Cambridge University Computer Laboratory

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

Compiling Techniques

http://www.cl.cam.ac.uk/users/am/teaching/cmptech

Alan Mycroft am@cl.cam.ac.uk

Lent 1998

These notes are largely based on those of Dr. M. Richards

Summary

This course starts with a brief survey of various features and concepts common to many programming languages, together with an outline of the kind of target code they require. The second part of the course covers the design of the various parts of a compiler.

The course is intended to study compilation of a range of languages and accordingly syntax for example constructs will be taken from various languages (with the intention that the particular choice of syntax is reasonably clear).

In terms of programming languages in which parts of compilers themselves are to be written, the preference varies between pseudo-code (as per the 'Data Structures and Algorithms' course) and language features (essentially) common to C/C++/Java. The language Standard ML (which the Diploma and Part II (general) students will see as part of the 'Functional Programming' course) is used when it significantly simplifies the code compared to C.

The following books contain material relevant to the course.

Compilers – Principles, Techniques, and Tools
A.V.Aho, R.Sethi and J.D.Ullman
Addison-Wesley (1986)
Ellis Horwood (1982)

Compiler Design in Java/C/ML (3 editions)
A.Appel
Cambridge University Press (1996)

Compiler Design
R.Wilhelm and D.Maurer
Addison Wesley (1995)

Introduction to Compiling Techniques
J.P.Bennett
McGraw-Hill (1990)

A Retargetable C Compiler: Design and Implementation
C.Frazer and D.Hanson
Benjamin Cummings (1995)

Compiler Construction
W.M.Waite and G.Goos
Springer-Verlag (1984)

High-Level Languages and Their Compilers
D.Watson
Addison Wesley (1989)

Teaching and Learning Guide

[This is a Computer Laboratory mandated section of all lecture courses.]

The lectures largely follow the syllabus for the course which is as follows:

```
Survey of language constructs and their implementation.
  Expressions, applicative structure, lambda expressions. Environments
  and a simple lambda evaluator. Evaluation of function calls using
  Dijkstra displays. Static and dynamic chains, optimisations.
  Situations where a simple stack is inadequate. Argument passing
  mechanisms.  Implementation of labels, jumps, arrays and exceptions.
  Object-oriented storage allocation.  Static and dynamic data types.
```

```
Lexical analysis and syntax analysis.
  Regular expressions and finite state machine implementations.
  Grammars, Chomsky classification of phrase structured grammars.
  Parsing algorithms: recursive descent, precedence and SLR(k).
  Parser matrix compaction. Syntax error recovery.

Translation phase.
  Intermediate code design. Translation of commands, expressions and
  declarations.

Code generation.
  Typical machine codes. Code generation from the parse tree and from
  intermediate code. Simple optimisation.

Compiler compilers.
  Summary of Lex and Yacc. A rule base parse tree pattern matching
  algorithm.

Runtime.
  Object modules and linkers. Resolving external references. Debuggers,
  break points and single step execution. Profiling, interpreters,
  portability.
```

A good source of exercises is the past 10 or 20 years' (sic) Tripos questions in that most of the basic concepts of block-structured languages and their compilation to stack-oriented code were developed in the 1960s. The course 'Optimising Compilation' in CST (part II) considers more sophisticated techniques for the later stages of compilation and the course 'Comparative Programming Languages' considers programming language concepts in rather more details.

Chapter 1

Introduction

1.1 Assignments, Lvalues and Rvalues

A feature common to all computers is that they have memory and the fundamental ability to access and update the contents of arbitrary words of memory. Correspondingly, a basic feature of most programming languages is the assignment command. For example,

```
x := 3;
x := y+1;
x := x+1;
```

In some languages the form of the assignment can be more general:

```
i := (a>b ? j : k);
v[i] := v[a>b ? j : k];
v[a>b ? j : k] := v[i];
(a>b ? j : k) := i;
```

In the above examples the notation `E ? E1 : E2` denotes a conditional expression, and `v[i]` is a subscripted expression.

We are tempted to describe the syntactic form of such assignments as

```
E1 := E2;
```

where `E1` and `E2` are both expressions, and to describe the meaning (semantics) of the command as the execution of three steps.

1. Evaluate E1 to give an address.
2. Evaluate E2 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).

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.).

1.2 Definition of Variables

Most languages allow the programmer to declare variables. In C/Java for example, variables may be declared as follows:

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

This causes a new storage cell to be found and associated with the name (or identifier) `p`. The contents of this cell is initially set to the **float** number `3.4`. Then a second new storage cell (identified by `q`) is initialised with the contents of `p` (clearly `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 in left hand mode (or Lmode) and the Lvalue obtained associated with the identifier, 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 this 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;
```

Observe that Lmode evaluation (like Rmode evaluation) in general requires some computation to be done. Consider the program:

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

Note that `m[i,i]` is an expression which (like `i`) can be evaluated in both L and R modes. Some expressions have only Rvalues (e.g. `x+1`) and so may not be used, for instance, on the left hand side of an assignment.

1.3 Names

The term name is often used to mean different things and confusion frequently results. It is sometimes, for instance, unwittingly used as a synonym for Lvalue. In these notes we will always use the word name to refer to the identifiers coined by a programmer and used in a program. The only property we require of names is that, given two of them, it is possible to determine whether they are the same (i.e. same sequence of letters, digits, etc.) or not.

1.4 Expressions and Commands

The simplest programming languages (e.g. assembly language) consist of sequences of commands. When obeyed they cause basic operations to take place, e.g. subtraction of Loc1 from Loc2 using the 'gas' assembly language for Intel Pentium architecture would be written

```
subl   Loc1,Loc2
```

In ordinary mathematics there are no commands as such. Expressions are used to build up complicated results, e.g. $x + y$ means:

"the sum of x and y"

and never

"add x to y"

As programming languages developed, ordinary mathematical notation found its way into commands, e.g.

```
x := (a+b)/c+d
```

which is shorter and easier to follow than the Pentium code which achieves the same effect:

```
movl 8(%ebp),%eax          get a
addl 12(%ebp),%eax         add b
cltd                       prepare division
idivl 16(%ebp)             divide by c
addl 20(%ebp),%eax         add d
movl %eax,x                store in x
```

It is the existence of *commands* which makes programming languages quite different from the rest of mathematics. We will first consider expressions in the absence of commands.

1.5 Expressions

The characteristic feature of an expression is that it yields a value as a result. We have seen that it can be either an Lvalue or an Rvalue (depending on the context). However, without the assignment commands Lvalues are of no interest, and as in ordinary mathematics we shall, for the time being, only be concerned with Rvalues.

One important property relating to expressions is called *referential transparency*. In essence, this means that if we wish to find the value of an expression that contains subexpressions, then the only thing we need to know about the subexpressions are their values. We are already familiar with this, e.g.

```
sin(6),  sin(5+1),  sin(30/5)
```

all have the same value. If an expression has this property then the order of evaluation does not matter.[1] Another example is

[1] C expressions like `(x++) + (x--)` are not referentially transparent.

```
sin(x+y/z)
```

which has the same value as

```
sin(p)
```

provided `p` has the same value as `x+y/z`. We could also say that

```
sin(x+y/z)
```

has the same value as

```
sin(p) where p = x+y/z
```

To be able to discuss this more formally within the concept of a programming language it is useful to extend the traditional idea of expression (name, number or operator applied to two smaller sub-expressions) to formalise such an idea of local definition (traditionally left informal (`where` above) in mathematics) by allowing an expression also to be of the form

`let` var = expression1 `in` expression2

so we more properly write

```
let p = x+y/z
in sin(p)
```

This process of replacing an expression by a name which has the same value is called *abstraction* and is of fundamental importance in programming languages. (It is the same idea as that which, in a language like Java, we use to replace multiple occurrences of command sequences with calls to a procedure whose body is one copy of the sequence).

1.6 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. 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
```

Yet another way is to use lambda calculus (described later):

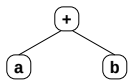
```
(λa. a+3/a)(2+3/7)
(λx. x+y+3/x)(y+2/y)
```

These two methods are exactly equivalent and have the same meaning. The name `t` 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 with local definitions are known as *bound variables*.

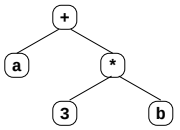
Note that in programming languages like ML, λ is often written `fn` and we will occasionally adopt this convention in these notes.

1.7 Applicative structure

Arithmetic expressions (and commands) can be represented as a structure which makes clear which operators act on which operands. For example, `a+b` can be represented as `+(a, b)` or diagrammatically as follows:



and `a+3*b` can be represented as `+(a, *(3,b))` or as

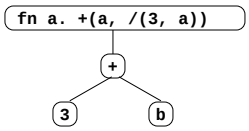


This structure is called the *applicative structure* of the expression. Applicative structures are difficult to read because of the deep nesting of brackets, but they help to emphasise the uniform way in which complicated expressions (and commands) can be built out of smaller ones. The rule for evaluation is simple and is as follows:

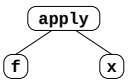
1. Evaluate the operands (in any order).

2. Apply the operator to the operand values.

One could regard a lambda abstraction as an operator of a complicated sort. For instance, the applicative structure of {λ`a. a+3/a`}`(3+b)` could be:



It is probably better to introduce a new basic operator `Apply` to deal with function applications, so that the structure for `f(x)` is:

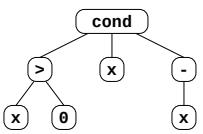


This maintains the convention that all applicative expression operators are constants of the system.

One apparent exception to the evaluation rule is the evaluation of the conditional expression. Consider

```
x>0 ? x : -x
```

One possible applicative form of this might be



where `Cond` is defined to have the rather special meaning of evaluating only one of the two alternatives depending on the result of the condition. This is necessary in `x=0 ? 1 : 1/x` for instance.

1.8 Lambda calculus

There are three main reasons for studying lambda calculus when considering programming languages. These are:

1. It is a useful notation for specifying the scope rules of identifiers in programming languages, and helps to demonstrate such notions as Algol 60 **own** variables and "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.

A lambda expression is an expression composed of *identifiers* (e.g. `x`, `y` and `z`), *abstractions* of the form $\lambda I.E$ where I is an identifier, and *applications* of the form FA where E, F and A are other (smaller!) lambda-expressions. We will also allow numeric constants as lambda-expressions. Parentheses and braces may be used for grouping. The following are examples of lambda expressions:

```
λx. x
f(λg. g x)
f(x, y)
{λ(h,x). (h x) y} ((λt. t),(λr. r))
(λf. (λg. g g) (λg. (f g) g))
```

To enhance readability, certain identifiers (e.g. those spelt with non-alphanumeric characters) may be used as infixed dyadic operators. For example, `x+y*z` may be written to denote `+(x,*(y,z))`.

In an abstraction $\lambda I.E$, the identifier I is called the *bound variable* and the expression E is called the *body*.

Evaluation of lambda expressions is by means of two simple rewrite rules.

- **α-conversion.** The expression $(\lambda \mathtt{x}.M)$ can be replaced by $(\lambda \mathtt{y}.M')$ where `y` is any identifier that does not occur in M and where M' is M rewritten with all occurrences of the identifier `x` replaced by the identifier `y`.
- **β-reduction.** The expression $(\lambda \mathtt{x}.M)A$ can be replaced by a copy of M with all occurrences of the bound variable `x` replaced by the argument A, provided that
  1. M contains no (inner) bound occurrences of identifier `x`, and
  2. A contains no free variables that are bound in M.

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.

$$\begin{aligned}
\{\lambda \mathtt{t}.\mathtt{t}+16\}(4) &= 20 \\
\{\lambda(\mathtt{a},\mathtt{b}).\mathtt{a}+\mathtt{b}\}(5,6) &= 11 \\
\{\lambda \mathtt{f}.\{(\lambda \mathtt{x}.\mathtt{f}(\mathtt{x}+1))(3)\}\}(\lambda \mathtt{y}.\mathtt{y}*2) &= (\lambda \mathtt{x}.\{\lambda \mathtt{y}.\mathtt{y}*2\}(\mathtt{x}+1))(3) \\
&= \{\lambda \mathtt{y}.\mathtt{y}*2\}(4) \\
&= 8
\end{aligned}$$

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 n

1. Find the smallest textually enclosing lambda expression.
2. Compare n 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.

1.9 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.

1.10 A short interlude on recursion

When a function is defined in terms of itself as in the following C/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`. Below we see how to use a stack for this purpose.

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 $5! = 120$.]

1.11 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` he was defining. 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 by the most ML systems to

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

1.12 The Y operator

At first sight, the `rec` construction seems to have no lambda calculus equivalent; however, postulating a new operator `Y` 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 `F(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. Any value of `g` satisfying `H(g)=g` is called a *fixed point* of `H`. Let us postulate a special function `Y` that yields the fixed point of its argument. If `Y(H)=g`, then `H(g)` will also `= g`. Notice that

```
H(Y(H)) = H(g) = g = Y(H)
```

Observe that we can now write

```
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 that we can evaluate a call of `f` knowing no more about `Y` than the property that `H(Y(H))=Y(H)`. 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 perhaps entertaining to observe that some ordinary lambda expressions (not involving `Y`) have the property we need. For example, define:

```
let Y = λf. (λg. g(g))(λg. f(g(g)))
```

With this definition of `Y`, we see that

```
Y(H)  =  {λf. (λg. g(g))(λg. f(g(g)))} (H)  [definition of Y]
      =  (λg. g(g)) (λg. H(g(g)))           [lambda reduction]
      =  (λg. H(g(g))) (λg. H(g(g)))        [lambda reduction]
      =  H( (λg. H(g(g))) (λg. H(g(g))) )   [lambda reduction]
      =  H( Y(H) )                          [combining the two lines above]
```

This helps us to believe that `Y` both exists and is a respectable operator.

1.13 Object-oriented languages: optional for 1998

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:

```
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 designed could follow simply; now such standards committees have to use their (not generally reliable!) powers of decision.

1.14 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 an applicative structure and evaluate it in an environment that is initially empty. The applicative structure that we shall use has 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;
```

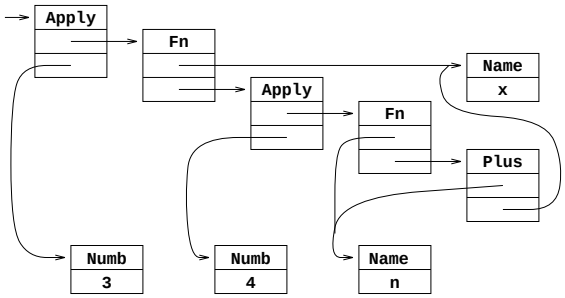
Later in the course we will see this also can be described by the context-free grammar

```
<Expr> -> <Name> | <Int> | <Expr>+<Expr>
                 | fn <Name>.<Expr> | <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[2] 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:

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

```
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* 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.

1.14.1 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.

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)
}
```

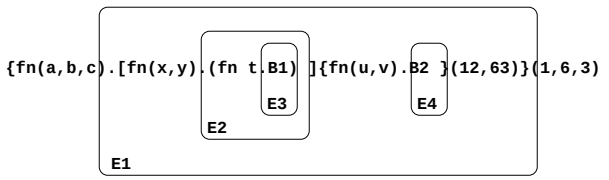
Apply the `eval` function by hand to this tree and an empty environment and draw the structure of every environment that is used in the course of the evaluation.

