UNIVERSITY OF
CAMBRIDGE
Computer Laboratory

Computer Science Tripos Part IB

Compiler Construction

http://www.cl.cam.ac.uk/Teaching/1011/CompConstr/

Timothy G. Griffin

tgg22@cam.ac.uk

2010–2011 (Lent Term)

Notes and Slides thanks to Prof Alan Mycroft and his predecessors.

Summary

The first part of this course covers the design of the various parts of a fairly basic compiler for a pretty minimal language. 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).

The target language of a compiler is generally machine code for some processor; this course will only use instructions common (modulo spelling) to x86, ARM and MIPS—with MIPS being favoured because of its use in the Part Ib course "Computer Design".

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/ML.

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 Martin Richards of the Computer Laboratory. Gavin Bierman provided the call-by-value lambda-form for Y. Tim Griffin's course in 2006/07 has been a source of ideas, and has a good set of alternative notes covering much of the same material from slightly difference perspective—find them on the "information for current students" part of the CL web site.

Additional Notes

The course web site contains auxiliary course notes. The material in these auxiliary notes is *not* examinable, but may provide useful background (or alternative techniques from those given here). The notes are:

1. A.C. Norman "Compiler Construction—Supplementary Notes". These notes explain how to use `JLex` and `cup` which are Java-oriented tools which replaces `lex` and `yacc` discussed in this course.

2. Neil Johnson "Compiler Construction for the 21st Century". These notes (funded by Microsoft Research's "Curriculum Development" project) introduce the tool `antlr`—an alternative to `lex` and `yacc` which combines their functionality. One particularly useful aspect is that `antlr` can output code for C, C++, Java and C♯. The notes also contain various longer exercises.

If there is enough demand (ask Jennifer or your student representative) then I will print copies.

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), JIT. Simple run-time structures (stacks). Structure of interpreters for result of each stage of compilation (tokens, tree, bytecode). [3 lectures]
- **Lexical analysis and syntax analysis.** Recall regular expressions and finite state machine acceptors. Lexical analysis: hand-written and machine-generated. Recall context-free grammars. Ambiguity, left- and right-associativity and operator precedence. Parsing algorithms: recursive descent and machine-generated. Abstract syntax tree; expressions, declarations and commands. [2 lectures]
- **Simple type-checking.** Type of an expression determined by type of subexpressions; inserting coercions. [1 lecture]
- **Translation phase.** Translation of expressions, commands and declarations. [1 lecture]
- **Code generation.** Typical machine codes. Code generation from intermediate code. Simple peephole optimisation. [1 lecture]
- **Object Modules and Linkers.** Resolving external references. Static and dynamic linking. [1 lecture]
- **Non-local variable references.** Lambda-calculus as prototype, Landin's principle of correspondence. Problems with `rec` and class variables. Environments, function values are closures. Static and Dynamic Binding (Scoping). [1 lecture]
- **Machine implementation of a selection of interesting things.** Free variable treatment, static and dynamic chains, ML free variables. Compilation as source-to-source simplification, e.g. closure conversion. Argument passing mechanisms. Objects and inheritance; implementation of methods. Labels, `goto` and exceptions. Dynamic and static typing, polymorphism. Storage allocation, garbage collection. [3 lectures]
- **Parser Generators.** A user-level view of Lex and Yacc. [1 lecture]
- **Parsing theory and practice.** Phrase Structured Grammars. Chomsky classification. LL(k) and LR(k) parsing. How tools like Yacc generate parsers, and their error messages. [2 lectures]

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: the course and these notes are in the process of being re-structured. 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 which translates the source form of a program into a semantically equivalent target form. Traditionally this was machine code or relocatable binary form, but nowadays the target form may be a virtual machine (e.g. JVM) or indeed another language such as C.

For many CST students, the idea of writing a compiler is liable to be rather daunting. Indeed to be able to do so requires total understanding of all the dark corners of the programming language (for example references to outer method arguments from methods defined in a Java inner class, or exactly which ML programs type-check). Moreover, a compiler writer needs to know how best to exploit instructions, registers and idioms in the target machine—after all we often judge a compiler by how quickly compiled programs run. Compilers can therefore be very large. In 2004[1] the Gnu Compiler Collection (GCC) was noted to "[consist] of about 2.1 million lines of code and has been in development for over 15 years".

On the other hand, the core principles and concepts of compilation are fairly simple (when seen in the correct light!) so the first part of this course explains how to construct a *simple* compiler for a *simple* language—here essentially just the features common to all three of C, Java and ML. This need not require significantly more than 1000–2000 lines of code. The standard conceptual problem is "where do I even start?". For example, deciding which version of an overloaded operator to use is hard enough, but doing it alongside determining when an inner expression finishes and choosing the appropriate x86, ARM or MIPS instruction to implement it seems at first require a brain the size of a planet. The (time-established) solution is to break down a compiler into *phases* or *passes*. Each pass does a relatively simple transformation on its input to yield its output, and the composition of the passes is the desired compiler (even GCC follows this model). This solution is called a *multi-pass compiler* and is ubiquitous nowadays. An analogy: juggling 5 balls for 1 minute is seriously hard, but juggling one ball for one minute, followed by another for one minute, and so on until all 5 balls have been juggled is much easier.

