UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Computer Science Tripos
Part IB [P]
Part II General [C]
Diploma in Computer Science [D]

Compiler Construction

(Part 2 of 2)

http://www.cl.cam.ac.uk/Teaching/2001/CompConstr

Alan Mycroft am@cl.cam.ac.uk

2001–2002 (Lent Term)

Part B: Implementing Language Features

In first section of the course we showed how to translate a subset of Java into both JVM-style code and to native machine code and how such latter code can be linked to form an executable. The subset of Java considered was a single class containing static methods and variables—this is very similar in expressiveness to the language C.

In this second part of the course we will try to crystalise various notions which appeared informally in the first part into formal concepts. In particular we investigate some of the interactions or equivalences which occur how these are reflected in a simple interpreter. We then consider how other aspects of programming languages might be implemented, in particular we focus on: how free variables (used by a function but not defined in it) are accessed, how exceptions are implemented and how inheritance may be implemented.

1 Declarations, Expressions and Commands

A idea common to most modern programming languages is the identification of three particular concepts: *expressions* whose principal purpose is to calculate a value; *commands* whose principal purpose is to update the values held in *variables* or to perform I/O; and *declarations* whose job is to introduce new variables and (optionally) to give them an initial value. Declarations can also be used to introduce new *functions* (sometimes called *procedures*) and to introduce new *types*. Some languages identify some subset of declaration and refer to them *definitions*, for example in C `int x;` is a declaration but `int x=2;` is a definition. We will use definition for declaration informally, saying "the function f is defined by $f(x) = e$" etc. Object-oriented languages often use the word "*methods*" to refer to functions; and phrases like "*class variables*" or "*attributes*" for variables declared within a class.

There is a circular dependence here: a command (e.g. `x:=`e`;`) may contain an expression which may refer to a variable introduced in a declaration which was initialised by an expression containing a function call which contains an expression etc. Although often convenient for programming, the mixing of commands and declarations under the concept of *statements* in Java is not generally helpful for understanding—declarations and commands serve very different rôles. One can see side-effects within expressions similarly—expressions like `A[i++]=`e`;` can be very convenient for the programmer, but the way that a (rather restricted) set of commands be executed during the evaluation of an expression can make it hard to reason about a program.

One can see the concept of variable as a high-level abstraction of the concept of a cell in computer memory, and those of expression and command as abstractions of sequences of machine instructions. In this view, the rôle of declarations is merely to give names to items (previously unused memory locations, functions, types and the like) but this machine-oriented view understates the importance for humans of structuring a large system in terms of named components.

Note that, in general, the left-hand-side of an assignment may be more complicated than a simple variable, for example in Java we may have

```
a[x>0 ? x : y+1] = 42;
```

whereas in C++, for suitable variable declarations, one can even have

```
(x>0 ? x : A[y+1]) = 42;
```

In such languages it is common to consider the left-hand-side of an assignment as a syntactically restricted form of expression; then an assignment statement e`:=`e'`;` can be seen as having meaning (*semantics* is the proper word here) as:

1. evaluate e to give an address;

2. evaluate e' to give a value;

3. update the addressed cell with the value.

To avoid the overtones and confusion that go with the terms *address* and *value* we will use the more neutral words *Lvalue* and *Rvalue* (first coined by C. Strachey); this is useful for languages like C where addresses are just one particular form of value. An Lvalue (left hand value) is the address (or location) of an area of store capable of holding the Rvalue. An Rvalue (right hand value) is a bit pattern used to represent an object (such as an integer, a floating point number, a function, etc.). In general, see the examples above, both Lvalues and Rvalues require work done when they are being evaluated—viewing `x=y+1;` as a 'typical' assignment gives an over-simplified view of the world.

Note that in Java, the above description of e`:=`e'`;` is precise, but other languages (such as C and C++) say that the exact ordering and possible interleaving of evaluation of e and e' is left to the implementation; this matters if expressions contain side-effects, e.g.

```
A[x] = f();
```

where the call to `f` updates `x`.

Warning: many inscrutable errors in C and C++ occur because the compiler, as permitted by the ISO language standards, chooses (for efficiency reasons) to evaluate an expression in a different order from what the programmer intended. If the order does matter then it is clearer (even in Java) and better for maintenance to break down a single complicated expression by using assignments, executed in sequence, to temporary named variables.

2 Variables and Names

Programmers use names (or identifiers) to declare variables (actual storage locations). Note that a single name may refer to more than one variable, e.g. the name of the parameter to procedure which is recursively called, or in examples like

```
void f() { ... { int n; ... } ... { int n; ... } ... }
```

In discussion below we will also see situations where multiple names refer to the *same* variable—this is called *aliasing*.

Consider the following C/C++/Java code:

```
float p = 3.4;
float q = p;
```

This causes a new storage cell to be allocated, initialised with the value 3.4 and associated with the name `p`. Then a second new storage cell (identified by `q`) is initialised with the (Rvalue) contents of `p` (clearly also 3.4). Here the defining operator `=` is said to *define by value*. One can also imagine language constructs permitting *definition by reference* (also called *aliasing*) where the defining expression is evaluated to give an Lvalue which is then associated with the identifier instead of a new storage cell being allocated, e.g.

```
float r ≃ p;
```

Since `p` and `r` have the same Lvalue they share the same storage cell and so the assignments: `p := 1.63` and `r := 1.63` will have the same effect, whereas assignments to `q` happen without affecting `p` and `r`. In C++ definition by reference is written:

```
float &r = p;
```

whereas (for ML experts) in ML mutable storage cells are defined explicitly so the above example would be expressed:

```
val p = ref 3.4;
val q = ref (!p);
val r = p;
```

Recall from the previous section that evaluation to an Lvalue may require computation to be performed:

```
int i = 2;
int x ≃ m[i+2];
...
i := 3;
...// here x still refers to m[4]
```

Finally, note that defining a new variable is quite different from assigning to a pre-existing variable. Consider the Java programs:

```
int a = 1;                              int a = 1;
int f() { return a; }                   int f() { return a; }
void g() { a = 2; println(f()); }       void g() { int a = 2; println(f()); }
```

3 Functional Programming—Life without Commands

In the next few sections we will consider the interplay between declarations and side-effect-free expressions—the so-called functional languages.[1] There is synergy between these and the ML and Computation Theory courses. The simpler scenario of declarations and side-effect-free expressions will enable us to study interesting issues (at the same time avoiding issues such as whether evaluation is to give an Lvalue or Rvalue—since if the values stored in Lvalues cannot change it does not matter at what time the Rvalue is taken from it or whether aliasing takes place). However, we ensure that declarations still associate each name with a memory location so that later introducing an assignment operator is easy.

A useful property of functional languages is *referential transparency* which means that the value of an expression only depends on the values of its subexpressions. This means that normal mathematical equivalences hold, e.g. $e + e = 2 * e$. However in the presence of side-effects this fails, e.g. in Java we have `(x++ + x++)` is very different from `2*(x++)`.

Because this part of the course is not concerned with any particular language, we will introduce an ML-like expression-based language which captures ideas common in other languages. We assume that the (abstract) syntax of expressions e is:

- c, an integer;
- x, a name;
- $e_1 + e_2$, provided e_1 and e_2 are (smaller) expressions;
- $e_1 - e_2$, provided e_1 and e_2 are (smaller) expressions;
- $e_1 ? e_2 : e_3$, provided e_1, e_2 and e_3 are (smaller) expressions;
- `let` $x = e_1$ `in` e_2, provided x is a name and e_1 and e_2 are (smaller) expressions. The phrase $x = e_1$ is seen as a *declaration*;
- $e_1\ e_2$, (an application) provided e_1 and e_2 are (smaller) expressions;
- $\lambda x.e_1$, (a lambda-abstraction) provided e_1 is a (smaller) expression. The identifier x is called the *bound variable* and the expression e_1 is called the *body* of the abstraction.

