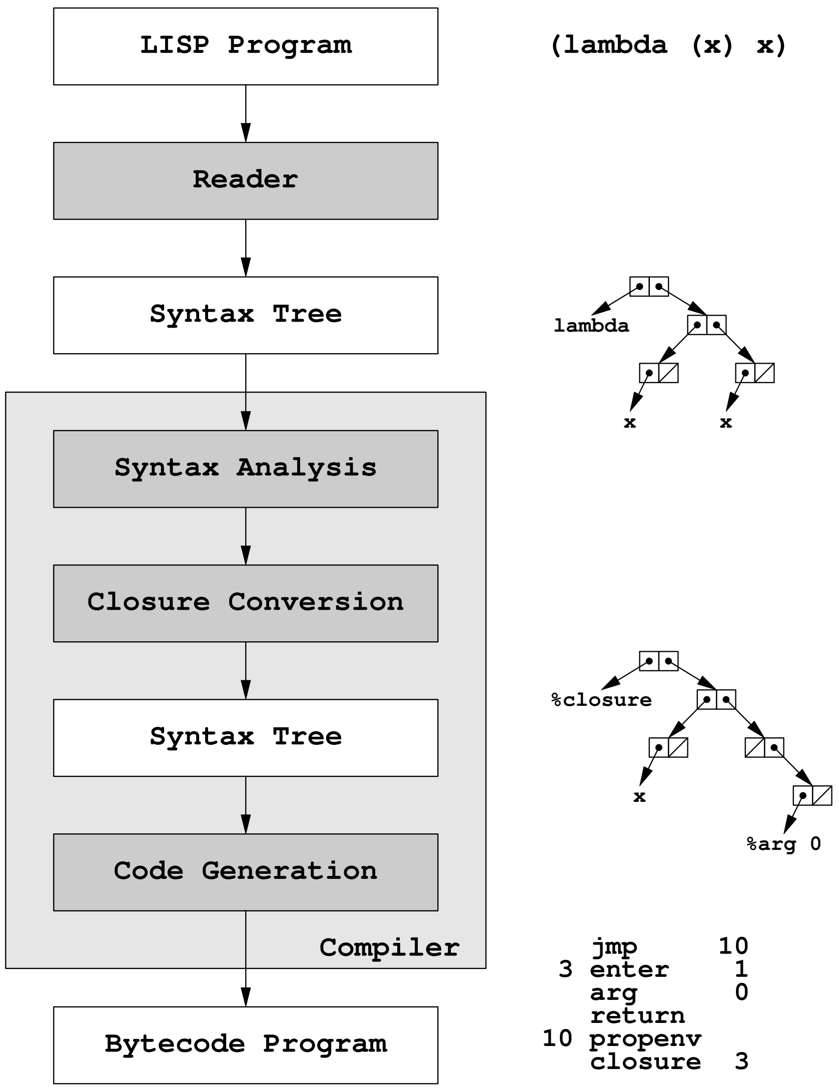
LISP SYSTEM IMPLEMENTATION

Nils M Holm

Table of Contents

APPENDIX

14. The Compiler



```
   jmp      10
 3 enter     1
   arg       0
   return
10 propenv
   closure   3
```

Fig. 26 – The LISP9 compiler

This chapter describes the *compiler* of the LISP9 system. The compiler is the part of the system that translates the internal representation of LISP programs (as produced by the *reader*) to *abstract machine code* that will be interpreted by an abstract machine in the last step of the process of evaluation.

Figure 26 illustrates the process of compilation with a textual representation of LISP programs as its input and bytecode for an abstract machine as its output.

The first step in the process is to translate the textual representation of the input program to a structure that is easier to manipulate in the subsequent steps. LISP is a *homoiconic* language, which means that code and data share the same representation. When a LISP expression is read by the **read** function, the *external* (textual) representation of the expression is translated to some LISP data object – typically a nested *list*.

In figure 26, the example program

```
(lambda (x) x)
```

is translated to the nested list structure in figure 27.

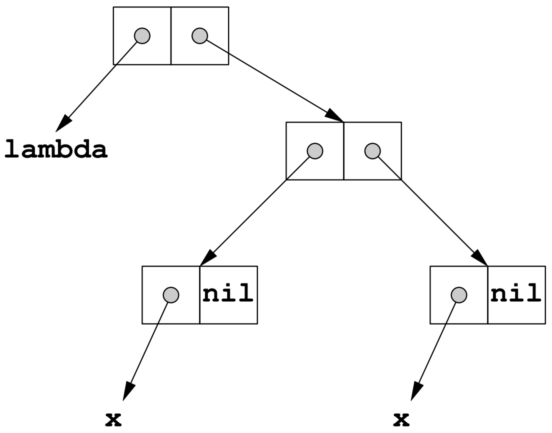


Fig. 27 – List structure as abstract syntax tree

Compilers typically operate on structures called *abstract syntax trees* (*AST*). Such structures represent the input program in tree form, where inner nodes indicate operations and outer nodes denote operands. The part of the process that converts a textual program to an AST is called a *parser*. Because LISP is homoiconic, the **read** function can serve as a parser for LISP.

In most LISP syntax trees, inner nodes will denote either application or concatenation. For instance, figure 28 shows the tree form of the expression **(cons foo bar)**, which applies **cons** to the concatenation of **foo**, **bar**, and **nil**. However, some trees, like the tree in figure 27, which was generated from the lambda expression **(lambda (x) x)**, have special meanings. For example, the **(x)** in the expression is *not* a function application.

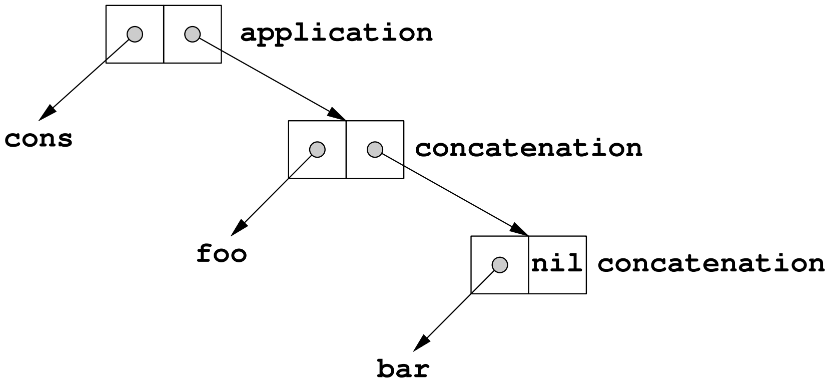


Fig. 28 – Application and concatenation

Hence some additional analysis has to be performed. The reader does not “know” about LISP’s syntax. It treats all its input as data and transforms it to trees (or other objects), no matter whether or not that input represents a valid LISP program. For example, it will happily turn expressions like **(lambda)** or **(quote foo bar baz)** or **(setq 5 7)** into corresponding list structures.

Hence the first phase of the compiler itself will be *syntax analysis*. This phase will examine the syntax trees generated by **read** and make sure that they represent syntactically correct LISP9 programs. When an error is found at this stage, compilation will be aborted and control will be passed back to the invoking process (either the REPL or, in case of batch processing, the operating system).

14.1 Syntax Analysis

One of the most obvious syntax tests is checking the number of arguments given to a particular keyword. For example,

(setq *x* *y***)**

might be correct, but at least x has to be inspected more closely. The expressions

(setq x**)**
(setq x y z**)**

though, do not need to be tested any further, because **setq** expects exactly two arguments.

