Notice that each of these operations may fail if the arguments are not consistent. The compiler makes sure that each of these functions is called with one argument. We ensure that the argument to the CAR and CDR operations are non–null lists.

- **IF–expressions.** IF–expressions cause some trouble in eager systems. If we just treat IF as a given system function defined in terms of LET, then in order to be consistent, we should evaluate all its arguments eagerly before we apply the IF operation. This is precisely what we want to avoid. So we build–in a special operator with 'better' semantics. Our definition states quite explicitly, that we first evaluate e_1. If it returns false, we evaluate and return e_3. Otherwise whatever e_1 evaluates to, we evaluate and return e_2. Either way we evaluate only one arm of the IF–expression. The compiler will type check that e_1 is of type *bool*. It does not check that e_2 and e_3 have the same type.

- **function calls.** For a function call, the definition of f is looked–up in the initial environment (its body must be a LAMBDA definition). Its arguments are evaluated with respect to the initial environment. The value of f will be applied to these arguments. The compiler checks to see that the number of arguments supplied equals the number expected and raises an error if not. See also LAMBDA definitions below.

- **LAMBDA definitions.** A λ definition will be applied to arguments which have already been evaluated and then performs textual substitution. The number of arguments to a lambda definition is saved by the compiler. Every Lispkit function body has to be a lambda definition. When a lambda expression is applied, the compiler checks to see that the

number of arguments supplied equals the number expected and raises an error if not. See also function calls above.

- **LETs and LETRECs.** LET(REC) expressions are syntactically sugared λ's.

  The semantics for LET here and LAMBDA (above) are equivalent. Both evaluate the arguments in the original context then do a β conversion. The classic definition for LET in terms of λ is

  $$\textbf{let } v = expr \textbf{ in } body \equiv (\lambda\ v\ .\ body\ )\ expr$$

  In our case, both evaluate the arguments prior to the substition in the body.

  LETREC has the classic interpretation (for a single binding) of

  $$\textbf{letrec } v = expr \textbf{ in } body \equiv \textbf{let } v = \mathbf{Y}(\lambda v.expr) \textbf{ in } body$$

  and is much harder to explain. Since the SECD machine implements recursion directly without using the Y operator, we present a sufficient surface explanation of its semantics without using Y. The body is evaluated in an environment augmented by a definition for each clause bound by the LETREC, just like a LET. The difference is that each clause is evaluated in the augmented environment as well. Thus, e_j will be evaluated in an environment where each of $x_1,...,x_k$ is already defined. The difficulty that x_j must be defined before e_j is evaluated is handled by the Y operator. For an indepth description of how Y works, see [FH88].

6 The translation schema

The translation schema from Lispkit to SECD machine code is given in table 5.

We use the notation $\mathcal{T}^{\sigma}$ *exp* to denote the translation of *exp* with respect to the environment σ. $\mathcal{T}^{\sigma+}$ *exp* denotes the translation of *exp* with respect to the an augmented environment $\sigma+$consisting of the bindings associated with x_1 ... x_k ... evaluated in $\sigma+$ appended onto σ.

The SECD code and architecture are given in a companion report [HBGS89], or else see [Hen80], pages 163–176.

Lispkit source	SECD code
$\mathcal{T}^{\sigma}$x	(LD (m.n)) where (m.n) is the position of x in σ
$\mathcal{T}^{\sigma}$(QUOTE s)	(LDC s)
$\mathcal{T}^{\sigma}$(ADD e_1 e_2)	$\mathcal{T}^{\sigma}e_1$ $\mathcal{T}^{\sigma}e_2$ (ADD)
$\mathcal{T}^{\sigma}$(ATOM e)	$\mathcal{T}^{\sigma}$e (ATOM)
$\mathcal{T}^{\sigma}$(CAR e)	$\mathcal{T}^{\sigma}$e (CAR)
$\mathcal{T}^{\sigma}$(CDR e)	$\mathcal{T}^{\sigma}$e (CDR)
$\mathcal{T}^{\sigma}$(CONS e_1 e_2)	$\mathcal{T}^{\sigma}e_2$ $\mathcal{T}^{\sigma}e_1$ (CONS)
$\mathcal{T}^{\sigma}$(DIV e_1 e_2)	$\mathcal{T}^{\sigma}e_1$ $\mathcal{T}^{\sigma}e_2$ (DIV)
$\mathcal{T}^{\sigma}$(EQ e_1 e_2)	$\mathcal{T}^{\sigma}e_1$ $\mathcal{T}^{\sigma}e_2$ (EQ)
$\mathcal{T}^{\sigma}$(LEQ e_1 e_2)	$\mathcal{T}^{\sigma}e_1$ $\mathcal{T}^{\sigma}e_2$ (LEQ)
$\mathcal{T}^{\sigma}$(MUL e_1 e_2)	$\mathcal{T}^{\sigma}e_1$ $\mathcal{T}^{\sigma}e_2$ (MUL)
$\mathcal{T}^{\sigma}$(REM e_1 e_2)	$\mathcal{T}^{\sigma}e_1$ $\mathcal{T}^{\sigma}e_2$ (REM)
$\mathcal{T}^{\sigma}$(SUB e_1 e_2)	$\mathcal{T}^{\sigma}e_1$ $\mathcal{T}^{\sigma}e_2$ (SUB)
$\mathcal{T}^{\sigma}$(IF e_1 e_2 e_3)	$\mathcal{T}^{\sigma}e_1$ (SEL $\mathcal{T}^{\sigma}e_2$ (JOIN) $\mathcal{T}^{\sigma}e_3$ (JOIN))
$\mathcal{T}^{\sigma}$(LAMBDA (x_1...x_k) body)	(LDF $\mathcal{T}^{\sigma+}$ body (RTN))
$\mathcal{T}^{\sigma}$(f e_1 ...e_k)	(LDC NIL) $\mathcal{T}^{\sigma}e_k$ (CONS) ... (CONS) $\mathcal{T}^{\sigma}e_1$(CONS) $\mathcal{T}^{\sigma}$f (AP)
$\mathcal{T}^{\sigma}$(LET body (x_1.e_1)...(x_k.e_k))	(LDC NIL) $\mathcal{T}^{\sigma}e_k$ (CONS) ... $\mathcal{T}^{\sigma}e_1$ (CONS) (LDF $\mathcal{T}^{\sigma+}$ body (RTN) AP)
$\mathcal{T}^{\sigma}$(LETREC body (x_1.e_1)...(x_k.e_k))	(DUM LDC nil) $\mathcal{T}^{\sigma+}$ e_k (CONS) ... $\mathcal{T}^{\sigma+}$ e_1 (CONS) (LDF $\mathcal{T}^{\sigma+}$ body (RTN) RAP)