As an aside, note that many modern languages *require* a multi-pass compiler: in both the two programs below, it is rather hard to emit code for function `f()` until the definition of `g()` is found. In Java

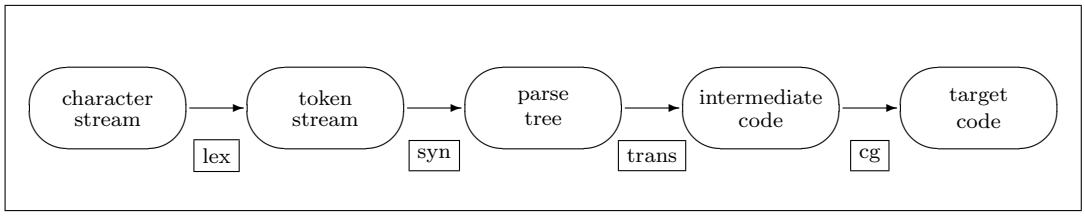
```
class A {
  public int f() { return g(1); }       // i.e. g(1.0)
  // ... many lines before we find ...
  public int g(double x) { ... }
}
```

or in ML:

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

1.1 The structure of a typical multi-pass compiler

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



[1] `http://www.redhat.com/magazine/002dec04/features/gcc/`.

1.2 The lexical analyser

The lexical analyser 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:

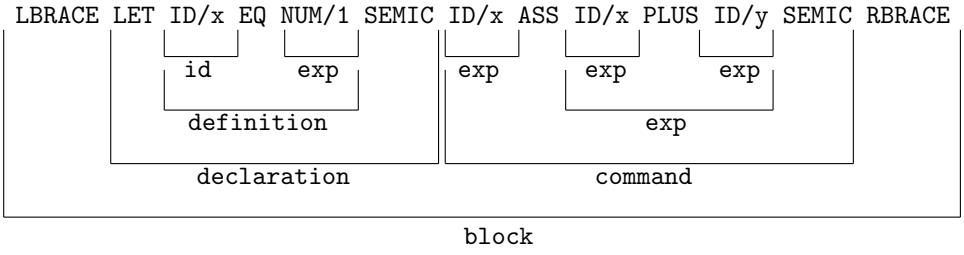
```
LBRACE LET ID/x EQ NUM/1 SEMIC ID/x ASS ID/x PLUS ID/y SEMIC RBRACE
```

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.

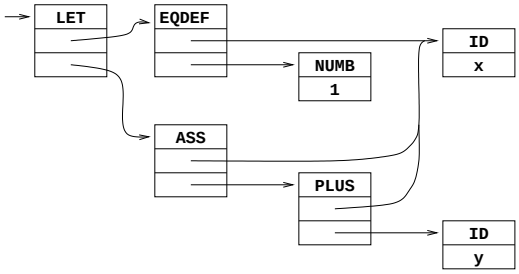
This is perhaps a rather old-fashioned and non-modular approach to returning multiple values from a function (return one and leave the additional values lying around), but it is often used in C and forms the model used by `lex` and `yacc`—see later. Of course, if you prefer a functional style, then an ML datatype for tokens is just what is required, or in Java a class `Token` which is `extend`ed to give classes `Token_Id` or `Token_Int` as required.

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:



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` occurring as part of `x+y` as an `<exp>` which is a `<factor>` which is a `<primary>` which is an `<identifier>`. This would take more tree space and would also make later processing less convenient. Similarly, nodes representing identifiers are often best 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

The translation pass/phase 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 before 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 5.

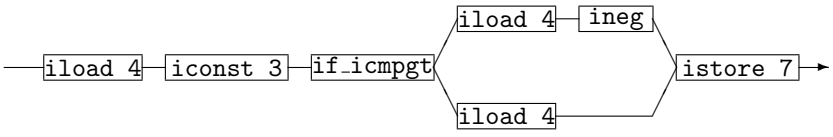
The intermediate object code for the statement:

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

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

```
iload 4         load x (4th load variable, say)
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, say)
```

Alternatively, the intermediate object code could be represented within the compiler as a directed graph[2] as follows:



1.5 The code generator

The code generation 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
```

[2] 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.

we can easily produce (simple if inefficient) MIPS code of the form using the traditional downwards-growing MIPS stack:[3]

```
     lw    $a0,-4-16($fp)    load x (4th local variable)
     ori   $a1,$zero,3       load 3
     slt   $t0,$a1,$a0       swap args for <= instead of <
     bne   $t0,$zero,L36     if greater then jump to L36
     lw    $a0,-4-16($fp)    load x
     sub   $a0,$zero,$a0     negate it
     addi  $sp,$sp,-4        first part of PUSH...
     sw    $a0,0($sp)        ... PUSH r0 (to local stack)
     B     L37               jump to L37
L36: lw    $a0,-4-16($fp)    load x
     addi  $sp,$sp,-4        first part of PUSH...
     sw    $a0,0($sp)        ... PUSH r0 (to local stack)
L37: lw    $a0,0($sp)        i.e. POP r0 (from local stack)...
     addi  $sp,$sp,4         ... 2nd part of POP
     sw    $a0,-4-28($sp)    store y (7th local variable)
```

Or on ARM