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

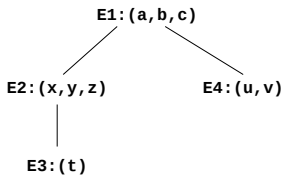
3. Modify the interpreter to use dynamic scoping; is it now possible to write a finite program that would cause this evaluator to attempt to create an infinitely long environment?

1.15 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:



The environment structure can be represented as a tree as follows.



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 `E3` which looks like:

```
E1:(a,b,c)              level 1
E2:(x,y,z)              level 2
E3:(t)                  level 3
```

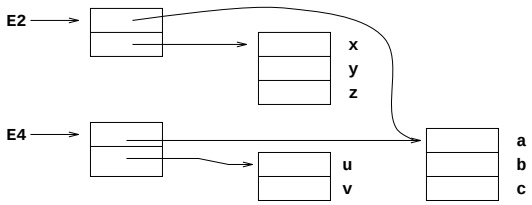
We can associate with any name used in B1 an 'address' consisting of a pair of numbers, namely, a level number and a position within that level. For example:

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

Similarly within B2:

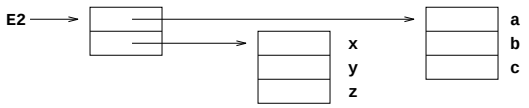
```
a:(1,1)   b:(1,2)   c:(1,3)
u:(2,1)   v:(2,2)
```

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

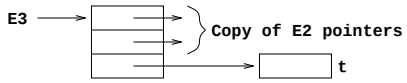


A vector such as the one pointed to by `E2` is called a *display* (first coined by Dijkstra). Notice that while evaluating `B2` in environment `E4` we may use the "address" (2,1) to access the variable u. It should be clear that the pointer to the current display provides sufficient information to access 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 `(fn t.B1)` in the last example is the pointer to the display vector for `E2`.

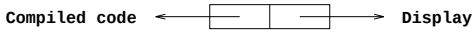


In order to apply this closure to an argument value `k`, we must first create a new display which consists of a copy of `E2` augmented with a new level. The new display will be as follows:



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 to construct the new display followed by code to evaluate the body and the closure is often represented as a pair of pointers:

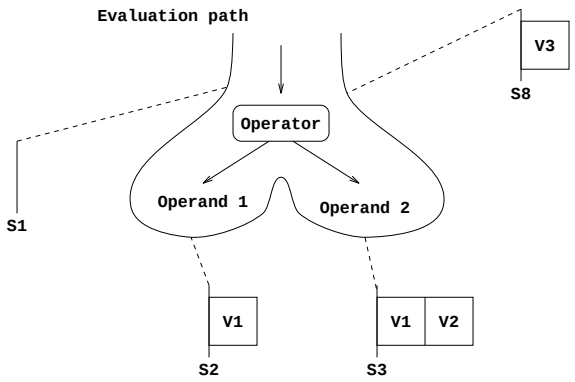


Exercise

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

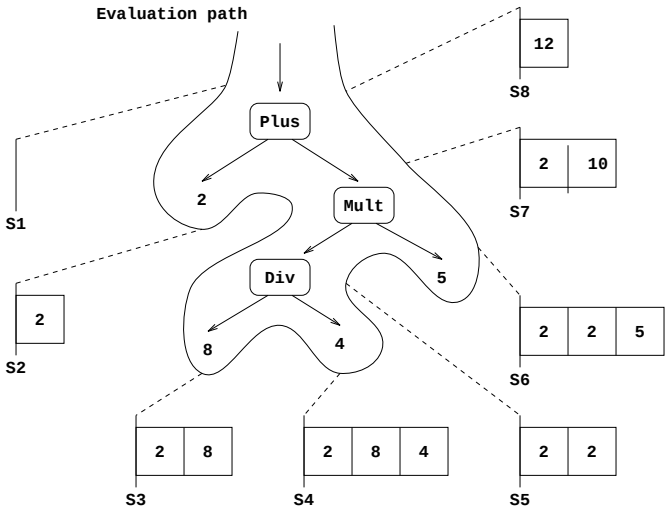
1.16 Evaluation using a stack

A stack has the property that items can be placed on the top or removed from the top. Such a storage organisation is sometimes called a LIFO (last in first out) store. Stacks are often used in the implementation of programming languages since they provide an efficient means of allocating work space used in the evaluation of expressions. To evaluate an expression consisting of an operator applied to two operands, the operands are first evaluated and then the operator applied to the results obtained. This can be shown diagrammatically as follows:



The evaluation proceeds along the evaluation path. The initial stack (S1) is empty. After evaluating the first operand the stack (S2) contains one value namely `V1`. The second operand causes a second value to be placed on the stack (S3) and, finally, the operator takes these two value from the stack replacing them by the single result `V3` as shown in stack (S4) ((S8) in the picture!).

This mechanism works for expression of any complexity; for example, the evaluation of `2+(8/4)*5` is as follows:



Observe that we only use the applicative structure to help us find the evaluation path. To perform the evaluation it is sufficient to know the order in which the items appear in the evaluation path, namely:

```
2 8 4 / 5 * +
```

This order is called the *reverse polish form* of the expression `2+(8/4)*5` and it is important since

- it is easy to obtain
- and it is an order of evaluation that works.

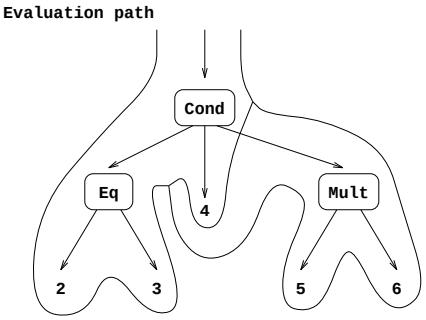
If we connect the operator to the path on the way down rather than on the way up we obtain the sequence:

```
+ 2 * / 8 4 5
```

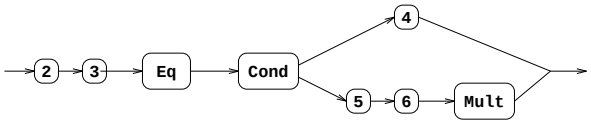
This is called the polish (or prefix) notation and is close to the functional form

```
plus(2, mult(div(8, 4), 5))
```

Notice that monadic operators can be transformed to reverse polish form. For example, the reverse polish form of `2 * (-3)` is `2 3 ~ *` where `~` denotes the monadic minus operator. Some expressions are not quite so susceptible. Consider the expression: `2=3 ? 4 : 5*6`.



The choice of evaluation path depends on the value of the boolean expression. The closest approach to reverse polish form might be



As far as compiled code is concerned, it does not matter that the reverse polish form cannot be written as a linear sequence of items.

1.17 The use of a stack in function calls

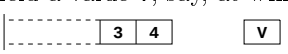
The observation that function calls are in many respects similar to applicative expression operators leads one to suspect that it possible to use a stack in the implementation of function applications. Consider the call `f(3,4)` of a function defined as follows:

```
let f(x, y) = E
```

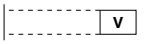
If we regard `f` as an ordinary operator we find that, at the time when `f` is to be applied, the top two items of the stack are `3` and `4`.

The evaluation of the call involves the evaluation of `E` and so one might provide a new stack on which to perform this evaluation.

Ultimately the evaluation of `E` will yield a value `V`, say, at which time the stacks will be as follows:



Then finally this result is carried back to the previous stack frame and the old control path resumed. At that moment the stack will look as follows:

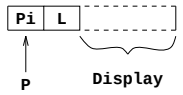


1.18 An implementation of function calls

What follows is a description of a simple mechanism for the implementation of function calls. Many variations and optimisations are possible and so the implementation given here should only be regarded as a guide.

When a function is called it is allocated a piece of stack (sometimes called a stack frame, or activation record) that is used to hold temporary values that are needed during the course of the evaluation of the body. A stack pointer (`P`) will be used to point to the base of the stack frame that is currently active.[3] When the function call is complete the previous stack frame must be re-instated and execution must resume at the point just after the call. For this to be possible the previous stack pointer (`Pi`) and the return address (`L`) are held in the current stack frame.

When a function is applied, a new display must be constructed so that the free variables of the function are accessible during the evaluation of the body. This new display could be placed in the stack frame allocated for the function call, and so combined with the previous stack pointer and return address the base of a stack frame is as follows:

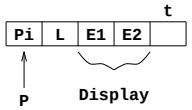


The size of the display depends on the textual depth of nesting of the function body and so is known when the function is being compiled. Local variables of the function can be allocated cells within the stack frame and accessed using constant offsets relative to the stack frame pointer.

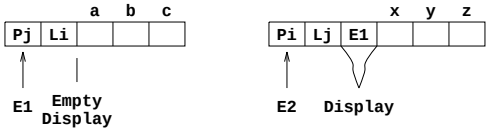
The purpose of the display is to allow for access to the free variables of the function. Since every free variable is a local variable of some other function it would be sufficient for each element of the display to be a pointer to the appropriate stack frame. The display can, thus, be a vector of stack pointers. In the example on page 11, the environment `E3` was as follows:

```
E1:(a,b,c)              level 1
E2:(x,y,z)              level 2
E3:(t)                  level 3
```

When execution is in the environment, the current stack will be as follows:



where `E1` and `E2` are pointers to the stack frames containing `(x,y,z)` and `(a,b,c)`.



Notice that all the variables `a`, `b`, `c`, `x`, `y`, `z` and `t` can be accessed via `P`.

```
a: P[2][2]   b: P[2][3]   c: P[2][4]
x: P[3][3]   y: P[3][4]   z: P[3][5]
t: P[4]
```

Thus `P` can be regarded as a representation of the current environment and so can be used, for example, as the environment part of a closure.

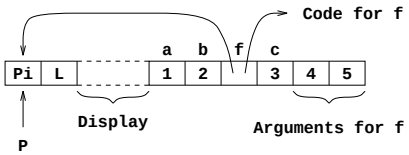
[3] It is convenient to assume the stack is essentially an array of 4-byte or 8-byte locations (capable of holding an integer or a pointer) and to describe accessing the 5th location in the current stack frame as `P[4]` etc.

An example

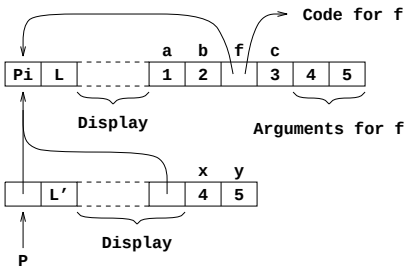
Consider the following fragment of program:

```
...
let r() =
{ 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:



If `E` is the environment part of the closure for `f`, the display can be constructed by compiled code at the start of `f` consisting of a sequence of assignments (assuming the size of the new display is `3`).

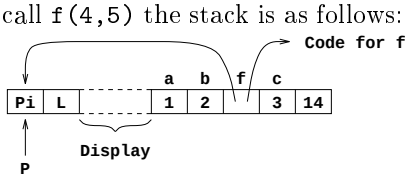
```
P[2] := E[2]      // make a copy
P[3] := E[3]      // of the display
P[4] := E         // add one more level
```

The arguments can now be popped from the calling stack frame and placed in their appropriate positions in the current frame.

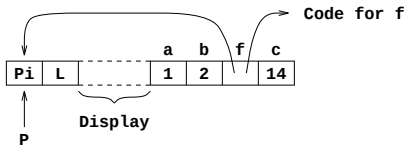
Notice that at this moment `a`, `b`, `x` and `y` can be addressed via `P` as follows:

```
a: P[4][4]     b: P[4][5]
x: P[5]        y: P[6]
```

Just after the return from the call `f(4,5)` the stack is as follows:



and after the assignment to c the stack is as follows:



The implementation of this mechanism can be reasonably efficient provided there are sufficient general registers available to hold the `P` pointer and all the display pointers. These registers could be set up on entry to the function. This would allow all the variables (both local and free) to be directly addressed. On some machines saving and restoring display registers can be done efficiently using the store-multiple and load-multiple instructions.

1.19 Static chain method

Instead of storing the entire display in each stack frame, the display could be held as a chain of pointers (called the *static chain*) through the stack frames corresponding to the different levels of the display. The base of a stack frame now consists of:

1. The previous stack pointer `P`.
2. The return address `L`.
3. The static chain pointer `S`.

The length of the static chain is equal to the textual depth of the current piece of code that is being executed and is, of course, equal to the size of the current display. The elements of the display are the pointers to all the stack frames that the static chain passes through. When a function is called or returned from, the `P` pointer changes and so does the current display. At such a moment the registers used to hold the display elements can be set from `P` by a short sequence of load instructions compiled into the code. A possible calling sequence for the call `f(args ...)` is as follows:

```
        code to deal with the arguments
        E    := <environment part of f>
        B    := <code address part of f>
        NEWP := <pointer to the first free stack position>
        L    := ret; JMP B       subroutine jump to f
ret:    D3   := P[2]   restore display (the number of
        D2   := D3[2]  instructions here depends on the textual
        D1   := D2[2]  depth of the call)
```

The code at the entry point of `f` could be:

```
        NEWP[0] := P  save the previous P pointer
        P := NEWP     change to the new P pointer
        P[1] := L     save the return address
        P[2] := E     save the static chain link
        D2 := E       establish the display
        D1 := D2[2]
```

and the code to return from `f` could be:

```
L := P[1]      find the return address
P := P[0]      restore the previous P pointer
JMP L          branch to the return address
```

One way to examine the possible actual code sequences for (say) the Pentium architecture is to create a file `foo.c` containing C such as

```
extern int g(int);
int f(int x) { return g(x+1)+2; }
```

and then to compile it using (say) gcc with