The first forms will be familiar to everyone. The final form $\lambda x.e_1$ is an (anonymous) function which takes an argument x and yields e_1. Thus the conventional function definition $f(x) = e$ would here be written $f = \lambda x.e$, i.e. one can view $\lambda x.e$ as equivalent to f `where` $f(x) = e$. We

[1] Sometimes to emphasise the difference between this simple framework full ML (which does have assignment) the phrase "pure functional language" is used.

will later note that for many purposes $(\lambda x.e_2)e_1$ and $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ can be treated almost identically.

In examples, we will allow abstractions and applications to take multiple arguments and use additional operators from the above.

4 Environments

In order to evaluate `a+5+b/a` we need to know the values of `a` and `b`. We speak of evaluating an expression in an *environment* which provides the values of the names in the expression. (An environment is the run-time equivalent (associating names to values) of the compile-time notion of symbol tables (associating names to locations where variables will reside).) One way to provide such an environment is by the above `let` form, e.g.

```
let a = 2+3/7 in a+3/a
let x = y+2/y in x+y+3/x
```

or, equivalently—as we will see—by the lambda form:

```
(λa. a+3/a)(2+3/7)
(λx. x+y+3/x)(y+2/y)
```

(Note that in programming languages like ML, λ is often written `fn` and we will occasionally adopt this convention in these notes.) These two methods are exactly equivalent and have the same meaning. The name `y` in the second expression is not bound and its value must still be found in the environment in which the whole expression is to be evaluated. Variables of this sort are known as *free variables.* Variables having local definitions in scope are known as *bound variables.*

Given an expression we can formally define its bound variables $BV(e)$ and its free variables $BV(e)$ inductively:

$$\begin{array}{rcl} BV(c) & = & \{\} \\ BV(x) & = & \{\} \\ BV(e_1 + e_2) & = & BV(e_1) \cup BV(e_2) \\ BV(\lambda x.e) & = & BV(e) \cup \{x\} \\ BV(\texttt{let}\ x = e_1\ \texttt{in}\ e_2) & = & BV(e_1) \cup BV(e_2) \cup \{x\} \end{array}$$

$$\begin{array}{rcl} FV(c) & = & \{\} \\ FV(x) & = & \{x\} \\ FV(e_1 + e_2) & = & FV(e_1) \cup FV(e_2) \\ FV(\lambda x.e) & = & FV(e) \setminus \{x\} \\ FV(\texttt{let}\ x = e_1\ \texttt{in}\ e_2) & = & FV(e_1) \cup (FV(e_2) \setminus \{x\}) \end{array}$$

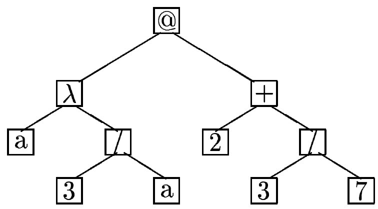
Note that the expected $BV(e) \cap FV(e) = \{\}$ does not always hold—consider

```
(let a = 2 in a)+a
```

This has $FV(e) = \{\texttt{a}\}$ because of the final `a` (which is unbound) but also $BV(e) = \{\texttt{a}\}$ because of the `let` which binds `a` to 2 within the parentheses. Note also that this shows the idea of scope already exists such a simple language.

5 Abstract Syntax Tree Representation

The phrase "applicative structure" used in the syllabus is now rather obsolete, and merely refers to the abstract syntax tree of an expression. As an example as to how lambda-forms and applications (we will use the symbol "@" to represent application since juxtaposition used above is otherwise invisible!) are represented as trees, the expression `(λa. a+3/a)(2+3/7)` above would have a tree:



6 Lambda calculus

The lambda calculus is the subset of expressions comprised solely of variables, applications, lambda-abstractions and (sometimes) constants. There are three main reasons for studying it when considering programming languages:

1. It is a useful notation for specifying the scope rules of identifiers in programming languages, and helps to demonstrate such notions as "holes in the scope of variables".
2. It helps one to understand what a function is, and what it means to pass a parameter.
3. There is a well established mathematical theory of lambda calculus.

Evaluation of lambda expressions is by means of two simple rewrite rules. We write $e \to e'$ to mean e can be rewritten to e':

- **α-conversion.** $\lambda x.e \to \lambda y.e'$ where y is any identifier that does not occur in e and where e' is a copy of e with all occurrences of the identifier x replaced by y;
- **β-reduction.** $(\lambda x.e_1)e_2 \to e_1[e_2/x]$ where the right-hand-side just means "a copy of of e_1 with all occurrences of x replaced by the argument e_2", provided that
  1. e_1 contains no (inner) bound occurrences of identifier $\mathtt{x}$, and
  2. e_2 contains no free variables that are bound in e_1.

The α-conversion rule is used to remove conflicts that prevent β-reductions from being applicable.

To enhance readability in these notes, lists of bound variables enclosed in parentheses will be allowed after λ instead of a single variable and lists of arguments will be allowed in applications; this follows the ML practice.

A few example evaluations are given below (there would normally be additional reduction rules to reduce arithmetic expressions):

$$
\begin{array}{rcl}
(\lambda\mathtt{t}.\mathtt{t}+16)(4) & \to & 4+16 \\
(\lambda(\mathtt{a},\mathtt{b}).\mathtt{a}+\mathtt{b})(5,6) & \to & 5+6 \\
(\lambda\mathtt{f}.((\lambda\mathtt{x}.\mathtt{f}(\mathtt{x}+1))(3)))(\lambda\mathtt{y}.\mathtt{y}*2) & \to & (\lambda\mathtt{x}.(\lambda\mathtt{y}.\mathtt{y}*2)(\mathtt{x}+1))(3) \\
 & \to & (\lambda\mathtt{y}.\mathtt{y}*2)(3+1) \\
 & \to & (3+1)*2 \\
(\lambda\mathtt{y}.\mathtt{yy})(\lambda\mathtt{x}.\mathtt{xx}) & \to & (\lambda\mathtt{x}.\mathtt{xx})(\lambda\mathtt{x}.\mathtt{xx}) \\
 & \to & (\lambda\mathtt{x}.\mathtt{xx})(\lambda\mathtt{x}.\mathtt{xx}) \\
 & \to & \cdots \langle\text{forever}\rangle
\end{array}
$$

An important property of a lambda expression is that one can determine from the text of the expression (i.e. without having to evaluate it) to which bound variable each occurrence of a name is bound (unless it is free). A suitable algorithm is as follows:
Given an occurrence of the name x

1. Find the smallest textually enclosing lambda (or let) expression.
2. Compare x with the bound variables names, if there is a match we have finished, otherwise repeat from (1) to try the lambda expression one level further out.

7 The correspondence between programming languages and lambda calculus

Here we show that the `let` notation, both for introducing definitions of simple names and functions, can be eliminated in favour of λ.

Consider the following expression:

```
{ let f(y) = y*2
  in let x = 3
  in f(x+1)
}
```

In this `f` is a function with one bound variable `y` whose body is the expression `y*2`. We might write: `let f = (λy. y*2)`. Similarly `x` is like a bound variable of a lambda expression whose body is `f(x+1)` and whose argument is `3`. Thus we could re-write the whole expression as:

```
{ let f = λy. y*2
  in (λx. f(x+1)) (3)
}
```

which further can be re-written as:

```
{λf. (λx. f(x+1)) (3)} (λy. y*2)
```

Hence, the lambda notation can completely express both simple variable and function definitions. Indeed it can usefully be seen as a machine code in its own right (there was even a machine built at Cambridge some years back which used essentially λ-calculus as its machine code!). Just as we chose to prefer higher level notation than (say) Pentium machine code, one prefers the more usual `let` and function forms rather than the rebarbarative λ form for real programming—its real benefit is that of understanding concepts like scoping.

8 A short interlude on recursion

When a function is defined in terms of itself as in the following Java definition

```
int scantree(Tree x) { ...
                       ... scantree(x.left) ...
                       ...
                       ... scantree(x.right) ...
                       ...
                     }
```

It is said to be *defined recursively.* If several functions are defined in terms of themselves they are said to be *mutually recursive.* Suppose there is a call `scantree(sometree)` to the function given above, then while this call is being evaluated it may happen that the call `scantree(x.left)` is executed. While this second call is active there are two *activations* of `scantree` in existence at once. The second call is said to be a *recursive call* of `scantree`. Note that therefore there will be two distinct variables called `x` (holding different values) in such circumstances; we therefore need to use fresh storage for variable `x` at each call to `scantree`. This was the purpose of using a new stack frame for each activation of a function in the JVM.