Table 5: Lispkit translation schema

- **variables.**
  $\mathcal{T}^{\sigma}$x $\rightarrow$ (LD (m.n))

  The eager SECD machine has static scoping. We follow Henderson and assume that the SECD machine saves values associated with definitions in an environment whose overall shape is that of a list of lists. The environment is accessed through the LD instruction which has as argument a pair of integers. The sublists of the environment (the *m*'s) are numbered 0, 1, 2, ... and each element of each sublist (the *n*'s) is also numbered 0, 1, 2, ... (but independently). The index pair *(m.n)* selects the nth pair from the mth sublist.

  The compiler will see to it that only legal pairs will be looked up. Assuming the environment is named σ the look–up algorithm is

  $$(\text{car } (\text{cdr}^{n} \ (\text{car } (\text{cdr}^{m} \ \sigma)))$$

  As an example, table 6 shows the environment on entry of the *body* when we execute the Lispkit program

```
(LET
   (LET body
      (u . (QUOTE 1)))
      (x . (QUOTE 2))
      (y . (QUOTE (A B)))
      (z . (LAMBDA (x) (QUOTE 0))))
```

  corresponding to the mathematical

  let x = 2 **and** y = (A B) **and** zero x = 0 **in**
  let u = 1 **in**
  body

  Within the body, the the values in the environment can be selected by using index pairs u = (0.0), x = (1.0), y = (1.1), and z = (1.2).

indices	value returned
(0.0)	1
(1.0)	2
(1.1)	(A B)
(1.2)	(LDC NIL LDF (LDC 0 RTN) CONS LDF (LD (0 . 0) RTN) AP)

Table 6: Look–up example

- **constants.**
  $\mathcal{T}^{\sigma}$(QUOTE s) $\rightarrow$ (LDC s)

  Constants appear directly in line.

- **binary operators.**
  $\mathcal{T}^{\sigma}$(op e_1 e_2) $\rightarrow$ $\mathcal{T}^{\sigma}e_1$ $\mathcal{T}^{\sigma}e_2$ (*op*)

  Lispkit source expressions are written in prefix form, that is the operator appears before its argument(s). The SECD machine uses its S–stack to evaluate expressions. Hence the code generated for SECD is argument(s) first, then the operator. The order of the arguments is preserved *except* for *CONS* which (faithfully following Henderson) rotates them. The translation schema merely states that the translation of an operator followed by two operands is simply the translation of the first operand, the translation of the second operation, followed by the (SECD) representation of the operator.

$$
\begin{array}{lcl}
\mathcal{T}^{\sigma}(\text{ADD } e_1\ e_2) & \rightarrow & \mathcal{T}^{\sigma}e_1\ \mathcal{T}^{\sigma}e_2\ (\text{ADD}) \\
\mathcal{T}^{\sigma}(\text{CONS } e_1\ e_2) & \rightarrow & \mathcal{T}^{\sigma}e_2\ \mathcal{T}^{\sigma}e_1\ (\text{CONS}) \\
\mathcal{T}^{\sigma}(\text{DIV } e_1\ e_2) & \rightarrow & \mathcal{T}^{\sigma}e_1\ \mathcal{T}^{\sigma}e_2\ (\text{DIV}) \\
\mathcal{T}^{\sigma}(\text{EQ } e_1\ e_2) & \rightarrow & \mathcal{T}^{\sigma}e_1\ \mathcal{T}^{\sigma}e_2\ (\text{EQ}) \\
\mathcal{T}^{\sigma}(\text{LEQ } e_1\ e_2) & \rightarrow & \mathcal{T}^{\sigma}e_1\ \mathcal{T}^{\sigma}e_2\ (\text{LEQ}) \\
\mathcal{T}^{\sigma}(\text{MUL } e_1\ e_2) & \rightarrow & \mathcal{T}^{\sigma}e_1\ \mathcal{T}^{\sigma}e_2\ (\text{MUL}) \\
\mathcal{T}^{\sigma}(\text{REM } e_1\ e_2) & \rightarrow & \mathcal{T}^{\sigma}e_1\ \mathcal{T}^{\sigma}e_2\ (\text{REM}) \\
\mathcal{T}^{\sigma}(\text{SUB } e_1\ e_2) & \rightarrow & \mathcal{T}^{\sigma}e_1\ \mathcal{T}^{\sigma}e_2\ (\text{SUB})
\end{array}
$$

- **unary operators.**
  $\mathcal{T}^{\sigma}$(op e) $\rightarrow$ $\mathcal{T}^{\sigma}$e (*op*)

  There are three unary operators, each concerned with list manipulations. The translation style is too similar to that of binary operators to require comment.

$$
\begin{array}{lcl}
\mathcal{T}^{\sigma}(\text{ATOM e}) & \rightarrow & \mathcal{T}^{\sigma}\text{e (ATOM)} \\
\mathcal{T}^{\sigma}(\text{CAR e}) & \rightarrow & \mathcal{T}^{\sigma}\text{e (CAR)} \\
\mathcal{T}^{\sigma}(\text{CDR e}) & \rightarrow & \mathcal{T}^{\sigma}\text{e (CDR)}
\end{array}
$$

- **IF–expressions.**
  $\mathcal{T}^{\sigma}$(IF e_1 e_2 e_3) $\rightarrow$ $\mathcal{T}^{\sigma}e_1$ (SEL $\mathcal{T}^{\sigma}e_2$ (JOIN) $\mathcal{T}^{\sigma}e_3$ (JOIN))

  IF–expressions take three arguments after the keyword – e_1 is taken as the condition, e_3 the expression to be evaluated should e_1 evaluate to false, e_2 the expression to be evaluated otherwise.