The **length()** function returns the length of the proper (non-dotted) list n. It will be used to count arguments.

```
int length(cell n) {
    int     k;

    for (k = 0; n != NIL; n = cdr(n))
        k++;
    return k;
}
```

The **ckargs()** function makes sure that the special form x has at least *min* and at most *max* arguments. When $max < 0$, then there is no upper limit on the arguments in x. **ckargs()** signals an error when the wrong number of arguments is found in x.

```
void ckargs(cell x, int min, int max)
{
    int     k;
    char    buf[100];

    k = length(x)-1;
    if (k < min || (k > max && max >= 0)) {
        sprintf(buf,
          "%s: wrong number of arguments",
          symname(car(x)));
        error(buf, x);
    }
}
```

The **ckseq()** function performs syntax analysis on each element of the list x. The *top* argument indicates that syntax checking is currently being applied to a form that appears at the top level of a program, i.e. a form that does not appear inside of another form.

The *comm* argument indicates that a *comment* may appear in the place of each of the subforms of x.

All syntax analysis functions return 0, but that value does not bear any significance. See the **syncheck()** function at the end of this section for an explanation.

```
int syncheck(cell x, int top, int comm);

int ckseq(cell x, int top, int comm) {
    for (; pairp(x); x = cdr(x))
        syncheck(car(x), top, comm);
    return 0;
}
```

The **ckapply()** function makes sure that **apply** has at least two arguments. That's all. Not much can be done in a dynamic programming language at this point without resorting to more complex static program analysis, like type inference. [Milner1978]

```
int ckapply(cell x) {
    ckargs(x, 2, -1);
    return 0;
}
```

A comment may only appear in a context where comments are allowed. This is what the **ckcomment()** function verifies.

```
int ckcomment(int comm) {
    if (!comm) error("comment in wrong context",
                     UNDEF);
    return 0;
}
```

A *definition* is always introduced by a **def** form in LISP9. All derived forms, like **defun**, will be transformed to applications of **def**. The **ckdef()** function makes sure that the first argument of **def** is a symbol and the second one is syntactically correct.

Note that *top=0* and *comm=0* are passed to **syncheck()**, because the *value* in **(def** *symbol value***)** is not at the top level (it is contained in **(def ...)**) and it may not be a comment. The value may *contain* comments, though, for instance when it is a **lambda** special form).

```
int ckdef(cell x, int top) {
    ckargs(x, 2, 2);
    if (!symbolp(cadr(x)))
        error("def: expected symbol", cadr(x));
    if (!top)
        error("def: must be at top level", x);
    return syncheck(caddr(x), 0, 0);
}
```

The **if** form has two or three arguments and the **if*** form has exactly two arguments. This is what the **ckif()** and **ckifstar()** functions test. They also make sure that those arguments are syntactically correct.

```
int ckif(cell x) {
    ckargs(x, 2, 3);
    return ckseq(cdr(x), 0, 0);
}

int ckifstar(cell x) {
    ckargs(x, 2, 2);
    return ckseq(cdr(x), 0, 0);
}
```

The **symlistp()** predicate returns truth, if its argument x is either a symbol or a list of symbols. If it is a list, the list may even be dotted or empty. When it is dotted, the object at the end of the list must also be a symbol.

This function is used to make sure that the *formal argument lists* of **lambda** are in the proper shape.

```
int symlistp(cell x) {
    cell    p;

    for (p = x; pairp(p); p = cdr(p)) {
        if (!symbolp(car(p)))
            return 0;
    }
    return symbolp(p) || NIL == p;
}
```

The **memq()** function (almost) mimics Scheme's **memq** function [R4RS]. It returns the first tail of x whose first element is **eq** to a. If if x does not contain a, it return **NIL**. The function assumes that a is a proper (NIL-terminated) list.

```
int memq(cell x, cell a) {
    for (; a != NIL; a = cdr(a))
        if (car(a) == x) return a;
    return NIL;
}
```

The **uniqlistp()** predicate returns truth, if no two elements of the (proper) list x are equal in the sense of **eq**. It is used to make sure that formal argument lists of **lambda** do not contain any duplicate symbols.

```
int uniqlistp(cell x) {
    if (NIL == x) return 1;
    while (cdr(x) != NIL) {
        if (memq(car(x), cdr(x)) != NIL)
            return 0;
        x = cdr(x);
    }
    return 1;
}
```

The **flatargs()** function turns the (potentially dotted) list a into a flat (proper) list. If a is a symbol, it returns a singular list containing that symbol. It is used to “flatten” formal argument list of **lambda**, e.g.:

```
(a b . c)  →  (a b c)
a          →  (a)
```

```
cell flatargs(cell a) {
    cell    n;

    protect(n = NIL);
    while (pairp(a)) {
        n = cons(car(a), n);
        car(Protected) = n;
        a = cdr(a);
    }
    if (a != NIL) n = cons(a, n);
    unprot(1);
    return nreverse(n);
}
```

The **cklambda()** function makes sure that the lambda form x

- has at least two arguments
- has a proper formal argument list as its second argument
- has no duplicate formal arguments
- has a syntactically correct body (2^{nd} and following arguments)

Note that comments are allowed in the body of **lambda**. In fact, the *only* expression in the body of a lambda function might be a comment, which would make the value of the function undefined. A more sophisticated syntax analyzer might catch this case. Feel free to implement the necessary code.

```
int cklambda(cell x) {
    ckargs(x, 2, -1);
    if (!symlistp(cadr(x)))
        error("lambda: invalid formals",
              cadr(x));
    if (!uniqlistp(flatargs(cadr(x))))
        error("lambda: duplicate formal",
              cadr(x));
    return ckseq(cddr(x), 0, 1);
}
```

A **macro** definition has basically the same shape as a **def** form, only with a different keyword as its first element. The **ckmacro()** function syntax-checks this form. The derived **defmac** form will be transformed to an application of **macro** during macro expansion.

```
int ckmacro(cell x, int top) {
    ckargs(x, 2, 2);
    if (!symbolp(cadr(x)))
        error("macro: expected symbol", cadr(x));
    if (!top)
        error("macro: must be at top level", x);
    return syncheck(caddr(x), 0, 0);
}
```

A **prog** form is correct, if all of its arguments are correct. **ckprog()** performs this test.

```
int ckprog(cell x, int top) {
    return ckseq(cdr(x), top, 1);
}
```

The **ckquote()** function makes sure that **quote** has exactly one argument.

```
int ckquote(cell x) {
    ckargs(x, 1, 1);
    return 0;
}
```

As indicated at the beginning of this section, **setq** has exactly two arguments and, given this case, the first one has to be a symbol. The **cksetq()** function performs the syntax analysis of **setq**.

```
int cksetq(cell x) {
    ckargs(x, 2, 2);
    if (!symbolp(cadr(x)))
        error("setq: expected symbol", cadr(x));
    return ckseq(cddr(x), 0, 0);
}
```

The **syncheck()** function performs syntax analysis of the form x. When $top \neq 0$, the form x appears at the top level and when $comm \neq 0$, the form may be a comment. **syncheck()** is called recursively by the above functions, passing the proper values for top and $comm$ along.