```
     LDR   r0,[fp,#-4-16]    load x (4th local variable)
     MOV   r1,#3             load 3
     CMP   r0,r1
     BGT   L36               if greater then jump to L36
     LDR   r0,[fp,#-4-16]    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,[fp,#-4-16]    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,[fp,#-4-28]    store y (7th local variable)
```

(x86 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. Similarly the translation and code generation phases *could* be combined for an ultra-simple compiler producing code of similar quality to our structure (they are left separate for the pedagogic gain of explaining stack code, JIT compilation, etc. as part of this course). Some compilers have additional passes, particularly for complex language features or if a high degree of optimisation

[3] Why does '4' turn into '-4-16'? Well `iload` counts from zero, the MIPS is byte addressed, and so '`iload 0`' corresponds to '`-4($fp)`'.

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 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 inside an Internet *browser*.

1.8 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, Ruby, Python 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 at the end of the course (§11.3).

1.9 Machine code

A compiler generally compiles to some form of machine code, so it is hard to understand what a compiler does unless we understand its output. While modern processors (particularly the x86 family) are complicated, there is a common (RISC-like) subset to most typical processors, and certainly for the x86, MIPS, ARM and SPARC.

All that this course really needs are target instructions which:

- load a constant to a register (possibly in multiple instructions) [MIPS `movhi/ori`].
- do arithmetic/logical/comparison operators on two registers and leaving the result in a register (possibly one of the source registers) [MIPS `add`, `or` etc.].
- load and store register-sized values to and from memory; these need to support two *addressing modes*—(*i*) absolute addressing (accessing a fixed location in memory) and (*ii*) indexed addressing (add a fixed offset to a register and use that as the address to access). Note that many instruction sets effectively achieve (*i*) by first loading a constant representing an address into a register and then use (*ii*) to complete the operation. [MIPS `lw`, `sw`].
- perform conditional branches, jump (possibly saving a return address) to fixed locations (or to a location specified in a register). [MIPS `beq/bne`, `j/jal`,`jr/jalr`].

Students are reminded of last term's 'Computer Design' course which introduced both MIPS and JVM code.

Reminder: the MIPS has 32 registers; registers `$a0`–`$a3` are used for arguments and local temporaries in a procedure; `$zero` always holds zero; `$fp` holds a pointer ("stack frame") in which the local variables can be found. Global variables are allocated to fixed memory locations. The exact locations (absolute addresses for global variables, offsets from `$fp` for locals) have been selected by the compiler (see later in these notes as to how this can be done).

Instructions are fetched from `pc`; this is not a user-visible register (unlike ARM), but *is* stored in `$ra` by subroutine jump instructions `jal/jalr`, a return from a subroutine can be effected by `jr $ra`.

So, for simple primitives

- constants are just loaded into a register, e.g. `0x12345678` by

```
movhi   $a0,0x1234
ori     $a0,$a0,0x5678
```

- local variables at offset `<nn>` are loaded/stored by

```
lw      $a0,<nn>($fp)
sw      $a0,<nn>($fp)
```

- global variables at address (say) `0x00be3f04` are loaded/stored with

```
movhi   $a3,0x00be
lw      $a0,0x3f04($a3)
sw      $a0,0x3f04($a3)
```

- operations like `plus` just use the corresponding machine operation:

```
add     $a2,$a0,$a1
```

- function calling is again more complicated (particularly on the MIPS or ARM where the first four arguments to a function are passed in registers `$a0`–`$a3` and subsequent ones left at the top of the callers' stack). We will ignore this here and assume a JVM-like convention: a *caller* pushes the arguments to a function on the stack and leaves register `$sp` pointing to the most recently pushed. The *callee* then makes a new *stack frame* by pushing the old value of `$fp` (and the return address—`pc` following caller) on the stack and then setting `$fp` to `$sp` to form the new stack frame.

- function return is largely the opposite of function call; on the MIPS place the result in `$v0` then de-allocate the locals by copying `$fp` into `$sp`, finally pop the caller's FP into `$fp` and pop the return address and branch (`jr` on MIPS) to it. On the MIPS there is a *caller-removes-arguments* convention (unlike the JVM) where the caller is responsible for popping arguments immediately after return.

1.10 Typical JVM instructions

The full JVM instruction set is quite daunting at first. However, because of our choice of a simple programming language, we need very few JVM instructions to express all its concepts. These are:

`iconst` $\langle n \rangle$ push integer n onto the stack. Because the JVM was designed to be interpreted, there are special opcodes `iconst_0`, `iconst_1` and variants taking 8-bit, 16-bit and 32-bit operands (widened to 32 bits). These make only efficiency differences and we will treat them as a single instruction.

`iload` $\langle k \rangle$ push the kth local variable onto the stack. This is also used to push arguments—for a n argument function when $0 \leq k < n-1$ then an argument is pushed, for $n > k$ a local.

`istore` $\langle k \rangle$ pop the stack into the kth local variable.

`getstatic` ⟨**class:field**⟩ push a static field (logically a global variable) onto the stack. One might wonder why it is not `igetstatic`; the answer is that the type 'int' is encoded in the ⟨class:field⟩ descriptor.

`putstatic` ⟨**class:field**⟩ pop the stack into a static field (logically a global variable).

`iadd`, `isub`, `ineg` etc. arithmetic on top of stack.

`invokestatic` call a function.

`ireturn` return (from a function) with value at top of stack

`if_icmpeq` ℓ**, also** `if_icmpgt`**, etc.** pop two stack items, compare them and perform a conditional branch on the result.

`goto` unconditional branch.

`label` ℓ not an instruction: just declares a label. In assembly code form (human readable) labels ℓ are strings, but in the binary form conditional branches have numeric offsets and `label` pseudo-instructions are not represented.

One of the reasons for selecting JVM code as intermediate code in our simple compiler is the simple way the above subset may be translated to MIPS code. The Microsoft language C# has a very close resemblance to Java and their `.NET` (or CLR) virtual machine code a similar relationship to JVM code. Their divergence owes more to commercial reasons than to technological ones (one person's "minimal change to keep this side of IPR law" is another person's "doing it properly after failing to agree commercial terms"!)

1.11 Variables, Stacks and Stack Frames

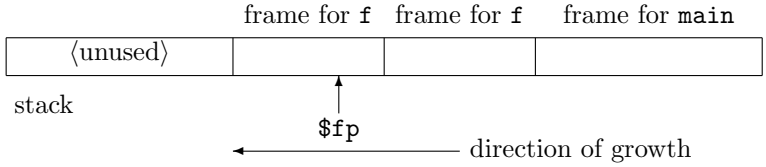
For simple programming languages (and hence the first part of this course), there are only two forms of variables—local variables defined within a function, and global (or top-level) variables which are accessible from multiple functions.

Global variables will be allocated to a fixed location in memory and typically accessed by 'absolute addressing'. Local variables are more tricky: when a function calls itself recursively a local variable, v, will need to be stored in a different memory location in the *callee* from where it is stored in the *caller*. This is resolved by using a *stack*.[4]

A stack is a block of memory in which *stack frames* are allocated. They will be de-allocated in the reverse order.[5] When a function is called, a new stack frame is allocated and a machine register, the *frame pointer* (`$fp` on the MIPS), is set up to point to it. When a function returns, its stack frame is deallocated and `$fp` restored to point to the stack frame of the caller. Just as a global variable is held in a fixed location in memory, a local variable is held at a fixed offset from `$fp`; i.e. local variables are accessed using *indexed addressing*. Thus loading the 5th local variable on the MIPS will map to something like