Notice that it is possible to call a function recursively without defining it recursively.

```
let f(g,n) = { ...
               ... g(g,n-1)
               ...
             }
in f(f, 5)
```

Here the call `g(g,n-1)` is a recursive call of `f`.
[Exercise: complete the body of `f` so that the call yields $5! = 120$.]

9 The need for the word `rec`

Consider the following expression:

```
{ let f(n) = n=0 ? 1 : n*f(n-1)
  in f(4)
}
```

The corresponding lambda expression is

```
(λf. f(4)) (λn. n=0 ? 1 : n*f(n-1) )
```

We observe that the scope of `f` is `f(4)`, and that the `f` in `f(n-1)` is unbound and certainly different from the `f` in `f(4)`. However, here the programmer presumably was trying to define the recursive factorial function and so meant the `f` on the right hand side to be the same as the `f` being defined. To indicate that the scope of x in (`let` x=e `in` e') extends to include both e as well as e' the keyword `rec` is normally used.

```
let rec f(n) = n=0 ? 1 : n*f(n-1)
```

which can be more primitively written as

```
let rec f = λn. n=0 ? 1 : n*f(n-1)
```

The linguistic effect of `rec` is to extend the the scope of the defined name to include the right hand side of the definition.

In ML, all `fun`-based definitions are assumed to be recursive, and so the definition

```
fun f(x) = e;
```

is first simplified (de-sugared) by the most ML systems to

```
val rec f = fn x => e;
```

10 The Y operator

At first sight, the `rec` construction seems to have no lambda calculus equivalent; however, postulating a new constant `Y` (just like `0`, `1`, ..., `+`, `cond`) operating on functions enables a solution to be found. Consider

```
let H = λf. λn. n=0 ? 1 : n*f(n-1)
```

`f` is now bound, but `H` is not the factorial function, for

```
H(λx. x)(6)  =  {λn. n=0 ? 1 : n*(λx. x)(n-1)} (6)
             =  {λn. n=0 ? 1 : n*(n-1)} (6)
             =  6*5
             =  30
```

However, if g were the factorial function, then

```
H(g)  =  λn. n=0 ? 1 : n*g(n-1)
      =  the factorial function
      =  g
```

thus `H(g)=g` if g is the factorial function. It therefore seems plausible that, if we can find a g for which `H(g)=g`, then the g we found would be the factorial function. Given an function, Φ say, any value v such that $\Phi(v) = v$ called a *fixed point* of Φ. (Observe that 1 and 2 are both fixed points of the ordinary function on reals given by $\phi(x) = x^2 - 2x + 2$, but note that our Φ will typically map functions to functions.)

In the same sense that the fixed points of a quadratic $ax^2 + bx + c$ can be found by a formula (this can be seen as a function which operates on ϕ and gives us x)

$$x = \frac{-(b-1) \pm \sqrt{(b-1)^2 - 4ac}}{2a}$$

we could *hope* for a function Y which returns the fixed point of its argument, e.g. $Y(\mathtt{H}) =$ factorial or more generally

$$Y\Phi = \text{some value } f \text{ such that } \Phi f = f.$$

Put more simply we want

$$Y\Phi = \Phi(Y\Phi).$$

If this can be done, then we can rewrite any recursive function simply using Y, e.g. writing

```
let rec f = λn. n=0 ? 1 : n*f(n-1)
```

as

```
let f = Y( λf. λn. n=0 ? 1 : n*f(n-1) )
```

and we can evaluate a call of `f` knowing no more about Y than the property $Y\Phi = \Phi(Y\Phi)$. For example, with `H = λf. λn. n=0 ? 1 : n*f(n-1)`

```
f(3)  =  Y(H) (3)                            [definition of f]
      =  H(Y(H)) (3)                         [property of Y]
      =  {λn. n=0 ? 1 : n*(Y(H)(n-1))} (3)   [lambda reduction]
      =  3 * Y(H)(2)
      =  3 * 2 * Y(H)(1)                     [similarly]
      =  3 * 2 * 1 * Y(H)(0)                 [similarly]
      =  3 * 2 * 1 * 1                       [similarly]
```

It is somewhat remarkable at first that such a Y exists at all and moreover that it can be written just using λ and *apply*. One can write[2]

$$Y = \lambda f.\ (\lambda g.\ (f(\lambda a.\ (gg)a)))(\lambda g.\ (f(\lambda a.\ (gg)a))).$$

(Please note that learning this lambda-definition for Y is *not* examinable for this course!) For those entertained by the "Computation Theory" course, this (and a bit more argument) means that the lambda-calculus is "Turing powerful".

Note that the definition of Y given here will only find fixed points of functions, like H above of type $(int \to int) \to (int \to int)$ and in doing so yield a function of type $int \to int$. It will not find the numeric fixed points of

$$\lambda x.\ x^2 - 2x + 2.$$

(Why?)

Finally, an alternative implementation of Y (there seen as a primitive rather as the above arcane lambda-term) suitable for an interpreter is given in section 12.

11 Object-oriented languages

The view that lambda-calculus provides a fairly complete model for binding constructs in programming languages has generally been well-accepted. However, notions in inheritance in object-oriented languages seem to require a generalised notion of binding. Consider the following C++ program:

[2] The form

$$Y = \lambda f.\ (\lambda g.\ gg)(\lambda g.\ f(gg))$$

is usually quoted, but (for reasons involving the fact that our lambda-evaluator uses call-by-value and the above definition requires call-by-name) will not work on the lambda-evaluator presented here.

```
const int i = 1;
class A { const int i = 2; };
class B : A { int f(); };
int B::f() { return i; }
```

There are two `i` variables visible to `f()`: one being `i=1` by lexical scoping, the other `i=2` visible via inheritance. Which should win? C++ defines that the latter is visible (because the definition of `f()` essentially happens in the scope of `B` which is effectively nested within `A`). The `i=1` is only found if the inheritance hierarchy has no `i`. Note this argument still applies if the `const int i=1;` were moved two lines down the page. The following program amplifies that the definition of the order of visibility of variables is delicate:

```
const int i = 1;
class A { const int j = 2; };
void g()
{   const int i = 2;
    class B : A { int f() { return i; }; }
    // which i does f() see?
}
```

The lambda-calculus for years provided a neat understanding of scoping which language designers could follow simply; now such standards committees have to use their (not generally reliable!) powers of decision.

Note that here we have merely talked about (scope) *visibility* of identifiers; languages like C/Java also have declaration qualifier concerning *accessibility* (`public`, `private`, etc.). It is for standards bodies to determine whether, in the first example above, changing the declaration of `i` in `A` to be `private` should invalidate the program or merely cause the `private i` to become invisible so that the `i=1` declaration becomes visible within `B::f()`. (Actually draft ISO C++ checks accessibility after determining scoping.)

We will later return to implementation of objects and methods as data and procedures.

12 Mechanical evaluation of lambda expressions

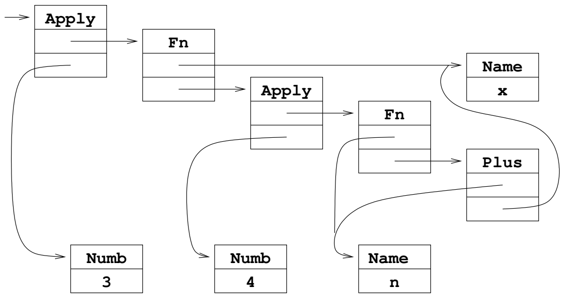
We will now describe a simple way in which lambda expressions may be evaluated in a computer. We will represent the expression as a parse tree and evaluate it in an environment that is initially empty. As above there will be tree nodes representing variables, constants, addition, function abstraction and function application. In ML this can be written:

```
datatype Expr = Name of string |
                Numb of int |
                Plus of Expr * Expr |
                Fn of string * Expr |
                Apply of Expr * Expr;
```

The expression: `(λx. (λn. n+x)(4)) (3)` would be written in ML (or C, assuming appropriate (constructor) functions like `Apply`, `Fn` etc. were defined to allocated and initialise structures) as:

```
Apply(Fn("x", Apply(Fn("n", Plus(Name("n"), Name("x"))),
                    Numb(4))),
      Numb(3))
```

and be represented as follows:



When we evaluate such an Expr we expect to get a value which is either a integer or a function. For non-ML experts the details of this do not matter, but in ML we write this as

```
datatype Val = IntVal of int | FnVal of string * Expr * Env;
```

(the justification for why functions consist of more than simply their text will become apparent when we study the evaluator 'eval' below).

