

For the sake of illustration we here assumed that we can use arithmetic on integers in the lambda calculus, although this was not included in the syntax above.

In fact, the natural numbers (non-negative integers with addition, subtraction, multiplication, and test for zero) can be encoded as so-called Church numerals as follows (and also in a number of other ways):

| | | |
|-------|----|---------------------------------|
| zero | is | $\lambda f.\lambda x.x$ |
| one | is | $\lambda f.\lambda f.f x$ |
| two | is | $\lambda f.\lambda f.f(f x)$ |
| three | is | $\lambda f.\lambda f.f(f(f x))$ |

and so on. Then successor (+1), addition and multiplication may be defined as follows:

| | | |
|------|----|------------------------------------------------------|
| succ | is | $\lambda m.\lambda f.\lambda x.f(m f x)$ |
| add | is | $\lambda m.\lambda n.\lambda f.\lambda x.m f(n f x)$ |
| mul | is | $\lambda m.\lambda n.\lambda f.\lambda x.m(n f) x$ |

Some of these encodings are possible only in the untyped lambda calculus, so the absence of type restrictions is important. In particular, the pure *simply typed* lambda calculus cannot encode unbounded iteration or recursion.

There is a rich literature on the lambda calculus, not least Henk Barendregt's comprehensive monograph [10].

Several different evaluation strategies are possible for the untyped lambda calculus. To experiment with some encodings and evaluation strategies, you may try an online lambda calculus reducer [58].

Chapter 6

Imperative languages

This chapter discusses *imperative programming languages*, in which the value of a variable can be modified by assignment. We first present a naive imperative language where a variable denotes an updatable store cell, and then present the environment/store model used in real imperative programming languages. Then we show how to evaluate micro-C, a C-style imperative language, using an interpreter, and present the concepts of expression, variable declaration, assignment, loop, output, variable scope, lvalue and rvalue, parameter passing mechanisms, pointer, array, and pointer arithmetics.

6.1 What files are provided for this chapter

| File | Contents |
|--------------------|-------------------------------------------|
| imp/Naivestore.sml | naive store model |
| imp/imp.sml | naive imperative language interpreter |
| imp/Sto.sml | a store mapping locations to values |
| imp/Absyn.sml | micro-C abstract syntax (Figure 6.6) |
| imp/grammar.txt | informal micro-C grammar |
| imp/Clex.lex | micro-C lexer specification |
| imp/Cpar.grm | micro-C parser specification |
| imp/parse.sml | micro-C parser |
| imp/c.sml | micro-C interpreter (Section 6.6) |
| imp/ex1.c-ex21.c | micro-C example programs (Figure 6.8) |
| imp/Parameters.cs | call-by-reference parameters in C# |
| imp/array.c | array variables and array parameters in C |

6.2 A naive imperative language

We start by considering a naive imperative language (file `imp/imp.sml`). It has expressions as shown in Figure 6.1, and statements as shown in Figure 6.2: assignment, conditional statements, statement sequences, for-loops, while-loops and a print statement.

```
datatype expr =
  CstI of int
  | Var of string
  | Prim of string * expr * expr
```

Figure 6.1: Abstract syntax for expressions

```
datatype stmt =
  Asgn of string * expr
  | If of expr * stmt * stmt
  | Block of stmt list
  | For of string * expr * expr * stmt
  | While of expr * stmt
  | Print of expr
```

Figure 6.2: Abstract syntax for statements

Variables are introduced as needed, as in sloppy Perl programming; there are no declarations. Unlike *C/C++/Java/C#*, the language has no blocks to delimit variable scope, only statement sequences.

For-loops are as in Pascal or Basic, not *C/C++/Java/C#*, so a for loop has the form

```
for i = startval to endval do
  stmt
```

where `start` and `end` values are given for the controlling variable `i`, and the controlling variable cannot be changed inside the loop.

The store naively maps variable names to values; see Figure 6.3. This is similar to a functional language, but completely unrealistic for imperative languages.