Any *atom* is a valid LISP9 program, so **syncheck()** returns immediately when given an atom. Dotted lists are never valid programs. All other cases will be delegated to the functions above. The default case is *function application*, which is simply checked for correctness of its function and arguments.

syncheck() returns an uninteresting value when x is a valid form. Otherwise it signals an error.

The reason why the syntax analyzing functions return a value at all is the structure of the **syncheck()** function. Because all the **ck...()** functions return a value, delegation to other functions can be done by statements like

```
if (car(x) == S_apply) return ckapply(x);
```

which looks less cumbersome than

```
if (car(x) == S_apply) { ckapply(x); return; }
```

Yes, equally readable alternatives (like nested “if”) exist.

```
int syncheck(cell x, int top, int comm) {
    cell    p;

    if (atomp(x)) return 0;
    for (p = x; pairp(p); p = cdr(p))
        ;
    if (p != NIL)
        error("dotted list in program", x);
    if (car(x) == S_apply) return ckapply(x);
    if (car(x) == S_comment)
        return ckcomment(comm);
    if (car(x) == S_def) return ckdef(x, top);
    if (car(x) == S_if) return ckif(x);
    if (car(x) == S_ifstar) return ckifstar(x);
    if (car(x) == S_lambda) return cklambda(x);
    if (car(x) == S_macro)
        return ckmacro(x, top);
    if (car(x) == S_prog) return ckprog(x, top);
    if (car(x) == S_quote) return ckquote(x);
    if (car(x) == S_setq) return cksetq(x);
    return ckseq(x, top, comm);
}
```

14.2 Closure Conversion

Once a form is known to be in the proper shape, its handling becomes much simpler, because lots of assumptions about its syntax tree can be made. For instance, the body of a lambda function f can now be referred to as $cddr(f)$ without having to make sure that the form f does indeed have a cddr field. Since the form has passed syntax analysis, it does. Similarly, it can be assumed safely that the first arguments of the **def**, **macro**, and

setq forms are symbols.

The next step in the compilation of LISP code is called *closure conversion*. A *closure* is a technical term that describes what LISP programmers typically call a *function* – given *lexical scoping*.

When a LISP function is created, it *closes over* all free variables referenced in its body. The *free variables* of a function are those variables that are referenced in a function, but are are not formal arguments of the same function. A variable that is not free in a function is called *bound* in that function (and vice versa).

For instance, the variable x is bound in

```
(lambda (x) (f x))
```

but the variable f is free in that expression – because it is not contained in the formal argument list **(x)**.

The function created by the lambda expression will "remember" the binding of f, though, thereby enabling definitions like this:

```
(defun (complement p)
  (lambda (x) (not (p x))))
```

The **complement** function takes a predicate p as its argument and creates a new (anonymous) function that returns the complement of p applied to x (its own argument). When **complement** is done evaluating, the variable p in **complement** no longer exists, but the resulting anonymous function will remember its binding. This is why the following code works:

```
(defun (even x) (= 0 (rem x 2)))
(def odd (complement even))
(odd 5)  =>  t
```

LISP textbooks, including this one, often talk about *bindings* of variables to values or of symbols to values. This terminology is a bit muddy, though, so it shall be clarified in the following.

A *variable* is a *symbol* that is bound to a *location*. A location is a storage cell – sometimes called a *box* – in which some value can be stored. So when a "variable is bound to a value", this actually means that the "variable is bound to a location containing the value".

The term “the binding of a variable” is sometimes used to refer to the box that holds the value of that variable. The binding of a variable is “in effect”, or exists, as long as the box associated with the name of the variable exists.

Bindings of symbols to locations are created by **def** forms or by lambda functions. All bindings have *indefinite extent*: once created, they exist forever. However, *local bindings*, as created by **lambda**, may be recycled by the garbage collector when it can prove that these bindings can no longer be accessed by any program in the system.

In programming languages without closures, a function is merely an address in memory. Application of the function is performed by temporarily transferring control to that address. It works the same way in LISP, but in addition the bindings that were in effect when the function was created also have to be in effect when the function is being applied.

The task of closure conversion is basically to replace all references to free variables in a function by references to a *local environment* that will become part of the data object representing a function. In LISP, this conversion can be though of as:

```
(lambda (z)  -->  (%closure (z)
  (z x y))          #(<binding of x>
                      <binding of y>)
                    (z (%ref 0) (%ref 1)))
```

The **%closure** keyword indicates a closure; x has been replaced by **(%ref 0)** and y by **(%ref 1)** in the body of the closure. In front of the body, there is another element in the closure: a vector containing the local environment. Given this vector v, **(%ref** n**)** essentially becomes **(vref** v n**)**.

In the above expression the free variables x and y have been replaced by references to locations in a local environment. When the closure is applied to a value, the local environment becomes the current environment for the duration of evaluating the body of the closure.

There is one problem, though: The bindings of x and y cannot necessarily be known at compile time, so the compiler cannot build the vector implementing the local environment. This has to be done *at run time*.

14.2.1 Building an Environment

Every free variable is bound *somewhere*, either in the environment of an *outer function* or in the formal argument list of the *immediate* outer function (*IOF*). An outer function of the function f is any function in which f is contained. Given $f = lambda\ (z)\ (z\ x\ y))$ and

```
(lambda (x) (lambda (y) (lambda (z) (z x y))))
```

both the entire above expression as well as

```
(lambda (y) (lambda (z) (z x y)))
```

are outer functions of f. However, only the latter is the *immediate* outer function of f, because it directly contains f without any other functions (lambda expressions) in between.

In the example, the function f contains one free variable (y) that is bound in its immediate outer function and another one (x) that is bound in the outermost function. Note that x is also free in the IOF of f.

So when $f = (lambda\ (z)\ (z\ x\ y))$ is created, its environment has to be populated with one free variable of its IOF and one bound variable of its IOF. The former can be achieved by copying the binding of x from an environment slot of the IOF to a slot in the environment of the new function.

However, the same cannot be done with a variable that is *bound* in the IOF, because it is located on the runtime stack and does not have any environment slot. So the another step is necessary to replace bound variables in closures as well:

```
(lambda (z)  -->  (%closure (z)
  (z x y))          #(<binding of x>
                      <binding of y>)
                    ((%arg 0)   ; z
                     (%ref 0)   ; x
                     (%ref 1))) ; y
```

An expression of the form **(%arg** n**)** now references the n^{th} argument of the closure. The formal argument list is still kept, through, for two reasons: it indicates the number of arguments that the closure expects and it indicates whether the number of arguments is fixed or variable.

Now that all variables can be identified using numeric indices, the compiler can build the *initialization map* or *initmap* of the closure. This map specifies a source slot for each binding of a free variable as well as its destination slot in the new local environment. It also specifies the type of slot from which the binding will be fetched: either from an outer environment (e) of from an argument slot (a) on the runtime stack.

Given that

- the binding of x is in environment slot 0 of the IOF
- the binding of y is in argument slot 0 of the IOF

the initmap would look like this:

```
((e 0 0)    ; env slot 0 --> slot 0
 (a 0 1))   ; arg slot 0 --> slot 1
```

meaning that when $(lambda\ (z)\ (z\ x\ y))$ is being created, the first slot of its local environment is populated from slot 0 of the environment of the IOF and slot 1 is populated from argument 0 of the IOF. The complete conversion would then look like this:

```
(lambda (z)  -->  (%closure (z)
  (z x y))          ((e 0 0) (a 0 1))
                    ((%arg 0) (%ref 0)
                              (%ref 1)))
```

The body of the closure is now completely free of symbols and all references to values of variables are done via *shallow binding*, i.e. by directly accessing a slot of an environment or a slot of the runtime stack. Therefore, variable access is constant-time. The local environment itself has been replaced by instructions in the initmap that specify how to create the environment at run time.