We will represent the environment of defined names (names in scope) as a linked list with the following structure:

```
datatype Env = Empty | Defn of string * Val * Env;
```

(I.e. an Env value is either Empty or is a 3-tuple giving the most recent binding of a name to a value and the rest of the environment.) The function to look up a name in an environment[3] could be defined in ML as follows.

```
fun lookup(n, Defn(s, v, r)) =
            if s=n then v else lookup(n, r);
  | lookup(n, Empty) = raise oddity("unbound name");
```

We are now ready to define the evaluation function itself:

```
fun eval(Name(s), r) = lookup(s, r)
  | eval(Numb(n), r) = IntVal(n)
  | eval(Plus(e, e'), r) =
      let val v = eval(e,r);
          val v' = eval(e',r)
      in case (v,v') of (IntVal(i), IntVal(i')) => IntVal(i+i')
                      | (v, v') => raise oddity("plus of non-number")
      end
  | eval(Fn(s, e), r) = FnVal(s, e, r)
  | eval(Apply(e, e'), r) =
      case eval(e, r)
        of IntVal(i) => raise oddity("apply of non-function")
         | FnVal(bv, body, r_fromdef) =>
             let val arg = eval(e', r)
             in eval(body, Defn(bv, arg, r_fromdef))
             end;
```

The immediate action of eval depends on the leading operator of the expression it is evaluating. If it is Name, the bound variable is looked up in the current environment using the function lookup. If it is Numb, the value can be obtained directly from the node (and tagged as an IntVal). If it is Plus, the two operands are evaluated by (recursive) calls of eval using the current environment and their values summed (note the slightly tedious code to check both values correspond to numbers else to report an error). The value of a lambda expression (tagged as a FnVal) is called a *closure*

[3]There is a tradition of using letters like r or ρ for 'environment' to avoid clashing with the natural use of e for 'expression'.

and consists of three parts: the bound variable, the body and the current environment. These three components are all needed at the time the closure is eventually applied to an argument. To evaluate a function application we first evaluate both operands in the current environment to produce (hopefully) a closure (`FnVal(bv, body, r_fromdef)`) and a suitable argument value (`arg`). Finally, the body is evaluated in an environment composed of the environment held in the closure (`r_fromdef`) augmented by `(bv, arg)`, the bound variable and the argument of the call.

At this point it is appropriate to mention that recursion via the Y operator can be simply incorporated into the interpreter. Instead of using the gory definition in terms of λ, we can implement the recursion directly by

```
| eval(Y(Fn(f,e)), r) =
    let val fv = IntVal(999);
        val r' = Defn(f, fv, r);
        val v = eval(e, r')
    in
        fv := v;        (* updates value stored in r' *)
        v
    end;
```

This first creates an extended closure `r'` for evaluating `e` which is `r` extended by the (false) assumption that `f` is bound to 999. e (which should really be an expression of the form $\lambda x.\ e'$ to ensure that the false value of `f` is not used) is then evaluated to yield a closure, which serves as result, but only after the value for `f` stored in the closure environment has been updated to its proper, recursive, value `fv`. This construction is sometimes known as "tying the knot [in the environment]" since the closure for `f` is circular in that its environment contains the the closure itself (under name `f`).

A more detailed working evaluator including Y and *let*) can be found on the web page for this course (see front cover).

12.1 Static and dynamic scoping

This final point is worth a small section on its own; the normal state in modern programming languages is that free variables in are looked up in the environment of existing at the time the function was *defined* rather than when it is *called*. This is called *static scoping* or *static binding* or even *lexical scoping*; the alternative of using the calling environment is called *dynamic binding* and was used in many dialects of Lisp. The difference is most easily seen in the following example:

```
let a = 1;
let f() = a;
let g(a) = f();
print g(2);
```

Check your understanding of static and dynamic scoping by observing that this prints `1` under the former and `2` under the latter.

You might be tempted to believe that rebinding a variable like '`a`' in dynamic scoping is equivalent to assigning to '`a`'. This is untrue, since when the scope ends (in the above by `g` being exited) the previous binding of '`a`' (of value one) again becomes visible, whereas assignments are not undone on procedure exit.

Exercises

1. Draw the tree structure representing the lambda expression form of the following program.

```
{ let x = 3
  in let f(n) = n+x
  in let x = 4
```

```
    in f(x)
}
```

Apply the `eval` function by hand to this tree and an empty environment and draw the structure of every environment that is used in the course of the evaluation.

2. Is it possible to write a finite program that would cause this evaluator to attempt to create an infinitely long environment?

3. Modify the interpreter to use dynamic scoping; is it now possible to write a finite program that would cause this evaluator to attempt to create an infinitely long environment?

13 A more efficient implementation of the environment

The previous lambda evaluator (also known as an *interpreter*) is particularly inefficient in its treatment of names since it searches a potentially long environment chain every time a name is used. This search can be done much more efficiently if the environment were represented differently; moreover the technique we describe is much more appropriate for a *compiler* which generates machine code for a target machine. Consider the following:

```
(λ(a,b,c).(λ(x,y,z).(λt.B1))((λ(u,v).B2),12,63))(1,6,3)
```

Modulo ignoring function names `f`, `g`, `h`, `k` this environment structure can also be generated by: can also be seen as:

```
let f(a,b,c) =
  ( let g(x,y,z) = (let h(t) = B1 in h)
    in g((let k(u,v) = B2 in k), 12, 63)
  )
in f(1,6,3)
```

The environment structure can be represented as a tree as follows (note that here the tree is logically backwards from usual in that each node has a single edge to its parent, rather than each node having an edge to its children):

```
              R1:(a,b,c)
             ↗          ↖
    R2:(x,y,z)          R4:(u,v)
        ↑
    R3:(t)
```

The levels on the right give the depth of textual nesting of lambda bodies, thus the maximum number of levels can be determined by inspecting the given expression. When evaluating `B1`, we are in environment `R3` which looks like:

```
R1:(a,b,c)          level 1
R2:(x,y,z)          level 2
R3:(t)              level 3
```

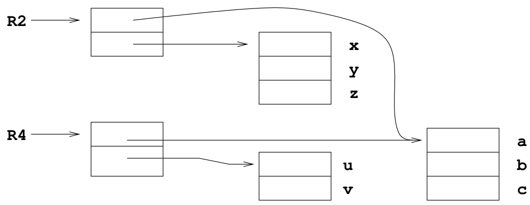
We can associate with any name used in `B1` an ‘address’ consisting of a pair of numbers, namely, a level number and a position within that level. For example:

```
a:(1,1)   b:(1,2)   c:(1,3)
x:(2,1)   y:(2,2)   z:(2,3)
t:(3,1)
```

Similarly within B2:

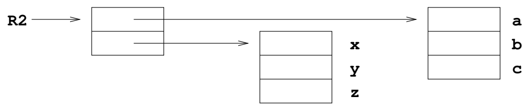
```
a:(1,1)   b:(1,2)   c:(1,3)
u:(2,1)   v:(2,2)
```

At execution time, the environment could be represented as a vector of vectors. For example,

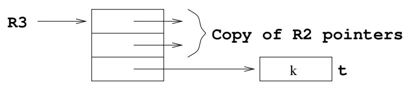


A vector such as the one pointed to by R2 is called a *display* (first coined by Dijkstra). Notice that while evaluating B2 in environment R4 we may use the "address" (2,1) to access the variable u. It should be clear that the pointer to the current display provides sufficient information to access any currently declared variable and so may be used in place of the environment chain used in the `eval` function.