```
lw      $a0,-20($fp)
```

given integers are 4 bytes wide, and also assuming which direction variables are allocated (I have followed the MIPS convention that local variables live *below* `$fp`). Suppose `main()` calls `f()` which then calls `f()` recursively, then during the recursive call to `f()` the stack will look like:[6]



Thus, the local variables of `f()` will be allocated twice (once for each active call), those of `main()` once, and those of all other functions zero times (as they are not currently being invoked). Note that from this diagram, it is apparent that the only variables which can be accessed are global variables, and those in the currently active function—doing more than this is an issue for Part C of these notes.

First we must return to more pressing issues: we have said nothing yet about *how* stack frames are allocated and de-allocated (particularly mysterious above is how `$fp` is restored on return) and we have made no mention of passing parameters between procedures. If we adopt the conventional FP/SP model, then both of these problems can be solved together. In the FP/SP model, an additional machine register (called `$sp` on the MIPS) is reserved for stack management. Confusingly at first, this is the called *stack pointer*.[7] The stack pointer always points to the lowest-allocated location in the entire stack (which is also the lowest-allocated location in the most recently allocated stack frame).

[4] Note that the word 'stack' is used with subtly different meanings—all have the notion of LIFO (last-in first-out), but sometimes as here (and Operating Systems) it abbreviates 'stack segment' and means a block of memory in which stack frames are allocated and sometimes (JVM) it abbreviates 'operand stack' which is used to evaluate expressions within a stack frame.

[5] Unlike the heap, introduced later, where there is no expectation at compile time of when the store will become free.

[6] This is a *downward-growing stack* in which the stack grows from higher addresses to lower addresses.

[7] Arguably, 'stack fringe pointer' might be a better name but sadly both 'fringe' and 'frame' begin with 'f'!

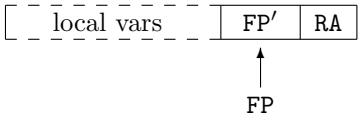
Hence, on a procedure call, allocating a new stack frame is simply a matter of copying `$sp` into `$fp`. But, we need to be able to return from the procedure too. Hence, before we corrupt `$fp`, we first need to save the old value of `$fp` and also the return address—`$pc` value immediately after the call site (on the MIPS this is stored in `$31`, also known as `$ra`, by the `jal` instruction). Together, this pair (old FP, RA) constitute the *linkage information* which now forms part of a stack frame. So, on entry to a procedure, we expect to see instructions such as

```
foo:    sw    $ra,-4(sp)    ; save $ra in new stack location
        sw    $fp,-8(sp)    ; save $fp in new stack location
        addi  $sp,$sp,-8    ; make space for what we stored above
        addi  $fp,$sp,0     ; $fp points to this new frame
```

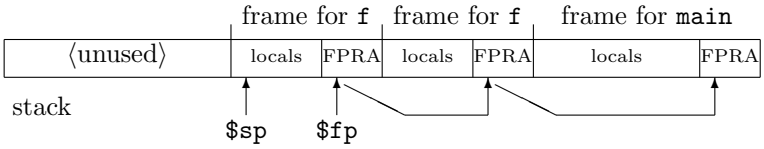
To return from a procedure we expect to see code to invert this:

```
fooxit: addi  $sp,$fp,8     ; restore $sp at time of call
        lw    $ra,-4(sp)    ; load return address
        lw    $fp,-8(sp)    ; restore $fp to be caller's stack frame
        jr    $ra           ; branch back to caller
```

There are many ways to achieve a similar effect: I have chosen one which draws nicely pictorially (`$fp` points *to* linkage information).[8] Local variables are allocated by decrementing `$sp`, and deallocated by incrementing `$sp`. A frame now looks like:



A stack now looks like:



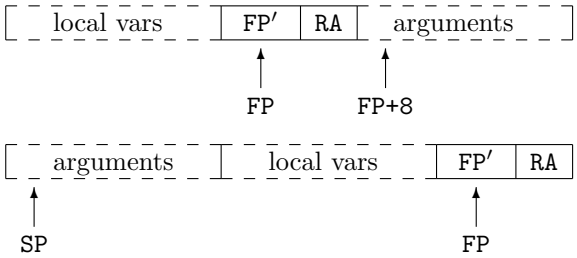
Finally, we need to deal with parameter passing. Because passing parameters in registers is much faster than passing them in memory, the MIPS procedure calling standard mandates that first first four arguments to a procedure are passing in registers `$a0`–`$a3`. For simplicity in this course (and also compatibility with JVM and traditional x86 calling standards) we ignore this and pass all parameters on the stack. The mechanism works like this: to call a procedure:

- the caller and callee agree that the parameters are left in memory cells at `$sp`, `$sp+4`, etc. This is achieved by:
- the caller evaluates each argument in turn,[9] pushing it onto `$sp`—i.e. decrementing `$sp` and storing the argument value at the newly allocated cell.
- the callee stores the linkage information contiguous with the received parameters, and so they can be addressed as `$fp`+8, `$fp`+12, etc. (assuming 2-word linkage information pointed at by `$fp`).

[8] In practice, debuggers will expect to see stack frames laid out in a particular manner (and the defined layout for the MIPS has a few quaintnesses), but to achieve maximum compatibility it is necessary to follow the tool-chain's ABI (application binary interface), part of the API (application programming interface).

[9] In Java this is done left-to-right; in C (and indeed for the MIPS calling standard) the convention (which will ignore in this course, but which is useful if you have real MIPS code to read) is that the arguments are pushed right-to-left which means, with a downward-growing stack, that they appear first-argument in lowest memory.

In a sense the caller allocates arguments to a call in memory in *its* stack frame but the callee regards this memory as being part of its stack frame. So here are the two views (both valid!):

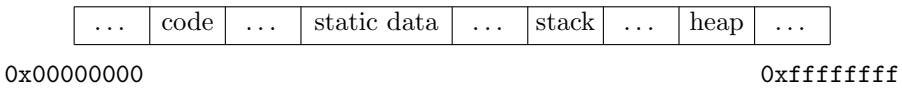


With this setup, local variables and (received) parameters are both addressed as local variables—parameters as positive offsets from `$fp`, and locally declared variables as negative offsets. Argument lists being formed are evaluated and pushed on `$sp`.

There remains one significant issue: a dichotomy as what to do on return from a procedure—is the callee or caller responsible for removing the arguments from the stack? For MIPS, the caller is responsible, on the JVM the callee.

1.12 Overall address space use

It is useful to remind ourselves that programs typically occupy various locations in a modern $0..2^{32}-1$ address space map:



The items listed above are often called *segments*: thus the *code segment* or the *stack segment*. We will only discuss the *heap segment* in Part C of these notes.

1.13 Stack frames in Java

The use of 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 code generated for the function `f` might be:

```
f:
  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.

I have omitted some subtleties here—such fine detail represents 'accidental' implementation artifacts and is neither examinable nor helpful to understanding (unless you actually need to build a JVM implementation following Sun's standard). These include:

- There is no stack management code shown at entry to `f`; this is stored as meta-data in the JVM file, in particularly there are two magic numbers associated with `f`: these are (n_p, n_v). The meta-data n_p is the number of parameters to f (this is used by `ireturn` to remove parameters and at entry to setup `fp`. The meta-data n_v is the number of local variables of f and is used on entry to decrement SP to make space for local variables (on JVM these are all allocated on entry, even if this "wastes space" for local variable only in-scope for part of the function body). Note also that knowing n_v at entry makes for simple convenient checking the stack has not overflowed the space allocated.

- Local variable access and parameter access both use `iload` with operand $(0..n_p + n_v - 1)$; when its operand is $(0..n_p - 1)$ it references parameters and when it is $(n_p..n_p + n_v - 1)$ it references locals—hence in the interpreter `fp` points to an offset from where it would be placed on the MIPS.

- As a consequence, the JVM linkage information cannot obviously be placed between parameters and locals (as done on most systems including MIPS). In an interpreter this can easily be resolved by placing linkage information on a separate (parallel) stack held in the interpreter; on a JIT compiler then the k in "`iload` $\langle k \rangle$" can be adjusted (according to whether or not $k < n_p$) to leave a gap for linkage information.

1.14 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
```

Using `.net` languages on linux, the `mondis` tool can be applied to an executable bytecode file to generate a disassembly of the contents.

Part A: A simple language and interpreters for it

2 A simple language and its execution structures

This course attempts to be language neutral (both source language and implementation language). As said before, a compiler can be *written in* a language of your choice, e.g. Java, C or ML. However, as we will see in Part C of the course, certain language features pose significant implementation problems, or at least require significant implementation effort in representing these features in terms of the simpler features we use here (examples are function definitions nested within other functions, non-static methods—dealing with the `this` pointer, exceptions and the like).

Accordingly for the first part of the course we consider a source language with

- only 32-bit integer variables (declared with `int`), constants and operators;
- no nested function definitions, but recursion is allowed.

Although we do not introduce the formal notation used below until later in the course, our source language has: *expressions*, *declarations* and *commands* of the following form.