  The SECD evaluates IF–expressions on the S–stack. First e_1 is evaluated and the result is placed on top of S. If the result is false, we want to

evalute e_3; if it is not false (we carefully avoid asking whether or not it actually is true – any non-false value is taken), we evaluate e_2. Either way, we continue on with the instruction after the IF–expression. The code generated is as follows. We plant code for each constituent expression in a special way. First we translate e_1, then we plant an *SEL* instruction, then a list consisting of the translated e_2 followed by a *JOIN* instruction, and similarly for e_3. The job of *SEL* is to examine the top of the stack S and see whether or not is false. If false, we continue with the code sequence for e_3. If non-false, we continue with the code sequence for e_2. The *JOIN* instruction ensures that we continue on correctly with the next instruction after the IF.

Function definitions and function calls.

- **function calls.**
  $\mathcal{T}^\sigma$(f e_1 ...e_k) →
  (LDC NIL) $\mathcal{T}^\sigma e_k$ (CONS) ... (CONS) $\mathcal{T}^\sigma e_1$ (CONS) $\mathcal{T}^\sigma$f (AP)
  Remember what we said about extensionality. That if

$$\text{f x y} = \text{body, then f} = \lambda \text{ x y . body}$$

A function call

$$(\text{f } e_1 \ldots e_k)$$

will operate as the lambda expression followed by its arguments.

$$(\text{f } e_1 \ldots e_k) = (\lambda \text{ } x_1 \ldots x_k \text{ . body}) \text{ } e_1 \ldots e_k$$

To compile code for a function call we first gather the arguments together on S in a single list, then push the closure on S and enter the function body. To gather the arguments together in one list we first push NIL on S (with LDC NIL). The we evaluate the last argument e_k ($\mathcal{T}^\sigma e_k$). The net effect of executing this code will be to place one value on top of S, that of e_k evaluated in the current environment σ. A CONS makes a single stack item (a dotted pair) out of the two top values. We now evaluate e_{k-1} in σ ($\mathcal{T}^\sigma e_{k-1}$) and CONS that, and so on until we have formed a single list from the code generated for the k arguments. The next step is to push the body of the function onto S. Typically we just give the name of a function which is already defined, say f. The translation of *f* in environment σ is just a LD and the object loaded will be the closure for f, a dotted pair consisting of the body of f and the environment for the evaluation of f. The call on AP will expect a closure and a single argument list on the stack.

It saves away the state in which the call was made, sets the machine state to reflect our execution of f (sets S to null, gets the base environment for the call from the closure and appends to it the argument list pertinent to this call) and then enters the body of f.

- **LAMBDA expressions.**
  $\mathcal{T}^{\sigma}$(LAMBDA (x_1...x_k) body) → (LDF $\mathcal{T}^{\sigma+}$ body (RTN))

  Each lambda expression introduces new variables x_1, ..., x_k which it uses in its body. When we compile the body of a lambda expression we must do so with respect to an environment which is σ augmented by these additional bindings. This we refer to as $\sigma+$.

  What do we want from the compilation of a lambda expression? Just generate the dotted pair representing its closure and leave it sitting on top of the stack S. It could be used on the fly, in which case arguments to it should have been prepared and already sit on the stack – all we now need to do is call AP. Or it could be the body of a local function declaration in a LET or LETREC (see below). We need the closure either way. The translation scheme here will cause the translated function body (translated with respect to $\sigma+$) to be generated as a single item and placed on the stack S. The preceding LDF will cause the creation of the closure on top of the stack. We append RTN to the body to enable a graceful return from each call. The code for the closure is now complete.

- **LET–expressions.**
  $\mathcal{T}^{\sigma}$(LET body (x_1.e_1)...(x_k.e_k)) →
  (LDC NIL) $\mathcal{T}^{\sigma}e_k$ (CONS) ... $\mathcal{T}^{\sigma}e_1$ (CONS)
  (LDF $\mathcal{T}^{\sigma+}$ body (RTN) AP)

  The compilation of a LET expression compiles a list of the bindings one by one, CONSing them together into a single list on S. Notice that the translations of the arguments are with respect to σ and may NOT refer to other arguments (no recursion). With the arguments compiled, the body is now compiled with respect to the augmented environment $\sigma+$ (which is not yet in place). The call on AP saves the calling state, empties the stack S, installs the closure environment and then appends the argument list to it. Then the code for the body of the LET is entered.

- **LETREC–expressions.**
  $\mathcal{T}^{\sigma}$(LETREC body (x_1.e_1)...(x_k.e_k)) →
  (DUM LDC NIL)
  $\mathcal{T}^{\sigma+}$ e_k (CONS) ... $\mathcal{T}^{\sigma+}$ e_1 (CONS)
  (LDF $\mathcal{T}^{\sigma+}$ body (RTN) RAP)

  The compilation of a LETREC is structurally similar to that of a LET. There are two significant differences. First the local definitions MUST be lambda definitions. When compiled lambda definitions tell you how

to carry out a computation, not the result. And this is what we want because a LETREC definition will in general depend upon definitions not yet treated. Both the definitions and the body are compiled with respect to the augmented environment $\sigma+$. Secondly, we have to build a circular environment. As explained in the companion report [HBGS89], this we do by creating a marker atom and adding it to the initial environment through DUM. Code generation then proceeds by our generating a single list of arguments (all lambda bodies), and then code for the body of the LETREC, prefixed by LDF and terminated by RTN. The closures generated all have their environment pointers set to refrence the marker atom generated by the call on DUM. The final piece of code is RAP. RAP saves the current state and prepares for this invocation. It clears the stack S. The argument list is appended to the environment for the LETREC contained in the closure AND is made circular by setting the CAR pointer of the marker atom to point to the head of the argument list. Finally we enter the body of the LETREC.