You will recall that a closure (the value representing a function) consists of three parts—the bound variable, the body and the environment information. If we are using the display technique then the environment part can be represented by a pointer to the appropriate display vector. For instance, the environment part of the closure for λt.B1 in the last example is the pointer to the display vector for R2.

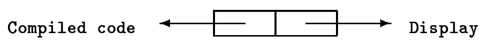


In order to apply this closure to an argument value k, we must first create a new display which consists of a copy of R2 augmented with a new level. The new display will be as follows:



The body of B1 is then evaluated in this new environment. When the application is complete the value is returned and the previous environment reinstated.

The compiled form of a function often consists of code which first constructs the new display and then evaluates the body; hence the closure is often represented as a pair of pointers:



The beauty of displays is that every free variable is accessible from any procedure (no matter how deeply nested) in two instructions. However, in practice, even in languages which permit such procedure nesting, we find that only about 3% of variable accesses are to variables which are neither local (addressable from FP) nor top-level (addressable using absolute addressing). Therefore the cost of setting them up on procedure entry can easily outweigh the saving over the alternative scheme (the 'static link' method) which we now consider.

Exercise

Re-implement the `eval` function defined above using a display mechanism for the environment (instead of the linked list).

14 Evaluation Using a Stack—Static Link Method

We saw in the first part of the notes how the JVM uses a stack to evaluate expressions and function calls. Essentially, two registers (FP and SP) respectively point to the current stack frame and its fringe.

Evaluation on a stack is more efficient than in the lambda-interpreter presented above in that no search happens for variables, they are just extracted from their location. However, the JVM as defined only provides instructions which can access *local* variables (i.e. those on the local stack frame accessed by FP) and *static* variables (often called *global or top-level variables* in other languages) which are allocated once at a fixed location.

Indeed Java forbids the textual nesting of one function within another, so the question of how to access the local variables of the outer function from within the inner function does not need to be addressed in the JVM. However, for more general languages we need to address this issue.

The usual answer is to extend the *linkage* information so that in addition to holding the return address L and the old frame pointer FP′, also known as the *dynamic link* as it points to the frame of its *caller*, it also holds a pointer S, the *static link* which points to the frame of its *definer*.

Because the definer also has its static link, access to a non-local variable (say the ith local variable in the routine nested j levels out from my current nesting) can be achieved by following the static link pointer j times to give a frame pointer from which the ith local can be extracted. Thus access to non-local variables can be slower than with a display, but the set-up time of the static link field is much quicker than creating a display. In the example in section 13, the environment for B1 was R3 with variables as follows:

```
R1:(a,b,c)              level 1
R2:(x,y,z)              level 2
R3:(t)                  level 3
```

Thus t is accessed relative to FP in one instruction, access to variables in R2 first load the S field from the linkage information and then access x, y or z from that (two instructions), and variables in R1 use three instructions first chaining twice down the S chain and then accessing the variable.

Hence the instruction effecting a call to a closure (now represented by a pair of the function entry point and the stack frame pointer in which it was defined) now merely copies this latter environment pointer from the closure to the S field in the linkage information in addition to the work detailed for the JVM.

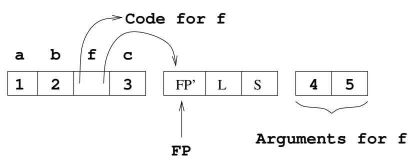
Exercise: give a simple example in which S and FP' pointers differ.

An example

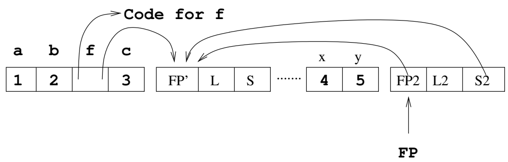
Consider the following fragment of program:

```
...
let G() =
{ let a, b = 1, 2
  let f(x, y) = a*x + b*y
  let c = 3
  c := f(4,5)
}
...
```

At the moment when f is just about to be entered the current stack frame is as follows:



At the moment just after f has been entered (when `a*x+b*y` is about to be evaluated) the state is as follows:



We see that `f` can now access `x` and `y` from `FP` (at offsets -1 and -2), and `a` and `b` from the definer's stack frame (offsets -4 and -3) which is available as `S2`. Beware: we cannot access `a` and `b` as a constant offset from `FP` since `f` may be called twice (or more) from within `G` (or even from a further function to which it was passed as a parameter) and so the stack frame for `G()` may or may not be contiguous with `x` as it is in the example.

You might wonder why we allocated `f`, or more properly its closure, to a local variable when we knew that it was constant. The answer was that we treated the local definition of `f` as if it were

```
let f = λ(x,y). a*x + b*y
```

and further that `f` was an updatable variable. This can help you see how first-class function-valued variables can be represented. In practice, if we knew that the call to `f` was calling a given piece of code (here $\lambda(\mathtt{x},\mathtt{y}).\mathtt{a}*\mathtt{x}+\mathtt{b}*\mathtt{y}$) with a given environment pointer (here the `FP` of the caller) then the calling sequence can be simplified.

15 Situations where a stack does not work

If the language allows the manipulation of pointers then erroneous situations are possible. Suppose we have the "address of" operator `&` which is defined so that `&x` yields the address of (or pointer to) the storage cell for `x`. Suppose we also have "contents of" operator `*` which takes a pointer as operand and yields the contents of the cell to which it refers. Naturally we expect `*(&x)=x`. Consider the program:

```
let f() = { let a = 0
            in &a
          }
let p = f()
...
```

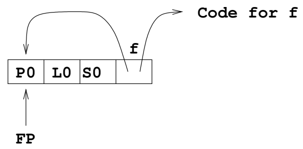
The result of `f` is a pointer to the local variable a but unfortunately when we return from the call this variable no longer exists and `p` is initialised to hold a pointer which is no longer valid and if used may cause an extremely obscure runtime error. Many languages (e.g. Pascal, Java) avoid this problem by only allowing pointers into the heap.

NB. Apart from the JVM, it is usually more convenient to arrange that arguments appear to the left of the linkage information, and first local variables and then local evaluation stack to the right of the linkage information. We sometimes adopt this convention in the examples below because they show more clearly such pointers pointing the 'wrong direction' up the stack.

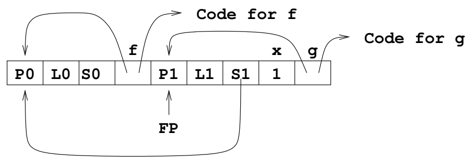
Some other objects such as functions and arrays contain implicit pointers to the stack and so have to be restricted if a stack implementation is to work. Consider:

```
let f(x) = { let g(t) = x+t
             in g
           }
let add1 = f(1)
...
```

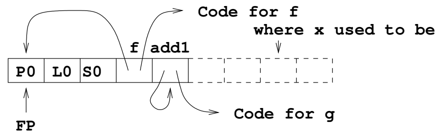
The result of `f(1)` should be a function which will add one to its argument. Thus one might hope that `add1(23)` would yield 24. It would, however, fail if implemented using a simple stack. We can demonstrate this by giving the state of the stack at various stages of the evaluation. Just after the `f` has been declared the stack is as follows:



At the time when g has just been declared in the evaluation of `f(1)` the stack is as follows:



After the declaration of `add1` the stack would be as follows:



Thus if we now try to use `add1` it will fail since its implicit reference to `x` will not work. If `g` had free variables which were also free variables of `f` then failure would also result since the static chain for `g` is liable to be overwritten.

The simple safe rule that many high level languages adopt to make a stack implementation possible is that no object with implicit pointers into the stack (procedures, arrays or labels) may be assigned or returned as the result of a procedure call. Algol-60 first coined these restrictions as enabling a stack-based implementation to work.

ML clearly does allow objects to be returned from procedure calls. We can see that the problem in such languages is that the above implementation would forbid stack frames from being deallocated on return from a function, instead we have to wait until the last use of any of its bound variables.[4] This implementation is called a "Spaghetti stack" and stack-frame deallocation is handled by a garbage collector. However, the overhead of keeping a whole stack-frame for possibly a single variable is excessive and we now turn to an efficient implementation.

