UNIVERSITY OF CAMBRIDGE
Computer Laboratory

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

Compiler Construction

http://www.cl.cam.ac.uk/Teaching/2003/CompConstr

Alan Mycroft am@cl.cam.ac.uk

2003–2004 (Lent Term)

Summary

The first part of this course covers the design of the various parts of a fairly basic compiler. The second part of the course considers various language features and concepts common to many programming languages, together with an outline of the kind of run-time data structures and operations they require.

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)

Acknowledgments

Various parts of these notes are due to or based on material developed by Dr M. Richards of the Computer Laboratory. Dr G. Bierman provided the call-by-value lambda-form for Y.

STOP PRESS. We have recently been awarded a grant from Microsoft's "Microsoft Research Curriculum Development Request for Proposal" fund which "seeks to fund creative Content and Curriculum projects from the academic community that will help to address some of the shortages in Microsoft technology-inclusive teaching materials, and to increase the quality of existing content in areas relevant to core informatics curricula."

So this means examples will appear on the course web-site in C♯ as well as Java and other languages, leaving readers to make their own comparison.

In due course materials from Cambridge and elsewhere may appear on the Microsoft Content and Curriculum Repository via `http://www.msdnaa.net` .

Teaching and Learning Guide

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

- **Survey of execution mechanisms.** The spectrum of interpreters and compilers; compile-time and run-time. Structure of a simple compiler. Java virtual machine (JVM).

- **Lexical analysis and syntax analysis.** Regular expressions and finite state machine implementations. Phrase Structured Grammars. Chomsky classification. Parsing algorithms: recursive descent and SLR(k)/LALR(k). Syntax error recovery. Abstract syntax tree; expressions, declarations and commands.

- **Simple type-checking.** Type of an expression determined by type of subexpressions; inserting coercions.

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

- **Intermediate code interpreter.** Essential structure of a JVM interpreter.

- **Code generation.** Typical machine codes. Code generation from intermediate code. Simple peephole optimisation.

- **Compiler compilers.** Compiler compilers. Summary of Lex and Yacc.

- **Object Modules and Linkers.** Resolving external references. Static and dynamic linking.

- **Variable binding and tree-based interpreter.** Lambda-calculus as prototype. Problems with `rec` and class variables. A simple lambda interpreter. Environments, function values are closures. Static and Dynamic Binding. Landin's principle of correspondence.

- **Machine implementation of a selection of interesting things.** Free variable treatment, static and dynamic chains, ML free variables. Argument passing mechanisms. Objects and inheritance; the virtual function table. Labels, `goto` and exceptions. Dynamic and static typing, polymorphism. Storage allocation, garbage collection.

A good source of exercises is the past 20 or 30 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.

Note: these notes have last year been re-written to use Java and the JVM as the intermediate code. I would be grateful for comments identifying errors or readability problems.

Contents

1 Introduction and Overview

Never put off till run-time what you can do at compile-time. [David Gries]

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 compilation 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 for a simple compiler and many million for serious optimising compilers) 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. Although such a compiler can be simple and fast, such a route is generally avoided because (a) it is hard to optimise the code and (b) the techniques developed for multi-pass compilation are better known.

Most languages have features that make compilation in a single pass either difficult or impossible. For example, in Java the definition of names may occur many lines after they are first used, as in:

```
class A {
  public int g() { return f(); }
  // ... many lines before we find ...
  public int f() { ... }
}
```

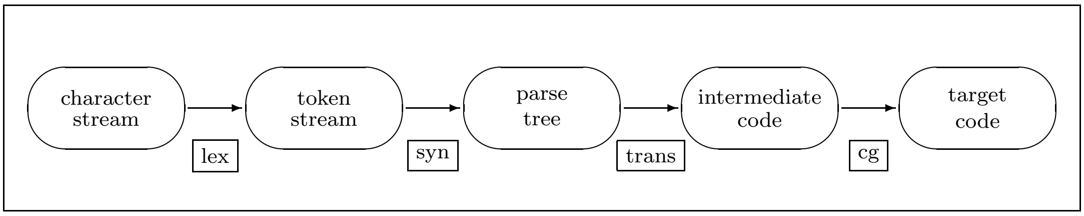
Similarly in ML, consider programs like:

```
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 phases, 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.*

1.1 The structure of a typical multi-pass compiler

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



1.2 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 words, 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 phrase:

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

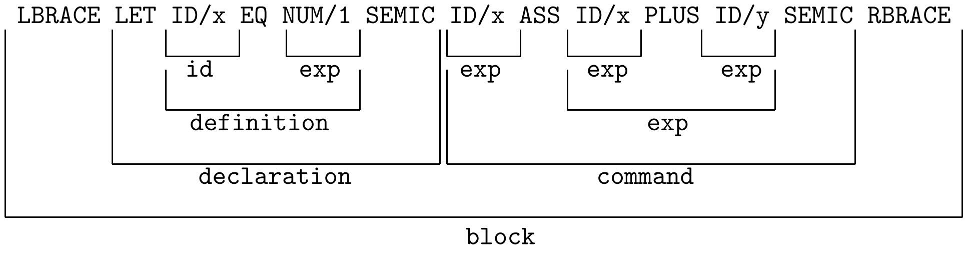
might be:

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

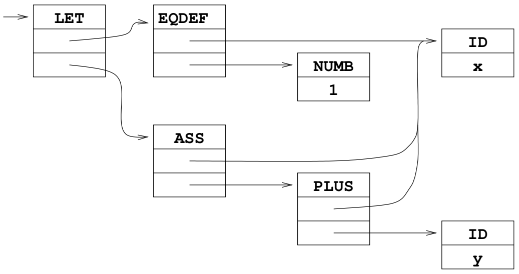
Lexical tokens are often represented in a compiler by small integers; for composite tokens such as identifiers, numbers, etc. additional information is passed by means of pointers into appropriate tables; for example calling a routine `lex()` might return the next token while setting a global variable `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.

1.3 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:


LBRACE LET ID/x EQ NUM/1 SEMIC ID/x ASS ID/x PLUS ID/y SEMIC RBRACE
id
exp
exp
exp
exp
definition
exp
declaration
command
block

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. Similarly, nodes representing identifiers are stored uniquely—this saves store and reduces the problem of comparing whether identifiers are equal to simple pointer equality. 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`.

1.4 The translation phase

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

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

However, nowadays often a separate 'type-checking' phase is run on the syntax tree below translation. This phase at least conceptually modifies the syntax tree to indicate which version of an overloaded operator is required. We will say a little more about this in section 6.7.

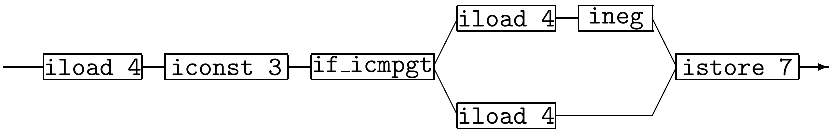
The intermediate object code for the statement:

```
y := x<=3 ? -x : x
```

might be as follows (using for JVM as an example intermediate code):

```
iload 4         load x (4th load variable)
iconst 3        load 3
if_icmpgt L36   if greater (i.e. condition false) then jump to L36
iload 4         load x
ineg            negate it
goto L37        jump to L37
label L36
iload 4         load x
label L37
istore 7        store y (7th local variable)
```

Alternatively, the intermediate object code could be represented within the compiler as a directed graph[1] as follows:



1.5 The code generator

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

```
y := x<=3 ? -x : x
```

we can easily produce (simple if inefficient) ARM code of the form using the traditional downwards-growing ARM stack:

```
     LDR   r0,[fp,#40-16]    load x (4th local variable, out of 10 say)
     MOV   r1,#3             load 3
     CMP   r0,r1
     BGT   L36               if greater then jump to L36
     LDR   r0,[sp,#40-16]    load x
     RSB   r0,r0,#0          negate it
     STMDB sp!,{r0}          i.e. PUSH r0 (to local stack)
     B     L37               jump to L37
L36: LDR   r0,[sp,#40-16]    load x
     STMDB sp!,{r0}          i.e. PUSH r0 (to local stack)
L37: LDMIA sp!,{r0}          i.e. POP r0 (from local stack)
     STR   r0,[sp,#40-28]    store y (7th local variable)
```

[1] The Part II course on optimising compilers will take this approach but here it would merely add to the weight of concepts for no clear gain.

(Pentium code would be very similar, if spelt differently—see Section 7.) This code has the property that it can simply be generated from the above JVM code on an instruction-by-instruction basis (which explains why I have not hand-optimised the `PUSH`es and `POP`s away).

When compilers produce textual output in a file (for example `gcc`) it is necessary to have a separate program (usually called an *assembler*) to convert this into an object file. An assembler is effectively a simple compiler which reads an instruction at a time from the input stream and writes the binary representation of these into the object file output. One might well argue that this is a separate pass, which separates the formatting of the binary file from the issue of deciding which code to generate.

1.6 Compiler Summary

The four passes just described form a clear-cut logical division of a 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. In simple compilers 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 for complex language features or if a high degree of optimisation is required. Examples might be separating the type-checking phase from the translation phase (for example as code which replaces source-level types in the syntax tree with more machine-oriented type information), or by adding additional phases to optimise the intermediate code structures (e.g. common sub-expression elimination which reworks code to avoid re-calculating common subexpressions).

The advantages of the multi-pass approach can be summarised as.

1. It breaks a large and complicated task into smaller, more manageable pieces. [Anyone can juggle with one ball at a time, but juggling four balls at once is much harder.]
2. 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.
3. A multi-pass compiler tends to be easier to describe and understand.
4. More of the design of the compiler is language independent. It is sometimes possible to arrange that all language dependent parts are in the lexical and syntax analysis (and type-checking) phases.
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.

1.7 Reading compiler output

Reading assembly-level output is often useful to aid understanding of how language features are implemented; if the compiler can produce assembly code directly then use this feature, for example

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

will write a file `foo.s` containing assembly instructions. Otherwise, use a *disassembler* to convert the object file back into assembler level form, e.g. in Java