Because there are no symbolic references in the body of the closure any longer, it can be translated to bytecode for an abstract machine. The closure can then be applied to some values by temporarily making its local environment the current environment and transferring control to the address of its bytecode.

Note that local environments are *always* being populated from the IOF and *never* from any other outer functions. This can be demonstrated as follows:

If a variable x is bound in a function f, it shadows any variables named x that are bound in outer functions of f, so x is never fetched from *any* outer function.

A variable that is free in f is either bound or free in the IOF of f. In either case, its binding will be fetched from the IOF.

The binding of a variable that is free in both the IOF and the IOF of the IOF already has been *propagated* from the IOF of the IOF to the IOF. Hence there is no need to refer to any function other than the IOF when building a new environment.

Alternatively: given a function c with IOF b and outermost function a, there cannot be any variable that exists (free or bound) in a, does *not* exist in b and is free in c. It must also be free (or bound) in b and therefore its binding will be fetched from b.

When there is no outer function from which a binding can be copied, then the binding will be copied from the *top level environment* in which all bindings created by **def** are located. For instance:

```
(lambda (x)     -->  (%closure (x)
  (foo x x))           ((e 6 0))
                       ((%ref 0) (%arg 0)
                                 (%arg 0)))
```

given that the symbol **foo** is bound to slot 6 of the top level environment.

When bindings are propagated from one environment to the next, they do not necessarily occupy the same slots in every environment. Each time a new environment is built, bindings may vanish or be added. Bindings are added when a variable that was previously bound becomes free, as in

```
(lambda (x) (lambda (y) (x y)))
```

Here the variable *x* is bound in the outer, but free in the inner function. Bindings vanish when a free variable exists in the outer function, but not in the inner, as in

```
(lambda () (foo (lambda (y) y)))
```

Here the variable *foo* is free in the outer function, but does not exist in the inner one. Consider the expression

```
(lambda (f g)          ; E1
  (lambda (h)          ; E2
    (f (lambda (x)     ; E3
         (foo (g x) (h x))))))
```

where various free variables appear and disappear as the number of outer functions increases. The corresponding local environments, argument slots, and initmaps are shown in figure 29. E1 is the outermost environment, E2 the middle one, and E3 the innermost one. Solid arrows denote propagation of bindings, dashed arrows indicate time.

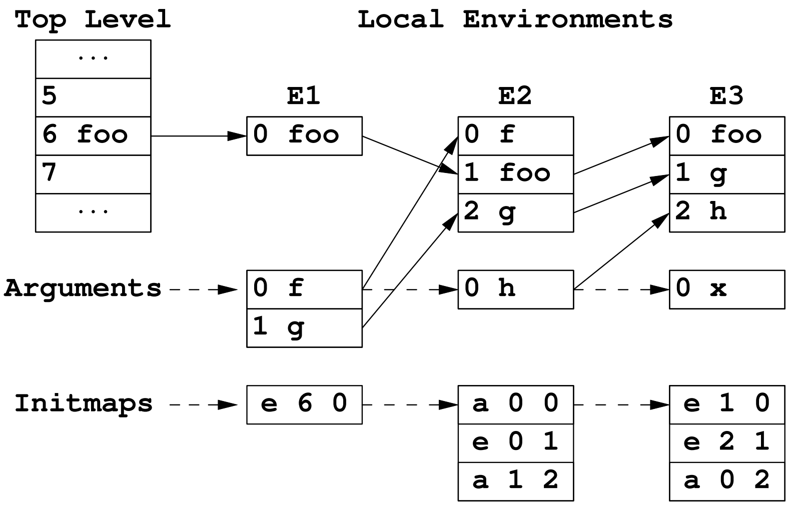


Fig. 29 – Propagation of bindings

Note that, although there are four free variables in total, no environment contains more than three bindings. This is because the binding of *f* is dropped at the same time as the binding of *h* is added. Local environments do not carry any dead weight. Only variables that are actually free in a function will be propagated.

A special case of the closure is the *combinator*, a function without any free variables, like

```
(lambda (x) x)
```

The closure resulting from a combinator has an empty initmap, so no local environment has to be created at all. Strictly speaking, the resulting closure is not even a closure, because it does not close over any free variables. Combinators are very efficient and under some circumstances, closures can be converted to combinators.

The following closure conversion code will do all of the above and a few more things. It is the heart of the compiler, where most of the complexity resides. But let us approach this part step by step...

14.2.2 Finding Free Variables

The **set_union()** function returns the *union* of the sets a and b. Each set is represented by a list of unique symbols. The function returns a fresh list that contains every element that is contained in either a or b or both. Since it operates on lists of symbols and the sets a and b are typically small, it uses the **memq** function (rather than a hash table) to test whether a given element is already contained in the result.

```
cell set_union(cell a, cell b) {
    cell    n;

    a = reverse(a);
    protect(a);
    protect(n = b);
    while (pairp(a)) {
        if (memq(car(a), b) == NIL)
            n = cons(car(a), n);
        car(Protected) = n;
        a = cdr(a);
    }
    if (a != NIL && memq(a, b) == NIL)
```

```
        n = cons(a, n);
    unprot(2);
    return n;
}
```

The **freevars()** function returns a list of *free variables* in the *S-expression* x given the bound variables listed in the *environment* e.

The term “S-expression” (“symbolic expression”) is an ancient umbrella term for all LISP expressions and forms, including special forms [McCarthy1960a]. Originally, the term applied only to symbols and pairs, but these days it is more broadly applied to all kinds of LISP data objects. The latter, more liberal usage will be adopted here.

The term “environment” is used rather loosely in the context of the **freevars()** function, because it is really only a list of bound symbols here and does not contain any bindings.

The function uses the **subrp()** predicate, which returns truth, when the symbol x names a *primitive* function. The terms shall be explained later. For now it is only important to know that the names of primitive functions are not considered to be variables when they appear in operator positions in function applications. Hence they cannot be free variables in this case.

The name *SUBR* is short for “subroutine”, another ancient name for functions that were not implemented in LISP, but in assembly language or, later, in various system programming languages [MaCarthy1960b].

Basically, finding free variables is simple [Holm2016], but in the specific case discussed here, there are many different cases to consider.

(1) If x is in e, it is bound; the result is **NIL**.

(2) If x is a symbol (and not in e), it is free; the result is (x), a singular list containing x.

(3) If x is an atom, but not a symbol, it is not a variable at all; the result is **NIL**.

(4) If x is an application of **quote**, the result is **NIL**. Quoted objects can never be free variables.

(5) If x is an application of **--** (a comment), the result is **NIL**.

(6) If x is an application of **apply**, **prog**, **if**, **if***, **setq**, or an application of any primitive function, the result is the union of the free variables of $cdr(x)$.

(7) If x is an application of **def** or **macro**, the result is the union of the free variables of $cddr(x)$. The symbol being *defined* in such a form ($cadr(x)$) can never denote a free variable, exactly *because* it is being defined by the form.

(8) If x is a lambda function, then the variables bound by it will be added to e and with that environment in effect, the result will be the union of the free variables of the body, $cddr(x)$, of the function. (The original environment will be restored after finding the free variables of the function).