7 An Example

In order to demonstrate the translation schema we will work through the compilation of the function *append* given earlier and restated below with a call imbedded in the body:

```
(LETREC append (CONS (QUOTE 1) (QUOTE nil)) (QUOTE nil)
   (append LAMBDA (x y)
      (IF (EQ x (QUOTE NIL))
         y
         (CONS (CAR x) (append (CDR x) y)
         )
      )
   )
)
```

Assume that the original environment, the empty list (), is called σ. We will define some abbreviations for portions of code to facilitate readability:

```
eqexp   =  (EQ x (QUOTE nil))
consexp =  (CONS (CAR x) (append (CDR x) y)
ifexp   =  (IF eqexp y consexp)
lamexp  =  LAMBDA (x y) ifexp
arg1    =  (CONS (QUOTE 1) (QUOTE nil))
arg2    =  (QUOTE nil)
```

As well, anticipating that *append* and *x,y* will be added to σ, let us define:

σ_1 = ((append))
σ_2 = ((x y) (append))

We can now state the example as:

$\mathcal{T}^{\sigma}$ (LETREC (append arg1 arg2) (append lamexp))

Applying the rule for LETREC we get

(DUM LDC nil ($\mathcal{T}^{\sigma_1}$ lamexp) CONS
    LDF (($\mathcal{T}^{\sigma_1}$ (append arg1 arg2)) RTN) RAP)

Compiling the LAMBDA expression in σ_1 proceeds as:

$\mathcal{T}^{\sigma_1}$ (LAMBDA (x y) ifexp)

$= \mathcal{T}^{\sigma_2}$ ifexp

Compiling the IF expression in σ_2:

$\mathcal{T}^{\sigma_2}$ (IF eqexp y consexp)

$= (\mathcal{T}^{\sigma_2}$ eqexp) (SEL (($\mathcal{T}^{\sigma_2}$ y) JOIN) (($\mathcal{T}^{\sigma_2}$ consexp) JOIN))

$\mathcal{T}^{\sigma_2}$ (EQ x (QUOTE nil))

$= (\mathcal{T}^{\sigma_2}$ x) ($\mathcal{T}^{\sigma_2}$ (QUOTE nil)) EQ
= LD (0.0) LDC nil EQ

$\mathcal{T}^{\sigma_2}$ y

= LD (0.1)

$\mathcal{T}^{\sigma_2}$ (CONS (CAR x) (append (CDR x) y))

$= (\mathcal{T}^{\sigma_2}$ (append (CDR x) y))) ($\mathcal{T}^{\sigma_2}$ (CAR x)) CONS
= LDC nil (($\mathcal{T}^{\sigma_2}$ y) CONS ($\mathcal{T}^{\sigma_2}$ (CDR x)) CONS
($\mathcal{T}^{\sigma_2}$ append)) AP ($\mathcal{T}^{\sigma_2}$ x) CAR CONS
= LDC nil LD (0.1) CONS ($\mathcal{T}^{\sigma_2}$ x) CDR CONS
LD (1.0) AP LD (0.0) CAR CONS
= LDC nil LD (0.1) CONS LD (1.0) CDR CONS
LD (1.0) AP LD (0.0) CAR CONS

Thus, the finished LAMBDA expression is

LD (0.0) LDC nil EQ SEL (LD (0.1) JOIN)
(LDC nil LD (0.1) CONS LD (0.0) CDR CONS
LD (1.0) AP LD (0.0) CAR CONS JOIN)

We can now compile the body of the LETREC in σ_1

$\mathcal{T}^{\sigma_1}$ (append arg1 arg2)

= (LDC nil ($\mathcal{T}^{\sigma_1}$ arg2) CONS ($\mathcal{T}^{\sigma_1}$ arg1) CONS
($\mathcal{T}^{\sigma_1}$ append) AP

$\mathcal{T}^{\sigma_1}$ (QUOTE nil)

= LDC nil

$\mathcal{T}^{\sigma_1}$ (CONS (QUOTE 1) (QUOTE nil))

= ($\mathcal{T}^{\sigma_1}$ (QUOTE nil)) ($\mathcal{T}^{\sigma_1}$ (QUOTE 1)) CONS
= LDC nil LDC 1 CONS

$\mathcal{T}^{\sigma_1}$ append

= LD (1.0)

So, the body compiles to

(LDC nil LDC nil CONS LDC nil LDC 1 CONS
CONS LD (1.0) AP)

Pulling together all the pieces, we get a final result of

(DUM (LDC nil
LDF (LD (0.0) LDC nil EQ SEL (LD (0.1) JOIN)
(LDC nil LD (0.1) CONS LD (0.0) CDR CONS LD (1.0) AP
LD (0.0) CAR CONS JOIN)) RTN)
CONS LDF (LDC nil LDC nil CONS LDC nil LDC 1 CONS
CONS LD (1.0) AP) RTN RAP) AP

Acknowledgements

This work is based upon a larger study of functional architectures carried out by Todd Simpson whilst an undergraduate at Calgary. The work proceeded in tandem with an effort on functional architectures undertaken by Mike Hermann. Mike supplied the workbench and the SECD emulator used in these experiments. We are happy to acknowledge help and support from other members of the VLSI team at Calgary (Cameron Patterson, Konrad Slind and Simon Williams) and to Jeff Joyce who started the SECD ball rolling at Calgary with a comprehensive study in 1985.

References

[Bur75] W. Burge. *Recursive programming techniques.* Addison–Wesley, New York, 1975.

[FH88] A. J. Field and P. G. Harrison. *Functional programming.* Addison–Wesley, New York, 1988.

[HBGS89] M. J. Hermann, G. Birtwistle, B. Graham, and T. Simpson. The architecture of Henderson's SECD machine. Research report 89/340/02, Computer Science Department, University of Calgary, 1989.

[Hen80] P. Henderson. *Functional programming; applications and implementation.* Prentice Hall, London, 1980.

[HGAJ83a] P. Henderson, Jones G. A, and S. B. Jones. The Lispkit manual, volume 1. Technical Monograph, PRG-32(1), Oxford University Computing Laboratory, 1983.

[HGAJ83b] P. Henderson, Jones G. A, and S. B. Jones. The Lispkit manual, volume 2. Technical Monograph, PRG-32(2), Oxford University Computing Laboratory, 1983.

[Lan64] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1964.

[Lan65a] P. J. Landin. The correspondence between ALGOL60 and Church's lambda calculus, Part 1. *Communications of the ACM*, 8(2):89–101, 1965.

[Lan65b] P. J. Landin. The correspondence between ALGOL60 and Church's lambda calculus, Part 2. *Communications of the ACM*, 8(3):158–165, 1965.

[Lan66] P. J. Landin. An abstract machine for designers of computing languages. In *Proceedings of the IFIP Congress 65, volume 2*, pages 438–439, Washington, 1966. Spartan Books.

[Plo75] G. D. Plotkin. Call–by–name, call–by–value, and the lambda calculus. *Theoretical Computer Science*, 1(1):125–159, 1975.