```
javac foo.java
javap -c foo
```

Note that the '`-c`' switch seems not to work on all versions of `javap`. It works at least on the Linux PWF facility and also on the following variants of `thor`:

```
Solaris Release 8 [hammer] Linux Red Hat Release 7.1 [belt, gloves] (Thor)
```

1.8 The linker

Most programs written in high-level languages are not self-contained. In particular they may be written in several modules which are separately compiled, or they may merely use library routines which have been separately compiled. With the exception of Java (or at least the current implementations of Java), the task of combining all these separate units is performed by a *linker* (on Linux the linker is called `ld` for 'loader' for rather historical reasons). A linker concatenates its provided object files, determines which library object files are necessary to complete the program and concatenates all these to form a single *executable* output file. Thus classically there is a total separation between the idea of *compile-time* (compiling and linking commands) and *run-time* (actual execution of a program).

In Java, the tendency is to write out what is logically the intermediate language form (i.e. JVM instructions) into a `.class` file. This is then dynamically loaded (i.e. read at run-time) into the running application. Because (most) processors do not execute JVM code directly, the JVM code must be interpreted (i.e. simulated by a program implementing the JVM virtual machine). An alternative approach is to use a so-called *just in time* (JIT) compiler in which the above code-generator phase of the compiler is invoked to convert the loaded JVM code into native machine instructions (selected to match the hardware on which the Java program is running). This idea forms part of the "write once, compile once, run anywhere" model which propelled Java into prominence when the internet enabled `.class` files (applets) to be down-loaded to execute under an Internet *browser*

1.9 Compilers and Interpreters

The above discussion on Java execution mechanism highlights one final point. Traditionally user-written code is translated to machine code appropriate to its execution environment where it is executed directly by the hardware. The JVM *virtual machine* above is an alternative of an *interpreter-based* system. Other languages which are often interpreted are Basic, various scripting languages (for shells, spreadsheets and the like), perl etc. The common thread of these languages is that traditional compilation as above is not completed and some data structure analogous to the input data-structure of one of the above compiler phases. For example, some Basic interpreters will decode lexical items whenever a statement is executed (thus syntax errors will only be seen at run-time); others will represent each Basic statement as a parse tree and refuse to accept syntactically invalid programs (so that run-time never starts). What perhaps enables Java to claim to be a compiled language is that compilation proceeds far enough that all erroneous programs are rejected at compile-time. Remaining run-time problems (e.g. de-referencing a NULL pointer) are treated as *exceptions* which can be handled within the language.

I will present a rather revisionist view on compilers and interpreters in the second part of the course.

Part A: A Simple Compiler

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.

2.1 Regular expressions

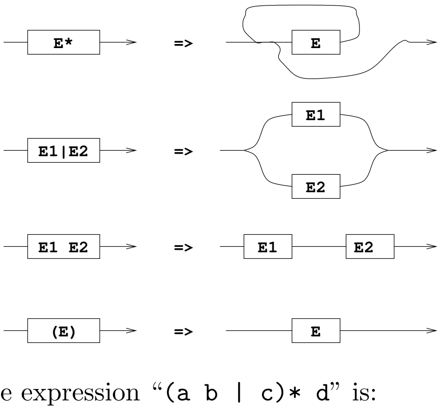
The recognition of lexical tokens is straightforward and does not require a sophisticated analyser. This results from the simple syntax of lexical tokens. It is usually the case that all the lexical tokens of a language can be described by *regular expressions*, which then implies that the recognition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

A *regular expression* is composed of characters, operators for concatenation (juxtaposition), 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:

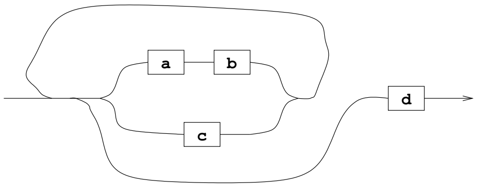
```
d
abd      cd
ababd    abcd   cabd   ccd
etc.
```

From other courses you know that, given any regular expression, there is a *finite state automaton* which will accept exactly those strings generated by the regular expression. A (deterministic) finite state automaton is a 5-tuple (A, Q, q_0, δ, F) where A is a alphabet (a set of symbols), Q is a set of states, $q_0 \in Q$ is the start state, $\delta : Q \times A \rightarrow Q$ is a transition function and $F \subseteq Q$ is the set of accepting states. It is often simpler first to construct a *non-deterministic* finite automaton which is as above, but the transition function is replaced by a transition *relation* $\delta \subseteq Q \times A \times Q$.

When such a automaton is drawn diagramatically we often refer to it as a *transition diagram.* Constructing the finite state automaton from a regular expression can be then seen with a starting with a single transition labelled (rather illegally for our definition of finite state automaton) with the regular expression and then repeatedly applying the following rules to the transition diagram.



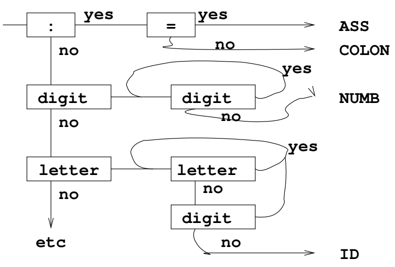
The transition diagram for the expression "`(a b | c)* d`" is:



A finite state automaton can be regarded as a *generator of strings* by applying the following algorithm:

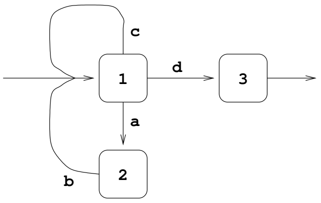
1. Follow any path from the starting point to any accessible box.
2. Output the character in the box.
3. Follow any path from that box to another box (possibly the same) and continue from step (2). The process stops when an exit point is reached (exit points correspond to accepting states).

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 (i.e. the finite state automaton is deterministic).

Note that finite state automata are alternatively (and more usually) written as a directed graph with nodes representing states and labelled edges representing the transition function or relation. For example, the graph for the expression "`(a b | c)* d`" can also be drawn as follows:



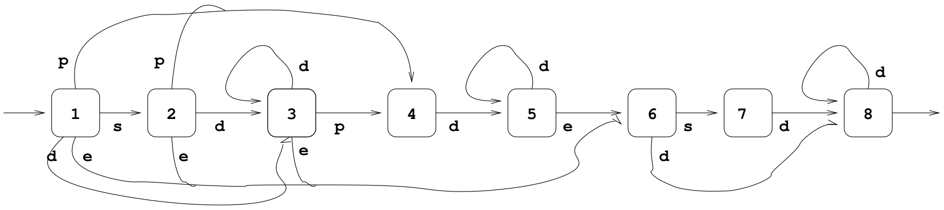
With state 3 designated an accepting state, this graph is a finite state automaton for the given regular expression. The automaton is easily implemented using a *transition matrix* to represent the graph.

We will demonstrate the method by considering the following syntax of floating point numbers (the '⟶' notation is introduced below in section 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[2] and note the next matrix row to be used. The entry `acc` would point to 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.

Later we will look at automated methods for producing such tables from a regular expression.

3 Phrase structured grammars

We start with an *alphabet*, Σ, of symbols (think of these as characters to lexing or tokens resulting from lexing). A *string* over this alphabet is just a finite sequence $\sigma_1 \cdots \sigma_n$ of symbols from Σ. A *language* is then merely a set of such strings. (Using the 'star' notation from regular expressions earlier, we can hence say that a language $\mathcal{L}$ over an alphabet Σ is a subset of Σ^*, i.e. $\mathcal{L} \subseteq \Sigma^*$.

To make this definition workable we need a rule to describe those strings in the language. For a rather boring language, consider the alphabet to be set of all letters `{a ... z}` and the rule to be "all strings of length three" then we would have the language whose strings are:

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

Note that this is a finite language, but some languages may be infinite (e.g. "all strings of even length"). Such informal rules (specified by English) are not terribly useful to Computer Science (e.g. think of the rule "all valid Java programs"), so we turn to *grammars*.

Informally a *grammar* (more precisely a *phrase structured grammar*) has additional symbols (non-terminals) which are not part of the language we wish to describe. The 'rule' for determining

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

which strings are part of the language is then a two-part process: strings containing such non-terminals can be re-written to other strings (perhaps also containing other non-terminals) using *production rules*; strings containing no non-terminal symbols are considered part of the language.

A *grammar* is can then be defined to be a 4-tuple (T, N, S, R) where T and N are disjoint sets of respectively terminal and non-terminal symbols, $S \in N$ is the start (or sentence) symbol, and R is a set of *productions*. T performs the rôle of Σ above, but now it is convenient to use the word 'symbol' to mean any symbol in $T \cup N$. The most general form of a production is:

$$A_1\ A_2 \cdots\ A_m \longrightarrow B_1\ B_2 \cdots\ B_n$$

where the A_i and B_i are symbols and $A_1\ A_2 \cdots\ A_m$ contains at least one non-terminal.

The above rule specifies that if $A_1\ A_2 \cdots\ A_m$ occurs in a string generated by the grammar then the string formed by replacing $A_1\ A_2 \cdots\ A_m$ by $B_1\ B_2 \cdots\ B_n$ is also generated by the grammar (note that the symbol '`::=`' is sometimes used as an alternative to '⟶'). The string consisting of just the start symbol S is defined to be trivially generated by the grammar. Any string that can be formed by the application of productions to S is called a *sentential form*. A sentential form containing no non-terminals is called a *sentence*. The language generated by a grammar is the set of sentences it generates. 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 important not to confuse T and N. Elements of T occur in programs, in examples terminal symbols may be `a`, `b`, `c` as in the length-3 string example above and occur in input text. *Non-terminals* like `Term` or `Declaration` do not occur in input text but instead are place holders for other sequences of symbols.

It is useful to impose certain restrictions on $A_1\ A_2 \cdots\ A_m$ and $B_1\ B_2 \cdots\ B_n$ 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 (commonly known as a context-free grammar for reasons which will become clear below).

3.1 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:

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

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.

3.2 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 `Cs` are at the right hand end.