The distinction between *statement* and *expression* has been used in imperative languages since the very first one, Fortran in 1956. The purpose of executing a statement is to modify the state of the computation (by modifying the

Naivestore

| | |
|---|----|
| a | 11 |
| b | 22 |
| y | 22 |

Figure 6.3: Naive store, a direct mapping of variables to values

store, by producing some output, or similar). The purpose of evaluating an expression is to compute a value. In most imperative languages, the evaluation of an expression can modify the store also, by a so-called *side effect*.

In Standard ML, there are no statements, state changes are produced by expressions that have side effects, such as `print "Hello!"`.

In Postscript, there are no expressions, so values are computed by statements (instruction sequences) that leave a result of the stack top, such as `4 5 add 6 mul`.

6.3 Environment and store

Real imperative languages such as C, Pascal and Ada, and imperative object-oriented languages such as C++, Java, C# and Ada95, have a more complex state (or store) model than functional languages:

- An environment maps variable names (`x`) to store locations (`0x34B2`)
- An updatable store maps locations (`0x34B2`) to values (`117`).

It is useful to distinguish two kinds of values in such languages. When a variable `x` or array element `a[i]` occurs as the target of an assignment statement:

```
x = e
```

or as the operand of an increment operator (in *C/C++/Java/C#*):

```
x++
```

or as the operand of an address operator (in *C/C++/C#*; see below):

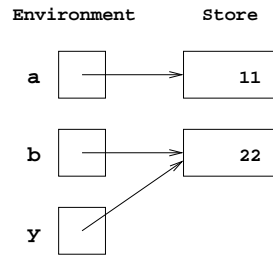


Figure 6.4: Environment (variable to location) and store (location to value)

$\&x$

then we use the *lvalue* ('left hand side value') of the variable or array element. The *lvalue* is the location (or address) of the variable or array element in the store.

Otherwise, when the variable x or array element $a[i]$ occurs in an expression such as this:

$x + 7$

then we use its *rvalue* ('right hand side value'). The *rvalue* is the value stored at the variable's location in the store. Only expressions that have a location in the store can have an *lvalue*. Thus in C/C++/Java/C# this expression makes no sense:

$(8 + 2)++$

because the expression $(8 + 2)$ has an *rvalue* (10) but does not have an *lvalue*.

In other words, the environment maps names to *lvalues*; the store maps *lvalues* to *rvalues*; see Figure 6.4.

When we later study the compilation of imperative programs to machine code (Chapter 7), we shall see that the environment exists only at compile-time, when the code is generated, and the store exists only at run-time, when the code is executed.

In all imperative languages the store is *single-threaded*: at most one copy of the store needs to exist at a time. That is because we never need to look back (for instance, by discarding all changes made to the store since a given point in time).

6.4 Parameter passing mechanisms

In a declaration of a procedure (or function or method)

```
void p(int x, double y) { ... }
```

the x and y are called *formal parameters* or just *parameters*. In a call to a procedure (or function or method)

```
p(e1, e2)
```

the expressions $e1$ and $e2$ are called *actual parameters*, or argument expressions.

When executing a procedure call $p(e1, e2)$ in an imperative language, the values of the argument expressions must be bound to the formal parameters x and y somehow. This so-called parameter passing can be done in several different ways:

- Call-by-value: a copy of the argument expression's value (*rvalue*) is made in a new location, and the new location is passed to the procedure. Thus updates to the corresponding formal parameter does not affect the actual parameter (argument expression).
- Call-by-reference: the location (*lvalue*) of the argument expression is passed to the procedure. Thus updates to the corresponding formal parameter will affect the actual parameter. Note that the actual parameter must have an *lvalue*. Usually this means that it must be a variable or an array element (or a field of an object or structure).
Call-by-reference is useful for returning multiple results from a procedure. It is also useful for writing recursive functions that modify trees, so some binary tree algorithms can be written more elegantly in languages that support call-by-reference (including Pascal, C++ and C#) than in Java (that does not).
- Call-by-value-return: a copy of the argument expression's value (*rvalue*) is made in a new location, and the new location is passed to the procedure. When the procedure returns, the current value in that location is copied back to the argument expression (if it has an *lvalue*).

Pascal, C++, C#, and Ada permit both call-by-value and call-by-reference. Fortran (at least some versions) uses call-by-value-return.