APPENDIX: The compiler listing

```
;
;  SSSSS    EEEEEE    CCCCCC     DDDDD     L
; S         E         C          D    D    L
; SSSSSS    EEEEE     C          D    D    L
;      S    E         C          D    D    L
; SSSSS     EEEEEE    CCCCCC     DDDDD     LLLLLL
;
; ---------------------------------------------------------------------
; LISPKIT TO SECD MACHINE CODE COMPILER
;
; Todd Simpson  - December 1986.
;
; COMPILE
; e - input lispkit code
;
; returns:  if no compiler errors, an S-expression containing the SECD code.
;           else, prints out the errors and returns nil
; ---------------------------------------------------------------------

(defun compile (e)
    (syntax_error)                          ; set up some constants
    (others)
    (setq result (cmp e                     ; try to compile the lispkit source
                      nil                   ; returns ((errors)(SECD code))
                      nil
                      (list nil (cons 'AP 'STOP))
                 )
    )
    (if (eq (car result) nil)               ; if no errors
        (car (cdr result))                  ;   return the SECD code
        (printlist (car result))            ; else print out the errors
    )
)


; ---------------------------------------------------------------------
; CMP
;   Handle each possible type of lispkit expression.
;
; e - lispkit code
; n - namelist
```

```
; p - position list
; c - ((errors)(SECD code))
; ----------------------------------------------------------------------

(defun cmp (e n p c)
    (if (atom e)
        (defined e n p c)        ; an atom is a (hopefully) variable
    (if (or (eq (car e) 'quote) (eq (car e) 'QUOTE))
        (constant e n p c)                ; QUOTE preceeds a constant
    (if (member (car e) monadic_ops)
        (mons e n p c)                        ; car, cdr, atom
    (if (member (car e) diadic_ops)
        (dias e n p c)                        ; cons, add, sub, rem, mul, div,
                                            ; leq, eq
    (if (member (car e) triadic_ops)
        (tris e n p c)                        ; if
    (if (eq (car e) 'lambda)
        (lamb e n p c)                        ; lambda constructs
    (if (eq (car e) 'let)
        (lets e n p c)                        ; let blocks
    (if (eq (car e) 'letrec)
        (letrecs e n p c)               ; letrec blocks
        (func e n p c)                        ; function call

    )                                 ; letrec
    )                                 ; let
    )                                 ; lambda
    )                                 ; triadic_ops
    )                                 ; diadic_ops
    )                                 ; monadic_ops
    )                                 ; quote
    )                                 ; atom
)

; ----------------------------------------------------------------------
; DEFINED
;     Make sure a variable is defined.
;
; e - lispkit source
; n - namelist
; p - position string for printing errors
; c - errors and SECD code
; ----------------------------------------------------------------------
```

```
(defun defined (e n p c)
     (if (eq e nil)                          ; if the variable is nil, do nothing
         c
     (if (eq n nil)                          ; if the namelist is empty
         (list                                  ;   then return an undefined variable
             (append                         ;   error and the existing SECD code.
                 (error err_undef e e p)
                 (car c)
             )
             (cdr c)
         )
         (if (not (null n))                  ;  If the namelist is not empty then
             (list                          ;     If e is in the top level of n
                 (car c)                ;       then return existing errors
                 (cons 'LD              ;         and (LD (m.n) code)
                       (cons
                           (location e n)
                           (car (cdr c))
                       )
                 )
             )
             (defined e (cdr n) p c)        ; If e is not in the top level on n
         )                                 ;     then check the next level back.
     )
     )
)

; ---------------------------------------------------------------------
; LOCATION
;
; Locate e in the namelist n.  Return relative position (m.n)
; ---------------------------------------------------------------------
(defun location(e n)
    (if (member e
                (car n)                         ; If e is in this level of namelist
        )
        (cons '0
              (posn e
                    (car n)                 ;   then return (0.n)
              )
        )
        (incar (location e
                         (cdr n)        ;   else add one to m and check again
               )
```

```
        )
    )
)

; ------------------------------------------------------------------------
; POSN
;
; Locate e in the list n and return the number of elements before e in n.
; ------------------------------------------------------------------------
(defun posn(e n)
    (if (eq e (car n))                ; if e is the first element
        '0                          ; return 0
        (add '1                   ; else add 1 and check the next atom.
             (posn e
                   (cdr n)
             )    .
        )
    )
)

; ------------------------------------------------------------------------
; INCAR
;
; Add one to the first element of (m.n)
; ------------------------------------------------------------------------
(defun incar(s)
    (cons (add '1                 ; add 1 to the first element of (m.n)
               (car s)
          )
          (cdr s)
    )
)

; ------------------------------------------------------------------------
; CONSTANT
;     Pull off the item after the QUOTE and LD it onto the Control list.
;
; e - lispkit source
; n - namelist
; p - position list
; c - ((errors)(SECD code))
; ------------------------------------------------------------------------

(defun constant (e n p c)
```

```
    (list                                ; return the list consiting of:
        (append                                ; errors which are:
            (count 2 (car e) e p)        ;     errors in # of args to QUOTE
            (car c)                        ;     existing errors
        )
        (cons 'LDC
              (cons (car (cdr e))        ; (LDC value . code)
                    (car (cdr c))
              )
        )
    )
)


; ---------------------------------------------------------------------
; MONS
;     Handle single argument operations.
; e - lispkit source
; n - namelist
; p - position list
; c - ((errors)(SECD code))
; ---------------------------------------------------------------------
(defun mons (e n p c)
    (cmp (car (cdr e))                        ; Compile 'e' as in (car e)
         n                                ;    with existing namelist
         p                                ;    and position string
         (list                                ;    and the list of errors in the
             (append                        ;    number of arguments + existing
                 (count 2 (car e) e p)  ;    errors, and the operation name
                 (car c)                ;    followed by existing SECD code.
             )
             (cons (car e) (car (cdr c)))
         )
    )
)

; ---------------------------------------------------------------------
; DIAS
;     Compile operations with two arguments.
;
; e - lispkit source
; n - namelist
; p - position
; c - ((errors)(SECD code))
```

```
; ---------------------------------------------------------------------
(defun dias (e n p c)
    (if (eq (car e) 'cons)
        (dias2 e n p c (car (cdr (cdr e))) (car (cdr e)))
        (dias2 e n p c (car (cdr e)) (car (cdr (cdr e))))
    )
)

(defun dias2 (e n p c one two)
    (cmp one                                        ; Compile e1 as in
         n                                          ;    (add e1 e2)
         p                                          ;    [for cons e2]
         (cmp two                                   ; Compile e2 as in
              n                                     ;    (add e1 e2)