```
aaaaaBBBBBCCCCC
```

Finally, use the last four productions to convert all the Bs and Cs 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.

3.3 Type 1 grammars

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

$$\underbrace{L_1 \cdots L_l}\ \ A\ \ \underbrace{R_1 \cdots R_r} \longrightarrow \underbrace{L_1 \cdots L_l}\ \ \overbrace{B_1 \cdots B_n}\ \ \underbrace{R_1 \cdots R_r}$$

where A is a single non-terminal symbol, and the $L_1 \cdots L_l$, $R_1 \cdots R_r$ and $B_1 \cdots B_n$ are sequences of terminal and non-terminal symbols. The sequence $B_1 \cdots B_n$ may not be empty. These grammars are called *context sensitive* since A can only be replaced by $B_1 \cdots B_n$ if it occurs in a suitable context (the L_i are the left context and the R_i the right context).

3.4 Type 3 grammars

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

$$\begin{array}{ccc} A & \longrightarrow & a \\ A & \longrightarrow & a\ B \end{array}$$

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` $\longrightarrow \epsilon$ otherwise the regular language consisting of the empty string (accepted by an automaton whose initial state is accepting, but any non-empty input sequence causes it to move to a non-accepting state) cannot be represented as a type 3 grammar.]

Finally, note that clearly every Type 3 grammar is a Type 2 grammar and every Type 2 grammar is a Type 1 grammar etc. Moreover these inclusions are strict in that there are languages which can be generated by (e.g.) a Type 2 grammar and which cannot be generated by any Type 3 grammar. However, just because a particular language can be described by (say) a Type 2 grammar does not automatically mean that there is no Type 3 grammar which describes the language. An example would be the grammar G given by

```
S  ⟶  a
S  ⟶  S a
```

which is of Type 2 (and not Type 3) but the grammar G' given by

```
S  ⟶  a
S  ⟶  a S
```

clearly generates the same set of strings (is *equivalent* to G) and is Type 3.

4 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.

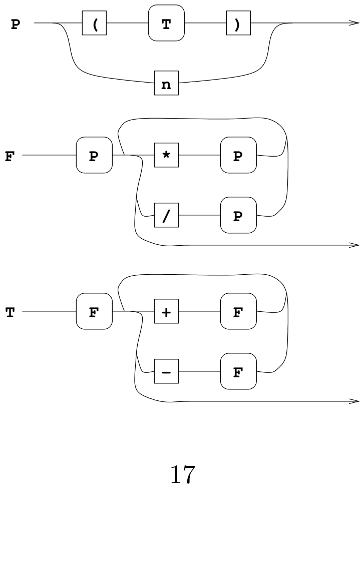
We will now look at two main parsing techniques, namely: *recursive descent* and *SLR(1)* (SLR(1) is a simpler form of LALR(1) used by `yacc` and `CUP`).

4.1 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
```

where the terminal symbol `n` represents name or number token from the lexer. The corresponding transition diagrams are:



Notice that the original syntax has been modified to avoid left recursion[3] 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:  return;
  }
}

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

4.2 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 = Add  of E * E  |  Sub 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 { E_Add, E_Sub, E_Mult, E_Div, E_Paren, E_Numb } flavour;
  union { struct { struct E *left, *right; } diad;
            // selected by E_Add, E_Sub, E_Mult, E_Div.
```

[3] 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.

```
          struct { struct E *child; } monad;
            // selected by E_Paren.
          int num;
            // selected by E_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 = E_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 1.

When there are many such operators like `+`, `-`, `*`, `/` with similar syntax it can often simplify the code to associate a binding power (or *precedence*) 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)`, `RdF()` to `RdE(1)` and `RdP()` to `RdE(2)`.

```
    struct E *RdP()
    {   struct E *a;
        switch (token)
        {   case ’(’: lex(); a = RdT();
                      if (token != ’)’) error("expected ’)’");
                      lex(); return a;
            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");
        }
    }

/* This example code includes a right associative ’^’ operator too...   */
/* ’^’ binds more tightly than ’*’ or ’/’.  For this example, The rule  */
/*          F ::= P | F * P | F / P                                     */
/* has been changed into the two rules                                  */
/*          F ::= G | F * G | F / G          G ::= P | P ^ G            */
    struct E *RdG()
    {   struct E *a = RdP();
        switch (token)
        {   case ’^’: lex(); a = mkE_Pow(a, RdG()); return a;
            default:  return a;
        }
    }

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

    struct E *RdT()
    {   struct E *a = RdF();
        for (;;) switch (token)
        {   case ’+’: lex(); a = mkE_Add(a, RdF()); continue;
            case ’-’: lex(); a = mkE_Sub(a, RdF()); continue;
            default:  return a;
        }
    }
```

Figure 1: Recursive descent parser yielding a parse tree

4.3 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 $\boxed{\texttt{eof}}$ which must not occur anywhere else in the grammar. (When you revisit this issue you will note that this ensures the value parsed is an `E` and what would be a reduce transition using rule `#0` is used for the `acc` accept marker.)

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` $\boxed{\texttt{eof}}$) 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
   }
```

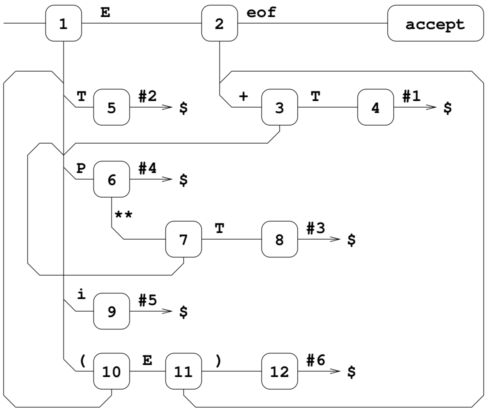
The other successor states are defined similarly, except that the successor of $\boxed{\texttt{eof}}$ is always the special state `accept`. If a new item set is identical to an already existing set then the existing set is used. The successor of a completed item is a special state represented by `$` and the transition is labelled 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. Note that for completed items the successor state is reached via the application of a production (whose number is given in the diagram).

```
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: {  T  ->  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: CFSM item sets

The CFSM can be represented diagrammatically as follows:



4.3.1 SLR(0) parser

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 `S`j.

2. If there is a transition under a non-terminal symbol A, say, from state i to state j, set `goto`$[i, A]$ to `S`j.

3. If state i contains a transition under $\boxed{\texttt{eof}}$ set `action`$[i, \boxed{\texttt{eof}}]$ to `acc`.

4. If there is a reduce transition `#p` from state i, set `action`$[i, k]$ to `#p` for all terminals k.

If any entry is multiply defined then the grammar is not SLR(0).

The example grammar given is not SLR(0), because state 6 admits both a shift transition and a reduce transition for the terminal $**$. In general right associative operators do not give SLR(0) grammars.

The key idea is to determine whether to shift or reduce according to the next terminal in the input stream—i.e. to use a 1-token *lookahead* to determine whether it is appropriate to perform reduce transition. This leads to the idea of an SLR(1) grammar to which we now turn.

4.3.2 SLR(1) parser

To construct an SLR(1) parser we must define and compute the sets `FOLLOW`(U) for all non-terminal symbols U. `FOLLOW`(U) is defined to be the set of all symbols (terminal and non-terminal) that can immediately follow the non-terminal symbol U in a sentential form. To do this, it is helpful to define the notion of the set of symbols `Left`(U) (again terminal and non-terminal) which can appear at the start of a sentential form generated from the non-terminal symbol U. I.e. if $U \xrightarrow{1+} B_1 \cdots B_n$ then B_1 is in `Left`(U) where the notation $\xrightarrow{1+}$ means using one or more production rules.

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

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

2. For each production $U \longrightarrow B_1 \cdots B_n$ enter B_1 into `Left`(U).

3. For each production $U \longrightarrow B_1 \cdots B_n$ where B_1 is also a non-terminal enter all the elements of `Left`(B_1) into `Left`(U)

4. Repeat 3. until no further change.

For the example grammar the `Left` sets are as follows:

U	`Left`(U)
S	E T P (i
E	E T P (i
T	P (i
P	(I

The sets `FOLLOW`(U) can now be formed using the following rules.

1. If there is a production of the form $X \longrightarrow \ldots YZ \ldots$ put Z and all symbols in `Left`(Z) into `FOLLOW`(Y).
2. If there is a production of the form $X \longrightarrow \ldots Y$ 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:

U	`FOLLOW`(U)
E	$\boxed{\texttt{eof}}$ +)
T	$\boxed{\texttt{eof}}$ +)
P	$\boxed{\texttt{eof}}$ +) **

The `action` and `goto` matrices are formed from the CFSM as in the SLR(0) case, but with rule 4 modified:

4' If there is a reduce transition `#p` from state i, set `action`$[i, k]$ to `#p` for all terminals k belonging to `FOLLOW`(U) where U 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	$\boxed{\texttt{eof}}$	(	i	)	+	**	P	T	E
S1	-	S10	S9	-	-	-	S6	S5	S2
S2	acc	-	-	-	S3	-	-	-	-
S3	-	S10	S9	-	-	-	S6	S4	-
S4	#1	-	-	#1	#1	-	-	-	-
S5	#2	-	-	#2	#2	-	-	-	-
S6	#4	-	-	#4	#4	S7	-	-	-
S7	-	S10	S9	-	-	-	S6	S8	-
S8	#3	-	-	#3	#3	-	-	-	-
S9	#5	-	-	#5	#5	#5	-	-	-
S10	-	S10	S9	-	-	-	S6	S5	S11
S11	-	-	-	S12	S3	-	-	-	-
S12	#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 $\boxed{\texttt{eof}}$. A typical situation is represented below:

`a A b B c C d D e E f | u v w x y z` $\boxed{\texttt{eof}}$

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 starts in state S1 with the whole program, i.e. configuration

1 | ⟨the whole program upto $\boxed{\texttt{eof}}$⟩

and then 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 $\boxed{\texttt{eof}}$

   to

   a A b B c C d D e E f u i | v w x y z $\boxed{\texttt{eof}}$

   This is called a *shift* transition.