Java, C, and Standard ML permit only call-by-value, but in C (and micro-C) one can pass variable x by reference just by passing the address $\&x$ of x and

making the corresponding formal parameter `xp` be a pointer. Note that Java does not copy objects and arrays when passing them as parameters, because it passes (and copies) only references to objects and arrays [89]. When passing an object by value in C++, the object gets copied. This is often not what is intended. For instance, if the object being passed is a file descriptor, the result is unpredictable.

Here are a few examples (in C#, see file `imp/Parameters.cs`) to illustrate the difference between call-by-value and call-by-reference parameter passing.

The method `swapV` uses call-by-value:

```
static void swapV(int x, int y) {
    int tmp = x; x = y; y = tmp;
}
```

Putting `a = 11` and `b = 22`, and calling `swapV(a, b)` has no effect at all on the values of `a` and `b`. In the call, the value 11 is copied to `x`, and 22 is copied to `y`, and they are swapped so that `x` is 22 and `y` is 11, but that does not affect `a` and `b`.

The method `swapR` uses call-by-reference:

```
static void swapR(ref int x, ref int y) {
    int tmp = x; x = y; y = tmp;
}
```

Putting `a = 11` and `b = 22`, and calling `swapR(ref a, ref b)` will swap the values of `a` and `b`. In the call, parameter `x` is made to point to the same address as `a`, and `y` to the same as `b`. Then the contents of the locations pointed to by `x` and `y` are swapped, which swaps the values of `a` and `b` also.

The method `square` below uses call-by-value for its `i` parameter and call-by-reference for its `r` parameter. It computes `i*i` and assigns the result to `r` and hence to the actual argument passed for `r`:

```
static void square(int i, ref int r) {
    r = i * i;
}
```

After the call `square(11, ref z)`, variable `z` has the value 121. Compare with the micro-C example in file `imp/ex5.c`: it passes an `int` pointer `r` by value instead of passing an integer variable by reference.

6.5 The C programming language

The C programming language [55], designed by Kernighan and Ritchie, USA in the early 1970s, is widely used, and its syntax is used in C++, Java, and C#.

The C programming language descends from B (designed by Brian Kernighan and Ken Thompson at MIT and Bell Labs 1971), which descends from BCPL (designed by Martin Richards at Cambridge UK and MIT, 1967), which descends from CPL, a research language designed by Christopher Strachey and others (at Cambridge UK, early 1960s). The ideas behind CPL also influenced other languages, such as Standard ML.

The primary aspects of C modelled here are functions (procedures), parameter passing, arrays, pointers, and pointer arithmetics. The language presented here has no type checker (so far) and therefore is quite close to B, which was untyped.

6.5.1 Integers, pointers and arrays in C

A variable `i` of type `int` may be declared as follows:

```
int i;
```

This reserves storage for an integer, and introduces the name `i` for that storage location. The integer is not initialized to any particular value.

A pointer `p` to an integer may be declared as follows:

```
int *p;
```

This reserves storage for a pointer, and introduces the name `p` for that storage location. It does not reserve storage for an integer. The pointer is not initialized to any particular value. A pointer is a store address, essentially. The integer pointed to by `p` (if any) may be obtained by dereferencing the pointer:

```
*p
```

An attempt to dereference an uninitialized pointer is likely to cause a Segmentation fault (or Bus error, or General protection fault), but it may instead just return an arbitrary value, which can give nasty surprises.

A dereferenced pointer may be used as an ordinary value (an `rvalue`) as well as the destination of an assignment (an `lvalue`):

```
i = *p + 2;
*p = 117;
```

A pointer to an integer variable `i` may be obtained by using the address operator (`&`):

```
p = &i;
```

This assignment makes `*p` an alias for the variable `i`. The dereferencing operator (`*`) and the address operator (`&`) are inverses, so `*&i` is the same as `i`, and `&*p` is the same as `p`.