16 Implementing ML free variables

In ML programs like

```
val a = 1;
fun g(b) = (let fun f(x) = x + a + b in f end);
val p = g 2;
val q = g 3;
```

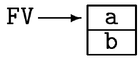
we have seen that an implementation which permanently allocates `b` to the stack location where it is passed will not work.

A mechanism originally proposed by Strachey is as follows. To declare a function such as

```
let f(x) = x + a + b
```

a tuple is constructed (called the *free variable list*) which contains the values (Lvalues or Rvalues whichever is appropriate) of the free variables. A pointer to this list is sufficient environment information for the closure. For `f` defined above the list would be as follows:

[4] More precisely, using static links, to the last use of any free variable of the called function.



During the evaluation of a function call, two pointers are needed: the `FP` pointer, as before, to address the arguments and local variables, and a pointer `FV` to point to the free variable list (although note that the `FV` pointer could be treated as an additional hidden argument to functions—this would be appropriate for expressing the translation as C code rather than machine code).

This mechanism requires more work at function definition time but less work within the call since all free variables can be accessed via a simple indirection. It is used in the Edinburgh SML implementation. (An additional trick is to store a pointer to the function code in offset 0 of the free variable list as if it were the first free variable. A pointer to the free variable list can then represent the whole closure as a single word.)

Note that this works most effectively when free variables are Rvalues and hence can be copied freely. When free variables are Lvalues we need to enter a pointer to the actual aliased location in the free variable list of each function which references it. It is then necessary also to allocate the location itself on the heap. (For ML experts: note that ML's use of `ref` for updateable variables means that this is already the case in ML.)

17 Parameter passing mechanisms

Strachey [Fundamental Concepts in Programming Languages. Oxford University Press, 1967] described the "Principle of Correspondence" in which, motivated by the *lambda*-calculus equivalence, he argued that simple declaration forms (e.g. of an initialised variable) and parameter passing mechanisms were two sides of the same coin.[5]

Thus if a simple variable may be defined (see section 2) to be either a copy or an alias of an existing variable, then so should a parameter passing mechanism. To put this another way, should parameter passing communicate the Lvalue of a parameter (if it exists) or the Rvalue?

Many languages (e.g. Pascal, Ada) allow the user to specify which is to be used. For example:

```
let f(VALUE x) = ...
```

might declare a function whose argument is an Rvalue. The parameter is said to be *called by value*. Alternatively, the declaration:

```
let f(REF x) = ...
```

might declare a function whose argument is an Lvalue. The parameter is said to be *called by reference*. The difference in the effect of these two modes of calling is demonstrated by the following example.

```
let r(REF x) = { x := x+1 }      let r(VALUE x) = { x := x+1 }
let a = 10                       let a = 10
r(a)                             r(a)
// a now equals 11               // a now equals 10
```

18 Note on Algol call-by-name

Algol 60 is a language that attempted to be mathematically clean and was influenced by the simple calling-as-substitution-of-argument-expression-into-function-body mechanism of lambda calculus. In the standard report on Algol 60 the procedure calling mechanism is described in terms of textually replacing a call by a copy of the appropriate procedure body. Systematic renaming

[5]You might care to note that even ML falls down here—you can declare a new type in a simple declaration, but not pass a type as an argument to a function!

of identifiers (α-conversion) is used to avoid problems with the scope of names. With this approach the natural treatment for an actual parameter of a procedure was to use it as a textual replacement for every occurrence of the corresponding formal parameter. This is precisely the effect of the lambda calculus evaluation rules and in the absence of the assignment command it is indistinguishable from call-by-value or call-by-reference.[6]

When an actual parameter in Algol is *called by name* it is not evaluated to give an Lvalue or Rvalue but is passed to the procedure as an unevaluated expression. Whenever this parameter is used within the procedure, the expression is evaluated. Hence the expression may be evaluated many times (possibly yielding a different value each time). Consider the following Algol program.

```
INTEGER a,i,b;
PROCEDURE f(x) INTEGER;
  BEGIN  a := x;
         i := i+1;
         b := x
  END;
a:=i:=b:=10;
f(i+2);
COMMENT a=12, i=11 and b=13;
```

ML and C/C++ have no call-by-name mechanism, but the same effect can be achieved by passing a suitable function by value. The following convention works:

1. Declare the parameter as a parameterless function (a 'thunk').

2. Replace all occurrences of it in the body by parameterless calls.

3. Replace the actual parameter expression by a parameterless function whose body is that expression.

The above Algol example then transforms into the following C program:

```
int a = 10, i = 10, b = 10;
int pointlessname() { return i+2;}
void f(int x(void)) { a = x();
                      i = i+1;
                      b = x();
                    }
f(pointlessname);
```

[C experts might care to note that this trick only works for C when all variables free to the thunk are declared at top level; Java cannot even express passing a function as a parameter to another function.]

19 A source-to-source view of argument passing

Many modern languages only provide call-by-value. This invites us to explain, as we did above, other calling mechanisms in terms of call-by-value (indeed such translations, and languages capable of expressing them, have probably had much to do with the disappearance of such mechanisms!).

For example, values passed by reference (or by result—Ada's `out` parameter) typically have to be Lvalues. Therefore they can be address-taken in C. Hence we can represent:

[6] Well, there is a slight difference in that an unused call-by-name parameter will never be evaluated! This is exploited in so-called 'lazy' languages and the Part II course looks at optimisations which select most appropriate calling mechanism for each definition in such languages.

```
void f1(REF int x) { ... x ... }
void f2(IN OUT int x) { ... x ... }     // Ada-style
void f3(OUT int x) { ... x ... }        // Ada-style
void f4(NAME int x) { ... x ... }
... f1(e) ...
... f2(e) ...
... f3(e) ...
... f4(e) ...
```

as

```
void f1'(int *xp) { ... *xp ... }
void f2'(int *xp) { int x = *xp; { ... x ... } *xp = x; }
void f3'(int *xp) { int x; { ... x ... } *xp = x; }
void f4'(int xf()) { ... xf() ... }
... f1'(&e) ...
... f2'(&e) ...
... f3'(&e) ...
... f4'(fn () => e) ...
```

It is a good exercise (and a frequent source of tripos questions) to write a program which prints different numbers based on which (unknown) parameter passing mechanism a sample language uses.

20 Modes of binding free variables

Consider a function definition such as

```
let f(x) = x+a
```

We have seen how `x` can be passed by value or by reference. It is also possible to distinguish the modes of association of free variables such as `a`. The language CPL provided syntactic means of specifying this process: a function whose free variables are called by value was defined using the = operator. If the free variables are called by reference then the function was defined using the == operator. For example,

```
let a = 3                           let a = 3
let f(x) = x+a                      let f(x) == x+a
// f(5) equals 8                    // f(5) equals 8
a := 10;                            a := 10;
// f(5) equals 8                    // f(5) equals 15
```

Nowadays languages provide the latter form only, leaving the former form to be simulated by the user by

```
let a = 3
    let private_a = a     // save 'a' at definition
    let f(x) = x + private_a
// f(5) equals 8
a := 10;
// f(5) equals 8
```

The variable `private_a` can be made truly private (visible in the body of `f` but not at `a:=10;`) by the ML 'local' construct or merely by taking a little care:

```
let a = 3
let f = (let private_a = a     // save 'a' at definition
         in fn x => x + private_a)
...
```

21 Labels and jumps

Many languages, like C and Java, provide the ability to label a statement. In C one can branch to such statements from anywhere in the current routine using a '`goto`' statement. (In Java this is achieved by the '`break`' statement which has rather more restrictions on its placement). In such situations the branch only involves a simple transfer of control (the `goto` instruction in JVM); note that because only `goto` is a statement and one can only label statements, the JVM local evaluation stack will be empty at both source and destination of the `goto`—this rather implicitly depends on the fact that statements cannot be nested within expressions.