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 matches 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 $\boxed{\texttt{eof}}$

   will transform to

   a A b B c P g | u v w x y z $\boxed{\texttt{eof}}$

   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. This is called a *reduce* transition.

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

   a Q f | $\boxed{\texttt{eof}}$

   and the parse will be complete. (Here Q will necessarily be the single non-terminal in the start symbol production (#0) and u will be the symbol $\boxed{\texttt{eof}}$.)

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 to use
1	i + i $\boxed{\texttt{eof}}$	
1 i 9	+ i $\boxed{\texttt{eof}}$	P ⟶ i
1 P 6	+ i $\boxed{\texttt{eof}}$	T ⟶ P
1 T 5	+ i $\boxed{\texttt{eof}}$	E ⟶ T
1 E 2	+ i $\boxed{\texttt{eof}}$	
1 E 2 + 3	i $\boxed{\texttt{eof}}$	
1 E 2 + 3 i 9	$\boxed{\texttt{eof}}$	P ⟶ i
1 E 2 + 3 P 6	$\boxed{\texttt{eof}}$	T ⟶ P
1 E 2 + 3 T 4	$\boxed{\texttt{eof}}$	E ⟶ E + T
1 E 2	$\boxed{\texttt{eof}}$	acc (E is result)

In practice a tree will be produced and stored attached to terminals and non-terminals on the stack. Thus the final E will in reality be a pair of values: the non-terminal E along with a tree representing i+i.

```
%%
[ \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 3: `calc.l`

4.3.3 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.

4.3.4 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 `action` and `goto` tables are thus likely to require over 40000 entries between them. There are good ways of compacting these by about a factor of ten.

5 Automated tools to write compilers

These tools are often known as compiler compilers (i.e. they compile a textual specification of part of your compiler into regular, if sordid, source code instead of you having to write it yourself).

Lex and Yacc are programs that run on Unix and provide a convenient system for constructing lexical and syntax analysers. JLex and CUP provide similar facilities in a Java environment. There are also similar tools for ML.

5.1 Lex

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 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.

5.2 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.

Like Lex, a Yacc program has three parts 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 4.

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. Note

```
%{
#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 4: calc.y

that in one sense it is not typical in that it does not construct a parse tree—instead the value of the input expression is evaluated as the expression is parsed. The first two productions for '`expr`' would more typically look like:

```
expr: '(' expr ')'     { $$ = $2; }
    | expr '+' expr    { $$ = mkbinop('+', $1, $3); }
```

where `mkbinop()` is a C function which takes two parse trees for operands and makes a new one representing the addition of those operands.

6 Translation to intermediate code

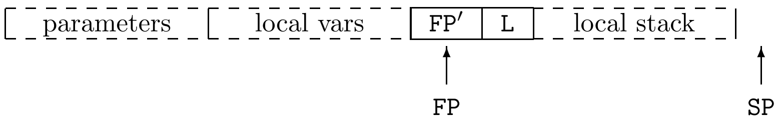
The translation phase of a compiler normally converts the abstract syntax tree representation of a program into intermediate object code which is usually either a linear sequence of statements or an internal representation of a flowchart. We will assume that the translation phase deals with (1) the scope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

Before we can give algorithms to translate a parse-tree into linear intermediate code form, we need to be a little more precise about the particular representation of the parse tree and also the intermediate code used. We present a JVM-style[4] intermediate code, which consists of a linear sequence of simple (virtual machine) instructions which act on a *run-time* stack. Note that the explanation given here corresponds to a subset of the JVM (e.g. we will not explore exceptions or non-static method invocation) and will prefer pedagogic simplicity over precision. Nevertheless, you should be able to read disassembled (using '`javap -c`') JVM `.class` files and I recommend you do this to aid your understanding.

6.1 Relevant JVM instruction set

The JVM has already been covered in the course "Computer Design". This section can be seen as a reminder which focuses on particular points relevant to compilation.

In these notes the stack is *upwards growing*, so that subsequent parameters and local variables occupy increasing memory locations (this is not necessary for the JVM but makes the pictures easier to understand). The version of the JVM we present has a stack-pointer `SP` register which points to the first free location on the stack. In addition, it has a so-called frame-pointer `FP` register which points to a constant place during the activation of a procedures. Although strictly not necessary, this simplifies explanations and aids debuggers, since local variables will remain at the same constant offset from `FP` but their offset from `SP` will vary during pushes and pops to the stack. (N.B. on both the Pentium and the ARM, the convention instead is to have the stack *downward growing*, and `SP` to hold the lowest currently used location on the stack.) Evaluation of a procedure uses a *stack frame* which will be addressed from `FP`. A stack frame has the following form:



Here `FP` points to (or rather into) the current stack frame. `FP` points to the so-called *linkage information* which allows a function call to return. This consists of the previous value of `FP`, here called `FP′` and a pointer `L` (the `return address`) to the machine instruction to be next evaluated after the function's return. To the left (at lower addresses) of the linkage informations are stored the functions parameters and (user-declared) local variables; on the right is the local stack (used for evaluating temporaries and for storing arguments), `SP` is a pointer to first free cell of this (which

[4] The Microsoft language C# has a very close resemblance to Java and their `.net` virtual machine code a similar relationship to JVM code. Their divergence owes more to commercial reasons that technological ones.

coincides with the first free cell of the whole stack). To clarify explanations of function call, we will use the word *parameters* to refer to the identifiers holding arguments to the current procedure and restrict the word *arguments* to refer to expressions are being, or have been, evaluated on the stack in order to be passed to a further function. At the end of each Java statement we can expect the local stack to be empty and hence SP to point exactly two words further along the stack from FP. In general these notes will assume that the JVM stack is an array of (32-bit) words, and so addresses of stack items will differ by one.

One more point is worthy of note. A Java routine has a fixed number of parameters (say n_p) and local variables (say n_v) and similarly the maximum depth of its local stack can be determined in advance. The latter means it is simple to check, at procedure invocation, that there is enough space beyond SP for a new stack frame, thus avoiding over-writing issues. The JVM does not have distinct instructions for addressing parameters and local variables, they are just accessed with the instruction iload i, using offsets $0..(n_p + n_v - 1)$.

Turning to the JVM instructions, variables and parameters are accessed by instructions iload i and istore i, with operand $0..(n_p+n_v-1)$. Thus "read the first parameter to a function" would be written iload 0.

More formally, and showing the use of the local stack, these JVM instructions have the following effects:

iload i	SP[0] = FP[$i-(n_p+n_v)$]; SP++;
istore i	SP--; FP[$i-(n_p+n_v)$] = SP[0];

Note that we have just written, following C, the name of a pointer as an array name, in particular SP[0] means "the cell pointed to by SP", SP[-5] means "the cell five cells before that pointed to by SP etc. If you find this (C-like) use of pointers confusing, then you may instead perfectly validly consider program data to be stored in a big Java array int [] dmem; . This this viewpoint can be exemplified by how iload is then represented:

iload i	dmem[SP+0] = dmem[FP$+i-(n_p+n_v)$]; SP++;

Note in this viewpoint that SP and FP are now integers (rather than pointers) which reflect the offset within dmem at which values are stored (which, in the pointer model, would have been represented by "pointed-to" memory locations).

Some of the other JVM instructions are as follows:

iadd	SP[-2] = SP[-2] + SP[-1]; SP--;	
isub	SP[-2] = SP[-2] - SP[-1]; SP--;	
ineg	SP[-1] = - SP[-1];	
if_icmpgt L	if (SP[-2] > SP[-1]) PC = L;	SP-=2;
if_icmpeq L	if (SP[-2] == SP[-1]) PC = L;	SP-=2;
goto L	PC = L;	

Function calls and return are more tricky because they have to create or deallocate a stack frame (the general principle is to avoid trampling on one's own feet, especially in ireturn if $n_p+n_v = 0$).

invokestatic L	SP = SP + $n_v^{\texttt{called-fn}}+2$; SP[-2] = FP; SP[-1] = PC;
	FP = SP - 2; PC = L;

ireturn	tempSP = FP - (n_p+n_v); PC = FP[1]; FP = FP[0];
	tempSP[0] = SP[-1]; SP = tempSP+1;

The use of the JVM instructions can be illustrated by considering the following Java program fragment:

```
class fntest {
public static void main(String args[]) {
    System.out.println("Hello World!" + f(f(1,2),f(3,4)));
    }
static int f(int a, int b) { int y = a+b; return y*a; }
}
```

The JVM function code generated for the function f might be:

```
iload 0       ; load a
iload 1       ; load b
iadd
istore 2      ; store result to y
iload 2       ; re-load y
iload 0       ; re-load a
imul
ireturn       ; return from fn with top-of-stack value as result
```

and the series of calls in the println in main as:

```
iconst_1
iconst_2
invokestatic f
iconst_3
iconst_4
invokestatic f
invokestatic f
```

Note how two-argument function calls just behave like binary operators in that they remove two items from the stack and replace them with one; the instructions invokestatic and ireturn both play their part in the conspiracy to make this happen. You really must work through the above code step-by-step to understand the function call/return protocol.