```
<expr> ::= <number>
        |  <var>
        |  <expr> <binop> <expr>     ;; e.g.  + - * / & | ^ &&
        |  <monop> <expr>            ;; unary operators: - ~ !
        |  <fnname>(<expr>*)
        |  <expr> ? <expr> : <expr>
<cmd>  ::=  <var> = <expr>;
        |  if (<expr>) <cmd> else <cmd>
        |  while (<expr>) <cmd>
        |  return <expr>;
        |  { <decl>* <cmd>* }
<decl> ::= int <var> = <expr>;
        |  int <fnname>(int <var> ... int <var>) <cmd>
<program> ::= <decl>*
```

We will assume many things normally specified in detail in a language manual, e.g. that `<var>` and `<fnname>` are distinct, that the `<cmd>` of a function body must be a (`{}`-enclosed) 'block' and that `<decl>`s appearing within a `<cmd>` may not define functions. (When you come to re-read these notes, you'll realise that this specifies the *abstract syntax* of our language, and the *concrete syntax* can be used to enforce many of these requirements as well as specifying operator precedence.)

The language given above is already a subset of C. Moreover, it is also a subset of ML modulo spelling.[10] It can also be considered as a subset of Java (wrap the program within a single class with all functions declared `static`). Note that static class members are the nearest thing Java has to a 'global' or 'top-level' variable—one accessible from all functions and typically created before any function is called.

Sometimes, even such a modest language is rather big for convenient discussion in lectures. It can then help to look at a functional 'subset': we skip the need to consider `<cmd>` and `<decl>` by requiring the `<cmd>` forming a function body to be of the form { `return` *e*`;` }. Because this makes the language rather too small (we have lost the ability to define local variables) it is then useful to include an 'let' form declaring a variable local to an `<expr>` as in ML:

[10] Care needs to be taken with variables; a `<var>` declaration needs to use ML's '*ref*', the `<var>` at the left of an assignment is unchanged and all `<var>`s in `<expr>`s need an explicit '!' to dereference.

```
<expr> ::= <number>
         |  ...
         |  let <var> = <expr> in <expr>
```

For the moment, we're going to assume that we already have a compiler and to consider how the various intermediate outputs from stages in the compiler may be evaluated. This is called *interpretation*. Direct interpretation by hardware is usually called *execution*. But of course, it is possible that our compiler has generated code for an obsolete machine, and we wish to interpret that on our shiny new machine. So, let us consider interpreting a program in:

character-stream form: while early Basic interpreters would have happily re-lexed and re-parsed a statement in Basic whenever it was encountered, the runtime overhead of doing so (even for our minimal language) makes this no longer sensible;

token-stream form: again this is very slow, memory is now so abundant and parsing so fast that it can be done when a program is read; historically BBC Basic stored programs in tokenised form, rather than storing the syntax tree (this was for space reasons)

syntax-tree form: this is a natural and simple form to interpret (the next section gives an example for our language). It is noteworthy (linking ideas between courses) to note that operational semantics are merely a form of interpreter. Syntax tree interpreters are commonly used for PHP or Python.

intermediate-code form the suitability of this for interpretation depends on the choice of intermediate language; in this course we have chosen JVM as the intermediate code—and historically JVM code was downloaded and interpreted. See subsequent sections for how to write a JVM interpreter.

target-code form if the target code is identical to our hardware then (in principle) we just load it and branch to it! Otherwise we can write an interpreter (normally interpreters for another physical machine are called *instruction set simulators* or *emulators*) in the same manner as we might write a JVM interpreter.

2.1 A syntax-tree interpreter

While writing a full interpreter for these notes is rather excessive, let us focus on the core aspects. We will have type definitions in the implementation language for the constructs in the tree. Expressing these in ML, then for our language[11] this might be:

```
datatype Expr = Num of int
              | Var of string
              | Add of Expr * Expr
              | Times of Expr * Expr
              | Apply of string * (Expr list)
              | Cond of Expr * Expr * Expr
              | Let of string * Expr * Expr;
datatype Cmd = ...
datatype Decl =...
```

(a fuller set of type definitions are given in Section 6.2). To evaluate an expression we need to be able to get the values of variables it uses (its *environment*). We will simply use a list of (name,value) pairs. Because our language only has integer values, it suffices to use the ML type `env` with interpreter function `lookup`:

[11] I've cheated a little by introducing `let`, but at this point I do not wish to spell out an interpreter for `Cmd` and `Decl` so `let` provides an easy way of showing how local variables can be dealt with.

```
type env = (string * int) list
fun lookup(s:string, []) = raise UseOfUndeclaredVar
  | lookup(s, (t,v)::rest) = if s=t then v else lookup(s,rest);
```

Now the interpreter for `Expr` (interpreters for expressions are traditionally called `eval` even if `Interpret_Expr` would be more appropriate for large-scale software engineering) is:

```
(* eval : Expr * env -> int *)
fun eval(Num(n), r) = n
  | eval(Var(s), r) = lookup(s,r)
  | eval(Add(e,e'), r) = eval(e,r) + eval(e',r)
  | eval(Times(e,e'), r) = eval(e,r) * eval(e',r)
  | eval(Cond(e,e',e''), r) = if eval(e,r)=0 then eval(e'',r)
                                             else eval(e',r)
  | eval(Let(s,e,e'), r) = let val v = eval(e,r) in
                               eval(e', (s,v)::r)
                           end
```

You'll see I have skipped the `apply` case. This is partly because function call with parameters is one of the more complicated operations in a compiler or interpreter, and partly because (as we'll see later) there are subtleties with nested functions. In our mini-language, we only have two forms of variables: global (defined at top level), and local (and these are local to the procedure being executed).[12] Parameters and local variables treated equivalently—a good understanding for execution models (and indeed for many languages) is that a parameter is just a local variable initialised by the caller, rather than explicitly within a function body. For understanding function calls in such simple language (and indeed for understanding local versus global variables), it is convenient to think of the interpreter having *two* environments—one (`rg`) holding the name and values of global variables, the other (`rl`) of local variables. Then the code for `apply` might be (assuming `zip` takes a pair of lists and forms a list of pairs—here a new environment):