              p                                     ;    [for cons e1]
              (list
                  (append
                      (count 3 (car e) e p)         ; Check for # of arg errors
                      (car c)                       ;    existing errors
                  )
                  (cons (car e) (car (cdr c)))      ; operation name + existing
              )                                     ;    code.
         )
    )
)

; ---------------------------------------------------------------------
; TRIS
;     Compile operations with three arguments.
;
; e - lispkit code
; n - namelist
; p - position list
; c - ((errors)(SECD code))
; ---------------------------------------------------------------------
(defun tris (e n p c)
    (let
       ((temp1 (cmp (car (cdr (cdr e)))             ; temp1 is ((errors)
                    n                               ;    (code JOIN)) from
                    p                               ;    e2 in (if e1 e2 e3)
                    (list nil 'JOIN)
             )
        )
        (temp2 (cmp (car (cdr (cdr (cdr e))))       ; temp2 is ((errors)
```

```
                    n                                   ;   (code JOIN)) from
                    p                                   ;   e in (if e1 e2 e3)
                    (list nil 'JOIN)
                )
        ))
        (cmp (car (cdr e))                               ; Compile e1 from
            n                                       ;   (if e1 e2 e3)
            p
            (list
                (append
                    (count 4 (car e) e p)                ;  Check for errors
                    (append                              ;   in # of args
                        (car temp1)
                        (append                          ;
                            (car temp2)                  ;   + errors from e3
                            (car c)                      ;   + existing errors
                        )
                    )
                )
                (cons 'SEL                               ; SEL +
                    (cons (car (cdr temp1))
                          (cons (car (cdr temp2))        ;   + code from e3
                                (car (cdr c))            ;   + existing code.
                          )
                    )
                )
            )
        )
    )
)

; ---------------------------------------------------------------------
; LAMB
;    Compile a lambda expression.
;
; e - lispkit source
; n - namelist
; p - position string
; c - ((errors)(SECD code))
; ---------------------------------------------------------------------
(defun lamb (e n p c)
    (let (( temp  (cmp                                   ; errors and code
                    (car (cdr (cdr e)))                   ;  for the body of
                    (cons (clean (car (cdr e))) n)       ;  the lambda exp.
```

```
                p                                       ;  (eq x in
                (list                                ;     (lambda (x) x))
                    (checkfun (car e) e n p)        ;  namelist includes
                    'RTN                                ;  parameters, which
                )                                       ;  are checked by
            )
        ))
        (list
            (append
                (car temp)                          ; return errors from temp +
                (car c)                          ;    existing errors
            )                                   ;  and
            (cons 'LDF
                (cons (car (cdr temp))          ;    (LDF (body) . original_code)
                    (car (cdr c))
                )
            )
        )
    )
)


; ---------------------------------------------------------------------
; COUNT
;      Make sure that the operation has the right number of arguments.
; n - the number of arguments the operation should have
; keyword - operation name
; l - lispkit source
; p - position list
; ---------------------------------------------------------------------

(defun count (n keyword l p)
    (cond
        ((atom l)                       ; if l is an atom, we should
            (cond                       ;   have no more arguments.
                ((eq l nil)
                    (cond
                        ((eq n 0)
                            nil                 ; return nil, we have right # of args
                        )
                        (t (error err_fewargs keyword l p))
                    )
                )
                (t  (error err_arglist keyword l p))
```

```
            )
        )
        (t  (cond                          ; if l is not an atom then if we
                                           ; have processed all the arguments
                                           ; then we have to many.
                ((eq n 0)
                    (error err_manyargs keyword l p)
                )
                (t (count (- n 1) keyword (cdr l) p))        ; check the next arg.
            )
        )
    )
)


;---------------------------------------------------------------------
; ERROR
;    Returns a list containing its paramters.
; --------------------------------------------------------------------
(defun error (n a e p)
    (cons (cons n
                (cons a
                      (cons e
                            (cons p
                                  nil
                            )
                      )
                )
          )
          nil
    )
)

; --------------------------------------------------------------------
; OTHERS
;     Set up some constants for the different types of operations.
; --------------------------------------------------------------------
(defun others()
    (setq monadic_ops '(car cdr atom))
    (setq diadic_ops '(add sub mul div rem leq eq cons))
    (setq triadic_ops '(if))
)

; --------------------------------------------------------------------
```

```
; SYNTAX ERROR
;     Set up some constants for the types of errors.
; ------------------------------------------------------------------------
(defun syntax_error()
    (setq err_undef '1)
    (setq err_fewargs '2)
    (setq err_manyargs '3)
    (setq err_arglist '4)
    (setq err_invform '5)
    (setq err_invdef '6)
    (setq err_deftwice '7)
    (setq err_formlist '8)
    (setq err_formarg '9)
    (setq err_actlist '10)
)

;------------------------------------------------------------------------
; PRINTLIST
;     Print out the errors.
; ------------------------------------------------------------------------
(defun printlist (l)
    (if (eq l nil)
        nil
        (append
            (print1 (car l))                ; print the first errors
            (printlist (cdr l))               ; check for more errors
        )
    )
)

;------------------------------------------------------------------------
; PRINT1
;    Print out one error message.
; ------------------------------------------------------------------------
(defun print1 (x)
    (cond
        ((eq (car x) err_undef)
            (send (cons
                      (car (cdr x))
                      '(used but not defined)
                  )
                (p x)
            )
        )
```

```
((eq (car x) err_fewargs)
    (send (cons
              (car (cdr x))
              '(has too few arguments)
          )
          (cons (e x) (p x))
    )
)
((eq (car x) err_manyargs)
    (send (cons
              (car (cdr x))
              '(has too many arguments)
          )
          (cons (e x) (p x))
    )
)
((eq (car x) err_arglist)
    (send (cons
              (car (cdr x))
              '(has an incorrect argument list)
          )
          (cons (e x) (p x))
    )
)
((eq (car x) err_invform)
    (send (cons
              'incorrect
              (cons
                  (car (cdr x))
                  'form
              )
          )
          (cons (e x) (p x))
    )
)
((eq (car x) err_invdef)
    (send
        '(incorrect form of definitions)
        (cons
            (dump 2 a)
            (p x)
        )
    )
)
```

```
        ((eq (car x) err_deftwice)
            (send (cons
                      (car (cdr x))
                      '(defined more than once)