Instructions, such the first few instructions of f above, will be generated by translation phase of the compiler using a series of calls such as:

```
Gen2(OP_iload, 0);
Gen2(OP_iload, 1);
Gen1(OP_iadd);
Gen2(OP_istore, 2);
```

where OP_iload etc. will be represented by different values in an enumeration, for example (using the barbarous Java syntax for this)

```
static final const OP_iload = 1;
static final const OP_istore = 2;
static final const OP_iadd = 3;
static final const OP_isub = 4;
static final const OP_itimes = 5;
```

The magic numbers can correspond directly to the bit patterns in a .class file, or can be decoded in the Gen*i* routines into readable strings. Alternatively successive instructions can be stored in memory ready to be translated into native machine instructions in the CG phase.

6.2 JVM Interpreter

This section illustrates the structure of an interpreter for JVM code. It is useful to help familiarisation with the JVM instruction set, but also stands as an example of a "byte-code interpreter".

The JVM is an example of a "byte-code" interpreter. Each instruction on the JVM consists of a single byte specifying the opcode, followed by one or more operands depending on this opcode.

The structure of a JVM interpreter will be of the form:

```
void interpret()
{   byte [] imem;           // instruction memory
    int [] dmem;            // data memory
    int PC, SP, FP;         // JVM registers
    int T;                  // a temporary
    ...
    for (;;) switch (imem[PC++])
    {
case OP_ICONST_0:   dmem[SP++] = 0; break;
case OP_ICONST_1:   dmem[SP++] = 1; break;
case OP_ICONST_B:   dmem[SP++] = imem[PC++]; break;
case OP_ICONST_W:   T = imem[PC++]; dmem[SP++] = T<<8 | imem[PC++]; break;

case OP_ILOAD_1:    dmem[SP++] = dmem[FP+1]; break;
case OP_ILOAD_B:    dmem[SP++] = dmem[FP+imem[PC++]]; break;

case OP_IADD:       dmem[SP-2] = dmem[SP-2]+dmem[SP-1]; --SP; break;

case OP_ISTORE_B:   dmem[FP+imem[PC++]] = dmem[--SP]; break;

case OP_GOTO_B:     PC += imem[PC++]; break;
                    // etc
    }
}
```

Note that, for efficiency, many JVM instructions come in various forms, thus while `iconst_w` is perfectly able to load the constant zero to the stack (taking 3 bytes of instruction), the compiler will prefer to use the 1-byte form `iconst_0`.

6.3 Example tree form used in this section

In this section we will use a simple example language reflecting a subset of Java without types or classes. It has the following abstract syntax tree structure (expressed in ML for conciseness):

```
type N = string;   (* shorthand for 'name' *)
datatype E = Var of N | Num of int | Apply of N * (E list) |
             Neg of E  |  Pos of E  |  Not of E  |
             Add  of E * E  |  Sub 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  |  (* for && and || *)
             Cond of E * E * E |

     and C = Seq of C * C  |  Assign of N * E
             If3 of E * C * C  |  While of E * C  |
             Block of D list * C list | Return of E

     and D = Vardef of N * E  |   Fndef of N * (N list) * C;
type P = D list;   (* shorthand for 'program' *)
```

A program in this language essentially consists of an interleaved sequence of initialised variable definitions `let` x `=` e and function definitions `let` f`(`$x_1, \ldots, x_k$`)` c.

6.4 Dealing with names and scoping

To generate the appropriate instruction for a variable or function reference (e.g. `iload 7` instead of `y`) we require the compiler to maintain a table (often called a *symbol table* although beware that this sometimes is used for for other things). This table keeps a record of which variables are currently in scope and how the compiler may access them. For example, in Java

```
class A {
  public static int g;
  public int n,m;
  public int f(int x) { int y = x+1; return foo(g,n,m,x,y); }
}
```

the variables `x` and `y` will be accessed via the `iload` and `istore` as above, but there will be another pair of instructions to access a variable like `g` which is logically a global variable which can live in a fixed position in memory and be addressed using absolute addressing. Accessing per-instance variables, such as `n` above, is really beyond the scope of this part of the course which deals with translation, but how a translation might work will be covered in the second part of the course.

Essentially, the routine `trdecl` will save the current state of the symbol table and add the new declared names to the table. The routine `trname` consults the symbol table to determine the access path for a given name. Finally, the compiler will arrange, when it has concluded treatment a scope which has definitions, that the symbol table is restored to that which was extant at the start of the scope (and saved by `trdecl`).

As an example for the above the table might contain

`"g"`	`static variable`
`"n"`	`class variable 0`
`"m"`	`class variable 1`
`"f"`	`method`
`"x"`	`local variable 0`
`"y"`	`local variable 1`

when compiling the call to `foo`, but just the first four items when merely in the scope of `A`. In more detail, the symbol table will be extended by the entry $(\mathtt{x}, \mathit{loc}, 0)$ when `f`'s parameters (`x`) are scanned, and then by $(\mathtt{y}, \mathit{loc}, 1)$ when the local definition of `y` is encountered. The issue of how *environments* (the abstract concept corresponding to our symbol table) behave will be given in the second part of the course.

6.5 Translation of expressions

Some of the functions used during translation are as follows:

`trexp(x)`	translate an expression
`trexplist(x)`	translate an expression list
`trname(op,x)`	translate a name, `op` is one of `OP_iload`, `OP_istore`, `OP_invokestatic`
`jumpcond(x,b,n)`	translate a conditional jump
`trcmd(x)`	translate a command
`trdecl(x)`	translate a declaration

The argument to `trexp` is the tree for the expression being translated. An outline of its definition is as follows:[5]

```
fun trexp(Num(k))       = gen2(OP_iconst, k);
  | trexp(Id(s))        = trname(OP_iload,s);
  | trexp(Add(x,y))     = (trexp(x); trexp(y); gen1(OP_iadd))
  | trexp(Sub(x,y))     = (trexp(x); trexp(y); gen1(OP_isub))
```

[5] We have adopted an ML-like syntax 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`.

```
  | trexp(Mult(x,y))   = (trexp(x); trexp(y); gen1(OP_imul))
  | trexp(Div(x,y))    = (trexp(x); trexp(y); gen1(OP_idiv))
  | trexp(Neg(x))      = (trexp(x); gen1(OP_ineg))
  | trexp(Apply(f, el))   =
                ( trexplist(el);           // translate args
                  trname(OP_Invokestatic, f)) // Compile call to f
  | trexp(Cond(b,x,y)) =
                let val p = ++label;       // Allocate two labels
                    val q = ++label in
                  jumpcond(b,false,p);     // (see below for jumpcond)
                  trexp(x);                // code to put x on stack
                  gen2(OP_goto,q);         // jump to common point
                  gen2(OP_Lab,p);
                  trexp(y);                // code to put y on stack
                  gen2(OP_Lab,q)           // common point; result on stack
                end;
etc...

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

6.6 Translation of short-circuit boolean expressions

In Java, the operators `&&` and `||` are required not to evaluate their second operand if the result of the expression is determined by the value of their first operand. For example, consider code like

```
     if (i>=0 && A[i]==42) { ... }
```

If `i>=0` is false then we are forbidden to evaluate `A[i]` as it may give rise to an exception. 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. We follow C and assume that a boolean is represented as an `int` value with 0 corresponding to `false` and all other values being treated as `true`. The definition of `jumpcond` is outlined below:

```
    fun jumpcond(Num(c), true, n) =  if (c!=0) gen2(OP_goto, n) else ();
      | jumpcond(Num(c), false, n) =  if (c==0) gen2(OP_goto, n) else ();
      | jumpcond(Le(x,y), b, n) = (trexp(x); trexpr(y);
                                   gen2((b ? OP_if_cmple:OP_if_cmpgt), n))
      | jumpcond(Ne(x,y), b, n) = (trexp(x); trexpr(y);
                                   gen2((b ? OP_if_cmpne:OP_if_cmpeq), n))
      (* the cases Lt, Eq, Gt, Gt have been omitted here *)
      | 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(OP_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(OP_Lab, m);
                        end
  | jumpcond(x, b, n) =  ( trexp(x);
                          gen2(OP_iconst, 0);
                          gen2((b ? OP_if_cmpne:OP_if_cmpeq), n))
```

6.7 Type checking

So far in this section we have ignored type information (or rather, just assumed every variable and operator is of type `int`—hence the integer operators `iadd`, `ineg`, `iload` etc). In a language like Java, every variable and function name is given an explicit type when it is declared. This can be added to the symbol table along with other (location and name) attributes. The language specification then gives a way of determining the type of each sub-expression of the program. For example, the language would typically specify that $e + e'$ would have type `float` if e had type `int` and e' had type `float`.

This is implemented as follows. Internally, we have a data type representing language types (e.g. Java types), with elements like `T_float` and `T_int` (and more structured values representing things like function and class types which we will not discuss further here). A function `typeof` gives the type of an expression. It would be coded :

```
fun typeof(Num(k))      = T_int
  | typeof(Float(f))    = T_float
  | typeof(Id(s))       = lookuptype(s)  // looks in symbol table
  | typeof(Add(x,y))    = arith(typeof(x), typeof(y));
  | typeof(Sub(x,y))    = arith(typeof(x), typeof(y));
   ...
fun arith(T_int, T_int  )   = T_int
  | arith(T_int, T_float)   = T_float
  | arith(T_float, T_int)   = T_float
  | arith(T_float, T_float) = T_float
  | arith(t, t') = raise type_error("invalid types for arithmetic");
```

So, when presented with an expression like $e + e'$, the compiler first determines (using `typeof`) the type t of e and t' of e'. The function `arith` tells us the type of $e + e'$. Knowing this latter type enables us to output either a `iadd` or a `fadd` JVM operation. Now consider an expression `x+y`, say, with `x` of type `int` and `y` of type `float`. It can be seen that the type of `x` differs from the type of `x+y`; hence a *coercion*, represented by a *cast* in Java, is applied to `x`. Thus the compiler (typically in `trexp` or in an earlier phase which only does type analysis) effectively treats `x+y` as `((float)x)+y`. These type coercions are also elementary instructions in intermediate code, for example in Java, `float f(int x, float y) { return x+y; }` generates