```
gcc -S foo.c
```

This creates a file `foo.s` containing generated machine for for the host architecture (so chose a Pentium machine to see Pentium code!). Beware: 'gcc' constructs a downwards-growing stack as opposed to the upwards-growing stack used in these notes (see variations below) so everything will not correspond exactly.

1.20 Variations

The calling sequence outlined above should only be regarded as a guide, since many variations and optimisations are possible. A few of the possibilities are as follows.

1. One may adopt the convention that a subroutine must restore all the registers that it uses. This will mean that there is no need to restore the display just after the call since these registers will have been restored by the return sequence. In general, this produces more compact code by transferring work from the call sequence to the save and return sequences since, in large programs, there are typically between five to ten times as many calls than procedure definitions.
2. The outermost display level is the same for every display and so may be regarded as constant needing no code to repeatedly set it. The innermost display level is represented by the P pointer. In practice only about 3% of variable references belong to the intermediate levels and so many compilers only set up these levels when needed. This improves the efficiency of most procedures.
3. An advantage of the static chain approach is that the position of the arguments in a stack frame depends only on their number and type and not on the textual depth of the procedure. This means that the code in the call to deal with the arguments can (possibly) place them directly in the correct positions in the new stack frame.
4. A decision must be made whether to allocate one (large enough) contiguous region of store for the stack, or to break the stack on each procedure call.
5. In most programming languages procedures are constant (not updatable by assignment) and so no storage cells need be allocated for them. When such a procedure is referenced the compiler can easily construct the environment pointer and entry address that would have been in the closure. Notice that the required environment pointer is the pointer to the stack frame that would have held the closure if a storage cell had been allocated. Proper closures are needed for procedure variables, and they are usually used when procedures are passed as parameters to other procedures.
6. Often extra information is stored in a stack frame to allow for improved runtime diagnostics. This requires a difficult compromise between space efficiency, execution efficiency and the effectiveness of the debugging aids.

1.21 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:

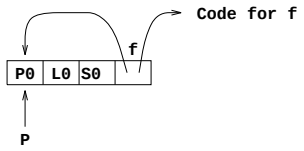
```
let f() = { let a = 0
            in &a
          }
let p = f()
...
```

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) avoid this problem by only allowing pointers into the heap.

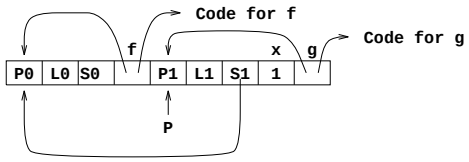
Some other objects such as functions and arrays contain implicit pointers to the stack and so have to be restricted if a stack implementation is to work. Consider:

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

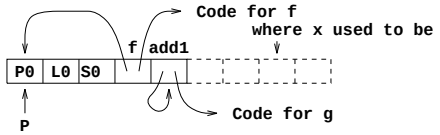
The result of `f(1)` should be a function which will add one to its argument. Thus one might hope that `add1(23)` would yield 24. It would, however, fail if implemented using a simple stack. We can demonstrate this by giving the state of the stack at various stages of the evaluation. Just after the `f` 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. ML clearly does allow objects to be returned from procedure calls; we shall see how this is achieved in section 1.26.

1.22 Parameter passing mechanisms

When we apply a function to an argument we write:

```
f(arg)
```

where `f` is a function (or more strictly an expression yielding a function) and `arg` is any expression. In a referentially transparent language all we need to know about arg is its value; but, as we have seen, there are two sorts of value that might be considered, namely an Lvalue or an 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 a = 10
r(a)
// a now equals 11
```

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

1.23 Note on Algol call by name

Algol 60 is a language that attempted to be mathematically clean and was influenced by the simple calling 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 of identifiers was 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.

When an actual parameter in Algol is *called by name* it is not evaluated in either Lmode or Rmode 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.]

1.24 Implementation of call by name

A parameter called by name is thus like a parameterless function and so can be represented in the machine as a closure consisting of a pointer to code and an environment pointer. Some optimisations are possible since there are no arguments to the closure and since its result can be handled in a special way. But one should notice that the full recursive calling mechanism may be necessary. Consider:

```
INTEGER PROCEDURE sum(p, n, x)
   VALUE n; INTEGER p, n, x;
   BEGIN sum := 0;
         FOR p := 1 STEP 1 UNTIL n DO sum := sum + x
   END;
INTEGER PROCEDURE f(a, b)
   VALUE a, b; INTEGER a, b;
   f := a+b;
INTEGER i, j;
sum(i, 10, sum(j, 10, f(i, j)))
```

Notice that the evaluation of the third parameter of the outer call of `sum` involve recursion.

1.25 Modes of calling free variables

Consider: `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
```

Section 1.27 explains how the `local` declaration in ML can be used to make `private_a` only visible in the body of `f` and not at `a := 10;`

1.26 Implementing ML free variables

In ML programs like

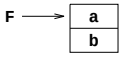
```
val a = 1;
fun g(b) = (let fun f(x) = x + a + b in f end);
val p = g 2;
val q = g 3;
```

we have already discussed that a 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 (L or R values 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:



During the evaluation of a function call, two pointers are needed: the `P` pointer, as before, to address the arguments and local variables, and a pointer `F` to point to the free variable list.

This mechanism requires more work at function definition time but less work at the call since there are no display registers to maintain. 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 free variable list can then represent the whole closure as a single word.)

1.27 Algol own variables

The purpose of Algol `OWN` variables is to allow the programmer to declare variables which may only be used in a region of program such as a block or procedure but which hold their values unchanged from one activation of the block or procedure until the next. An `OWN` variable may be declared at the head of a block using the word `OWN`, e.g.

```
BEGIN OWN REAL x;
...
END
```

The scope of the defined variable is the body of the block; however, when control leaves the block the storage cell that was allocated for the `OWN` variable remains in existence and is used when this same block is re-entered. Thus, in the above example, the value of `x` had when the block was last left will be the initial value of `x` when the block is next entered.

There are three major drawbacks to Algol `OWN` variables.

1. `OWN` variables are not initialised and there is no way of initialising them without the use of another `OWN` (with the same problem) or a more globally defined local variable. Notice that

   ```
   BEGIN OWN REAL a;
   a := 5;
   ...
   END
   ```

   defeats the purpose of the OWN declaration.

2. A second problem is caused by array declarations. Consider:

   ```
   INTEGER n; n := 10;
    L: BEGIN OWN REAL ARRAY a[1:n];
       ...
       END;
       n := n+10;
       GOTO L
   ```

   The first time a is declared the array bounds will be 1 and 10; however, the second time the block is entered the bounds are different. Presumably `a[1]` should hold its previous value, but what about `a[11]`? The vector a is called a *dynamic own array* and it is difficult to find a sensible meaning for it that can be implemented with reasonable efficiency. Most implementations of Algol disallowed dynamic own arrays.

3. A third problem with Algol OWN variables is one of interpretation. Consider:

   ```
   PROCEDURE r(x) REAL x;
        BEGIN OWN REAL a;
   ```

```
...
r(3);
...
END
```

The meaning of the call `r(3)` is defined to be the same as an equivalent program in which `r(3)` has been replaced by the text of the body of `r` (with appropriate substitutions and name replacements). The program thus produced will have two occurrences of the declaration: `OWN REAL a`. Do these both refer to the same `OWN` variable or not? If they share the same cell we have what is called the *static* interpretation of owns; if they are distinct variables we have the *dynamic* interpretation. The static interpretation is almost unanimously favoured by implementors.

Part of the trouble with Algol `OWN` variables is caused by associating `OWN` declarations with blocks rather than with procedures. CPL had a more satisfactory mechanism which now appears in ML; in ML it is possible to declare a variable whose scope is another declaration. The syntactic form is: '`local` D_1 `in` D_2' where D_1 and D_2 are declarations. For example,

```
let local val a = 1
    in fun f(x) = a+x
in BODY
end
```

The scope of `a` is just the right hand side of the definition of `f` and does not include the body of the block. In lambda notation this program is as follows:

$$(\lambda \texttt{f. BODY})\ [\ \{\lambda \texttt{a. } (\lambda \texttt{x. a+x})\}(1)\ ]$$

Thus `a` is declared and initialised just before `f` is defined, and it can be used only in the body of `f`, and `f` can be used in the body of the block. This mechanism is well defined and provides an `OWN` facility without any serious drawbacks.

At first sight this mechanism looks like one of the examples that cannot be implemented using a stack (since it involves a function producing a function as a result). However, if we implement the above example as though it were:

```
{ let a = 1
  in let f(x) = a+x
  in BODY
}
```

but do not tie up occurrences of `a` in BODY with the declaration `let a = 1` then there is no problem in producing a stack implementation. Note for comparison that the lambda calculus version of this latter form is:

$$\{\lambda \texttt{a. } [(\lambda \texttt{f. BODY})(\lambda \texttt{x. a+x})]\}(1)$$

which to some extent justifies the claim that the lambda-calculus can explain binding constructs in programming languages.

1.28 Labels and jumps

In Algol, if the destination label of a jump is in the same block as the jump, then the jump only involves a simple transfer of control. However, if the destination is in an outer block then the

jump also involves the removal of some declared variables from the stack, and if the destination is global to the current procedure then the jump will cause an exit from one or more procedures. Consider:

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

The call `r(M)` is equivalent to obeying the body of `r` with appropriate parameter substitutions. Thus the call is equivalent to:

```
{ ...
  ... goto M
  ...
}
```

and so the jump clearly should cause control to be resumed at the point labelled `M`.

In terms of the stack implementation it is necessary to reset the `P` pointer and the display (if one is used) to the values they had at the moment when execution entered the scope of `M`. Notice that, at the time when the jump is about to be made, the `P` pointer and display may have different settings. 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 P pointer. The action to take at the jump is then:

1. reset the P pointer,
2. establish the display, and
3. transfer control.

It may be more convenient to establish the display after the transfer of control since it is at the label (and not the jump) that the size of the display is known.

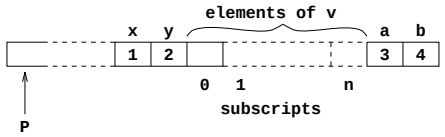
We should notice that the value of a label (like the value of a function) contains an implicit stack pointer and so some restrictions must be imposed to avoid nonsensical situations. Typically labels (as in Algol) may not be assigned or returned as results of functions. This will ensure that all jumps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

1.29 Arrays

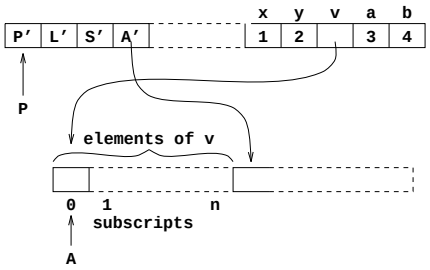
When an array is declared space must be allocated for its elements. In most languages the lifetime of an array is the same as that of a simple variable declared at the same point, and so it would be natural to allocate space for the array on the runtime stack. This is indeed what many implementations 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:



(This assumes, as in C, that `v` itself is a constant and does not require a storage cell.) In this example, `n` may be large and so the variables `a` and `b` may be a great distance from `P`. On some machines access to such variables is less efficient. Moreover, if `n` is not a compile-time constant,[4] the position of `a` and `b` relative to `P` will not be known until runtime, again causing inefficiency. Many implementations allocate such array elements on a separate stack, often 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:



When execution leaves the current procedure `P`, `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.

1.30 Object-oriented language storage layout: optional for 1998

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 `int`s are 4 bytes wide). Now after

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

objects of type `B` have 3 integer fields `a1` and `a2` (by inheritence) 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.

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

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

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.

The above details only dealt with ordinary members and inheritance. Suppose we now add member functions (methods), 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 and so 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 refered to Ellis and Stroustrup "The annotated C++ reference manual".

1.31 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 1.14 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 transformations can be inserted automatically. For instance, `x := i` is converted to `x := itof(i)` where `itof` is the transformation 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 transfers 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 being 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).

Chapter 2

Compiling Techniques

A *compiler* is a program to translate the source form of a program into its equivalent machine code or relocatable binary form. The number of compilers that exist is very large and considerable human effort has been expended in constructing them. As a result most parts of the compilations process have become well understood and the job of writing a compiler is no longer the difficult task it once was. A compiler tends to be a large program (typically 10,000 to 50,000 machine instructions and sometimes as high as 300,000 to a million instructions) and it is wise to structure it in order to make its individual components small enough to think about and handle conveniently.

If a language is sufficiently simple it and is designed suitably, it can be compiled by a *one pass compiler*, that is, it can be compiled a small piece (often just a statement) at a time. During the compilation process the data types and allocated locations of variables are remembered together with any other information that may be needed later on during the compilation. The space required for this is substantially less than the space needed to hold the entire program and there is usually no limit on the size of the program than may be compiled in this way. In general such a compiler is simple in design and fast in execution.

Most languages have features that make compilation in a single pass either difficult or impossible. For example, in Algol 60 the declaration of names may occur many line or even pages after they are first used, as in:

```
BEGIN REAL m;
      PROC g;
      BEGIN REAL x;
            f(m);
            ...
         m: ...
            ...
      END
      ...
END
```

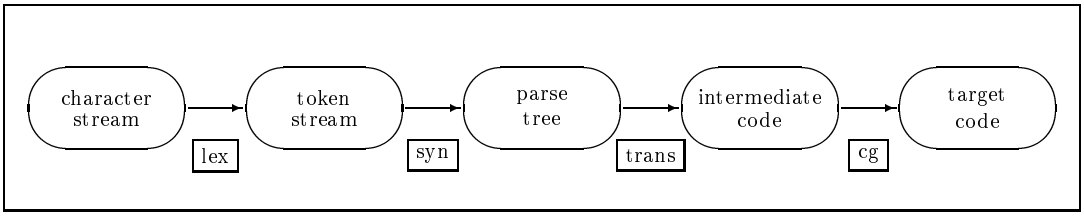
In this example it would be difficult to compile the call `f(m)` using a single pass compiler since although `m` seems to be declared as a REAL at the time of the call this is superseded by the label declaration some lines later. Similarly in ML, consider programs like:

```
val g = 3;
fun f(x) = ... g...
and g(x) = ...;
```

For most current programming languages, it is normal to compile in a number of stages (or passes) with the output of one pass being the input of the next. A compiler designed in this way is called a *multi-pass compiler*.

2.1 The structure of a typical multi-pass compiler

We will take as an example a compiler with four passes.



2.1.1 The lexical analyser

This reads the characters of the source program and recognises the basic syntactic components that they represent. It will recognise identifiers, reserved word, numbers, string constants and all other basic symbols (or tokens) and throw away all other ignorable text such as spaces, newlines and comments. For example, the result of lexical analysis of the following program:

```
{ let x = 1
  x := x + y
}
```

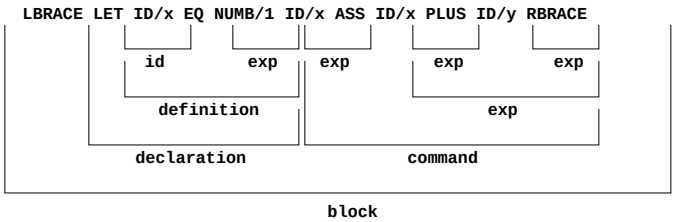
might be:

```
LBRACE LET ID/x EQ NUM/1 ID/x ASS ID/x PLUS ID/y RBRACE
```

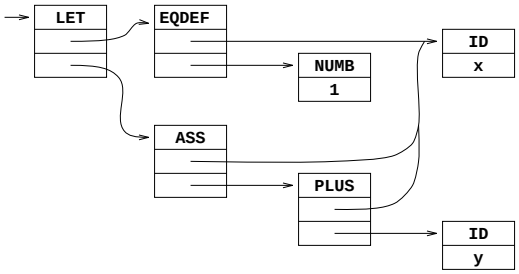
Lexical tokens are often represented in a compiler by small integers and for composite tokens such as identifiers, numbers, etc. additional information is passed by means of pointers into appropriate tables; for example calling a routine `lex()` might return the next token while setting a global variable `lex_aux_string` to the string form of an identifier when `ID` is returned, similarly `lex_aux_int` might be set to the binary representation of an integer when `NUM` is returned.

2.1.2 The syntax analyser

This will recognise the syntactic structure of the sequence of tokens delivered by the lexical analyser. The result of syntax analysis is often a tree representing the syntactic structure of the program. This tree is sometime called an *abstract syntax tree*. The syntax analyser would recognise that the above example parses as follows:



and it might be represented within the compiler by the following tree structure:



where the tree operators (e.g. `LET` and `EQDEF`) are represented as small integers.

In order that the tree produced is not unnecessarily large it is usually constructed in a condensed form as above with only essential syntactic features included. It is, for instance, unnecessary to represent the expression `x` as a `<sum>` which is a `<factor>` which is a `<primary>` which is an `<identifier>`. This would take more tree space space and would also make later processing less convenient. The phrase 'abstract syntax tree' refers to the fact the only semantically important items are incorporated into the tree; thus `a+b` and `((a)+(((b)))` might have the same representation, as might `while (e) C` and `for(;e;) C`.

2.1.3 The translation phase

This pass flattens the tree into a linear sequence of intermediate object code. At the same time it deals with

1. the scopes of identifiers,
2. declaration and allocation of storage,
3. selection of overloaded operators and the insertion of automatic type transfers.

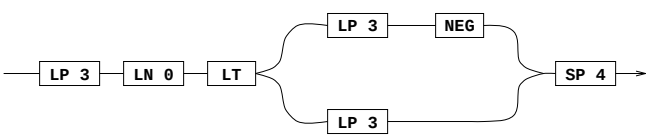
The intermediate object code for the command:

```
y := x<0 ? -x : x
```

might be as follows:

```
LP 3        load x
LN 0        load 0
LT          less than
JF L36      jump if false to L36
LP 3        load x
NEG         negate it
JUMP L37    jump to L37
LAB L36
LP 3        load x
LAB L37
SP 4        store y
```

Alternatively, the intermediate object code could be represented within the compiler as a directed graph as follows:



2.1.4 The code generator

This pass converts the intermediate object code into machine instructions and outputs them in either assembly language or relocatable binary form. The code generator is mainly concerned with local optimisation, the allocation of registers and the selection of machine instructions.

The four passes just described for a clear-cut logical divisions of the compiler, but are not necessarily applied in sequence. It is, for instance, common for the lexical analyser to be a subroutine of the syntax analyser and for it to be called whenever the syntax analyser requires another lexical token. It is also quite common for the translation phase and the code generator to be merged into one pass. Some compilers have additional passes, particularly if a high degree of optimisation is required.

2.1.5 Advantages of the multi-pass approach

1. It breaks a large and complicated task into smaller, more manageable pieces.
2. It is possible to reduce the store requirements of the compiler by overlaying the code for a later pass in the space occupied by the code of an earlier pass.
3. Modifications to the compiler (e.g. the addition of a synonym for a reserved word, or a minor improvement in compiler code) often require changes to one pass only and are thus simple to make.
4. A multi-pass compiler tends to be easier to describe and understand.
5. More of the design of the compiler is machine independent. It is sometimes possible to arrange that all machine dependent parts are in the code generator.
6. The job of writing the compiler can be shared between a number of programmers each working on separate passes. The interface between the passes is easy to specify precisely.

2.2 Lexical analysis

This is a critical part of a simple compiler since it can account for more than 50% of the compile time. This is because:

1. character handling tends to be expensive,
2. there are a large number of characters in a program compared with the number of lexical tokens, and
3. the lexical analyser usually constructs name tables and performs the binary conversion of constants.

For production compilers it was traditional to allocate one's most skilful programmers to the lexical analyser.

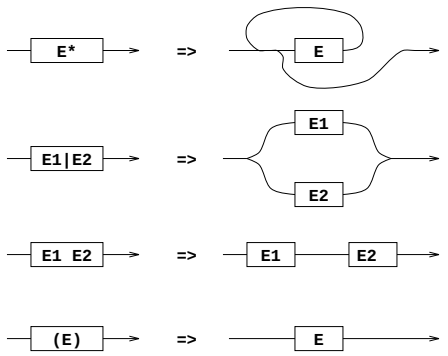
2.2.1 Regular expressions

The recognition of lexical tokens is straightforward and does not require a sophisticated analyser. This results from the simple syntax of lexical tokens. It is usually the case that all the lexical tokens of a language can be described by *regular expressions*, which then implies that the recognition can be performed by a *finite state algorithm*.

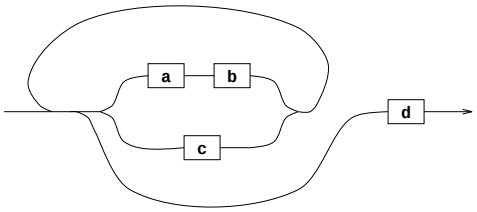
A *regular expression* is composed of characters, operators for concatenation (space), alternation (|) and repetition *, and parentheses are used for grouping. For example, `(a b | c)* d` is a regular expression. It can be regarded as a specification of a potentially infinite set of strings, in this case:

```
d
abd      cd
ababd    abcd   cabd   ccd
etc.
```

This is best derived by constructing the corresponding *transition diagram* by repeated application of the following rules.



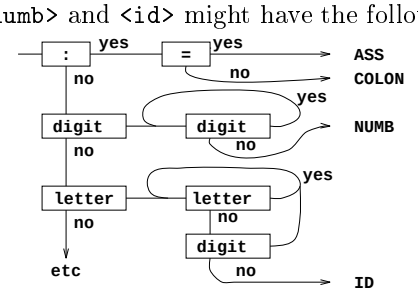
The transition diagram for the expression "`(a b | c)* d`" is:



This can be regarded as a generator of strings by applying the following algorithm:

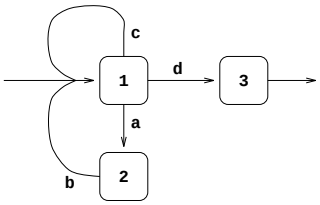
1. Follow any path from the starting point to any accessible box.
2. Output the character in the box.
3. Follow any path from that box to another box (possibly the same) and continue from step (2). The process stops when the exit point is reached.

We can also use the transition diagram as the basis of a recogniser algorithm. For example, an analyser to recognise `:=`, `:`, `<numb>` and `<id>` might have the following transition diagram:



Optimisation is possible (and needed) in the organisation of the tests. This method is only satisfactory if one can arrange that only one point in the diagram is active at any one time.

It is sometimes convenient to draw the transition diagram as a directed graph with labelled edges. For example, the graph for the expression "`(a b | c)* d`" can be represented as follows:

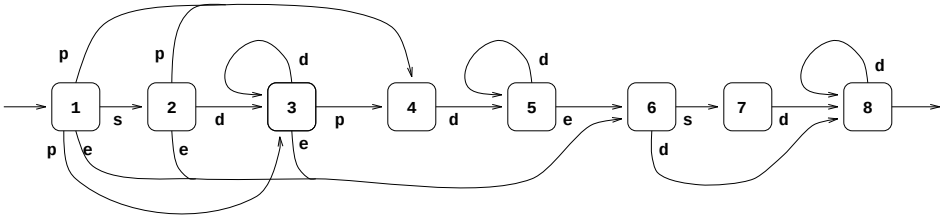


With state 3 designated an accepting state, this graph is a finite state acceptor for the given regular expression. The acceptor is easily implemented using a *transition matrix* to represent the graph.

We will demonstrate the method by considering the following syntax of floating point numbers (the '`->`' notation is introduced below in section 2.3):

```
      N  ->  U   |  s U             Number
      U  ->  D   |  E   |  D E      Unsigned number
      D  ->  J   |  F   |  J F      Unsigned decimal number
      E  ->  e I                    Exponent part
      F  ->  p J                    Decimal fraction
      I  ->  J   |  s J             Integer
      J  ->  d   |  d J             Unsigned integer

where  s  is a sign                 + or -
       e  is the exponent symbol    E
       p  is the decimal point      .
       d  is a digit                0-9
```

The corresponding graph is:



The corresponding matrix is as follows:

	s	d	p	e	other
S1	S2	S3	S4	S6	.
S2	.	S3	S4	S6	.
S3	.	S3	S4	S6	acc
S4	.	S5	.	.	.
S5	.	S5	.	S6	acc
S6	S7	S8	.	.	.
S7	.	S8	.	.	.
S8	.	S8	.	.	acc

In a program that uses this technique each matrix entry would specify the address of some code to deal with the transition[1] and note the next matrix row to be used. The entry `acc` would point to

[1] E.g. multiply the current total by 10 and add on the current digit

the code that processes a complete floating point number. Blank entries correspond to syntactic error conditions.

In general, this technique is fast and efficient, but if used on a large scale it requires skill and cunning to reduce the size of the matrix and to reduce the number of separate transition routines.

2.3 Phrase structured grammars

A *grammar* consists of an *alphabet* of symbols (think of these as characters to lexing or tokens resulting from lexing) and set of rules for generating a *language* (set of strings) of such symbols. For example, if the alphabet were the set of all letters `{a ... z}` and the rule were "generate all strings of length three" we would have a language whose strings are:

```
aaa, aab, ... zzy, zzz
```

A more useful form of grammar is the *phrase structured grammar* where the generation rule is given as a set of *productions*. It is first necessary to break the alphabet into two sets of symbols: *terminal symbols* like `a`, `b`, `c` above which may occur in the input text and *non-terminals* like `Term` or `Declaration` which do not occur in input text but summarise the structure of a sequence of symbols. The most general form of a production is:

```
A B ... C  ->  P Q ... R
```

where `A...Z` are symbols and `A B ... C` contains at least one non-terminal. This rule specifies that if `A B ... C` occurs in a string belonging to the grammar then the string formed by replacing `A B ... C` by `P Q ... R` also belongs to the grammar (note that the symbolx '`::=`' is sometimes used as an alternative to '`->`'). There must be a unique non-terminal `S`, say, called the *sentence symbol* that occurs by itself on the left hand side of just one production. Any string that can be formed by the application of productions is called a *sentential form*. A sentential form containing no non-terminals is called a *sentence*. The problem of syntax analysis is to discover which series of applications of productions that will convert the sentence symbol into the given sentence.

It is useful to impose certain restrictions on `A B ... C` and `P Q ... R` and this has been done by Chomsky to form four different types of grammar. The most important of these in the Chomsky Type 2 grammar.

2.4 Type 2 grammar

In the Chomsky type 2 grammar the left hand side of every production is restricted to just a single non-terminal symbol. Such symbols are often called *syntactic categories*. Type 2 grammars are known as *context free grammars* and have been used frequently in the specification of the syntax of programming languages, most notably Algol 60 where it was first used. The notation is sometime called *Backus Naur Form* or BNF after two of the designers of Algol 60. A simple example of a type 2 grammar is as follows:

```
S  ->  A B
A  ->  a
A  ->  A B b
B  ->  b c
B  ->  B a
```

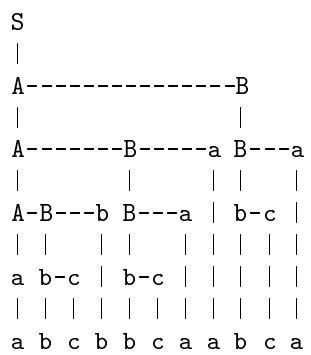
A slightly more convenient way of writing the above grammar is:

```
S  ->  A B
A  ->  a    |  A B b
B  ->  b c  |  B a
```

The alphabet for this grammar is `{S, A, B, a, b, c, d}`. The non-terminals are `S, A, B` being the symbols occurring on the left-hand-side of productions, with `S` being identified as the start symbol. The terminal symbols are `a, b, c, d`, these being the characters that only appear on the right hand side. Sentences that this grammar generates include, for instance:

```
abc
abcbbc
abcbca
abcbbcaabca
```

Where the last sentence, for instance, is generated from the sentence symbol by means of the following productions:



A grammar is ambiguous if there are two or more ways of generating the same sentence. Convince yourself that the follow three grammars are ambiguous:

```
a)   S  ->  A B
     A  ->  a  |  a c
     B  ->  b  |  c b

b)   S  ->  a T b  |  T T
     T  ->  a b  |  b a

c)   C  ->  if E then C else C  |  if E then C
```

Clearly every type 2 grammar is either ambiguous or it is not. However, it turns out that it is not possible to write a program which, when given an arbitrary type 2 grammar, will terminate with a result stating whether the grammar is ambiguous or not. It is surprisingly difficult for humans to tell whether a grammar in ambiguous. One example of this is that the productions in (c) above appeared in the original Algol 60 published specification. As an exercise, determine whether the example grammar given above is ambiguous.

For completeness, the other grammars in the Chomsky classification are as follows.

2.5 Type 0 grammars

Here there are no restrictions on the sequences on either side of productions. Consider the following example:

```
S    ->  a S B C  |  a B C
C B  ->  B C
a B  ->  a b
b B  ->  b b
b C  ->  b c
c C  ->  c c
```

This generates all strings of the form $a^n b^n c^n$ for all $n \geq 1$.

To derive `aaaaabbbbbccccc`, first apply `S -> aSBC` four times giving:

```
aaaaSBCBCBCBC
```

Then apply `S -> aBC` giving:

```
aaaaaBCBCBCBCBC
```

Then apply `CB -> BC` many times until all the `C`s are at the right hand end.