```
(* eval : Expr * env * env -> int *)
fun eval(Num(n), rg,rl) = n
  | eval(Var(s), rg,rl) = if member(s,rl) then lookup(s,rl)
                                          else lookup(s,rg)
  | ...
  | eval(Apply(s,el), rg,rl) =
      let val vl = <evaluate all members of el>  (* e.g. using 'map' *)
          val (params,body) = lookupfun(s)
          val rlnew = zip(params,vl)             (* new local env    *)
      in eval(body, rg,rlnew)
      end
```

Again, I have cheated—I've assumed the function body is just an `Expr` whereas for my mini-language it should be a `Cmd`—and hence the call to `eval` should really be a call to `Interpret_Cmd`. However, I did warn in the introduction that for simplicity I will sometimes only consider the functional subset of our mini-language (where the `Cmd` in a function is of the form { `return` e; } and `lookupfun()` now merely returns e and the function's parameter list. Extending to `Interpret_Cmd` and `Interpret_Decl` opens additional cans of worms, in that `Interpret_Cmd` needs to be able to assign to variables, so (at least in ML) environments now need to have modifiable values:

```
type env = (string * int ref) list
```

These notes have attempted to keep the structure clear; see the course web page for a full interpreter.

[12] Subtleties later include access to local variables other than those in the procedure being executed, and access to instance variables in object-oriented languages

At this point it is appropriate to compare (non-examinably) this idea of a "syntax tree interpreter" for a given language with that of an "operational semantics" for the same language (as per the Part IB course of the same name).
At some level an interpreter *is* exactly an operational semantics (and vice versa). Both are written in a formal language (a program in one case and mathematical rules in the other) and precisely specify the meaning of a program (beware: care needs to be taken here if the source language contains non-deterministic constructs, such as race conditions, which an interpreter might resolve by always taking one possible choice of the many possible allowed by the semantics).
The essential difference in usage is that: an interpreter is written for program *execution* (and typically for efficient execution, but sometimes also for semantic explanation), while an operational semantics aims at *simplicity in explanation* and *ability to reason* about programs without necessarily being executable. This difference often leads to divergence in style; for example the simplest operational semantics for the lambda-calculus uses substitution of one term for a variable within another. However, our `eval` function uses an environment which effectively eliminates the need to do substitution on syntax trees by deferring it to variable access where it is a trivial `lookup` operation. Substitution is often relatively expensive and creates new syntax trees at run-time (hence also pretty incompatible with compilation which relies on having a fixed program). On the other hand the *correctness* of using an environment rather than substitution can best be established by exhibiting two operational semantics—one close to our interpreter (a 'big-step' operational semantics in addition to using an environment) and one using substitution—and then to prove them equivalent using mathematical techniques.

2.2 A 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.

First we define enumeration constants, one for each virtual machine operation: for example (using the barbarous Java syntax[13] for this)

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

The structure of a JVM interpreter is of the form (note that here we use a downwards growing stack):

```
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;
```

[13] Java post-version 1.5 now supports type-safe `enum` types.

```
case OP_iconst_W:   T = imem[PC++]; dmem[--SP] = T<<8 | imem[PC++]; break;

/* Note use of FP-k below because we use a downwards growing stack */
/* see also below */
case OP_iload_0:    dmem[--SP] = dmem[FP]; 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+1] = dmem[SP+1]+dmem[SP]; SP++; break;

case OP_istore_0:   dmem[FP] = dmem[SP++]; break;
case OP_istore_1:   dmem[FP-1] = dmem[SP++]; break;
case OP_istore_B:   dmem[FP-imem[PC++]] = dmem[SP++]; break;

case OP_goto_B:     PC += imem[PC++]; break;
/* etc etc 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), a JVM compiler will prefer to use the 1-byte form `iconst_0`.

Java bytecode can be compiled to real machine code at execution time by a JIT-ing virtual machine. This might compile the whole program to native machine code before commencing execution, but such an approach would give a start-up delay. Instead, a lightweight profiling system can detect frequently-executed subroutines or basic blocks and selectively compile them. On a modern multicore machine, the compilation need not slow down the interpreting stage since it is done by another core.

2.3 Interpreting 'real' machine code

Sometimes we have a compiler for an obsolete machine (or even just an executable file from an architecture other than that of our current machine). This can be executed using an instruction set simulator that emulates a real architecture.

Part B: A Simple Compiler

3 Lexical analysis

Lexical analysis, also known as *lexing* or *tokenisation* is an important part of a simple compiler since it can account for more than 50% of the compile time. While this is almost irrelevant for a *compiler* on a modern machine, the painful wait for many a program to read XML-formatted input data shows that lessons are not always learnt! The reason lexing can be slow is because:

1. character handling tends to be expensive, and
2. there are a large number of characters in a program compared with the number of lexical tokens.

Nowadays it is pretty universally the case that programming language tokens are defined as regular expressions on the input character set. Lexical tokens are amenable to definition with regular expressions since they have no long-range or nested structure. This compares with general syntax analysis that requires parenthesis matching and so on.