Note that all the cases that compute the union of the free variables of some subexpression of x do so by "falling through" to the **while** loop at the end of **freevars()**. The function recurses only through that loop in order to collect free variables of subexpressions.

x	e	Result	Notes
`foo`	`nil`	`(foo)`	
`bar`	`(bar)`	`nil`	*bar* is bound
`(quote foo)`	`nil`	`nil`	
`(f 1 2)`	`nil`	`(f)`	1,2 are not symbols
`(+ x y)`	`nil`	`(x y)`	+ is a primitive operator
`(lambda (x) (f (g x)))`	`nil`	`(f g)`	x is bound in lambda

Fig. 30 – Finding free variables, examples

Figure 30 lists the results of some sample applications of **freevars()**.

```
int     subrp(cell x);

cell freevars(cell x, cell e) {
    cell    n, u, a;
```

```
int     lam;

lam = 0;
if (memq(x, e) != NIL) {
    return NIL;
}
else if (symbolp(x)) {
    return cons(x, NIL);
}
else if (!pairp(x)) {
    return NIL;
}
else if (car(x) == S_quote) {
    return NIL;
}
else if (car(x) == S_apply ||
         car(x) == S_prog ||
         car(x) == S_if ||
         car(x) == S_ifstar ||
         car(x) == S_setq
) {
    x = cdr(x);
}
else if (car(x) == S_def ||
         car(x) == S_macro
) {
    x = cddr(x);
}
else if (car(x) == S_comment) {
    return NIL;
}
else if (subrp(car(x))) {
    x = cdr(x);
}
else if (car(x) == S_lambda) {
    protect(e);
    a = flatargs(cadr(x));
    protect(a);
```

```
        n = set_union(a, e);
        protect(n);
        e = n;
        x = cddr(x);
        lam = 1;
    }
    protect(u = NIL);
    while (pairp(x)) {
        n = freevars(car(x), e);
        protect(n);
        u = set_union(u, n);
        unprot(1);;
        car(Protected) = u;
        x = cdr(x);
    }
    n = unprot(1);
    if (lam) e = unprot(3);
    return n;
}
```

14.2.3 Lambda Functions

This section deals with the transformation of S-expressions representing *functions* (“lambda functions”) to S-expressions representing *closures*. It will also introduce new variables by adding their bindings to the top level environment.

The **posq()** function is like **memq()**, but returns the offset of x in a (or **NIL**). It is used to find the location (slot number) of a variable given its symbol.

New symbols are always added to the end of an environment and the offset of a symbol in the environment equals the slot number that will later be used to access the value of the corresponding variable.

Environments are still lists of symbols in this context!

```
int posq(cell x, cell a) {
    int     n;
    n = 0;
    for (; a != NIL; a = cdr(a)) {
        if (car(a) == x) return n;
        n++;
    }
    return NIL;
}
```

In fact, closure conversion maintains two environments: one containing *bound variables* (*formal arguments*) and another one containing *free variables* in local environments. The top level environment is just the outermost local environment in this context.

Argument symbols are carried along in the variable a and environment symbols in the variable e, where a contains *only* the arguments of the function currently being converted and e contains *only* the free variables of that function.

The **initmap()** function computes an *initmap* for the free variables fv given the arguments a of the immediate outer function (IOF) and the environment e of the IOF. For instance, for $fv = (x\ y)$, $e = (v\ w\ x)$, and $a = (y)$, **initmap()** would return

$((e\ 2\ 0\ x)\ (a\ 0\ 1\ y))$

(The variable names in the initmap will be used in later steps.)

The variables **I_a** and **I_e**, which are defined here (not just declared), will be bound to the symbols “**a**” and “**e**”, which are used to indicate the source vector in initmap entries.

```
cell    I_a, I_e;

cell initmap(cell fv, cell e, cell a) {
    cell    m, n, p;
    int     i, j;

    protect(m = NIL);
    i = 0;
    while (fv != NIL) {
        p = cons(car(fv), NIL);
        protect(p);
        n = mkfix(i);
        p = cons(n, p);
        car(Protected) = p;
        if ((j = posq(car(fv), a)) != NIL) {
            n = mkfix(j);
            p = cons(n, p);
            unprot(1);
            p = cons(I_a, p);
        }
        else if ((j = posq(car(fv), e)) != NIL) {
            n = mkfix(j);
            p = cons(n, p);
            unprot(1);
            p = cons(I_e, p);
        }
        else {
            error("undefined symbol", car(fv));
        }
        m = cons(p, m);
        car(Protected) = m;
        i++;
        fv = cdr(fv);
    }
    return nreverse(unprot(1));
}
```

The **lastpair()** function returns the last pair (cons cell) of the list x. E.g., the last pair of the list **(x y z)** would be **(z)**. When given **NIL**, the function returns **NIL**.

```
cell lastpair(cell x) {
    if (NIL == x) return NIL;
    while (cdr(x) != NIL)
        x = cdr(x);
    return x;
}
```

The variable **Env** is bound to the *top level environment*, where “environment” still indicates a list of symbols without bindings. The variable **Envp** always points to the last pair in **Env**. It is the place where new symbols will be added. This is merely a performance hack; maintaining an append pointer is a constant-time operation, while using **conc** to append symbols would result in quadratic time behavior.

```
cell    Env = NIL,
        Envp = NIL;
```

The **newvar()** function adds the symbol x to the top level environment. If the symbol already is in the environment, it will not be added again.

The **newvars()** function adds multiple symbols at once – those contained in the list x.

Top-level bindings are sometimes introduced implicitly by lambda functions, as the following example illustrates:

```
(lambda (x) (foo x))
```

If the variable *foo* has not been defined before, the above expression will do so. However, it does not store any value in the location bound to *foo*, so the symbol *foo* will be bound at this point, but the variable *foo* will still be undefined. Only a slot has been reserved for it.

This is necessary in order to allow for mutually recursive definitions, as in

```
(defun (f x) (if (pair x) (g x)))
(defun (g x) (f (cdr x)))
```

In this program, g is defined implicitly inside of the definition of f. Only *after* defining f, an explicit definition for g is supplied. The value of g will then be stored in the slot that was pre-allocated in the definition of f.

During conversion of lambda functions, **newvars()** will be used to pre-allocate environment slots.

```
void newvar(cell x) {
    cell    n;

    if (memq(x, Env) != NIL) return;
    if (NIL == Envp) Envp = lastpair(Env);
    n = cons(x, NIL);
    cdr(Envp) = n;
    Envp = n;
}

void newvars(cell x) {
    while (x != NIL) {
        newvar(car(x));
        x = cdr(x);
    }
}
```

The **mapconv()** function maps closure conversion over the list x. In terms of LISP, it does

```
(mapcar (lambda (x) (cconv x e a)) x)
```

The **cconv()** function performs the complete closure conversion. It will be defined at the end of this section.

```
cell cconv(cell x, cell e, cell a);

cell mapconv(cell x, cell e, cell a) {
    cell    n, new;

    protect(n = NIL);
    while (pairp(x)) {
        new = cconv(car(x), e, a);
        n = cons(new, n);
        car(Protected) = n;
        x = cdr(x);
    }
    return nreverse(unprot(1));
}
```

The **lamconv()** ("lambda conversion") function converts an S-expression x, representing a lambda function, to an S-expression representing a closure. The function basically extracts the free variables and arguments of x, builds an initmap, and uses these parts to construct a new S-expression. This process is simple when shown in LISP:

```
(defun (lamconv x e a)
   (let ((fv   (freevars x))
         (args (flatargs (cadr x))))
     (newvars fv)
     @(%closure ,(cadr x)
                ,(initmap fv e a)
                ,@(mapconv (cddr x) fv args))))
```

The corresponding C code has to do a lot of GC protection of intermediate results, though, so it has to be serialized carefully.