                  )
                (cons (e x) (p x))
            )
        )
        ((eq (car x) err_formlist)
            (send
                '(incorrect formal argument list)
                (cons (e x) (p x))
            )
        )
        ((eq (car x) err_formarg)
            (send
                '(incorrect formal argument)
                (cons (e x) (p x))
            )
        )
        ((eq (car x) err_actlist)
            (send
                '(incorrect actual argument list)
                (cons (e x) (p x))
            )
        )
        (t
            (send
                (append
                    '(unexpected error number)
                    (cons (car x) nil)
                )
                (p x)
            )
        )
    )
)

;---------------------------------------------------------------------
; E
;     The position of the error in the program.
; --------------------------------------------------------------------
(defun e (x)
    (dump 2 (car (cdr (cdr x))))
```

```
)

;---------------------------------------------------------------------
; P
; --------------------------------------------------------------------
(defun p (x)
    (car (cdr (cdr (cdr x))))
)

;---------------------------------------------------------------------
; SEND
;    Actually print the error to the screen.
; --------------------------------------------------------------------
(defun send (m p)
    (terpri)
    (print
        (append
            m
            (if (eq p nil)
                '(in the body of the program)
                 (position p)
            )
        )
    )
)

;---------------------------------------------------------------------
; POSITION
;    Gives the exact location in the program where the error occured.
; --------------------------------------------------------------------
(defun position (p)
    (if (eq p nil)
        nil
        (cons
            'in
            (cons
                (car p)
                (position (cdr p))
            )
        )
    )
)

; --------------------------------------------------------------------
```

```
; DUMP
;    Simplify the position by inserting *'s for long expressions.
;
; n - the number of levels to go
; f - lispkit code
; ---------------------------------------------------------------------
(defun dump (n f)
    (if (atom f)
        f
        (if (not (> n 0))
            '*
            (if
                (and
                    (eq (dump n (cdr f)) '*)
                    (if (atom (dump n (cdr f)))
                        (eq (dump n (cdr f)) '*)
                        (and
                            (eq (car (dump n (cdr f))) '*)
                            (eq (cdr (dump n (cdr f))) nil)
                        )
                    )
                )
                '*
                (cons (dump (- n 1) (car f)) (dump n (cdr f)))
            )
        )
    )
)


;---------------------------------------------------------------------
; Checking LAMBDA
;     Check the arguments to lambda, and make sure lambda has the right
; number of arguments.
;
; keyword - 'lambda'
; e - lambda lispkit code
; n - namelist
; p - position string
; ---------------------------------------------------------------------
(defun checkfun (keyword e n p)
    (append
        (formallist                          ; Check the parameter list
           (car (cdr e))
```

```
            nil
        )
        (count 3 keyword e p)               ; Check for right # of args
    )
)

; ---------------------------------------------------------------------
; FORMALLIST
;     Check the argument list for the LAMBDA expression.
;
; f - the input parameter list
; l - build a new paramter list
; ---------------------------------------------------------------------

(defun formallist (f l)
    (if (atom f)
        (if (eq f nil)                       ; If no parameters, better have nil
            nil
            (error err_formlist e e p)
        )
        (if (atom (car f))                ; First element of f should be an
                                          ; atom.
            (if (member (car f) l)        ; If it is already in l, it was
                                          ; declared twice
                (append
                    (error err_deftwice (car f) e p)
                    (formallist (cdr f) l)
                )
                (formallist               ; If it is not in l then check the
                    (cdr f)               ; rest of the parameters
                    (cons (car f) l)
                )
            )
            (append                       ; If the first element of f is not
                                          ; an atom, we have an error
                (error err_formarg (car f) e p)
                (formallist (cdr f) l)
            )
        )
    )
)

;---------------------------------------------------------------------
; CLEAN
```

```
;    Ensures new levels of the namelist have a string of atoms.
; ---------------------------------------------------------------------

(defun clean (l)
    (if (atom l)
        nil
        (if (atom (car l))
            (cons (car l) (clean (cdr l)))
            (clean (cdr l))
        )
    )
)

; ---------------------------------------------------------------------
; LETS
;    Compile a lst block.
;
; e - lispkit code
; n - namelist
; p - position string
; c - ((errors)(SECD code))
; ---------------------------------------------------------------------
(defun lets (e n p c)
    (let ((let1 (cmp                                ; let1 is the compilation
                (car (cdr e))                   ; of the body of the let.
                (cons (vars (cdr (cdr e)))          ; The new names are included
                    n                           ; on the namelist
                )
                p
                (list                           ; Check the definition of
                                                ; let.
                    (append
                        (checkdef (car e) e n p)
                        (car c)
                    )
                    'RTN
                )
            )
        ))
        (complis (exprs (cdr (cdr e)))          ; Compile each component of the let
            n
            p
            (list
                (append
```

```
                    (checkdef (car e) e n p)
                    (car let1)                ; Errors from compiling the body
                )
                (cons 'LDF
                    (cons
                        (car (cdr let1))        ; Code from the body
                        (cons
                            'AP
                            (car (cdr c))        ; Existing code
                        )
                    )
                )
            )
        )
    )
)


; ---------------------------------------------------------------------
; LETRECS
;    Compile a letrec expression.
;
; e - lispkit code
; n - namelist
; p - position list
; c - ((errors)(SECD code))
; ---------------------------------------------------------------------
(defun letrecs (e n p c)
    (setq letrec1 (cmp                                    ; Compile the body
                    (car (cdr e))                        ; of the letrec.
                    (cons (vars (cdr (cdr e)))        ; New names on
                        n                                ; namelist
                    )
                    p
                    (list                                ; Check definition
                        (append
                            (checkdef (car e) e n p)
                            (car c)
                        )
                        'RTN
                    )
                )
    )
    (setq letrec2 (complis                            ; Compile each
```

```
                    (exprs (cdr (cdr e)))               ; component of the
                    (cons (vars (cdr (cdr e)))        ; letrec with the
                        n                               ; extended namelist
                    )                                 ; for recursion.
                    p
                    (list
                        (append
                            (checkdef (car e) e n p)