```
aaaaaBBBBBCCCCC
```

Finally, use the last four productions to convert all the `B`s and `C`s to lower case giving the required result. The resulting parse tree is as follows:

```
S
a-S----------------------B-C
| a-S-----------------B-C | |
| | a-S-----------B-C | | | |
| | | a-S-----B-C | | | | | |
| | | | a-B-C | | | | | | | |
| | | | a-b B-C B-C B-C B-C |
| | | | | b-b B-C B-C B-C | |
| | | | | | b-b B-C B-C | | |
| | | | | | | b-b B-C | | | |
| | | | | | | | b-b | | | | |
| | | | | | | | | b-c | | | |
| | | | | | | | | | c-c | | |
| | | | | | | | | | | c-c | |
| | | | | | | | | | | | c-c |
| | | | | | | | | | | | | c-c
| | | | | | | | | | | | | | |
a a a a a b b b b b c c c c c
```

As a final remark on type 0 grammars, it should be clear that one can write a grammar which essentially specifies the behaviour of a Turing machine, and syntax analysis in this case is equivalent to deciding whether a given string is the answer to some program. This is undecidable and syntax analysis of type 0 grammars is thus, in general, undecidable.

2.6 Type 1 grammars

A production in a type 1 grammar takes the following form:

```
A ... B P C ... D  -> A ... B U ... V C ... D
```

where P is a single non-terminal symbol, and the sequences A ... B, C ... D and U ... V are sequences of terminal and non-terminal symbols. The sequence U ... V may not be empty. These grammars are called *context sensitive* since P can only be replaced by U ... V if it occurs in a suitable context.

2.7 Type 3 grammars

This is the most restrictive of the phrase structured grammars. In it all productions are limited to being in one of the following forms:

```
      A  ->  a
or    A  ->  a B
```

That is, the right hand side must consist of a single terminal symbol possibly followed by a single non-terminal. It is sometimes possible to convert a type 2 grammar into an equivalent type 3 grammar. Try this for the grammar for floating point constants given earlier.

Type 3 grammars can clearly be parsed using a finite state recogniser, and for this reason they are often called *regular grammars*. [To get precise correspondence to regular languages it is necessary also to allow the empty production S -> ϵ otherwise the empty regular language (accepted by an automaton with no accepting states!) cannot be represented as a type 3 grammar.]

2.8 Syntax analysis

The type 2 (or context free) grammar is the most useful for the description of programming languages since it is powerful enough to describe the constructions one typically needs and yet is sufficiently simple to be analysed by a small and generally efficient algorithm. Some compiler writing systems use BNF (often with slight extensions) as the notation in which the syntax of the language is defined. The parser is then automatically constructed from this description.

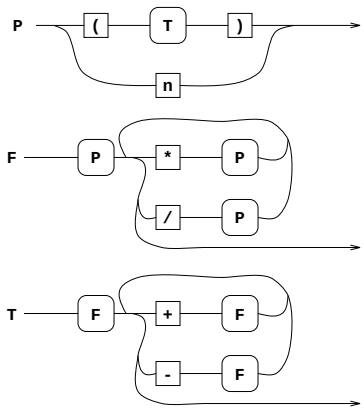
We will now look at three main parsing techniques, namely: *recursive descent*, *precedence* and *SLR(1)*.

2.9 Recursive descent

In this method the syntax is converted into transition diagrams for some or all of the syntactic categories of the grammar and these are then implemented by means of recursive functions. Consider, for example, the following syntax:

```
P  ->  ( T )  |  n
F  ->  F * P  |  F / P  |  P
T  ->  T + F  |  T - F  |  F
```

The corresponding transition diagrams are:



Notice that the original syntax has been modified to avoid left recursion[2] to avoid the possibility of a recursive loop in the parser. The recursive descent parsing functions are outlined below (implemented in C):

```
void RdP()
{ switch (token)
  { case '(': lex(); RdT();
              if (token != ')') error("expected ')');
              lex(); return;
    case 'n': lex(); return;
    default:  error("unexpected token");
  }
}

void RdF()
{ RdP();
  for (;;) switch (token)
  { case '*': lex(); RdP(); continue;
    case '/': lex(); RdP(); continue;
    default:  break;
  }
}

void RdT()
{ RdF();
  for (;;) switch (token)
  { case '+': lex(); RdF(); continue;
    case '-': lex(); RdF(); continue;
    default:  break;
  }
}
```

[2] By replacing the production

```
F -> F * P | F / P | P
```

with

```
F -> P * F | P / F | P
```

which has no effect on the strings accepted, although it does affect their parse tree—see later.

2.10 Data structures for parse trees

It is usually best to use a data structure for a parse tree which corresponds closely to the *abstract syntax* for the language in question rather than the *concrete syntax*. The abstract syntax for the above language is

```
E  ->  E + E  |  E - E  |  E * E  |  E / E  |  ( E ) | n
```

This is clearly ambiguous seen as a grammar on strings, but it specifies parse trees precisely and corresponds directly to ML's

```
datatype E = Plus of E * E  |  Minus of E * E  |
             Mult of E * E  |  Div of E * E  |
             Paren of E  |  Num of int;
```

Indeed one can go further and ignore the `( E )` construct in the common case parentheses often have no semantic import beyond specifying grouping. In C the construct tends to look like:

```
struct E {
  enum { T_Plus, T_Minus, T_Mult, T_Div, T_Paren, T_Numb } flavour;
  union { struct { struct E *left, *right; } diad;
            // selected by T_Plus, T_Minus, T_Mult, T_Div.
          struct { struct E *child; } monad;
            // selected by T_Paren.
          int num;
            // selected by T_Numb.
        } u;
};
```

It is not generally helpful to reliability and maintainability to make a single datatype which can represent all sub-structures of a parse tree. For parsing C, for example, one might well expect to have separate abstract parse trees for `Expr`, `Cmd` and `Decl`.

It is easy to augment a recursive descent parser so that it builds a parse tree while doing syntax analysis. The ML datatype definition defines constructor functions, e.g. `Mult` which maps two expression trees into one tree which represents multiplying their operands. In C one needs to work a little by defining such functions by hand:

```
struct E *mkE_Mult(E *a, E *b)
{   struct E *result = malloc(sizeof (struct E));
    result->flavour = T_Mult;
    result->u.diad.left = a;
    result->u.diad.right = b;
    return result;
}
```

A recursive descent parser which builds a parse tree for the parsed expression is given in Figure 2.1.

When there are many such operators like `+`, `-`, `*`, `/` with similar syntax it can often simplify the code to associate a binding power with each operator and to define a single routine `RdE(int n)` which will read an expression which binds at least as tightly as `n`. In this case `RdT()` might correspond to `RdE(0)`, `RdT()` to `RdE(1)` and `RdP()` to `RdE(2)`.

This idea is can be pushed further to produce a table-driven parser to which topic we now turn.

```
    struct E *RdP()
    {   struct E *a;
        switch (token)
        {   case '(': lex(); a = RdT();
                      if (token != ')') error("expected ')'");
                      lex(); return;
            case 'n': a = mkE_Numb(lex_aux_int); lex(); return a;
/* do names by
**          case 'i': a = mkE_Name(lex_aux_string): lex(); return a;
*/
            default:  error("unexpected token");
        }
    }

    /* We should have also had a right-associative '**' operator here! */

    struct E *RdF()
    {   struct E *a = RdP();
        for (;;) switch (token)
        {   case '*': lex(); a = mkE_Mult(a, RdP()); continue;
            case '/': lex(); a = mkE_Div(a, RdP()); continue;
            default:  break;
        }
        return a;
    }

    struct E *RdT()
    {   struct E *a = RdF();
        for (;;) switch (token)
        {   case '+': lex(); a = mkE_Plus(a, RdF()); continue;
            case '-': lex(); a = mkE_Minus(a, RdF()); continue;
            default:  break;
        }
        return a;
    }
```

Figure 2.1: Recursive descent parser yielding a parse tree

2.11 Simple precedence

For simple arithmetic grammars a parser based on the precedence of the operators is possible. Consider the token stream:

```
[ x * y + a / t - c ** d ]
```

Let us define two relations `LT` and `GT` which hold as follows:

	+	-	*	/	**	]
[	LT	LT	LT	LT	LT	
+	GT	GT	LT	LT	LT	GT
-	GT	GT	LT	LT	LT	GT
	GT	GT	GT	GT	LT	GT
/	GT	GT	GT	GT	LT	GT
*	GT	GT	GT	GT	LT	GT

then we can parse the above expression by means of the following steps:

```
We start with:     [ x * y + a / t - c ** d ]
[ LT *  GT +    => [ (x*y) + a / t - c ** d ]
+ LT /  GT -    => [ (x*y) + (a/t) - c ** d ]
[ LT +  GT -    => [ ((x*y)+(a/t)) - c ** d ]
- LT ** GT ]    => [ ((x*y)+(a/t)) - (c**d) ]
[ LT -  GT ]    => [ (((x*y)+(a/t))-(c**d)) ]
```

It is worth noting that this method allows both the precedence and associativity of operators to be specified. For example `a-b-c` parses as `(a-b)-c`, but `a**b**c` as `a**(b**c)`.

2.12 General precedence

For some grammars it is possible to define relations `EQ`, `GT` and `LT` between alphabet characters (terminal or non-terminal) of the grammar in such a way that, if

```
... U A B C D V ...
```

is a sentential form, and

```
U LT A EQ B EQ C EQ D GT V
```

then there must be a production `X -> A B C D` and so

```
... U X V ...
```

is a simpler sentential form. A matrix defining these relations then forms the basis of a very simple parser.

The definitions of `EQ`, `LT` and `GT` are as follows:

```
1)   A EQ B  <=>  P -> ... A B ...   is a production
```

```
2)   A LT B  <=>  P -> ... A U ...   is a production
                  and  U => B ...    (using one or more
                                      productions)

3a)  A GT B  <=>  P -> ... U B ...   is a production
                  and  U => ... A

3b)  A GT B  <=>  P -> ... U V ...   is a production
                  and  U => ... A
                  and  V => B ...
```

The grammar is a *precedence grammar* if for all pairs of alphabet characters at most one of the relations holds. The following grammar in not a precedence grammar.

```
S  ->  < E >
E  ->  T  |  E + T
T  ->  P  |  T * P
P  ->  ( E )  |  I
```

since, for instance `< EQ E` and `< LT E`. It can, however, be modified into the following equivalent grammar which is a precedence grammar.

```
S  ->  < E' >
E' ->  E
E  ->  T'  |  E + T'
T' ->  T
T  ->  P  |  T * P
P  ->  ( E' )  |  I
```

2.13 Construction of the precedence matrix

Before we construct the matrix it is convenient to form, for each non-terminal `U` in the grammar, two sets `Left(U)` and `Right(U)` of symbols that can start and end strings derived from `U`. If `U => B ...` then `B` is in `Left(U)`. Similarly, if `U => ... B` then `B` is in `Right(U)`.

`Left(U)` can be derived for all non-terminals in the grammar by the following algorithm:

```
1)  Initialise all sets Left(U) to empty.

2)  For each production  U -> A ...
    enter A into Left(U)

3)  For each production  U -> V ...
    enter all the elements of Left(V) into Left(U)

4)  Repeat (3) until no further change.
```

`Right(U)` can be derived similarly. For the example grammar the sets are as follows:

U	Left(U)	Right(U)
E'	E T' T P (I	E T' T P) I
E	E T' T P (I	T' T P) I
T'	T P (I	T P) I
T	T P (I	P) I
P	(I	) I

The following algorithm constructs the precedence matrix. For each pair `A B` occurring consecutively in a production (i.e. `P -> ... A B ...` is a production) do the following:

```
1)   Enter  A EQ B  into the matrix

2)   Enter  A LT X  into the matrix
                    for all X in Left(B)

3a)  Enter  X GT B  into the matrix
                    for all X in Right(A)

3b)  Enter  X GT Y  into the matrix
                    for all X in Right(A)
                    and all Y in Left(B)
```

For the example grammar, there are 8 pairs to consider:

< E'	E' >	E' +	+ T'	T *	* P	(E'	E')

and the resulting matrix is:

	E'	E	T'	T	P	(	I	*	+	)	>
E'	-	-	-	-	-	-	-	-	-	EQ	EQ
E	-	-	-	-	-	-	-	-	EQ	GT	GT
T'	-	-	-	-	-	-	-	-	GT	GT	GT
T	-	-	-	-	-	-	-	EQ	GT	GT	GT
P	-	-	-	-	-	-	-	GT	GT	GT	GT
)	-	-	-	-	-	-	-	GT	GT	GT	GT
I	-	-	-	-	-	-	-	GT	GT	GT	GT
	-	-	-	-	EQ	LT	LT	-	-	-	-
+	-	-	EQ	LT	LT	LT	LT	-	-	-	-
(	EQ	LT	LT	LT	LT	LT	LT	-	-	-	-
<	EQ	LT	LT	LT	LT	LT	LT	-	-	-	-

For this grammar it is possible to find two functions `f` and `g` with the property that:

```
if  A EQ B   then   f(A) = g(B)
if  A LT B   then   f(A) < g(B)
if  A GT B   then   f(A) > g(B)
```

Define such functions for this grammar and show that it is not possible in general. Why are such functions useful?

The following program will perform the parse using the precedence matrix. `P[k]` is the `k`th symbol of source text, and `S[i]` is the `i`th element of a stack.