The transformation performed by the function is as follows: given $e = (g)$, and $a = (f)$, it converts

```
(lambda (x) (f (g x)))
```

to

```
(%closure (x)
  ((a 0 0 f) (e 0 1 g))
  ((%ref 0) ((%ref 1) (%arg 0))))
```

where the conversion of variable symbols to applications of **%arg** and **%ref** will be done by functions to be discussed later in this chapter.

The **l_closure** variable, which is being defined below, is bound to the symbol **%closure**. This symbol is used to indicate a closure in the S-expression returned by **lamconv()**.

```
cell    I_closure;

cell lamconv(cell x, cell e, cell a) {
    cell    cl, fv, args, m;

    fv = freevars(x, NIL);
    protect(fv);
    newvars(fv);
    args = flatargs(cadr(x));
    protect(args);
    m = initmap(fv, e, a);
    protect(m);
    cl = mapconv(cddr(x), fv, args);
    cl = cons(m, cl);
    cl = cons(cadr(x), cl);
    cl = cons(I_closure, cl);
    unprot(3);
    return cl;
}
```

14.2.4 Lambda Lifting

Lambda lifting [Johnsson1985] is a transformation that converts closures to *combinators*, i.e. to functions without any free variables. The benefit of this transformation is reduced pressure on the *garbage collector*, because no *local environment* has to be created for a function without any free variables. In the ideal case, the application of a combinator will evaluate without allocating any memory at all.

Originally, lambda lifting was called so, because it was supposed to “lift local lambda functions to the global level”. However, this explanations seems inaccurate, which is why a different model will be offered here.

A function being “local” or “global” is unrelated to it being a combinator or not. Local functions can be combinators and top level functions can have free variables. This is why the focus in the following explanation will be on the “combinator” aspect of the

14.3 The Abstract Machine

The remainder of this chapter will deal with code generation, hence it makes sense to introduce the *target architecture* of the LISP9 compiler at this point. The target architecture of a compiler is the *CPU* and *host environment* for which code is being generated. For native code compilers, the CPU is mostly a machine that is cast in silicon and the host environment is an interface to services supplied by an operating system.

The LISP9 compiler targets a virtual architecture called the *LISP9 abstract machine* (*LAM*), which provides both, instructions for evaluating expressions and services that are typically offered by an operating system. It is called a *virtual* architecture, because it does not exist in hardware, but forms an additional layer of abstraction between the hardware CPU and the program being evaluated.

A LISP program is interpreted by the abstract machine and the abstract machine is interpreted by a CPU. The LAM is a *virtual machine* running on top of a "real" (hardware) CPU. In addition, the LAM maps some of its instructions to operating system services rather than instructions of the CPU. Finally, of course, the operating system is also interpreted by the CPU. See figure 32.

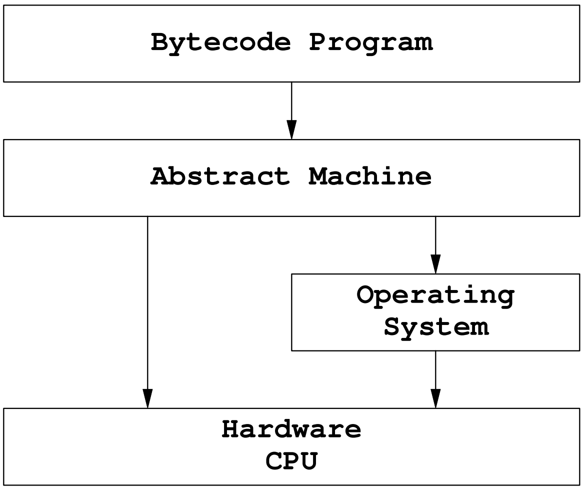


Fig. 32 – Abstraction layers of the LISP9 abstract machine

An abstract machine, like most “real” machines, is defined by its registers, storage, instructions, and other, often language-specific facilities. A common abstract machine for functional programming languages is the *SECD machine* [Landin1964], which consists of four components indicated by the letters of its name:

- stack
- environment
- control
- dump

The “stack” is a push-down list where arguments to instructions and functions as well as intermediate results are stored. The “environment” is an associative array, just like the deep-binding environment of LISP9. “Control” is a list containing the program being evaluated and “dump” is another push-down list where $(stack, environment, control)$ triplets are stored as activation frames for functions whose evaluation has been suspended in order to apply another function. I.e. the “dump” serves as a separate stack holding *continuations*, or points to which control will return after evaluating a function.

The LISP9 abstract machine is only loosely modeled after the SECD machine, which is natural, because the SECD machine is a very abstract model, while the LAM is a concrete implementation. But even then, there are some differences. The LAM has only a single stack that stores both, arguments of function applications and activation frames. The environment is not an associative array, but a set of vectors holding locations, i.e. it uses shallow binding instead of deep binding. Finally, it defines more built-in operations than the original SECD machine.

Many implementations of the SECD machine use data objects that are native to LISP (e.g. [Kogge1991]), maybe because the original design used lambda calculus [Church1941] as its theoretical foundation and/or because it is often used to implement LISP. The LAM also uses LISP data objects in its own implementation. For instance, its stack is a vector, its programs are strings, and its environments are also vectors.

The entire design of the LAM is driven by *simplicity*, *efficiency*, and the attempt to mimic a hardware CPU. Of course, the first two goals are contradictory to some degree, so some compromises have to be made. The last goal is pursued in order to facilitate the translation of LAM code to native machine code.

14.3.1 The Components of the LAM

The LISP9 abstract machine consists of the following components, which are outlined in figure 33:

- an interpreter of LAM instructions (control)
- a set of 7 registers
- a combined control and data stack
- any number of closures
- a top level environment

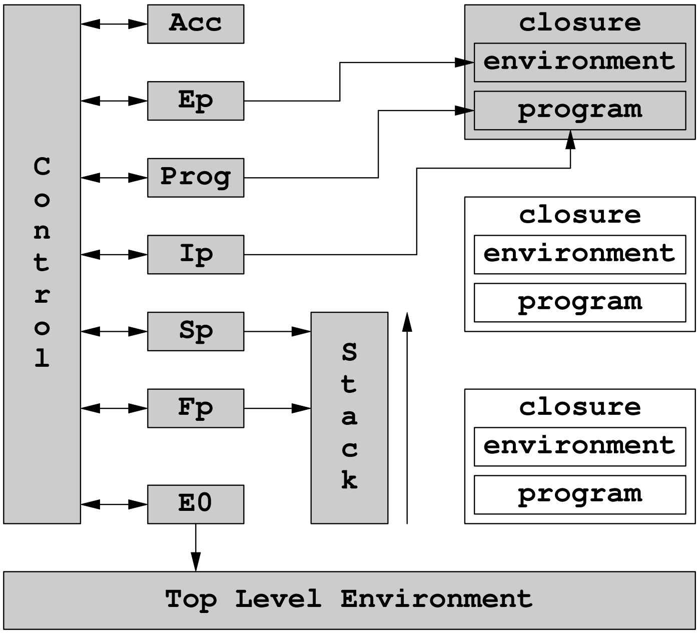


Fig. 33 – The components of the LAM

The definition of the LAM does not include any means of *storage*, except for the stack, the *top level environment* (*TLE*), and the closures. Memory is unbounded in the model: the stack and TLE can grow indefinitely and there can be any number of closures. In practice memory is managed dynamically, because the data structures used by the machine are subject to garbage collection. In the end, of course, the memory of the LAM is limited either by the underlying hardware or by a static limit imposed during compilation of the LAM.