However, if the destination is in a outermore procedure (either by static nesting or passing a label-valued variable) then the branch will cause an exit from one or more procedures. Consider:

```
 { let r(lab) = { ...
                  ... goto lab;
                  ...
                }
   ...
   r(M);
   ...
M: ...
 }
```

In terms of the stack implementation it is necessary to reset the `FP` pointer to the value it had at the moment when execution entered the body of `M`. Notice that, at the time when the jump is about to be made, the current `FP` pointer may differ. One way to implement this kind of jump is to represent the value of a label as a pair of pointers—a pointer to compiled code and a FP pointer (note the similarity to a function closure—we need to get to the correct code location and also to have the correct environment when we arrive). The action to take at the jump is then:

1. reset the FP pointer,

2. transfer control.

We should notice that the value of a label (like the value of a function) contains an implicit frame pointer and so some restrictions must be imposed to avoid nonsensical situations. Typically labels (as in Algol) may not be assigned or returned as results of functions. This will ensure that all jumps are jumps to activations that dynamically enclose the jump statement. (I.e. one cannot jump back into a previously exitted function!)

22 Exceptions

ML and Java exceptions and their handlers are conveniently seen as a restricted form of `goto`, albeit with an argument.

This leads to the following implementation: a `try` (Java) or `handle` (ML) construct effectively places a label on the handler code. Entering the `try` block pushes the label value (recall a label/frame-pointer pair) onto a stack (`H`) of handlers and successful execution of the `try` block pops `H`. When an exception occurs its argument is stored in a reserved variable (just like a procedure argument) and the label at the top of `H` is popped and a `goto` executed to it. The handler code then checks its argument to see if it matches the exceptions intended to be caught. If there is no match the exception is re-raised therefore invoking the next (dynamically) outermore handler. If the match succeeds the code continues in the handler and then with the statement following the `try-except` block.

For example given `exception foo;` we would implement

```
try C1 except foo => C2 end; C3
```

as

```
    push(H, L2);
    C1
    pop(H);
    goto L3:
L2: if (raised_exc != foo) doraise(raised_exc);
    C2;
L3: C3;
```

and the `doraise()` function looks like

```
void doraise(exc)
{   raised_exc = exc;
    goto pop(H);
}
```

Now, an alternative to an explicit stack is to implement `H` as a linked list of handlers `(label-value, next)` and keep a pointer to its top item. This has the advantage that each element can be stored in the stack frame which is active when the `try` block is entered; thus a single stack suffices for function calls and exception handlers.

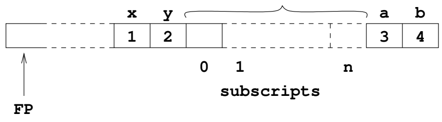
Finally, sadly ANSI C labels cannot be used as values as indicated above, and so code shown above would have to be implemented using the library function `setjmp()` instead.

23 Arrays

When an array is declared space must be allocated for its elements. In most languages the lifetime of an array is the same as that of a simple variable declared at the same point, and so it would be natural to allocate space for the array on the runtime stack. This is indeed what many implementations do. However, this is not always convenient for various reasons. Consider, for example, the following:

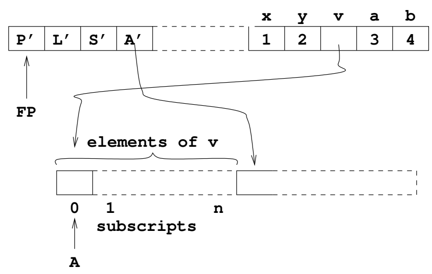
```
...
{ int x=1, y=2;
  int v[n];     // an array from 0 to n-1
  int a=3, b=4;
  ...
}
...
```

Within the body of the above block the current stack frame might look like the following (again note we are putting local variables to the right of `FP`):



In this example, `n` may be large and so the variables `a` and `b` may be a great distance from `FP`. On some machines access to such variables is less efficient. Moreover, if `n` is not a compile-time constant,[7] the position of `a` and `b` relative to `FP` will not be known until runtime, again causing inefficiency. Implementations can allocate such array elements on a separate stack, working from the other end of store. This necessitates a separate array-stack pointer (`A`, say) and space in the main stack to save and restore it. For the above example the two stacks may look as follows:

[7] C requires `n` to be a compile-time constant.



When execution leaves the current procedure FP, L, S and A must all all restored. On many implementations A is restored at the end of the execution of any block in which an array is declared.

In Java, arrays are all allocated on the heap (like other objects) and so the above techniques are restricted to languages of the C family.

24 Object-oriented language storage layout

Declarations (in C++) like

```
class A { int a1,a2; } x;
```

allocate storage for two integers and record the fact that a1 is at offset zero, and a2 is at offset 4 (assuming ints are 4 bytes wide). Now after

```
class B : A { int b; };
```

objects of type B have 3 integer fields a1 and a2 (by inheritance) normally stored at offsets 0 and 4 so that (pointers to) objects of type B can be passed to functions expecting objects of type A with no run-time cost. The member b would then be at offset 8. The following definition is similar.

```
class C : A { int c; };
```

[Note that Java uses the word 'extends' instead of ':'.]

Now, suppose one has multiple inheritance (as in C++) so we can inherit the members and methods from two or more classes and writes:

```
class D : B,C { int d; };
```

Firstly there is the observation that passing an object of type D to a routine expecting C must involve a run-time cost of an addition so that element c can be accessed at offset 8 in the received C. (This assumes that B is stored at offset zero in D.)

There is also the more fundamental question as to what are the members of objects of type D. Does it have 7 (3 in both B and C and also d)? Or maybe 5 (a1, a2, b, c, d)? C++ by default has 7, i.e. the two copies of A are separate. In C++ we can cause the two copies of A to share by replacing the definitions for B and C by

```
class B : virtual A { int b; };
class C : virtual A { int c; };
class D : B,C { int d; };
```

But now the need to treat objects of type D as objects of type B or C means that the storage layout for D is likely to be implemented as

```
struct { A *__p, int b; A *__q, int c; A x; } s =
          { &s.x, 0, &s.x, 0, { 0, 0 }};
```

I.e. there is a single A object and both the __p field of the logical B object and the __q field of the logical C object share it. This is necessary so that a D object can be passed to routines which expect a B or a C object—but note that is causes declarations like B x to be of 16 bytes: 8 for the A, 4 for the indirect pointer (after all, routines need to be compiled which access the elements of a B not knowing whether is it a 'true' B or actually a D).

Such arguments are one reason why Java omits multiple inheritance. Its interface facility provides similar facilities.

The above details only dealt with ordinary members and inheritance. Suppose we now add member functions (methods). Firstly consider the implementation of a method like:

```
class C {
  int a;
  static int b;
  int f(int x) { return a+b+x;}
};
```

How is f() to access its variables? Recall that a static variable is per-class, and a non-static one per-instance. Hence the code could be re-written as:

```
int unique_name_for_b_of_C;
class C {
  int a;
  int f(int x) { return a + unique_name_for_b_of_C + x;}
};
```

Now consider a call to f() such as c.f(x) where c is of class C. This is typically implemented as an ordinary procedure call unique_name_for_f_of_C(c,x) and the definition of f() implemented as:

```
 int unique_name_for_f_of_C(C c, int x)
 {   return c.a                      // fixed offset from c
          + unique_name_for_b_of_C  // global variable
          + x;                       // argument
};
```

Let us now turn to how inheritance affects this model of functions, say in Java:

```
class A { void f() { printf("I am an A"); }};
class B:A { void f() { printf("I am a B"); }};
A x;
B y;
void g(A p) { p.f(); }
main() { x.f();        // gives: I am an A
         y.f();        // gives: I am a B
         g(x);         // gives I am an A
         g(y);         // gives what?
       }
```

There are two cases to be made; should the fact that in the call p.f(); we have that p is of type A cause A::f(); to be activated, or should the fact that the value of p, although now an A was originally a B cause B::f(); to be activated and hence "I am a B" to be printed? In Java the latter happens; by default in C++ the former happens, to achieve the arguably more useful Java effect it is necessary to use the virtual keyword:

```
class A { virtual void f() { printf("I am an A"); }};
class B:A { virtual void f() { printf("I am a B"); }};
```

So how is this implemented? Although it appears that objects of type `A` have no data, they need to represent that fact that one or other `f` is to be called. This means that their underlying implementation is of a storage cell containing the address of the function to be called. (In practice, since there may be many virtual functions a *virtual function table* is used whereby a class which has one or more virtual functions has a single additional cell which points to a table of functions to be called by this object. This can be shared among all objects declared at that type, although each type inheriting the given type will in general need its own table).