```
S[0] = P[0]; i = 0, k = 1;
while (P[k] != '>')
{ S[++i] = P[k++];
  while (S[i] GT P[k])
  { int j = i;
    while (S[j-1] EQ S[j]) j--;
    S[j] = Leftpart(S[j] ,..., S[i]);
    i = j;
  }
}
```

The function `Leftpart` finds the subject of the production whose right part is given as its argument(s). Syntactic errors are detected by either encountering a blank entry in the matrix or by a failure in the function `Leftpart`.

NB. Note that the parsing program above is independent of the grammar to be used—all the grammatical details are stored in the tables. One can say that here the grammar is coded as data whereas in the recursive descent parser it was coded as program.

2.14 SLR parsing

Various parsing algorithms based on the so called LR(k) approach have become become popular. These are specifically LR(0), SLR(1), LALR(1) and LR(1). These four methods can parse a source text using a very simple program controlled by a table derived from the grammar. The methods only differ in the size and content of the controlling table.

To exemplify this style of syntax analysis, consider the following grammar (here `E`, `T`, `P` abbreviate 'expression', 'term' and 'primary'—an alternative notation would use names like `<expr>`, `<term>` and `<primary>` instead):

```
#0     S  ->  E  eof
#1     E  ->  E  +  T
#2     E  ->  T
#3     T  ->  P  **  T
#4     T  ->  P
#5     P  ->  i
#6     P  ->  (  E  )
```

The form of production `#0` is important. It defines the sentence symbol `S` and its RHS consists of a single non-terminal followed by the special terminal symbol `eof` which must not occur anywhere else in the grammar.

We first construct what is called the *characteristic finite state machine* or CFSM for the grammar. Each state in the CFSM corresponds to a different set of *items* where an *item* consists of a production together with a position marker (represented by `.`) marking some position on the right hand side. There are, for instance, four possible items involving production `#1`, as follows:

```
E  -> .E  +  T
E  ->  E .+  T
E  ->  E  + .T
E  ->  E  +  T .
```

If the marker in an item is at the beginning of the right hand side then the item is called an *initial* item. If it is at the right hand end the the item is called a *completed* item. In forming item

sets a *closure* operation must be performed to ensure that whenever the marker in an item of a set precedes a non-terminal, `E` say, then initial items must be included in the set for all productions with `E` on the left hand side.

The first item set is formed by taking the initial item for the production defining the sentence symbol (`S -> .E eof`) and then performing the closure operation, giving the item set:

```
1: {  S  -> .E  eof
      E  -> .E  +  T
      E  -> .T
      T  -> .P  **  T
      T  -> .P
      P  -> .i
      P  -> .(  E  )
   }
```

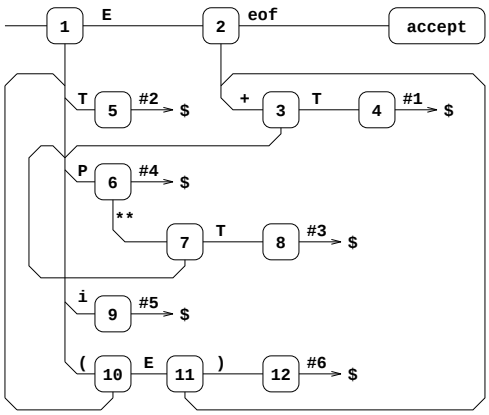
States have *successor* states formed by advancing the marker over the symbol it precedes. For state 1 there are successor states reached by advancing the marker over the symbols `E`, `T`, `P`, `i` or `(`. Consider, first, the `E` successor (state 2), it contains two items derived from state 1 and the closure operation adds no more (since neither marker precedes a non terminal). State 2 is thus:

```
2: {  S  ->  E .eof
      E  ->  E .+  T
   }
```

The other successor states are defined similarly, except that the successor of `eof` is always the special state `accept`. If a new item set is identical to an already existing set then the existing set is used. The successor of a completed item is a special state represented by `$` and the transition is labeled by the production number (`#i`) of the production involved. The process of forming the complete collection of item sets continues until all successors of all item sets have been formed. This necessarily terminates because there are only a finite number of different item sets.

For the example grammar the complete collection of item sets given in Figure 2.2. Note that for completed items the successor state is reached via the application of a production (whose number is given in the diagram).

The CFSM can be represented diagrammatically as follows:



Before we can construct an SLR(1) parser we must define and compute the sets `FOLLOW(A)` for all non-terminal symbols `A`. `FOLLOW(A)` is defined to be the set of all symbols (terminal and non-terminal) that can immediately follow the non-terminal symbol `A` in a sentential form. They can be formed iteratively by repeated application of the following rules.

```
1: {  S  -> .E  eof          \
      E  -> .E  +  T         /    E  => 2
      E  -> .T                    T  => 5
      T  -> .P  **  T        \
      T  -> .P               /    P  => 6
      P  -> .i                    i  => 9
      P  -> .(  E  )              (  => 10
   }
2: {  S  ->  E .eof                eof => accept
      E  ->  E .+  T               +  => 3
   }
3: {  E  ->  E  + .T               T  => 4
      T  -> .P  **  T        \
      T  -> .P               /    P  => 6
      P  -> .i                    i  => 9
      P  -> .(  E  )              (  => 10
   }
4: {  E  ->  E  +  T .             #1 => $
   }
5: {  E  ->  T .                   #2 => $
   }
6: {  T  =>  P .**  T              ** => 7
      T  ->  P .                   #4 => $
   }
7: {  T  ->  P  ** .T              T  => 8
      T  -> .P  **  T        \
      T  -> .P               /    P  => 6
      P  -> .i                    i  => 9
      P  -> .(  E  )              (  => 10
   }
8: {  P  ->  P  **  T .            #3 => $
   }
9: {  P  ->  i .                   #5 => $
   }
10:{  P  ->  ( .E  )          \
      E  -> .E  +  T         /    E  => 11
      E  -> .T                    T  => 5
      T  -> .P  **  T        \
      T  -> .P               /    P  => 6
      P  -> .i                    i  => 9
      P  -> .(  E  )              (  => 10
   }
11:{  P  ->  (  E .)               )  => 12
      E  ->  E .+  T               +  => 3
   }
12:{  P  ->  (  E  ) .             #6 => $
   }
```

Figure 2.2: CFSM item sets

1. If there is a production of the form `X -> ... Y Z ...` put `Z` and all symbols that can start `Z` into `FOLLOW(Y)`.
2. If there is a production of the form `X -> ... Y` put all symbols in `FOLLOW(X)` into `FOLLOW(Y)`.

We are assuming here that no production in the grammar has an empty right hand side. For our example grammar, the `FOLLOW` sets are as follows:

```
FOLLOW(E)  =  { eof  +  ) }
FOLLOW(T)  =  { eof  +  ) }
FOLLOW(P)  =  { eof  +  )  ** }
```

From the CFSM we can construct the two matrices `action` and `goto`:

1. If there is a transition from state `i` to state `j` under the terminal symbol `k`, then set `action[i,k]` to `Sj`.
2. If there is a transition under a non-terminal symbol `C`, say, from state `i` to state `j`, set `goto[i, C]` to `j`.
3. If state `i` contains a transition under `eof` set `action[i, eof]` to `acc`.
4. If there is a reduce transition `#p` from state `i`, set `action[i, k]` to `#p` for all terminals `k` belonging to `FOLLOW(A)` where `A` is the subject of production `#p`.

If any entry is multiply defined then the grammar is not SLR(1). Blank entries are represented by dash (-).

	action						goto		
state	eof	(	i	)	+	**	P	T	E
1	-	S10	S9	-	-	-	6	5	2
2	acc	-	-	-	S3	-	-	-	-
3	-	S10	S9	-	-	-	6	4	-
4	#1	-	-	#1	#1	-	-	-	-
5	#2	-	-	#2	#2	-	-	-	-
6	#4	-	-	#4	#4	S7	-	-	-
7	-	S10	S9	-	-	-	6	8	-
8	#3	-	-	#3	#3	-	-	-	-
9	#5	-	-	#5	#5	#5	-	-	-
10	-	S10	S9	-	-	-	6	5	-
11	-	-	-	S12	S3	-	-	-	-
12	#6	-	-	#6	#6	#6	-	-	-

The parsing algorithm used for all LR methods uses a stack that contains alternately state numbers and symbols from the grammar, and a list of input terminal symbols terminated by `eof`. A typical situation is represented below:

```
a A b B c C d D e E f  |  u v w x y z eof
```

Here `a ... f` are state numbers, `A ... E` are grammar symbols (either terminal or non-terminal) and `u ... z` are the terminal symbols of the text still to be parsed. If the original text was syntactically correct, then

```
A B C D E u v w x y z
```

will be a sentential form.

The parsing algorithm repeatedly applies the following rules until either a syntactic error is found or the parse is complete.

1. If `action[f, u] = Si`, then transform

   ```
   a A b B c C d D e E f  |  u v w x y z eof
   ```

   to

   ```
   a A b B c C d D e E f u i  |  v w x y z eof
   ```

2. If `action[f, u] = #p`, and production `#p` is of length 3, say, then it will be of the form `P -> C D E` where `C D E` exactly match the top three symbols on the stack, and `P` is some non-terminal, then assuming `goto[c, P] = g`

   ```
   a A b B c C d D e E f  |  u v w x y z eof
   ```

   will transform to

   ```
   a A b B c P g  |  u v w x y z eof
   ```

   Notice that the symbols in the stack corresponding to the right hand side of the production have been replaced by the subject of the production and a new state chosen using the `goto` table.

3. If `action[f, u] = acc` then the situation will be as follows:

   ```
   a Q b  |  eof
   ```

   and the parse will be complete.

4. If `action[f, u] = -` then the text being parsed is syntactically incorrect.

Note again that there is a single program for all grammars; the grammar is coded in the `action`and `goto` matrices.

As an example, the following steps are used in the parsing of `i + i`:

Stack	text	production used
1	i + i eof	
1 i 9	+ i eof	
1 P 6	+ i eof	P -> i
1 T 5	+ i eof	T -> P
1 E 2	+ i eof	E -> T
1 E 2 + 3	i eof	
1 E 2 + 3 i 9	eof	
1 E 2 + 3 P 6	eof	P -> i
1 E 2 + 3 T 4	eof	T -> P
1 E 2	eof	E -> E + T

```
%%
[ \t]  /* ignore blanks and tabs  */ ;

[0-9]+ { yylval = atoi(yytext); return NUMBER; }

"mod" return MOD;
"div" return DIV;
"sqr" return SQR;
\n|.  return yytext[0]; /* return everything else */
```

Figure 2.3: `calc.l`

2.15 Errors

A syntactic error is detected by encountering a blank entry in the `action` or `goto` tables. If this happens the parser can recover by systematically inserting, deleting or replacing symbols near the current point in the source text, and choosing the modification that yields the most satisfactory recovery. A suitable error message can then be generated.

2.16 Table compaction

In a typical language we can expect there to be over 200 symbols in the grammar and perhaps rather more states in the CFSM. The table `action` and `goto` are thus like to require over 40000 entries between them. There are good ways of compacting these by about a factor of ten.

2.17 Lex

Lex and Yacc are programs that run on Unix and provide a convenient system for constructing lexical and syntax analysers.

Lex takes as input a file (e.g. `calc.l`) specifying the syntax of the lexical tokens to be recognised and it outputs a C program (normally `lex.yy.c`) to perform the recognition. The syntax of each token is specified by means of a regular expression and the corresponding action when that token is found is supplied as a fragment of C program that is incorporated into the resulting lexical analyser. Consider the lex program `calc.l` in Figure 2.3. The regular expressions obey the usual unix conventions allowing, for instance, `[0-9]` to match any digit, the character `+` to denote repetition of one or more times, and dot (`.`) to match any character other than newline. Next to each regular expression is the fragment of C program for the specified token. This may use some predefined variables and constants such as `yylval, yytext` and `NUMBER`. `yytext` is a character vector that holds the characters of the current token (its length is held in `yyleng`). The fragment of code is placed in the body of an external function called `lex`, and thus a `return` statement will cause a return from this function with a specified value. Compound tokens such as `NUMBER` return auxiliary information in suitably declared variables. For example, the converted value of a `NUMBER` is passed in the variable `lexlval`. If a code fragment does not explicitly return from `lex` then after processing the current token the lexical analyser will start searching for the next token.

In more detail, a Lex program consists of three parts separated by %%s.

```
declarations
%%
translation rules
%%
auxiliary C code
```

The declarations allows a fragment of C program to be placed near the start of the resulting lexical analyser. This is a convenient place to declare constants and variables used be the lexical analyser. One may also make regular expression definitions in this section, for instance:

```
ws        [ \t\n]+
letter    [A-Za-z]
digit     [0-9]
id        {letter}({letter}|{digit})*
```

These named regular expressions may be used by enclosing them in braces (`{ or }`) in later definitions or in the translations rules.

The translation rules are as above and the auxiliary C code is just treated as a text to be copied into the resulting lexical analyser.

2.18 Yacc

Yacc (yet another compiler compiler) is like Lex in that it takes an input file (e.g. `calc.y`) specifying the syntax and translation rule of a language and it output a C program (usually `y.tab.c`) to perform the syntax analysis.

A Yacc program has three part separated by %%s.

```
declarations
%%
translation rules
%%
auxiliary C code
```

Within the declaration one can specify fragments of C code (enclosed within special brackets `%{` and `%}`) that will be incorporated near the beginning of the resulting syntax analyser. One may also declare token names and the precedence and associativity of operators in the declaration section by means of statements such as:

```
%token NUMBER
%left '*' DIV MOD
```

The translation rules consist of BNF-like productions that include fragments of C code for execution when the production is invoked during syntax analysis. This C code is enclosed in braces (`{ and }`) and may contain special symbols such as `$$`, `$1` and `$2` that provide a convenient means of accessing the result of translating the terms on the right hand side of the corresponding production.

The auxiliary C code section of a Yacc program is just treated as text to be included at the end of the resulting syntax analyser. It could for instance be used to define the main program.

An example of a Yacc program (that makes use of the result of Lex applied to `calc.l`) is `calc.y` listed in Figure 2.4.

```
%{
#include <stdio.h>
%}

%token NUMBER

%left '+' '-'
%left '*' DIV MOD
  /* gives higher precedence to '*', DIV and MOD  */
%left SQR

%%
comm: comm '\n'
    | /* empty  */
    | comm expr '\n'  { printf("%d\n", $2); }
    | comm error '\n' { yyerrok; printf("Try again\n"); }
    ;

expr: '(' expr ')'    { $$ = $2; }
    | expr '+' expr   { $$ = $1 + $3; }
    | expr '-' expr   { $$ = $1 - $3; }
    | expr '*' expr   { $$ = $1 * $3; }
    | expr DIV expr   { $$ = $1 / $3; }
    | expr MOD expr   { $$ = $1 % $3; }
    | SQR expr        { $$ = $2 * $2; }
    | NUMBER
    ;

%%

#include "lex.yy.c"

yyerror(s)
char *s;
{printf("%s\n", s);
}

main()
{ return yyparse();
}
```

Figure 2.4: `calc.y`

Yacc parses using the LALR(1) technique. It has the interesting and convenient feature that the grammar is allowed to be ambiguous resulting in numerous shift-reduce and reduce-reduce conflicts that are resolved by means of the precedence and associativity declarations provided by the user. This allows the grammar to be given using fewer syntactic categories with the result that it is in general more readable.

The above example uses Lex and Yacc to construct a simple interactive calculator; the translation of each expression construct is just the integer result of evaluating the expression.

Chapter 3

Translation

The translation phase of a compiler normally converts the abstract syntax tree representation of a program into intermediate object code which is usually either a linear sequence of statements or an internal representation of a flowchart. We will assume that the translation phase deals with (1) the scope and allocation of variables, (2) determining the type of all expressions, (3) the selection of overloaded operators, and (4) generating of the intermediate code.

In this section we will use a simple but real-feeling example language (it is actually based on BCPL) with the following abstract syntax tree structure:

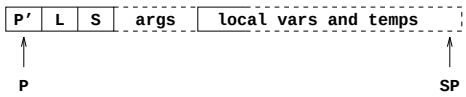
```
type N = string;   (* shorthand for 'name' *)
datatype E = Var of N | Num of int | Ap of N * (E list) |
             True | False |
             Neg of E  |  Pos of E  |  Not of E  |
             Subscript of E * E |
             Plus of E * E  |  Minus of E * E  |
             Mult of E * E  |  Div of E * E  |
             Eq of E * E  |  Ne of E * E  |
             Lt of E * E  |  Gt of E * E  |
             Le of E * E  |  Ge of E * E  |
             And of E * E  |  Or of E * E  |
             Cond of E * E * E  | Valof of C

     and C = Seq of C * C  |  Unless of E * C  |
             If1 of E * C  |  If2 of E * C * C  |
             While of E * C  |  Until of E * C  |
             Assign of N * E  | AssignSubscript of E * E * E  |
             For of N * E * E * C  | Result of E  |
             Call of N * E list  |  Let of D * C

     and D = And of D * D  |  Valdef of N * E  | Arraydef of N * int  |
             Fndef of N * (N list) * E  |  Rtdef of (N * N list) * C;
```

An example program fragment that can be represented by the above abstract syntax tree is given later.

We will assume that this language is essentially typeless and so all expression values are of the same size. It is to be implemented using a simple stack and the static chain method will be used for accessing free variables. We will also assume that functions may not be passed as

arguments or returned as results. Most of these restrictions could easily be removed by decorating the abstract syntax tree with more type information.

The intermediate object code for this language which we adopt is a linear sequence of simple statements that describe instructions acting on the runtime stack. This is based on OCODE from the BCPL compiler but note the Java Virtual Machine code is similar. A stack frame will has the following form:



Here `S` is the static chain pointer, `P'` is the dynamic stack pointer and `L` is the return address. Variables are accessed by the following intermediate code statements:

```
Statement      Meaning

Lv n 0         SP++; SP[0] := P[n];
Sv n 0         P[n] := SP[0]; SP--;
Lv n 1         SP++; SP[0] := P[2][n];
Sv n 1         P[2][n] := SP[0]; SP--;
Lv n 2         SP++; SP[0] := P[2][2][n];
Sv n 2         P[2][2][n] := SP[0]; SP--;
etc.
```

The address of a variable (or zeroth element of an array) can be computed using `Laddr`, for example

```
Laddr n 2      SP++; SP[0] := &P[2][2][n]
```

Some of the other intermediate code statements are as follows:

```
Ln k           SP++; SP[0] := k;
Plus           SP[-1] := SP[-1] + SP[0]; SP--;
Neg            SP[0] := - SP[0];
Eq             SP[-1] := SP[-1] = SP[0]; SP--;
Jt Ln          B := SP[0]; SP--; if  B goto Ln;
Jf Ln          B := SP[0]; SP--; if !B goto Ln;
Jmp Ln         goto Ln;
Lab Ln         Ln:
Setsp n        SP = P+n;
Call Ln d k    // Call the function or routine whose
               // entry point is labelled Ln and which
               // was declared at a textual depth d levels
               // less than the current depth,
               // incrementing the P pointer by k positions.
               NP := P+k
               NP[0] := P   // the old P
               NP[1] := Li  // the return address
               if d=0 do NP[2] := P        // current level
               if d=1 do NP[2] := P[2]     // one level out
               if d=2 do NP[2] := P[2]P[2] // two levels out
               etc.
               P := NP
               goto Ln            // Jump to the entry point
            Li:SP := P+k-1        // return here
```

```
Entry Ln    Ln:
Rtn            L := P[1]    // get return address
               P := P[0]    // get old P pointer
               goto L
SetRes         // RES for function results
               RES := SP[0]; SP--;
LdRes          SP++; SP[0] := RES;
```

The use of this intermediate code can be shown by considering the following C program fragment:

```
int i = 36;                          // i in P[3]

int g(a,b) { i = a*i+b; }            // Entry L1
int f(x,y) { return x==0 ? 0 :       // Entry L2
             x*y + f(x-1,y+1); }

int p(int n)                         // Entry L3 n in P[3]
{ if (n==0) return 1;
  else
  {   int v[11];                     // v in P[4]..P[14]
      int h(x) { return f(n,x); }    // Entry L4
      int j;                         // j in P[15]
      for (j = 0; j<11; j++)
      { g(n, j); v[j] = h(j) }
      return v[10];
  }
}
```

The code generated for the function g might be:

```
Entry L1  // entry to g(a,b)
Setsp 4   // Leave space for a in P[3] and b in P[4]
Lv 3 0    //     a
Lv 3 1    //          i
Mult      //       *
Lv 4 0    //               b
Add       //            +
Sv 3 1    // i :=
Rtn
```

and the code for the definition of h might be:

```
Entry L4    // entry to h(x)
Setsp 3     // Leave space for x in P[3]
Setsp 7     // Leave linkage space for call f(n, x)
Lv 3 1      // n  (one level out, pos 3)
Lv 3 0      // x  (current level, pos 3)
Call L2 1 4 // call f (entry L2, declared one level out,
            //         new P at pos P+4)
Rtn         // the result will be in RES
```

These statements will be generated within the translation phase by means of calls such as the following:

```
Gen3(Lv, 1, 3);
Gen3(Lv, 0, 3);
gen4(Call, 2, 1, 4);
gen1(Rtn);
```

The different intermediate code operators can be represented by distinct integers and so the intermediate code can be written to a file as just a sequence of integers.

3.1 Scope and allocation of variables

In a block structured language, variables can be declared and their scope is limited to the region of program for which the declaration is valid. When an identifier is encountered during translation it must be looked up in a stack of currently declared identifiers to discover its datatype and how it is to be accessed (i.e. its address). This stack of name cells is maintained by the translation phase. New cells are added when a declaration is encountered and they are removed when the translator moves out of the scope of the declaration. When an identifier is encountered during translation it is looked up by searching down the name cell stack. The first matching name cell will be the one required.

For our example language, name cells would contain: a pointer to the name node, the type (integer, function or routine), the textual level, and the allocated position within its stack frame if it were an integer or the entry label number if it were a function or routine.

3.2 Translation of expression

Some of the functions used during translation are as follows:

```
trexp(x)        translate an expression in Rmode
trexplist(x)    translate an expression list in Rmode
trlexp(x)       tranlate an expression in Lmode
trname(op,x)    translate a name, op is one of
                Lv, Sv, Laddr or Call
jumpcond(x,b,n) translate a conditional jump
trcom(x)        translate a command
declnames(x)    declare the names in a declaration,
                allocate stack space where necessary, and put
                suitable items in the declaration vector
trdecl(x)       perform a second pass over a
                declaration to compile code to
                initialise variables
```

During translation certain global variables hold information about the current state. For instance, `ssp` is the *simulated stack pointer* holding the distance between `P` and `SP` at the current point in the compiled code. Another variable is `depth` that hold the textual depth of the current piece of code being compiled. It is incremented when starting to translate the body of a function, and decremented when the translation of the body is complete. The argument to `trexp` is the tree for the expression being translated. An outline of its definition is as follows:[1]

[1] We have adopted ML to describe this code since we can exploit pattern matching to make the code more concise than C or Java would be. For ML experts there are still things left undone, like defining the `++` and `--` operators of type `int ref -> int`.

```
fun trexp(Num(k))      = (gen2(Ln, k); ++ssp)
  | trexp(Id(s))       = (trname(Lv,s); ++ssp)
  | trexp(Plus(x,y))   = (trexp(x); trexp(y); gen1(Plus); --ssp)
  | trexp(Minus(x,y))  = (trexp(x); trexp(y); gen1(Minus); --ssp)
  | trexp(Mult(x,y))   = (trexp(x); trexp(y); gen1(Mult); --ssp)
  | trexp(Div(x,y))    = (trexp(x); trexp(y); gen1(Div); --ssp)
  | trexp(Neg(x))      = (trexp(x); gen1(Neg))
  | trexp(Not(x))      = (trexp(x); gen1(Not))
  | trexp(Ap(f, el))   =
                let val s = ssp in
                  gen2(Setsp, s+3);  // leave space for linkage
                  trexplist(el);     // translate args
                  trname(Call, f);   // Compile Call Lf d
                  gen1(s+1);         // Compile           k
                  ssp := s;          // Restore saved ssp
                  gen1(Ldres);
                  ++ssp
                end
  | trexp(Cond(b,x,y)) =
                let val p = ++label;  // Allocate two labels
                    val q = ++label in
                  jumpcond(b,false,p);
                  trexp(x);           // code to put x on stack
                  gen2(Jmp,q);        // jump to common point
                  --ssp;              // at Lab stack is one less
                  gen2(Lab,p);
                  trexp(y);           // code to put y on stack
                  gen2(Lab,q)         // common point; result on stack
                end;
etc...

fun trexplist[] = ()
  | trexplist(e::es) = (trexp(e); trexplist(es));
```

3.3 Translation of boolean expressions

If a boolean expression occurs in a context where a conditional jump is to be made on the outcome of the evaluation then some optimisation is possible. For example, consider code like

```
IF x>0 AND A[i]=0 THEN ...
```

If `x>0` is false then there is no need to evaluate `A[i]=0` (assuming it is side-effect free). Some languages (like C) even provide a special operator '`&&`' which prescribes such *short-cut* evaluation. We will use the function `jumpcond` to compile such expressions. Its first argument is the tree structure of the expression, the second is a truth value stating whether a jump is to be made on true or on false, and the third argument is the number of the label to jump to. The definition of `jumpcond` is outlined below:

```
fun jumpcond(True,       true, n) =  gen2(Jmp, n)
  | jumpcond(True,          _, _) =  ()
  | jumpcond(False,     false, n) =  gen2(Jmp, n)
  | jumpcond(False,         _, _) =  ()
```

```
  | jumpcond(Not(x),        b, n) =  jumpcond(x, not b, n)
  | jumpcond(And(x, y),  true, n) =
                        let val m = ++label in
                          jumpcond(x, false, m);
                          jumpcond(y,  true, n);
                          gen2(Lab, m)
                        end
  | jumpcond(And(x, y), false, n) =  (jumpcond(x, false, n);
                                      jumpcond(y, false, n))
  | jumpcond( Or(x, y),  true, n) =  (jumpcond(x,  true, n);
                                      jumpcond(y,  true, n))
  | jumpcond( Or(x, y), false, n) =
                        let val m = ++label in
                          jumpcond(x,  true, m);
                          jumpcond(y, false, n);
                          gen2(Lab, m);
                        end
  | jumpcond(x, b, n) =  ( trexp(x);
                          gen2( (b ? Jt:Jf), n);
                          --ssp)
```

3.4 Translation to machine code from intermediate code

The part II course on 'Optimising Compilation' will cover this topic in an alternative manner, but let us for now merely observe that each intermediate intstruction listed above can be mapped into a small number of Pentium instructions. For example, if the current offset of `SP` above `P` is 5 words then

```
LAB L3
LP 3
LN 7
MINUS
SP 3
LP 3
LN 0
LT
JT L3
```

can be translated in a instruction-by-instruction manner (using a Pentium register here called `fp` as the stack pointer `P` since the Pentium stack pointer does not fit well into our upward growing stack model.

```
L3: movl 12(fp),%eax            LP 3
    movl %eax,20(fp)            (ssp = 5)
    movl 7,24(fp)               LN 7
    movl %eax,24(fp)            SUB
    subl %eax,20(fp)
    movl 20(fp),%eax            SP 3
    movl %eax,12(fp)
    ...
```

With a little more care in remembering values in registers between translations on individual intermediate instructions quite reasonable, though far from optimal, code can be produced.

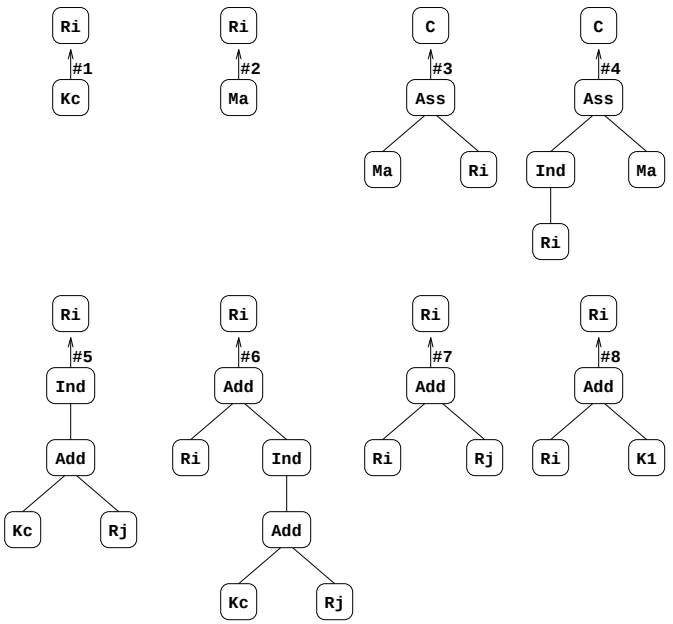


Figure 3.1: Tree version of rules

3.5 Translation using tree matching and rewriting

This section gives an alternative method of generating code for CISC-like target architectures directly from parse trees.

Ref: Code Generation Using tree matching and Dynamic programming by Aho, A.V., Ganapathi, M. and Tjiang, S.W.K. ACM Transations on Programming Languages and Systems, Vol 11, No 4, October 1989.

A slightly simplified version of the algorithm is presented here. The algorithm uses a collection of tree rewrite rules to define the resulting translation. Each rule has four components as follows:

```
replacement  <-  template        cost      code
```

where `replacement` is a single node, `template` is a tree, `cost` is the cost of using this rule, `code` is a fragment of compiled code. For example:

```
       Rule                            Cost      Code

#1  Ri  <-  Kc                           2      MOV #c,Ri
#2  Ri  <-  Ma                           2      MOV a,Ri
#3  C   <-  Ass(Ma, Ri)                  2      MOV Ri,a
#4  C   <-  Ass(Ind(Ri),Ma)              2      MOV a,*Ri
#5  Ri  <-  Ind(Add(Kc,Rj))              2      MOV c(Rj),Ri
#6  Ri  <-  Add(Ri, Ind(Add(Kc,Rj)))     2      ADD c(Rj),Ri
#7  Ri  <-  Add(Ri, Rj)                  1      ADD Rj,Ri
#8  Ri  <-  Add(Ri, K1)                  1      INC Ri
```

The tree pattern could be drawn as in Figure 3.1. The tree for the command: `v[i] := x` might be as in Figure 3.2. A third representation of the same tree is

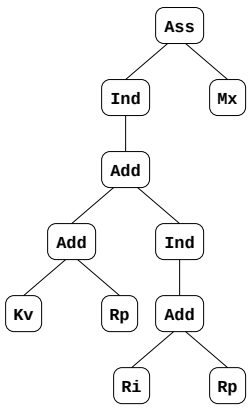


Figure 3.2: Tree for `v[i] := x`

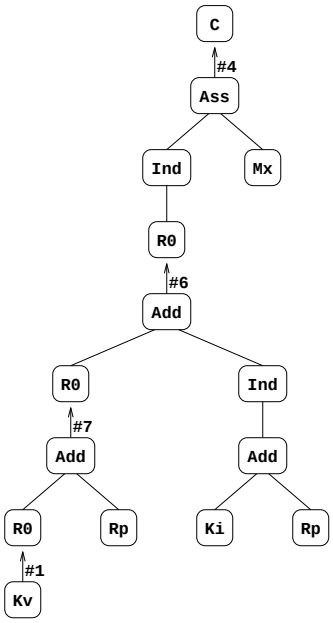


Figure 3.3: Minimum cost covering