All data objects are bound to variables, either in the TLE, on the stack, or in one of the closures. The symbol table and object table are implementation details and not important when discussing the LAM, which operates at a more concrete (or less abstract) level than those data structures.

The following *registers* exist in the LISP9 abstract machine:

- **Acc** (“accumulator”), used for arguments and result values
- **Ep** (“environment pointer”), points to the current environment
- **Prog** (“program”), points to the current program fragment
- **Ip** (“instruction pointer”), points to the current instruction in **Prog**
- **Sp** (“stack pointer”), points to the top of the stack
- **Fp** (“frame pointer”), points to the current stack frame
- **E0** (“base environment”), points to the top level environment

All registers of the LAM, except for **Acc**, are used to refer to components of the machine. They are typically used to read and modify those parts of the machine.

The *accumulator* Acc is the only general purpose register of the machine. It is used to exchange values with built-in operators. When an operator has one argument, it receives it in the **Acc** register. When there are multiple arguments, the then *first* one will be passed in **Acc**. In any case, the result of the operation will be returned in the accumulator. When an evaluation finishes, its final result will be stored in **Acc**.

The *environment pointer* **Ep** always points to the environment of the closure whose program is currently evaluating. When no closure is evaluating, it points to the top level environment **E0**.

The *program register* **Prog** always points to the program of the closure that is currently evaluating. Of course, a LISP program is usually composed of multiple closures. In the context of the LAM, a program (or, more correctly, a *program fragment*) is the code associated with a specific closure. The terms “program” and “program fragment” are used interchangeably in this section.

When no closure is currently evaluating, **Prog** points to a “free” program that is not contained in any closure. This is considered to be an implementation detail, though. As far as the model is concerned, the “free program” could be part of a separate closure that is reserved for this purpose.

The *instruction pointer* **Ip** points to the *next* instruction to interpret inside of **Prog**. Modifying this register passes control to a different part of the same program.

The *stack pointer* **Sp** points to the element most recently pushed to the *runtime stack* (*RTS*). The stack grows “upward”, i.e. towards larger values of **Sp**. When the stack is empty, the value of **Sp** is -1. The RTS will grow dynamically during evaluation.

The *frame pointer* **Fp** points *into* the stack frame of the most recently applied function (closure). More specifically, it points to the first argument passed to the function.

A *stack frame* is built whenever a function is being applied to values. The following steps will be performed:

- arguments are pushed to the stack in *reverse* (right-to-left) order
- the number of arguments is pushed to the stack
- the environment pointer is pushed to the stack
- the return continuation is pushed to the stack
- the frame pointer is pushed to the stack
- the frame pointer is pointed to the first argument

These steps leave the top of the run time stack in the configuration depicted in figure 34.

Each *closure* in the LAM consists of two components called its *program fragment* and its *local environment*. The local environment is an environment that has been constructed during closure conversion. (Or, rather, *instructions* for building that environment have been generated during conversion.)

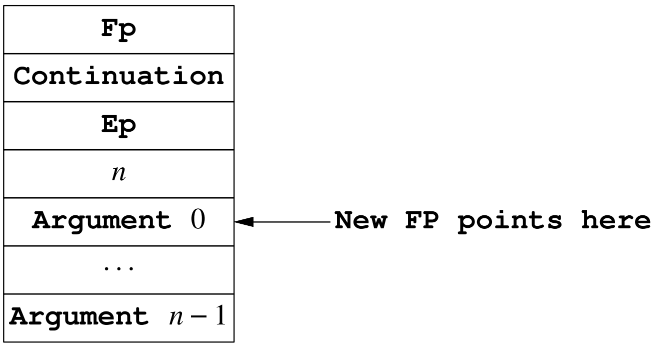


Fig. 34 – Stack frame layout

Whenever a closure is being invoked by passing control to its *program fragment*, the local environment of the closure becomes the current environment, and when the closure is done evaluating, the environment what was in effect before will be restored.

The program fragment of a closure contains instructions for performing a computation. These instructions will be discussed in the next subsection.

In the present implementation, closures contain another piece of information, and that is their *entry point*. Due to the way in which the LISP9 compiler emits code, program fragments will contain instructions that are not part of the computation performed by a closure. The entry point marks the first instruction that actually belongs to the code of the closure.

14.3.2 The Instruction Set of the LAM

The instructions of the LISP9 abstract machine will be described semi-formally in this subsection, using an approach that is loosely based on structural operational semantics [Plotkin1981/2004].

Structural operational semantics describes the meaning of a model of program evaluation at the "microscopic" level, by specifying exactly what each instruction does and how it ts being interpreted. In order to accomplish this goal, the following notation shall be introduced.

21. Derived Operations

```
;;; LISP9 Derived Syntax and Functions
;;; Nils M Holm, 2018,2019
;;; In the public domain
;;;
;;; If your country does not have a concept like
;;; the public domain, the Creative Common Zero
:;; (CC0) license applies, see
;;; https://creativecommons.org/publicdomain/zero/1.0
```

This part of the LISP system is referred to as its *library*, but it is not a library in the sense of a collection of functions which can be compiled into a program on demand. The LISP 9 library is a collection of interdependent functions and macros that will be compiled into the default *image file* of the system. The functions are always present and cannot be loaded separately. When the LISP9 system does not find an image file, it will load the code described in this part from its source file. When the source file is not present, the system cannot start up. All functions and derived special forms described here are a fixed part of the LISP9 language.

21.1 Simple Definitions

Reading the symbols **t** and **nil** at the beginning of this file will store them in *object table* slots 0 and 1. This is not strictly necessary, but facilitates debugging and may also enable certain optimizations, because the indexes 0 and 1 can then be used as truth values. None of these optimizations are performed in this implementation, though.

```
nil t
```

In the library code, a lot of definitions of the form

```
(defun (f x) (f x))
```

will appear. These definitions work only if f is the name of a *primitive function*. Since the LISP9 compiler will inline primitive functions, the application **(f x)** in the body of such a definition will generate inline code and not a function application. For instance, the definition

```
(defun (car x) (car x))
```

will inline a *car* instruction of the LISP9 abstract machine in the body of the **car** function. Hence applying the **car** function will actually perform a “car” operation. Definitions like these are necessary in order to compile expressions like

```
(mapcar car x)
```

where the name **car** does not appear in an operator position. Without redefining **car** as a function, the name **car** would be undefined in the above expression.