An array `ia` of 10 integers can be declared as follows:

```
int ia[10];
```

This reserves a block of storage with room for 10 integers, and introduces the name `ia` for the storage location of the first of these integers. Thus `ia` is actually a pointer to an integer. The elements of the array may be accessed by the subscript operator `ia[...]`, so

```
ia[0]
```

refers to the location of the first integer; thus `ia[0]` is the same as `*ia`. In general, since `ia` is a pointer, the subscript operator is just an abbreviation for dereferencing in combination with so-called *pointer arithmetics*. Thus

```
ia[k]
```

is the same as

```
*(ia+k)
```

where `(ia+k)` is simply a pointer to the `k`'th element of the array, obtained by adding `k` to the location of the first element, and clearly `*(ia+k)` is the contents of that location.

6.5.2 Type declarations in C

In C, type declarations for pointer and array types have a tricky syntax, where the type of a variable `x` surrounds the variable name:

| Declaration | Meaning |
|---------------------------|--------------------------------------------------------|
| <code>int x</code> | <code>x</code> is an integer |
| <code>int *x</code> | <code>x</code> is a pointer to an integer |
| <code>int x[10]</code> | <code>x</code> is an array of 10 integers |
| <code>int x[10][3]</code> | <code>x</code> is an array of 10 arrays of 3 integers |
| <code>int *x[10]</code> | <code>x</code> is an array of 10 pointers to integers |
| <code>int *(x[10])</code> | <code>x</code> is an array of 10 pointers to integers |
| <code>int (*x)[10]</code> | <code>x</code> is a pointer to an array of 10 integers |
| <code>int **x</code> | <code>x</code> is a pointer to a pointer to an integer |

The C type syntax is so obscure that there is a standard Unix/Linux program called `cdecl` to help explain it. For instance,

```
cdecl explain "int *x[10]"
```

prints

```
declare x as array 10 of pointer to int
```

By contrast,

```
cdecl explain "int (*x)[10]"
```

prints

```
declare x as pointer to array 10 of int
```

The expression syntax for pointer dereferencing and array access is consistent with the declaration syntax, so if `ipa` is declared as

```
int *ipa[10]
```

then `*ipa[2]` means the (integer) contents of the location pointed to by element 2 of array `ipa`, that is, `*(ipa[2])`, or in pure pointer notation, `*(*(ipa+2))`.

Similarly, if `iap` is declared as

```
int (*iap)[10]
```

then `(*iap)[2]` is the (integer) contents of element 2 of the array pointed to by `iap`, or in pure pointer notation, `*(*(iap)+2)`.

Beware that the C compiler will not complain about the expression

```
*iap[2]
```

which means something quite different, and most likely not what one intends. It means `*(*(iap+2))`, that is, add 2 to the address `iap`, take the contents of that location, use that contents as a location, and get its contents. This may cause a Segmentation fault, or return arbitrary garbage.

This is one of the great risks of C: neither the type system (at compile-time) nor the runtime system provide much protection for the programmer.

```

void main(int i) {
    int r;
    fac(i, &r);
    print r;
}

void fac(int n, int *res) {
    print &n;           // Show n's address
    if (n == 0)
        *res = 1;
    else {
        int tmp;
        fac(n-1, &tmp);
        *res = tmp * n;
    }
}

```

Figure 6.5: A micro-C program to compute and print factorial of *i*

6.6 The micro-C language

Micro-C is a small subset of the C programming language, but large enough to illustrate notions of evaluation stack, arrays, pointer arithmetics, and so on. Figure 6.5 shows a small program in micro-C (file `imp/ex9.c`).

The recursive function `fac` computes the factorial of `n` and returns the result using a pointer `res` to the variable `r` in the `main` function.

The abstract syntax of micro-C considerably more complex than that of the functional languages and the naive imperative language. It is shown in Figure 6.6 and in file `imp/Absyn.sml`.

As in real C, a micro-C program is a list of top-level declarations. A top-level declaration is either a function declarations or a variable declaration. A function declaration (`Fundec`) consists of an optional return type, a function name, a list parameters (type and parameter name), and a function body which is a statement. A variable declaration (`Vardec`) consists of a type and a name.

A statement (`stmt`) is an `if`-, `while`-, `expression`-, `return`- or `block`-statement. An expression statement `e;` is an expression followed by a semicolon as in C, C++, Java and C#. A block statement is a list of statements or declarations.