```
iload 0
i2f
fload 1
fadd
freturn
```

Overloading (having two simultaneous active definitions for the same name, but distinguished by type) of user defined names can require careful language design and specification. Consider the C++ `class` definition

```
class A
{   int f(int, int) { ... }
    float f(float, char) { ... }
```

```
        void main() { ... f(1,'a'); ... }
    }
```

The C++ rules say (roughly) that, because the call (with arguments of type `char` and `int`) does not match any declaration of `f` exactly, the *closest in type* variant of `f` is selected and appropriate coercions are inserted, thus the definition of `main()` corresponds to one of the following:

```
        void main() { ... f(1, (int)'a'); ... }
        void main() { ... f((float)1, 'a'); ... }
```

Which is a matter of fine language explanation, and to avoid subtle errors I would suggest that you do not make your programs depend on such fine details.

7 Code Generation for Target Machine

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 instruction listed above can be mapped into a small number of ARM or Pentium instructions, essentially treating JVM instructions as a `macro` for a sequence of Pentium instructions. Doing this naïvely will produce very unpleasant code, for example recalling the

```
y := x<=3 ? -x : x
```

example and its intermediate code with

```
iload 4        load x (4th load variable)
iconst 3       load 3
if_icmpgt L36  if greater (i.e. condition false) then jump to L36
iload 4        load x
ineg           negate it
goto L37       jump to L37
label L36
iload 4        load x
label L37
istore 7       store y (7th local variable)
```

could expand to (assuming a descending stack and 10 stack locations used for parameters and local variables):

```
movl   %eax,40-16(%fp) ; iload 4
pushl  %eax            ; iload 4
movl   %eax,#3         ; iconst 3
pushl  %eax            ; iconst 3
popl   %ebx            ; if_icmpgt
popl   %eax            ; if_icmpgt
cmpl   %eax,%ebx       ; if_icmpgt
bgt    L36             ; if_icmpgt
movl   %eax,40-16(%fp) ; iload 4
...
```

However, delaying output of `PUSH`es to stack by caching values in registers and having the compiler hold a table representing the state of the cache can improve the code significantly:

```
      movl   %eax,40-16(%fp) ; iload 4       stackcache=[%eax]
      movl   %ebx,#3         ; iconst 3      stackcache=[%eax,%ebx]
      cmpl   %eax,%ebx       ; if_icmpgt     stackcache=[]
      bgt    L36             ; if_icmpgt     stackcache=[]
      movl   %eax,40-16(%fp) ; iload 4       stackcache=[%eax]
      negl   %eax            ; ineg          stackcache=[%eax]
      pushl  %eax            ; (flush/goto)  stackcache=[]
      b      L37             ; goto          stackcache=[]
L36:  movl   %eax,40-16(%fp) ; iload 4       stackcache=[%eax]
      pushl  %eax            ; (flush/label) stackcache=[]
L37:  popl   %eax            ; istore 7      stackcache=[]
      movl   40-28(%fp),%eax ; istore 7      stackcache=[]
```

I would claim that this code is near enough to code one might write by hand, especially when we are required to keep to the JVM allocation of local variables to `%fp`-address storage locations. The generation process is sufficiently simple to be understandable in an introductory course such as this one; and indeed we would not in general to seek to produce ‘optimised’ code by small adjustments to the instruction-by-instruction algorithm we used as a basis. (For more powerful techniques see the Part II course "Optimising Compilers").

However, were one to seek to improve this scheme a little, then the code could be extended to include the concept that the top of stack cache can represent integer constants as well as registers. This would mean that the `movl #3` could fold into the `cmpl`. Another extension is to check jumps and labels sufficiently to allow the cache to be preserved over a jump or label (this is quite an effort, by the way). Register values could also be remembered until the register was used for something else (we have to be careful about this for variables accessible by another thread or `volatile` in C). These techniques would jointly give code like:

```
    movl  %eax,40-16(%fp) ; iload 4     stackcache=[%eax], memo=[]
                          ; iconst 3    stackcache=[%eax,3], memo=[%eax=local4]
    cmpl  %eax,#3         ; if_icmpgt   stackcache=[], memo=[%eax=local4]
    bgt   L36             ; if_icmpgt   stackcache=[], memo=[%eax=local4]
    negl  %eax            ; ineg        stackcache=[%eax], memo=[]
    b     L37             ; goto        stackcache=[%eax], memo=[]
L36:                      ; (label)     stackcache=[], memo=[%eax=local4]
                          ; iload 4     stackcache=[%eax], memo=[%eax=local4]
L37:                      ; (label)     stackcache=[%eax], memo=[]
    movl  40-28(%fp),%eax ; istore 7    stackcache=[], memo=[%eax=local7]
```

This is now about as good as one can do with this strategy.

7.1 Table-Driven Translation to Target Code

Having shown how parser-generator tools often made it easier to produce a parser than hand-writing a recursive-descent parser, you might expect this part of the course to suggest a corresponding replacement of the translation and code-generation phases by a simple table-driven tool—just feed in a description of the target architecture get a module to perform translation directly from parse-tree to target code.[6]

In practice it is not so simple—machine instructions have detailed effects and moreover there are often idiomatic way of achieving certain effects (e.g. generating `bool y=(x<0);` as if it were `bool y=(x>>>31);` (assuming `bool` values are stored as 0 or 1 and `x` is a 32-bit value) or code optimisations based on information about the code which it is hard to represent in such tables. The search for near-perfect code leads to the tables becoming more and more complicated until they become a programming language, in which case we may as well (instead of writing code for this language) write in some more mainstream language!).

An additional reason is that modern optimising compilers tend to do have many more phases and the tree is first translated to an intermediate code which is then repeatedly hit with optimisations before being finally translated into target-specific code.

[6] If you are interested (i.e. non-examinable) in knowing more, then a technique for generating CISC-like code directly from a tree based on tree matching and rewriting techniques is given in:
Aho, A.V., Ganapathi, M. and Tjiang, S.W.K. *Code Generation Using tree matching and Dynamic programming"*. ACM Transactions on Programming Languages and Systems, Vol 11, No 4, October 1989.

8 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 m = 37;
extern int h(void);
int f(int x) { return x+1; }
int g(int x) { return x+m+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 `m` 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 `m` 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 `m`' 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 5 for the format ELF often used on Linux (we only summarise the essential features of ELF).

8.1 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.)

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 5: Summary of ELF

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.

8.2 Dynamic linking

Consider a situation in which a user has many small programs (maybe 50k bytes each in terms of object files) each of which uses a graphics library which is several megabytes big. The classical idea of linking (*static linking*) presented above would lead to each executable file being megabytes big too. In the end the user's disc space would fill up essentially because multiple copies of library code rather than because of his/her programs. Another disadvantage of static linking is the following. Suppose a bug is found in a graphics library. Fixing it in the library (`.OBJ`) file will only fix it in my program when I re-link it, so the bug will linger in the system in all programs which have not been re-linked—possibly for years.

An alternative to static linking is *dynamic linking*. We create a library which defines *stub* procedures for every name in the full library. The procedures have forms like the following for (say) `sin()`:

```
static double (*realsin)(double) = 0;  /* pointer to fn */
double sin(double x)
{   if (realsin == 0)
    {   FILE *f = fopen("SIN.DLL");    /* find object file */
        int n = readword(f);           /* size of code to load */
        char *p = malloc(n);           /* get new program space */
        fread(p, n, 1, f);             /* read code */
        realsin = (double (*)(double))p;  /* remember code address */
    }
    return (*realsin)(x);
}
```

Essentially, the first time the `sin` stub is called, it allocates space and loads the current version of the object file (`SIN.DLL` here) into memory. The loaded code is then called. Subsequent calls essentially are only delayed by two or three instructions.

In this scheme we need to distinguish the stub file (`SIN.OBJ`) which is small and statically linked to the user's code and the dynamically loaded file (`SIN.DLL`) which is loaded in and referenced at run-time. (Some systems try to hide these issues by using different parts of the same file or generating stubs automatically, but it is important to understand the principle that (a) the linker does some work resolving external symbols and (b) the actual code for the library is loaded (or possibly shared with another application on a sensible virtual memory system!) at run-time.)

Dynamic libraries have extension `.DLL` (dynamic link library) on Microsoft Windows and `.so` (shared object file) on Linux. Note that they should incorporate a version number so that an out-of-date DLL file cannot be picked up accidentally by a program which relies on the features of a later version.

The principal disadvantage of dynamic libraries is the management problem of ensuring that a program has access to acceptable versions of all DLL's which it uses. It is sadly not rare to try to run a Windows `.EXE` file only to be told that given DLL's are missing or out-of-date because the distributor forgot to provide them or assumed that you kept your system up to date by loading newer versions of DLL's from web sites! Probably static linking is more reliable for executables which you wish still to work in 10 years' time—even if you cannot find the a hardware form of the processor you may be able to find an emulator.

Part B: Implementing Language Features

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

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

9 Foundations

First we look at foundational issues and how these are important for properly understanding the fine details of a programming language.

9.1 Declarations, Expressions and Commands

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

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

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

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

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

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

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

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

1. evaluate e to give an address;

2. evaluate e' to give a value;

3. update the addressed cell with the value.

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

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

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

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

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

9.2 Variables and Names

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

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

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

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

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

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