Most of these definitions are placed at the end of the library. Some of them are needed in the library code itself, though, and those are listed here.

```
(defun (cons x y) (cons x y))
(defun (car x) (car x))
(defun (cdr x) (cdr x))

(defun (caar x) (caar x))
(defun (cadr x) (cadr x))
(defun (cdar x) (cdar x))
(defun (cddr x) (cddr x))
```

Nested pair accessors with three levels of nesting (**caaar** ... **cdddr**) are also defined here, because they are used frequently in the following code. The single-level and two-level accessors are inlined by the compiler.

```
(defun (caaar x) (car (caar x)))
(defun (caadr x) (car (cadr x)))
```

```
(defun (cadar x) (car (cdar x)))
(defun (caddr x) (car (cddr x)))
(defun (cdaar x) (cdr (caar x)))
(defun (cdadr x) (cdr (cadr x)))
(defun (cddar x) (cdr (cdar x)))
(defun (cdddr x) (cdr (cddr x)))
```

The **list**, **string**, and **vector** functions create the corresponding type of object from a variable number or arguments. Using optional arguments, they are easy to implement.

```
(defun (list . x) x)
(defun (vector . x) (listvec x))
(defun (string . x) (liststr x))
```

The **rever** function reverses its argument, which must be a (proper) list. **Nrever** does the same, more efficiently, but at the cost of destroying its argument. They are easily implemented on top of **reconc** and **nreconc**.

```
(defun (rever a)
  (reconc a nil))

(defun (nrever a)
  (nreconc a nil))
```

21.2 Bootstrapping

A lot of definitions in the LISP9 library depend on each other, which creates the classical problem of *bootstrapping*: definition A depends on definition B and B depends on A. In the case of derived special forms, there is an easy (but cumbersome) way around: use a manually expanded form of B in A. The unwieldiness of manually expanded definitions can be mitigated by introducing definitions gradually. For instance, even a simplified **let** or **cond** special form can be very helpful, and the full implementation can be postponed until more expressive tools are available.

The same approach can be used with functions. For instance, a simplified **mapcar** function will be defined early in the bootstrapping process. It is sufficient for all tasks to be solved in the library itself, and the full implementation only appears rather late in the process.

As a first step in the bootstrapping process, a simplified form of **cond** is defined. This variant accepts only clauses of the form

```
(predicate expression ...)
```

and expands as follows:

```
(cond ($p_1$  $x_{1,1}$ $\cdots$)    $\rightarrow$  (if $p_1$
       $\cdots$                          (prog $x_{1,1}$ $\cdots$)
      ($p_n$  $x_{n,1}$ $\cdots$))           (if $\cdots$
                                   $\cdots$
                                     (if $p_n$
                                         (prog $x_{n,1}$ $\cdots$)
                                         nil) $\cdots$ ))
```

The predicate of a clause may be the keyword **else**, which is always true. Such a clause expands in this way:

```
(else $x_1$ ...)  $\rightarrow$  (prog $x_1$ ...)
```

The definition of a minimal **cond** special form is straightforward, but the expanded form has to be constructed by using **list** and **cons**, because quasiquotation is not yet available at this point. Once it is, a complete **cond** macro will be defined.

```
(defmac (cond . cs)
  (if (null cs)
      nil
      (if (eq 'else (caar cs))
          (cons 'prog (cdar cs))
          (list 'if (caar cs)
                    (cons 'prog (cdar cs))
                    (cons 'cond (cdr cs))))))
```

The **and** special form implements the *short-circuit logical and*, i.e. it returns the first of its arguments that is **nil** or the last argument when no argument is **nil** (and **t**, if no arguments are given). It implements a *short-circuit* operator, because it will not evaluate any further arguments as soon as the result can no longer change: when one argument is **nil**, it does not make any sense to evaluate anything that follows.

The **and** special form expands in the following way:

```
(and x₁ x₂ ··· xₙ)  →  (if x₁
                           (and x₂ ··· xₙ)
                          nil)
```

Multiple arguments expand recursively.

```
(defmac (and . xs)
  (cond ((null xs) 't)
        ((null (cdr xs)) (car xs))
        (else (list 'if (car xs)
                         (cons 'and (cdr xs))
                        nil))))
```

Quasiquotation is a convenient tool in the transformation of derived special forms. It is similar to quotation, but allows to insert dynamic elements into list and vector structures. The structure being processed by quasiquotation is called a *quasiquotation template*.

Quasiquotation consists of the **qquote** special form and the **unquote** and **splice** subforms, which are only valid inside of **qquote**. The following abbreviations are commonly used:

```
 @x  =  (qquote x)
 ,x  =  (unquote)
,@x  =  (splice x)
```

Note that Scheme and Common Lisp would use the backtick (‘) character in the place of “@”, but not in the place of “**,**@”. E.g., it would use **‘(f ,@x)** in the place of **@(f ,@x)**. The backtick character can also be used in LISP9.

Given the above abbreviations, this is how quasiquotation works formally:

```
     @x   =  'x
    @,x   =  x
  @(x y)  =  (cons @x @(y))
 @(,x y)  =  (cons x @(y))
@(,@x y)  =  (conc x @(y))
```

Note particularly that there is a **conc** in the last equation, but a **cons** in the other formulae. The difference might not be obvious, so here are some examples:

```
@(1 ,(list 2 3) 4)
```

would expand to

```
(cons '1 (cons (list 2 3) (cons '4 'nil)))
  =>  (1 (2 3) 4)
```

but

```
@(1 ,@(list 2 3) 4)
```

would expand to

```
(cons '1 (CONC (list 2 3) (cons '4 'nil)))
  =>  (1 2 3 4)
```

That is, the **splice** operator *splices* multiples values into the place of its application instead of inserting a list of values (which is what the **unquote** operator would do).

Because **qquote** expands recursively, it can be used to insert elements into templates of any complexity. For instance, given a list of variables $vs = (x\ y\ z)$ and a list of arguments $as = (A\ B\ C)$ and a list of expressions $body = (x_1\ x_2\ x_3)$, **qquote** can be used to create an S-expression denoting function application:

```
@((lambda ,vs ,@body) ,@as)
  =>  ((lambda (x y z) x1 x2 x3) A B C)
```

Note that splicing is not really necessary at the end of a list, so the following template would also work:

```
@((lambda ,vs . ,body) . ,as)
```

Splicing only has to be used when multiple values are to be inserted into a list where further list elements follow after the place where the insertion takes place. E.g.:

```
@(a b . (unquote '(0 1)))   =>   (a b 0 1)
@(a (splice '(0 1)) b)      =>   (a 0 1 b)
@((splice '(0 1)) a b)      =>   (0 1 a b)
```

See the definition of **labels** for a more practical example.

The **qquote** special form itself is implemented as an ordinary macro. Only support for the abbreviations listed above has to be provided by the reader.

The code is straight-forward, it basically implements the formal rules outlined earlier in this description. A special case that was not discussed above, though, is the use of a vector in a template. This case is handled by converting the vector to a list, expanding the list, and then converting it back to a vector.

```
(defmac (qquote x)
  (cond ((vectorp x)
          (list 'listvec
                (list 'qquote (veclist x))))
        ((not (pair x))
          (list 'quote x))
        ((eq 'unquote (car x))
          (cadr x))
        ((and (pair (car x))
              (eq 'unquote (caar x)))
          (list 'cons (cadar x)
                      (list 'qquote (cdr x))))
        ((and (pair (car x))
              (eq 'splice (caar x)))
          (list 'conc (cadar x)
                      (list 'qquote (cdr x))))
        (else
          (list 'cons (list 'qquote (car x))
                      (list 'qquote (cdr x))))))
```

30. List of Figures