All expressions have an `rvalue`, but only three kinds of expressions have an `lvalue`: a variable `x`, a pointer dereferencing `*p`, and an array element `a[e]`. An expression of one of these forms may be called an *access expression*; such

```

datatype constant =
    CstI of int                (* Integer constant *)
  | CstN                       (* The null pointer *)
datatype typ =
    TypI                       (* Type int *)
  | TypC                       (* Type char *)
  | TypA of typ * int option   (* Array type *)
  | TypP of typ                (* Pointer type *)
and expr =
    Access of access          (* Variable or element access *)
  | Assign of access * expr   (* Assignment to var or elt *)
  | Addr of access           (* Get address *)
  | Cst of constant          (* Constant *)
  | Prim1 of string * expr   (* Unary primitive operator *)
  | Prim2 of string * expr * expr (* Binary primitive operator *)
  | Andalso of expr * expr   (* Sequential and *)
  | Orelse of expr * expr    (* Sequential or *)
  | Call of string * expr list (* Function call f(...) *)
and access =
    AccVar of string          (* Variable access x *)
  | AccDeref of expr         (* Dereferencing of a pointer *p *)
  | AccIndex of access * expr (* Array indexing a[e] *)
and stmt =
    If of expr * stmt * stmt (* Conditional *)
  | While of expr * stmt     (* While-loop *)
  | Expr of expr             (* Expression (as in C or Java) *)
  | Return of expr option    (* Return from method *)
  | Block of stmtordec list  (* Block: grouping and scope *)
and stmtordec =
    Dec of typ * string      (* Declaration of local variable *)
  | Stmt of stmt             (* A statement *)
and topdec =
    Fundec of typ option * string * (typ * string) list * stmt
  | Vardec of typ * string
and program =
    Prog of topdec list

```

Figure 6.6: Abstract syntax of micro-C

expressions are represented by the type `access` in the abstract syntax.

An expression (`expr`) may be a variable `x`, a pointer dereferencing `*p`, or an array element access `a[e]`. The value of such an expression is the rvalue of the access expression (`x` or `*p` or `a[e]`).

An expression may be an assignment `x=e` to a variable, or an assignment `*p=e` to a pointed-to cell, or an assignment `a[e]=e` to an array element. The assignment uses the lvalue of the access expression (`x` or `*p` or `a[e]`).

An expression may be an application `&a` of the address operator to an expression `a`, which must have an lvalue and so must be an access expression.

An expression may be a constant (an integer literal or the null pointer literal); or an application of a primitive; or a short-cut logical operator `e1 && e2` or `e1 || e2`; or a function call.

A micro-C type is either `int` or `char` or array `t[]` with element type `t`, or pointer `t*` to a value of type `t`.

6.6.1 Interpreting micro-C

File `imp/c.sml` and other files mentioned in Section 6.1 provide an interpretive implementation of micro-C, a tiny subset of the C programming language. The interpreter's state is split into environment and store as described in Section 6.3. Variables must be explicitly declared (as in C), but there is no type checking (as in B). The scope of a variable extends to the end of the innermost block enclosing its declaration. In the interpreter, the environment is used to keep track of variable scope and the next available store location, and the store keeps track of the locations' current values.

We do not model the `return` statement in micro-C functions because it represents a way to abruptly terminate the execution of a sequence of statements. This is easily implemented by translation to a stack machine (Chapter 7), or by using a continuation-based interpreter (Chapter 8), but it is rather cumbersome to encode in the direct-style interpreter in `imp/c.sml`.

The main functions of the direct-style micro-C interpreter are shown in Figure 6.7.

Later we shall compile micro-C to bytecode for a stack machine (Chapter 7), and even to bytecode for the Java virtual machine (Chapter 10).