```
float r ≃ p;
```

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

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

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

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

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

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

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

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

9.3 Functional Programming—Life without Commands

In the next few sections we will consider the interplay between declarations and side-effect-free expressions—the so-called functional languages.[7] There is synergy between these and the ML and Computation Theory courses. The simpler scenario of declarations and side-effect-free expressions will enable us to study interesting issues (at the same time avoiding issues such as whether evaluation is to give an Lvalue or Rvalue—since if the values stored in Lvalues cannot change it does not matter at what time the Rvalue is taken from it or whether aliasing takes place). However, we ensure that declarations still associate each name with a memory location so that later introducing an assignment operator is easy.

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

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

- c, an integer;
- x, a name;
- $e_1 + e_2$, provided e_1 and e_2 are (smaller) expressions;
- $e_1 - e_2$, provided e_1 and e_2 are (smaller) expressions;
- $e_1 ? e_2 : e_3$, provided e_1, e_2 and e_3 are (smaller) expressions;
- `let` $x = e_1$ `in` e_2, provided x is a name and e_1 and e_2 are (smaller) expressions. The phrase $x = e_1$ is seen as a *declaration*;
- $e_1\ e_2$, (an application) provided e_1 and e_2 are (smaller) expressions;

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

- $\lambda x.e_1$, (a lambda-abstraction) provided e_1 is a (smaller) expression. The identifier x is called the *bound variable* and the expression e_1 is called the *body* of the abstraction.

The first forms will be familiar to everyone. The final form $\lambda x.e_1$ is an (anonymous) function which takes an argument x and yields e_1. Thus the conventional function definition $f(x) = e$ would here be written $f = \lambda x.e$, i.e. one can view $\lambda x.e$ as equivalent to f `where` $f(x) = e$. We will later note that for many purposes $(\lambda x.e_2)e_1$ and `let` $x = e_1$ `in` e_2 can be treated almost identically.

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

9.4 Environments

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

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

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

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

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

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

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

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

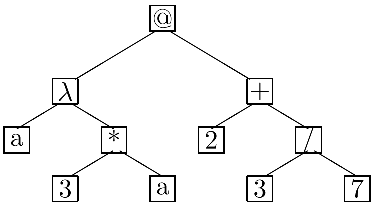
Note that the expected $BV(e) \cap FV(e) = \{\}$ does not always hold—consider

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

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

9.5 Abstract Syntax Tree Representation

As an example as to how lambda-forms and applications (we will use the symbol "@" to represent application since juxtaposition used above is otherwise invisible!) are represented as trees, the expression `(λa. 3*a)(2+3/7)` would have a tree:



9.6 Lambda calculus

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

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

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

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

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

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

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

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

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

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

9.7 The correspondence between programming languages and lambda calculus

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

Consider the following expression:

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

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

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

which further can be re-written as:

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

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

9.8 A short interlude on recursion

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

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

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

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

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

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

9.9 The need for the word `rec`

Consider the following expression:

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

The corresponding lambda expression is

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

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

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

which can be more primitively written as

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

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

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

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

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

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

9.10 The Y operator

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

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

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

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

However, if `g` were the factorial function, then

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

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

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

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

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

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

Put more simply we want

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

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

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

as

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

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

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

It is somewhat remarkable at first that such a Y exists at all and moreover that it can be written just using λ and *apply*. One can write[8]

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

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

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

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

(Why?)

Finally, an alternative implementation of Y (there seen as a primitive rather as the above arcane lambda-term) suitable for an interpreter is given in section 9.12.

[8] The form

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

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

9.11 Object-oriented languages

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

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

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

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

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

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

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

9.12 Mechanical evaluation of lambda expressions

We will now describe a simple way in which lambda expressions may be evaluated in a computer.[9] We will represent the expression as a parse tree and evaluate it in an environment that is initially empty. As above there will be tree nodes representing variables, constants, addition, function abstraction and function application. In ML this can be written:

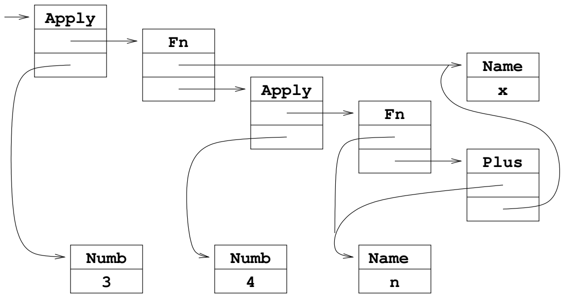
```
datatype Expr = Name of string |
                Numb of int |
                Plus of Expr * Expr |
                Fn of string * Expr |
                Apply of Expr * Expr;
```

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

[9] We do not do the evaluation by textual re-writing (as in Section 9.6 or the Part 1B course on semantics) because we later to translate the programs into machine code and this is only possible if we have a fixed set of expressions rather than dynamically re-writing potentially new expressions during evaluation.

```
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[10] could be defined in ML as follows.

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

We are now ready to define the evaluation function itself:

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

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

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.

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

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

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

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

9.13 Static and dynamic scoping

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

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

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

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

Exercises

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

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

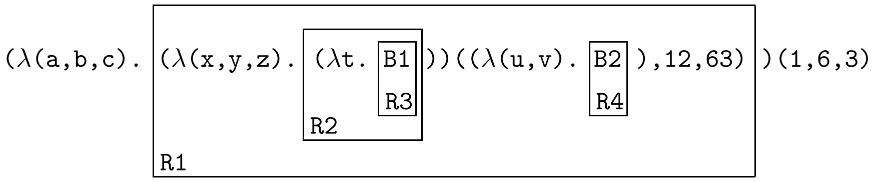
   Apply the `eval` function by hand to this tree and an empty environment and draw the structure of every environment that is used in the course of the evaluation.

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

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

9.14 A more efficient implementation of the environment

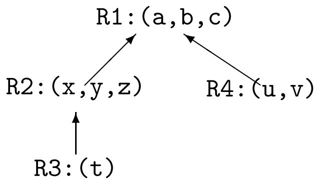
The previous lambda evaluator (also known as an *interpreter*) is particularly inefficient in its treatment of names since it searches a potentially long environment chain every time a name is used. This search can be done much more efficiently if the environment were represented differently; moreover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



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

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

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



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

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

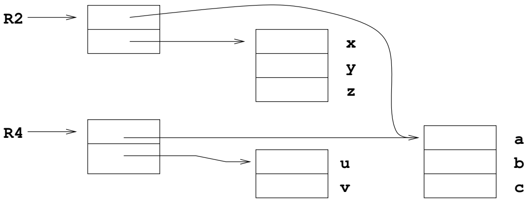
We can associate with any name used in B1 an ‘address’ consisting of a pair of numbers, namely, a level number and a position within that level. For example:

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

Similarly within B2:

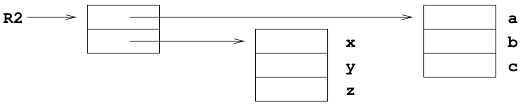
```
a:(1,1)   b:(1,2)   c:(1,3)
u:(2,1)   v:(2,2)
```

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

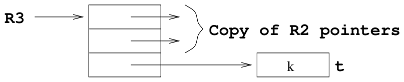


A vector such as the one pointed to by R2 is called a *display* (first coined by Dijkstra). Notice that while evaluating B2 in environment R4 we may use the “address” (2,1) to access the variable u. It should be clear that the pointer to the current display provides sufficient information to access any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

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

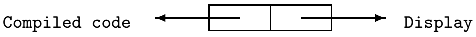


In order to apply this closure to an argument value k, we must first create a new display which consists of a copy of R2 augmented with a new level. The new display will be as follows:



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

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



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

Exercise

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

9.15 Landin's Principle of Correspondence

Landin first emphasised the connection between declarations (e.g. `let a=3 in e` and argument passing `f(3) where f(a)=e` which, in this form arises naturally for call-by-value in the λ-calculus. He suggested that well-designed languages would extend this to other situations: so if a language can pass a parameter by reference then it should have a `let`-construct to alias two variables, similarly, if it can define a procedure in a declaration it should be passable as a parameter. ML embodies much of this principle for values.

However, many languages break it for types; we often allow a type or class to be defined locally to a procedure, but do not have constructs like

```
f(int, fn x=>x+1) where f(t: type, g: t->t) = ...
```

Even if the principle is often violated, it often gives a good perspective to ask 'what if' questions about programming languages.

10 Machine Implementation of Various Interesting Things

In this section we address how various concepts in high-level languages can be implemented in terms of data structures and instructions of on a typical modern CPU, e.g. the Pentium or the ARM.

10.1 Evaluation Using a Stack—Static Link Method

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

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

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

The usual answer is to extend the *linkage* information so that in addition to holding the return address `L` and the old frame pointer `FP'`, also known as the *dynamic link* as it points to the frame of its *caller*, it also holds a pointer `S`, the *static link*[11] which points to the frame of its *definer*.

Because the definer also has its static link, access to a non-local variable (say the ith local variable in the routine nested j levels out from my current nesting) can be achieved by following the static link pointer j times to give a frame pointer from which the ith local can be extracted. Thus access to non-local variables can be slower than with a display, but the set-up time of the static link field is much quicker than creating a display. In the example in section 9.14, the environment for `B1` was `R3` with variables as follows:

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

[11] Note that the similarity to *static linking* is totally accidental.

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

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

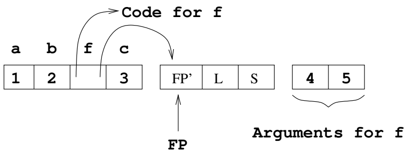
Exercise: give a simple example in which S and FP' pointers differ.