```
Ass(Ind(Add(Add(Kv,Rp),Ind(Add(Ki,Rp))), Mx)
```

This tree can be 'covered' by the templates in several ways, but one that gives the least cost is given in Figure 3.3. The total cost in this case is 7. Using this covering, the code and resulting trees produced by a depth first left to right scan are as follows:

```
                Ass(Ind(Add(Add(Kv,Rp), Ind(Add(Ki,Rp))), Mx)
MOV #v,R0       Ass(Ind(Add(Add(R0,Rp), Ind(Add(Ki,Rp))), Mx)
ADD Rp,R0       Ass(Ind(Add(R0,         Ind(Add(Ki,Rp))), Mx)
ADD i(Rp),R0    Ass(Ind(R0),                              Mx)
MOV x,*R0       C
```

Other coverings are possible but these would give different costs and different code sequences. The algorithm given here is designed to find a least cost covering efficiently.

The template `#6 Add(R?, Ind(Add(K?,R?)))` contains paths from the root to its three leaf

nodes, namely:

```
Add.1--R?
Add.2--Ind.1--Add.1--K?
Add.2--Ind.1--Add.2--R?
```

The integers specify which edge is being taken from an operator node. The length of a path is the number of edges it contains. The path `Add.1--R?` is of length 1 and occurs in template `#6` (this is denoted by `#6/1`). We can combine all the paths from all the templates to form a tree as follows:

```
                                          Rule/Length

n00-Add.1-n01-R?--a00                     #8/1,#7/1,#6/1
  |    .2-n02-Ind.1-n03-Add.1-n04-K1-a01  #6/3,#5/2,#1/0
  |         |              |    *-K?-a02   #6/3,#5/2,#1/0
  |         |              .2-n05-R?-a03   #7/1,#6/3,#5/2
  |         *-K1--a04                     #8/1,#1/0
  |         *-R?--a05                     #7/1
  *-Ass.1-n06-Ind.1-n07-R?--a06           #4/2
  |    |    *-M?--a07                     #3/1,#2/0
  |    .2-n08-M?--a08                     #4/1,#2/0
  |         *-R?--a09                     #3/1
  *-Ind.1-n09-Add.1-n10-K1--a10           #5/2,#1/0
  |              |    *-K?--a11           #5/2,#1/0
  |              .2-n11-R?--a12           #7/1,#5/2
  *-K1--a13                               #1/0
  *-K?--a14                               #1/0
  *-M?--a15                               #2/0
```

The internal nodes of the tree are labelled `n00` to `n11`, and the leaf nodes are labelled `a00` to `a15`. Notice that the path `Add.1--R?` is of length 1 and occurs in three different templates, namely `#8`, `#7` and `#6` which accounts for why `#8/1,#7/1,#6/1` is attached to node `a00`. The node `a01` is at the end of the path `Add.2--Ind.1--Add.1--K1` which is a path of length three belonging to template `#6`. However, `a01` is also at the end of `Ind.1--Add.1--K1` which is a path of length 2 belonging to template `#5`, and it also ends a path of length zero belonging to template `#1`. This accounts for `#6/3,#5/2,#1/0` being attached to `a01`.

This tree forms the basis of a finite state acceptor with the leaf nodes being the accepting states. Extra transitions must be added, but the number of nodes remains unchanged. Consider the path string: `Ind.1--Add.2--Ind.1--Add.1--K1`. Starting from `n00`, `Ind.1` gets us to `n09`, then `add.2` gets us to `n11`, but from here the only transisition in the tree is on `R?` to `a12`. This tells us that there are no path strings starting with `Ind.1--Add.2--Ind.1` belonging to any of our set of templates. What we should do is delete the first item of our path string, giving us `Add.2--Ind.1--Add.1--K1` and see if this leads to an accepting state. Thus from state `n11`, a transition on `Ind.1` should lead to `n03`. One way of viewing this is if state `n11` does not encounter `R?` then it should behave like state `n02`. Looking carefully at the tree we see that:

```
State       Not followed by          Behaves like

n01         R?                       n00
n02         Ind.1,  K1  or R?        n00
n03         Add.1 or Add.2           n09
n04         K1 or K?                 n10
n05         R?                       n11
```

```
n06         Ind.1 or M?               n00
n07         R?                        n09
n08         M? or R?                  n00
n09         Add.1 or Add.2            n00
n10         K1 or K?                  n01
n11         R?                        n02
```

This easily leads to the resulting finite state acceptor described by the following table:

```
     | Add.1  Add.2  Ass.1 Ass.2  Ind.1    K1   K?   R?   M? | like
-----+-------------------------------------------------------+-----
n00  |  n01*   n02*   n06*  n08*   n09*   a13* a14*  -   a15*|  -
n01  |  n01    n02    n06   n08    n09    a13  a14  a00* a15 | n00
n02  |  n01    n02    n06   n08    n03*   a04* a14  a05* a15 | n00
n03  |  n04*   n05*   n06   n08    n09    a13  a14   -   a15 | n09
n04  |  n01    n02    n06   n08    n09    a01* a02* a00  a15 | n10
n05  |  n01    n02    n06   n08    n03    a04  a14  a03* a15 | n11
n06  |  n01    n02    n06   n08    n07*   a13  a14   -   a07*| n00
n07  |  n10    n11    n06   n08    n09    a13  a14  a06* a15 | n09
n08  |  n01    n02    n06   n08    n09    a13  a14  a09* a08*| n00
n09  |  n10*   n11*   n06   n08    n09    a13  a14   -   a15 | n00
n10  |  n01    n02    n06   n08    n09    a10* a11* a00  a15 | n01
n11  |  n01    n02    n06   n08    n03    a04  a14  a12* a15 | n02
```

The asterisks (*) indicate transitions that are in the original tree. The non-asterisked entries of a row are copied from the row it (otherwise) behaves like. Compare, for instance, row n11 with row n02.

The rule/length information associated with accepting states are encoded as bit patterns, as shown in the following table.

```
     #8 #7   #6  #5  #4 #3 #2 #1  Path                  Rule/Length

a00  10 10 0010 000 000 00  0  0  Add.1-R?              #8/1,#7/1,#6/1
a01  00 00 1000 100 000 00  0  1  Add.2-Ind.1-Add.1-K1  #6/3,#5/2,#1/0
a02  00 00 1000 100 000 00  0  1  Add.2-Ind.1-Add.1-K?  #6/3,#5/2,#1/0
a03  00 10 1000 100 000 00  0  0  Add.2-Ind.1-Add.2-R?  #7/1,#6/3,#5/2
a04  10 00 0000 000 000 00  0  1  Add.2-K1              #8/1,#1/0
a05  00 10 0000 000 000 00  0  0  Add.2-R?              #7/1
a06  00 00 0000 000 100 00  0  0  Ass.1-Ind.1-R?        #4/2
a07  00 00 0000 000 000 10  1  0  Ass.1-M?              #3/1,#2/0
a08  00 00 0000 000 010 00  1  0  Ass.2-M?              #4/1,#2/0
a09  00 00 0000 000 000 10  0  0  Ass.2-R?              #3/1
a10  00 00 0000 100 000 00  0  1  Ind.1-Add.1-K1        #5/2,#1/0
a11  00 00 0000 100 000 00  0  1  Ind.1-Add.1-K?        #5/2,#1/0
a12  00 10 0000 100 000 00  0  0  Ind.1-Add.2-R?        #7/1,#5/2
a13  00 00 0000 000 000 00  0  1  K1                    #1/0
a14  00 00 0000 000 000 00  0  1  K?                    #1/0
a15  00 00 0000 000 000 00  1  0  M?                    #2/0
```

The number of bits allocated for a template is one greater than the length of the longest path in the template. The position of a one indicates the length of an accepted path string. Bit strings allow overlapping matches to the same tree template to be recorded.

3.6 The Algorithm

Perform a left to right depth first scan over the subject tree attaching (context) states of the acceptor to each node. For the given example this gives:

```
Replacement              Ri Ri   Ri   Ri   C  C Ri Ri
Rule number              #8 #7   #6   #5  #4 #3 #2 #1

Ass                n00   00 00 0000 000 001 00  0  0   #4/0
(#4->C)                  00 00 0000 000 000 00  0  0
*-Ind              n06   00 00 0000 000 010 00  0  0   #4/1
| *-Add            n01   00 01 0001 000 000 00  0  0   #6/0, #7/0
|   (#6->Ri)             00 00 0000 000 100 00  0  0   #4/2
|   *-Add          n01   00 01 0000 000 000 00  0  0   #7/0
|   | (#7->Ri)           10 10 0010 000 000 00  0  0   #8/1,#7/1,#6/1
|   | *-Kv         n01   00 00 0000 000 000 00  0  1   #1/0
|   | | (#1->Ri)         10 10 0010 000 000 00  0  0   #8/1,#7/1,#6/1
|   | *-Rp         n02   00 10 0000 000 000 00  0  0   #7/1
|   *-Ind          n02   00 00 0010 001 000 00  0  0   #6/1,5/0
|     (#5->Ri)           00 10 0000 100 000 00  0  0   #7/1,#5/2
|     *-Add        n03   00 01 0100 010 000 00  0  0   #7/0,#6/2,#5/1
|       (#7->Ri)         00 00 0000 000 000 00  0  0
|       *-Ki       n04   00 00 1000 100 000 00  0  1   #6/3,#5/2,#1/0
|       | (#1->Ri)       10 10 0010 000 000 00  0  0   #8/1,#7/1,#6/1
|       *-Rp       n05   00 10 1000 100 000 00  0  0   #7/1,#6/3,#5/2
*-Mx               n08   00 00 0000 000 010 00  1  0   #4/1,#2/0
  (#2->Ri)               00 00 0000 000 000 10  0  0   #3/1
```

An item of the form `(#i->Op)` indicates that the current branch of the parse tree can be matched by rule `#i`, and if it is, the branch would be replaced by a leaf node with operator `Op`.

The bit patterns indicate for each position in the parse tree which rules are matched and to what depths. When the least significant bit is a one then that point in the tree, it can be matched by the corresponding rule. For instance, the bit pattern indicates that the root node can be matched by rule `#4`. The value obtained for a node is computed by ANDing together the bit patterns for its children and shifting the result right by one position. When a template matches that position in the tree can be replaced by a leaf node. This may lead to another accepting state whose bit pattern should be regarded as being ORed with the bit pattern for the node itself. Careful study of the above table should make the mechanism clear.

As each possible replacement is found its cost is computed and, if found to be lower than the cost of a previously discovered replacement (yielding the same leaf node), the new cost and the number of the rule that made it possible is recorded in the tree.

When the depth first scan is complete the root node will contain a list of possible leaf nodes that it can be replaced by, together with the minimum cost for each replacement and the corresponding rule that was used. A second pass over the tree can generate the code corresponding to this minimum cost covering.

Chapter 4

Object Modules and Linkers

We have shown how to generate assembly-style code for a typical programming language using relatively simple techniques. What we have still omitted is how this code might be got into a state suitable for execution. Usually a compiler (or an assembler, which after all is only the word used to describe the direct translation of each assembler instruction into machine code) takes a source language and produces an *object file* or *object module* (`.o` on Unix and `.OBJ` on MS-DOS). These object files are linked (together with other object files from program libraries) to produce an *executable file* (`.EXE` on MS-DOS) which can then be loaded directly into memory for execution. Here we sketch briefly how this process works.

Consider the C source file:

```
int a = 37;
extern int h(void);
int f(int x) { return x+1; }
int g(int x) { return x+a+h(); }
```

Such a file will produce a *code segment* (often called a *text segment* on Unix) here containing code for the functions `f` and `g` and a *data segment* containing static data (here `a` only).

The data segment will contain 4 bytes probably `[0x25 00 00 00]`.

The code for `f` will be fairly straightforward containing a few bytes containing bit-patterns for the instruction to add one to the argument (maybe passed in a register like `%eax`)) and return the value as result (maybe also passed in `%eax`). The code for `g` is more problematic. Firstly it invokes the procedure `h()` whose final location in memory is not known to `g` so how can we compile the call? The answer is that we compile a 'branch subroutine' instruction with a dummy 32-bit address as its target; we also output a *relocation entry* in a *relocation table* noting that before the module can be executed, it must be linked with another module which gives a definition to `h()`.

Of course this means that the compilation of `f()` (and `g()`) cannot simply output the code corresponding to `f`; it must also register that `f` has been defined by placing an entry to the effect that `f` was defined at (say) offset 0 in the code segment for this module.

It turns out that even though the reference to `a` within `g()` is defined locally we will still need the linker to assist by filling in its final location. Hence a relocation entry will be made for the 'add `A`' instruction within `g()` like that for 'call `h`' but for 'offset 0 of the current data segment' instead of 'undefined symbol `h`'.

A typical format of an object module is shown in Figure 4.1 for the format ELF often used on Linux (we only summarise the essential features of ELF).

Header information; positions and sizes of sections
`.text` segment (code segment): binary data
`.data` segment: binary data
`.rela.text` code segment relocation table: list of (offset,symbol) pairs showing which offset within `.text` is to be relocated by which symbol (described as an offset in `.symtab`)
`.rela.data` data segment relocation table: list of (offset,symbol) pairs showing which offset within `.data` is to be relocated by which symbol (described as an offset in `.symtab`)
`.symtab` symbol table: List of external symbols used by the module: each is listed together with attribute 1. undef: externally defined; 2. defined in code segment (with offset of definition); 3. defined in data segment (with offset of definition). Symbol names are given as offsets within `.strtab` to keep table entries of the same size.
`.strtab` string table: the string form of all external names used in the module

Figure 4.1: Summary of ELF

4.1 The linker

Having got a sensible object module format as above, the job of the linker is relatively straightforward. All code segments from all input modules are concatenated as are all data segments. These form the code and data segments of the executable file.

Now the relocation entries for the input files are scanned and any symbols required, but not yet defined, are searched for in (the symbol tables of) the library modules. (If they still cannot be found an error is reported and linking fails.) Object files for such modules are concatenated as above and the process repeated until all unresolved names have been found a definition.

Now we have simply to update all the dummy locations inserted in the code and data segments to reflect their position of their definitions in the concatenated code or data segment. This is achieved by scanning all the relocation entries and using their definitions of 'offset-within-segment' together with the (now know) absolute positioning of the segment in the resultant image to replace the dummy value references with the address specified by the relocation entry.

(On some systems exact locations for code and data are selected now by simply concatenating code and data, possibly aligning to page boundaries to fit in with virtual memory; we want code to be read-only but data can be read-write.)

The result is a file which can be immediately executed by *program fetch*; this is the process by which the code and data segments are read into virtual memory at their predetermined locations and branching to the *entry point* which will also have been marked in the executable module.

[The end]