6.6.2 Example programs in micro-C

Several micro-C example programs illustrate various aspects of the language, the interpreter (Section 6.6) and the compilers presented in later chapters. The example programs are summarized in Figure 6.8.

```
run : program -> int list -> int sto
    Execute an entire micro-C program by initializing global variables and
    then calling the program's main function.

exec : stmt -> lenv -> genv -> int sto -> int sto
    Execute a micro-C statement stmt in the given environments and store,
    producing an updated store.

stmtordec : stmtordec -> lenv -> genv -> int sto -> lenv * int sto
    Execute a micro-C statement or declaration (as found in a statement
    block { ... }), producing an updated local environment and an up-
    dated store.

eval : expr -> lenv -> genv -> int sto -> int * int sto
    Evaluate a micro-C expression expr in the given environments and store,
    producing a result (an integer) and a possibly updated store.

access : access -> lenv -> genv -> int sto -> int * int sto
    Evaluate a micro-C access expression (variable x, pointer dereferencing
    *p, or array indexing a[e]), producing a result (an int, representing a
    number or address), and a possibly updated store.

allocate : typ * string -> lenv -> int sto -> venv * int sto
    Given a micro-C type and a variable name, bind the variable in the
    given environments and set aside space for it in the given store, pro-
    ducing an updated environment and an updated store.
```

Figure 6.7: Main functions of the micro-C interpreter. A `lenv` is a pair of a (local) environment and a counter indicating the next free store location. A `genv` is a global environment: a pair of an environment for global variables and an environment for global functions.

| File | Contents, illustration | Use |
|--------|-------------------------------------------------------------------|-----|
| ex1.c | while-loop that prints the numbers $n, n-1, n-2, \dots, 1$ | ICJ |
| ex2.c | declaring and using arrays and pointers | IC |
| ex3.c | while-loop that prints the numbers $0, 1, 2, \dots, n-1$ | ICJ |
| ex4.c | compute and print array of factorials $0!, 1!, 2!, \dots, (n-1)!$ | IC |
| ex5.c | compute square, return result via pointer; nested blocks | IC |
| ex6.c | recursive factorial function; returns result via pointer | IC |
| ex7.c | infinite while-loop, followed by dead code | ICJ |
| ex8.c | while-loop that performs 20 million iterations | ICJ |
| ex9.c | recursive factorial function; returns result via pointer | IC |
| ex10.c | recursive factorial function with ordinary return value | C |
| ex11.c | find all solutions to the n -queens problem | ICJ |
| ex12.c | perform n tail calls | C |
| ex13.c | decide whether n is a leap year; logical 'and' and 'or' | ICJ |
| ex14.c | compute integer square root; globally allocated integer | C |
| ex15.c | perform n tail calls and print $n, n-1, \dots, 2, 1, 999999$ | IC |
| ex16.c | conditional statement with empty then-branch | ICJ |
| ex17.c | call the Collatz function on arguments $0, 1, 2, \dots$ | C |
| ex18.c | nested conditional statements; backwards compilation | ICJ |
| ex19.c | conditional badly compiled by forwards compiler | ICJ |
| ex20.c | compilation of a logical expression depends on context | ICJ |
| ex21.c | the tail call optimization is unsound in micro-C | IC |

Figure 6.8: Example programs in micro-C; n is a command line argument. Examples marked I can be executed by the micro-C interpreter (Section 6.6); those marked C can be compiled to the micro-C stack machine (Chapter 7 and Chapter 9); and those marked J can be compiled to bytecode for the Java Virtual Machine (Section 10.2).

6.7 Notes on Strachey's *Fundamental concepts*

Christopher Strachey's lecture notes *Fundamental Concepts in Programming Languages* [97] from the Copenhagen Summer School on Programming in 1967 were circulated in manuscript and highly influential, although they were not formally published until 25 years after Strachey's death. They are especially noteworthy for introducing concepts such as lvalue, rvalue, ad hoc polymorphism, and parametric polymorphism, that shape our ideas of programming languages even today. Moreover, a number of the language constructs discussed in the notes made their way into CPL, and hence into BCPL, B, C, C++, Java, and C#.