An example

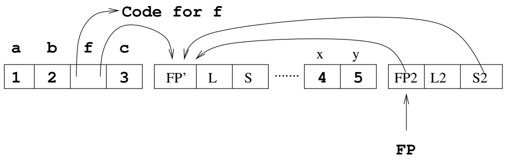
Consider the following fragment of program:

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

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



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



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

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

```
let f = λ(x,y). a*x + b*y
```

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

10.2 Situations where a stack does not work

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

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

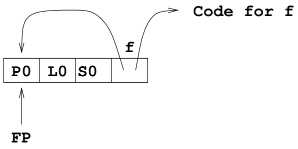
The result of f is a pointer to the local variable a but unfortunately when we return from the call this variable no longer exists and p is initialised to hold a pointer which is no longer valid and if used may cause an extremely obscure runtime error. Many languages (e.g. Pascal, Java) avoid this problem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

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

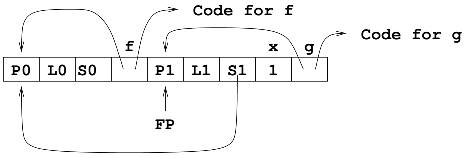
Some other objects such as functions and arrays contain implicit pointers to the stack and so have to be restricted if a stack implementation is to work. Consider:

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

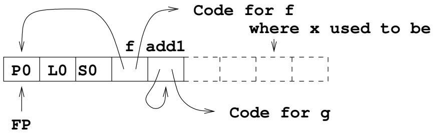
The result of f(1) should be a function which will add one to its argument. Thus one might hope that add1(23) would yield 24. It would, however, fail if implemented using a simple stack. We can demonstrate this by giving the state of the stack at various stages of the evaluation. Just after the f has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



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



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



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

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

ML clearly does allow objects to be returned from procedure calls. We can see that the problem in such languages is that the above implementation would forbid stack frames from being deallocated on return from a function, instead we have to wait until the last use of any of its bound variables.[12] This implementation is called a "Spaghetti stack" and stack-frame deallocation is handled by a garbage collector. However, the overhead of keeping a whole stack-frame for possibly a single variable is excessive and we now turn to an efficient implementation.

10.3 Implementing ML free variables

In ML programs like

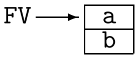
```
val a = 1;
fun g(b) = (let fun f(x) = x + a + b in f end);
val p = g 2;
val q = g 3;
```

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

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

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

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



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

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

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

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

10.4 Parameter passing mechanisms

Strachey [Fundamental Concepts in Programming Languages. Oxford University Press, 1967] described the ”Principle of Correspondence” in which, motivated by the *lambda*-calculus equivalence, he argued that simple declaration forms (e.g. of an initialised variable) and parameter passing mechanisms were two sides of the same coin.[13]

Thus if a simple variable may be defined (see section 9.2) to be either a copy or an alias of an existing variable, then so should a parameter passing mechanism. To put this another way, should parameter passing communicate the Lvalue of a parameter (if it exists) or the Rvalue?

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

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

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

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

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

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

10.5 Note on Algol call-by-name

Algol 60 is a language that attempted to be mathematically clean and was influenced by the simple calling-as-substitution-of-argument-expression-into-function-body mechanism of lambda calculus. In the standard report on Algol 60 the procedure calling mechanism is described in terms of textually replacing a call by a copy of the appropriate procedure body. Systematic renaming of identifiers (α-conversion) is used to avoid problems with the scope of names. With this approach the natural treatment for an actual parameter of a procedure was to use it as a textual replacement for every occurrence of the corresponding formal parameter. This is precisely the effect of the lambda calculus evaluation rules and in the absence of the assignment command it is indistinguishable from call-by-value or call-by-reference.[14]

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

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

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

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

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.]

10.6 A source-to-source view of argument passing

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

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

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

as

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

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

10.7 Labels and jumps

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

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

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

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

1. reset the FP pointer,
2. transfer control.

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

10.8 Exceptions

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

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

For example given `exception foo;` we would implement

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

as

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

and the `doraise()` function looks like

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

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

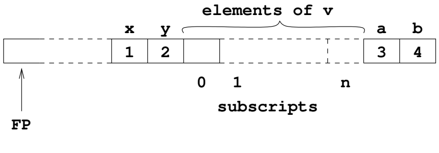
Finally, sadly ISO C labels cannot be used as values as indicated above, and so code shown above would have to be implemented using the library function `setjmp()` instead.

10.9 Arrays

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

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

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



In this example, `n` may be large and so the variables `a` and `b` may be a great distance from `FP`. On some machines access to such variables is less efficient. Moreover, if `n` is not a compile-time constant,[15] the position of `a` and `b` relative to `FP` will not be known until runtime, again causing inefficiency.

For this reason, large or compile-time-unknown size arrays are normally allocated on the heap[16]

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

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

[16] Experts might care to look at the (non-ISO) Linux C function `alloca()` for an interesting trick of allocating such arrays in the current stack frame between the received formal parameters and the out-going actual parameters. I am *not* recommending its use as not all implementations provide it and there is the hidden cost of the waste of a register on many implementations.

10.10 Object-oriented language storage layout

Declarations (in C++) like

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

allocate storage for two integers and record the fact that `a1` is at offset zero, and `a2` is at offset 4 (assuming `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 inheritance) normally stored at offsets 0 and 4 so that (pointers to) objects of type `B` can be passed to functions expecting objects of type `A` with no run-time cost. The member `b` would then be at offset 8. The following definition is similar.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

10.11 Heap Allocation and `new`

Languages like C++ and Java which provide an operator `new` to allocate new storage generally allocate such storage from a *heap*. A heap[17] is a storage area (separate from the stack and from statically allocated global variables) from which storage blocks can be allocated to a program (i.e. the heap data-structure also contains a record of which parts of it are in use and which parts are free to be allocated). You might care to note that a heap is very similar to that part of a filing

[17] Note this is a completely use of the work 'heap' meaning implementation of a priority queue within an array.

system which records which blocks on disc are used and which are available for allocation for new files.

The expression `new C` allocates enough storage for an object of type C by making a request of the heap allocation function. In C++ we have that the above request is very similar to `malloc(sizeof(C))` , although note some calculation may be necessary for requests like `new A[n+1]` to allocate an array of unknown size.

Systems may either have explicit de-allocation (C++ provides a `delete`) operator and a `free()` function which returns storage to the heap for subsequent re-allocation) or may provide implicit de-allocation via a *Garbage Collector*. In the latter case storage is simply allocated until the area allocated for the heap is full. Then the garbage collector is called. Its job is to first scan the global variables, the stack and the heap, marking which allocated storage units are reachable from any future execution of the program and flagging the rest as 'available for allocation'. It can then simply (logically) call the heap de-allocation function on these before returning. If garbage collection did not free any storage then you are out of memory!

This is how Sun's JVM default Garbage Collector works; it is called a *conservative* garbage collector in that it does not care whether a value on the stack (say 0x001b3460) is a pointer, or an integer. All such items are presumed to point to (or into!) valid objects which are thus marked as used (hence the name—it marks at least as much as it should). Note that the above "de-allocate all unmarked heap items" is as good as one can do with a conservative garbage collector. (Why?) Note also that a conservative garbage collector may signal out-of-memory when there is plenty of unused memory—in a 16Mb heap, first allocate 1 million 16-byte objects, stop using every second one, and then ask for a 1Mb array allocation!

On the other hand, if the garbage collector has access to sufficient type information to know which global variable, stack locations and heap-allocated object offsets hold pointers and which hold non-pointer data, then it is possible to move (contiguify, improving both cache and virtual memory performance) used data so that after garbage collection the unused data forms a single block of store which can be allocated sequentially. The moving process can be achieved by *compaction* (squeezing in the same space, like a disc de-fragmenter), or by *copying* from an old heap into a new heap (the rôles of these are reversed in the next garbage collection). This latter process is called a two-space garbage collector and generally works better than a conservative collector with respect to cache and virtual memory.

There are many exotic types of garbage collectors, including generational garbage collectors (exploiting the fact that allocated data tends to be quite short-lived or last for a significant time) and concurrent garbage collectors (these run in a separate thread, preferably using a separate CPU and are a significantly challenge to program, especially if minimising time wasted on locking concurrently accessed date is an issue).

10.12 Data types

With a small exception in Section 6.7, the course so far has 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 9.12 where errors occur if we apply an integer as a function or attempt to add a function to a value).

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

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

Knowing the type of a variable has the following advantages:

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

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

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

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

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

One advantage of dynamic typing over static typing is that it is easy to write functions which take a list of any type of values and applies a given function to it (usually called the `map` function). Many statically typed languages render this impossible (one can see problems might arise if lists of (say) characters were stored differently from lists of integers). Some languages (most notably ML) have `polymorphic types` which are static types[18] 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

[18] One might note with some sadness that if functions like `map` are compiled to one piece of code for all types then values will need to have type-determining tags (like dynamic compilation above) to allow garbage collection.

is treated as required by the operator which is applied to it. E.g. in `f(x+1,y[z])` we treat the values in `f`, `x`, `y`, `z` as function entry point, integer, pointer to array of words, and integer respectively. Although such languages are not common today, one can see them as being in the intersection of dynamically and statically type languages. Moreover, they are often effectively used as intermediate languages for typed languages whose type information has been removed by a previous pass (e.g. in intermediate code in a C compiler there is often no difference between a pointer and an integer, whereas there is a fundamental difference in C itself).

10.13 Source-to-source translation

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

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

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

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

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

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

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

11 Compilation Revisited and Debugging

11.1 Spectrum of Compilers and Interpreters

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

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

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

```
    iload 3
```

we execute

```
    iload(3);
```

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

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

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

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

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

11.2 Debugging

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

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

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

11.3 The Debugging Illusion

Source-level debuggers (like `gdb`) attempt to give the user the impression that the source code is being interpreted. The more optimising the compiler, the harder this is to achieve and the more information debugger tables need to hold. (Do you want to be able to put a breakpoint on a branch-to-a-branch which might have been optimised into a single branch? What if user-code has been duplicated, e.g. loop unrolling, or shared, e.g. optimising several computations of `a+b` into a single one?).

[The end]