                            (car letrec1)               ; Errors from body
                        )
                        (cons
                            'LDF
                            (cons
                                (car (cdr letrec1))       ; Code from body
                                (cons
                                    'RAP
                                    (car (cdr c))
                                )
                            )
                        )
                    )
                )
    )
    (list                                 ; return the list with all the
        (car letrec2)                         ; errors, and
        (cons 'DUM
            (car (cdr letrec2))         ; (DUM all_the_code)
        )
    )
)


; ---------------------------------------------------------------------
; FUNC
;    Handle a function call.
;
; e - lispkit source.
; n - namelist
; p - position string
; c - ((errors)(CESD code))
; ---------------------------------------------------------------------
(defun func (e n p c)
    (complis (cdr e)               ; Compile the parameters
```

```
        n
        p
        (cmp                        ; Compile the function call
            (car e)
            n
            (cons (car e) p)
            (list
                (append
                    (checklist e e n p)
                    (car c)                ; Existing errors
                )
                (cons
                    'AP
                    (car (cdr c))        ; Existing code
                )
            )
        )
    )
)

; ---------------------------------------------------------------------
; CHECKLIST
;     Check the definition of a parameter list.
;
; l - list of remaining parameters
; e - list of parameters
; n - namelist
; p - position string
; ---------------------------------------------------------------------
(defun checklist (l e n p)
    (if (atom l)
        (if (eq l nil)
            nil
            (error err_actlist l e p)
        )
        (checklist (cdr l) e n p)
    )
)


; ---------------------------------------------------------------------
; COMPLIS
;     Compile each of the components of the e list.
;
```

```
; e - all the lispkit routines to be compiled
; n - namelist
; p - position string
; c - ((errors)(SECD code))
; ---------------------------------------------------------------------
(defun complis (e n p c)
    (if (eq e nil)                          ; If we have compiled all of them
        (list
            (car c)                          ; Return the errors
            (cons
                'LDC
                (cons                          ; And the code.
                    'nil
                    (car (cdr c))
                )
            )
        )
        (complis (cdr e)                ; Make sure the rest are compiled
                 n
                 p
                 (cmp (car e)                ; Compile the first one
                      n
                      p
                      (list                  ; Check this ones definition
                          (car c)
                          (cons                ; And pass the existing code
                              'CONS
                              (car (cdr c))
                          )
                      )
                 )
        )
    )
)

; ---------------------------------------------------------------------
; VARS
;
; Select (x1,...,xk) from ((x1,e1),...,(xk,ek)).
; ---------------------------------------------------------------------
(defun vars(d)
   (if (eq d nil)
        nil
        (cons (car (car d))
```

```
            (vars (cdr d))
        )
    )
)

; ----------------------------------------------------------------------
; EXPRS
;
; Select (e1,...,ek) from ((x1,e1),...,(xk,ek)).
; ----------------------------------------------------------------------
(defun exprs(d)
   (if (eq d nil)
       nil
       (cons (cdr (car d))
             (exprs (cdr d))
       )
   )
)

; ----------------------------------------------------------------------
; checking LET and LETREC
; ----------------------------------------------------------------------
(defun checkdef (keyword e n p)
    (if (eq keyword nil)                          ; If the definition in nil,
        nil                                         ; nil
        (if (atom (cdr e))                        ;   If the function is not
                                                  ;    defined
            (error err_invform keyword e p)         ;    we have an error
            (checkdefs                                 ; Else we should check the
                (cdr (cdr e))                         ; parameters.
                nil
                e
                (if (eq keyword 'letrec)          ; Pass different n lists for
                    (nprime n e)                  ; let and letrec
                    n
                )
                p
                keyword
            )
        )
    )
)
```

```
; ------------------------------------------------------------------------
; checking LETREC
;    With letrec, we need a recursive definition for n.  This is given
;    by definiends.
;
; e - (eg.  (letrec x (x QUOTE a)))
; ------------------------------------------------------------------------

(defun nprime (n e)
    (cons (definiends (cdr (cdr e))) n)
)

; ------------------------------------------------------------------------
; checking LETREC
;     Build the resursive definition for letrec.
;
; d - (eg. (x QUOTE a))
; ------------------------------------------------------------------------

(defun definiends (d)
    (if (atom d)                        ; finished?
        nil
        (if (if (atom (car d))                ; If this is not a function
                nil
                (atom (car (car d)))        ;   And the next one is a function
            )
            (cons
                (car (car d))                ;   Add the next name to the list
                (definiends (cdr d))
            )
            (definiends (cdr d))
        )
    )
)

; ------------------------------------------------------------------------
; checking LET and LETREC
;    Check each of the functions declared in let/letrec
;
; d - list of functions in let/letrec
; l - nil
; e - lispkit source
; n - namelist
; p - position string
```

```
; keyword - 'let'/'letrec'
; ------------------------------------------------------------------------

(defun checkdefs (d l e n p keyword)
    (if (atom d)                                    ; No more functions?
        (if (eq d nil)                                  ; Better have the nil list
            nil
            (error err_invform keyword e p)          ; of it is an error.
        )
        (if
            (if        (atom (car d))
                nil
                (atom (car (car d)))                  ; If this is a function def
            )
            (append
                (if (member (car (car d)) l)          ; Check to make sure it is
                    (error                           ; not already defined.
                        err_deftwice
                        (car (car d))
                        e
                        p
                    )
                    nil
                )
                (checkdefs                           ; Check the rest of the
                    (cdr d)                           ; functions.
                    (cons (car (car d)) l)
                    e
                    n
                    p
                    keyword
                )
            )
            (error err_invdef (car d) e p)          ; If this is not a function
                                                    ; definition, then it is an
                                                    ; error.
        )
    )
)

; ------------------------------------------------------------------------
; NUMATOMS - Count the number of atoms in the input string.
; ------------------------------------------------------------------------
```

```
(defun numatoms (e)
    (cond
        ((eq e nil)
            0
        )
        ((atom e)
            1
        )
        (t (+ (numatoms (car e))
              (numatoms (cdr e))
           )
        )
    )
)
```