Here we discuss some of the subtler points in Strachey's notes:

- The CPL assignment:

```
i := (a > b -> j, k)
```

naturally corresponds to this assignment in C, C++, Java, C#:

```
i = (a > b ? j : k)
```

Symmetrically, the CPL assignment:

```
(a > b -> j, k) := i
```

can be expressed in GNU C (the gcc compiler) like this:

```
(a > b ? j : k) = i
```

and it can be encoded using pointers and the dereferencing and address operators in all versions of C and C++:

```
*(a > b ? &j : &k) = i
```

In Java, C#, and standard (ISO) C, conditional expressions cannot be used as lvalues. In fact the GNU C compiler (gcc -c -pedantic assign.c) says:

```
warning: ISO C forbids use of conditional expressions as lvalues
```

- The CPL definition in Strachey's section 2.3:

```
let q =- p
```

defines the lvalue of `q` to be the lvalue of `p`, so they are aliases. This feature exists in C++ in the guise of an *initialized reference*:

```
int& q = p;
```

Probably no other language can create aliases that way, but call-by-reference parameter passing has exactly the same effect. For instance, in C#:

```
void m(ref int q) { ... }
... m(ref p) ...
```

When `q` is a formal parameter and `p` is the corresponding argument expression, then the lvalue of `q` is defined to be the lvalue of `p`.

- The semantic functions `L` and `R` in Strachey's section 3.3 are applied only to an expression ϵ and a store σ , but should in fact be applied also to an environment, as in our `imp/c.sml`, if the details are to work out properly.
- Note that the CPL block delimiters `§` and `§` in Strachey's section 3.4.3 are the grandparents (via a BCPL and B) of C's block delimiters `{` and `}`. The latter are used also in C++, Java, Javascript, Perl, C#, and so on.
- The discussion in Strachey's section 3.4.3 (of the binding mechanism for the free variables of a function) can appear rather academic until one realizes that in Standard ML a function closure always stores the rvalue of free variables, whereas in Java an object stores essentially the lvalue of fields that appear in a method. In Java an instance method (non-static method) `m` can have as 'free variables' the fields of the enclosing object, and the methods refer to those fields via the object reference `this`. As a consequence, subsequent assignments to the fields affect the (r)value seen by the field references in `m`.

Moreover, when a Java method `mInner` is declared inside a local inner class `CInner` inside a method `mOuter`, then Java requires the variables and parameters of method `mOuter` referred to by `mInner` to be declared `final` (not updatable):

```
class COuter {
  void mOuter(final int p) {
    final int q = 20;
```

```
class C {
  void mInner() {
    ... p ... q ...
  }
}
}
```

In reality the rvalue of these variables and parameters is passed, but when the variables are non-updatable, there is no observable difference between passing the lvalue and the rvalue. Thus the purpose of this 'final' restriction in Java is to make free variables from the enclosing method appear to behave the same as free fields from the enclosing object.

C# does not have local classes, but C# version 2.0 has anonymous methods, and in contrast to Java's inner classes and Standard ML's function closures, these anonymous methods capture the lvalue of the enclosing method's local variables. Since an anonymous method may outlive the call to the method that created it, such captured local variables must be allocated in an object on the heap.

- The type declaration in Strachey's section 3.7.2 is quite cryptic, but roughly corresponds to this declaration in Standard ML:

```
datatype LispList =
  LAtom of Atom
| LCons of Cons
and Atom =
  Atom of { PrintName : string, PropertyList : Cons }
and Cons =
  CNil
| Cons of { Car : LispList, Cdr : Cons }
```

or these declarations in Java (where the Nil pointer case is implicit):

```
abstract class LispList {}

class Cons extends LispList {
  LispList Car;
  Cons Cdr;
}

class Atom extends LispList {
  String PrintName;
```

```
    Cons PropertyList;  
}
```

In addition, constructors and fields selectors should be defined.

- Note that Strachey's section 3.7.6 describes the C and C++ pointer dereferencing operator and address operator: `Follow[p]` is just `*p`, and `Pointer[x]` is `&x`.
- The 'load-update-pairs' mentioned in Strachey's sections 4.1 are called *properties* in Common Lisp Object System, Visual Basic, and C#: get-methods and set-methods.

6.8 History and literature

Many concepts in programming languages can be traced back to Strachey's 1967 summer school lecture notes [97], discussed in Section 6.7. Brian W. Kernighan and Dennis M. Ritchie wrote the authoritative book on the C programming language [55]. The development of C is recounted by Ritchie [83]. Various materials on the history of B (including a wonderfully short User Manual from 1972) and C may be found from Dennis Ritchie's home page [82]. A modern portable implementation of BCPL — which must otherwise be characterized as a dead language — is available from Martin Richards's homepage [81].