For more details on this topic the interested reader is referred to Ellis and Stroustrup "The annotated C++ reference manual".

25 Data types

In the course so far we have essentially ignored the idea of data type. Indeed we have used '`int x = 1`' and '`let x = 1`' almost interchangeably. Now we come to look at the possibilities of typing. One possibility (adopted in Lisp, Prolog and the like) is to decree that types are part of an Rvalue and that the type of a name (or storage cell) is the value last stored in it. This is a scheme of *dynamic types* and in general each operation in the language need to check whether the value stored in the cell is of the correct type. (This manifested itself in the lambda calculus evaluator in section 12 where errors occur if we apply an integer as a function or attempt to add a function to a value).

Most mainstream languages associate the concept of data type with that of an identifier. This is a scheme of *static types* and generally providing an explicit type for all identifiers leads to the data type of all expressions being known at compile time. The *type* of an expression can be thought of as a constraint on the possible values that the expression may have. The type is used to determine the way in which the value is represented and hence the amount of storage space required to hold it. The types of variables are often declared explicitly, as in:

```
float x;
double d;
int i;
```

Knowing the type of a variable has the following advantages:

1. It helps the compiler to allocate space efficiently, (`int`s take less space than `double`s).

2. It allows for *overloading*. That is the ability to use the same symbol (e.g. `+`) to mean different things depending on the types of the operands. For instance, `i+i` performs integer addition while `d+d` is a `double` operation.

3. Some type conversions can be inserted automatically. For instance, `x := i` is converted to `x := itof(i)` where `itof` is the conversion function from `int` to `float`. Similarly, `i+x` is converted to `itof(i)+x`.

4. Automatic type checking is possible. This improves error diagnostics and, in many cases, helps the compiler to generate programs that are incapable of losing control. For example, `goto L` will compile into a legal jump provided `L` is of type label. Nonsensical jumps such as `goto 42` cannot escape the check. Similar considerations apply to procedure calls.

Overloading, automatic type conversions and type checking are all available to a language with dynamic types but such operations must be handled at runtime and this is like to have a drastic effect on runtime efficiency. A second inherent inefficiency of such languages is caused by not knowing at compile time how much space is required to represent the value of an expression. This leads to an implementation where most values are represented by pointers to where the actual value is stored. This mechanism is costly both because of the extra indirection and the need for a garbage collecting space allocation package. In implementation of this kind of language the type of a value is often packed in with the pointer.

One advantage of dynamic typing over static typing is that it is easy to write functions which take a list of any type of values and applies a given function to it (usually called the `map` function). Many statically typed languages render this impossible (one can see problems might arise if lists of (say) characters were stored differently from lists of integers). Some languages (most notably ML) have `polymorphic types` which are static types but which retain some flexibility expressed as parameterisation. For example the above `map` function has ML type

$$(\alpha \texttt{ -> } \beta) \texttt{ * } (\alpha \texttt{ list}) \texttt{ -> } (\beta \texttt{ list})$$

If one wishes to emphasise that a statically typed system is not polymorphic one sometimes says it is a *monomorphic type system.*

Polymorphic type systems often allow for *type inference*, often called nowadays *type reconstruction* in which types can be omitted by the user and reconstructed by the system. Note that in a monomorphic type system, there is no problem in reconstructing the type of λ`x. x+1` nor λ`x. x ? false:true` but the simpler λ`x. x` causes problems, since a wrong 'guess' by the type reconstructor may cause later parts of code to fail to type-check.

We observe that overloading and polymorphism do not always fit well together: consider writing in ML λ`x. x+x`. The `+` function has both type

`(int * int -> int)` and `(real * real -> real)`

so it is not immediately obvious how to reconstruct the type for this expression (ML rejects it).

It may be worth talking about inheritance based polymorphism here.

Finally, sometimes languages are described as *typeless.* BCPL (a forerunner of C) is an example. The idea here is that we have a single data type, the word (e.g. 32-bit bit-pattern), within which all values are represented, be they integers, pointers or function entry points. Each value is treated as required by the operator which is applied to it. E.g. in `f(x+1,y[z])` we treat the values in `f`, `x`, `y`, `z` as function entry point, integer, pointer to array of words, and integer respectively. Although such languages are not common today, one can see them as being in the intersection of dynamically and statically type languages. Moreover, they are often effectively used as intermediate languages for typed languages whose type information has been removed by a previous pass (e.g. in intermediate code in a C compiler there is often no difference between a pointer and an integer, whereas there is a fundamental difference in C itself).

26 Source-to-source translation

It is often convenient (and you will have seen it done several times above in the notes) to explain a higher-level feature (e.g. exceptions or method invocation) in terms of lower-level features (e.g. gotos or procedure call with a hidden 'object' parameter).

This is often a convenient way to specify precisely how a feature behaves by expanding it into phrases in a 'core' subset language. Another example is the definition of

$$\texttt{while } e \texttt{ do } e'$$

construct in Standard ML as being shorthand (syntactic sugar) for

$$\texttt{let fun f() = if } e \texttt{ then (}e'\texttt{; f()) else () in f() end}$$

(provided that `f` is chosen to avoid clashes with free variables of e and e').

A related idea (becoming more and more popular) is that of compiling a higher-level language (e.g. Java) into a lower-level language (e.g. C) instead of directly to machine code. This has advantages of portability of the resultant system (i.e. it runs on any system which has a C compiler) and allows one to address issues (e.g. of how to implement Java `synchronized` methods) by translating them by inserting mutex function calls into the C translation instead of worrying about this and keeping the surrounding generated code in order.

27 Spectrum of Compilers and Interpreters

One might think that it is pretty clear whether a language is compiled (like C say) or interpreted (like BASIC say). Even leaving aside issues like microcoded machines (when the instruction set is actually executed by a lower-level program at the hardware level "Big fleas have little fleas upon their backs to bite them") this question is more subtle than first appears.

Consider Sun's Java system. A Java program is indeed compiled to instructions (for the Java Virtual Machine—JVM) which is then typically interpreted (one tends to use the word 'emulated' when the structure being interpreted resembles a machine) by a C program. One recent development is that of Just-In-Time—JIT compilers for Java in which the 'compiled' JVM code is translated to native code just before execution.

If you think that there is a world of difference between emulating JVM instructions and executing a native translation of them then consider a simple JIT compiler which replaces each JVM instruction with a procedure call, so instead of emulating

```
iload 3
```

we execute

```
iload(3);
```

where the procedure `iload()` merely performs the code that the interpreter would have performed.

Similarly, does parsing our simple expression language into trees before interpreting them cause us to have a compiler, and should we reserve the word 'interpreter' for a system which interprets text (like some BASIC systems)?

So, we conclude there is no line-in-the-sand difference between a compiled system and and an interpreted system. Instead there is a spectrum whose essential variable is how much work is done statically (i.e. before execution starts) and how much is done during execution.

In our simple lambda evaluator earlier in the notes, we do assume that the program-reading phase has arranged the expression as a tree and faulted any mismatched brackets etc. However, we still arrange to search for names (see `lookup`) and check type information (see the code for $e_1 + e_2$) at run-time.

Designing a language (e.g. its type system) so that as much work as possible *can* be done before execution starts clearly helps one to build efficient implementations by allowing the compiler to generate good code.

28 Debugging

One aspect of debugging is to allow the user to set a 'breakpoint' to stop execution when control reaches a particular point. This is often achieved by replacing the instruction at the breakpointed instruction with a special `trap` opcode.

Often extra information is stored the ELF object file and/or in a stack frame to allow for improved runtime diagnostics. Similarly, in many languages it is possible to address all variables with respect to `SP` and not use a `FP` register at all; however then giving a 'back-trace' of currently active procedures at a debugger breakpoint can be difficult.

Debuggers often represent a difficult compromise between space efficiency, execution efficiency and the effectiveness of the debugging aids.

[The end]