3.1 Regular expressions

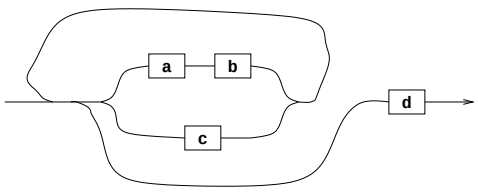
When all the lexical tokens of a language can be described by *regular expressions* it follows that the recognition can be performed by a *finite state algorithm*.

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.
```

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 an automaton is drawn diagrammatically we often refer to it as a *transition diagram*. Constructing the finite state automaton from a regular expression can be then seen as starting with a single transition labelled (rather illegally for our definition of finite state automaton) with the regular expression; and then repeatedly applying the re-write rules in Fig. 1 to the transition diagram (this gives *a* FSA, not necessarily the minimal one!): 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:

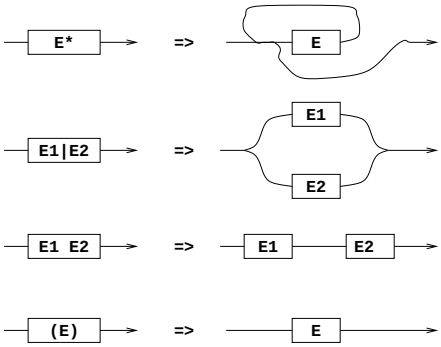
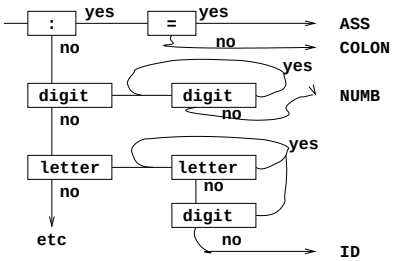


Figure 1: Re-write rules converting a regular expression to an automaton

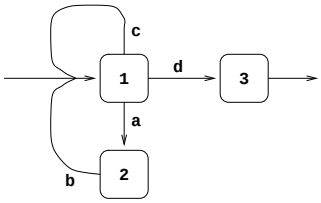
1. Follow any path from the starting point to any accessible box.
2. Output the character in the box.
3. Follow any path from that box to another box (possibly the same) and continue from step (2). The process stops when an exit point is reached (exit points correspond to accepting states).

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.

3.2 Example: floating point tokens

We will demonstrate the method by expressing one (slightly unusual) possible syntax of floating point numbers as a regular expression. Rather than use a single regular expression, we will allow

(*non-recursively*!) named regular expressions to be defined and used (this is harmless as they can be thought of as macros to be expanded away, but it does help human readability). First, some named regular expressions for basic constructs:

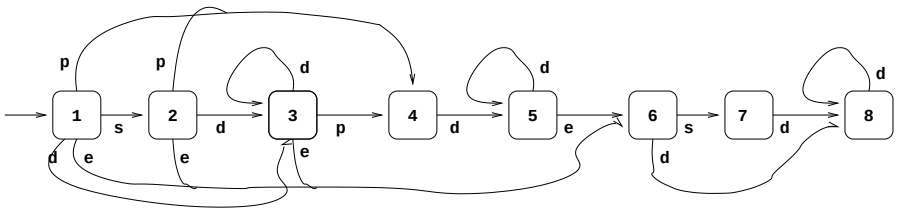
s	=	$+ \mid -$	sign
e	=	`E`	exponent symbol
p	=	.	decimal point
d	=	$0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$	digit

I've used lower case here not for any strong formal reason, but because s, e, p, d all represent exactly one character and hence when hand-implementing a lexer it is often convenient to see them as a character class (cf. `isdigit()`). So, now let's define a floating point number F step by step:

J	=	$d\,d^*$	unsigned integer
I	=	$s\,J \mid J$	signed integer
H	=	$J \mid p\,J \mid J\,p\,J$	digits maybe with '.'
G	=	$H \mid e\,I \mid H\,e\,I$	H maybe with exponent
F	=	$G \mid s\,G$	G optionally signed

Note that some of the complexity is due to expressing things precisely, e.g. H allows three cases: a digit string, a digit string preceded a point, or a point with digits either side, but disallows things like "`3.`".[14]

With some thought we can represent regular expression F as the following deterministic finite-state automaton (this can be automated but requires significant effort to get this minimal FSA):



with states `S3`, `S5` and `S8` being accepting states. Formally, the need to recognise the *longest* string matching the input (e.g. '3.1416' rather than '3.1' leaving '416' unread; since 3.1416 and 3.1 both result in state 8) means that we only accept when we find a character other than one which can form part of the number. 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 might also specify the address of some code to deal with the transition[15] 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 some effort to keep the size of the matrix under control (note for example that treating characters

[14] There are other oddities in the syntax given, such as an integer without '.' or 'E' being allowed, and a string like '`E42`' being allowed. *Exercise 1:* give a better syntax.

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

$\{+, -, E, \cdot, 0, \ldots, 9\}$ as separate entries rather than character groups $\{s, e, p, d\}$ would have made the table several times larger.

As an aside, it is worth noting that for human-engineering reasons we rely on having names for regular expressions: while theoretically these can just be macro-expanded to give a large regular expression (you should note that none of my names, like F, I or d were recursively used), in practice this regular expression is exponentially larger than using named regular expressions and a good deal harder for humans to understand.

Later we will look at an automated tool (lex) for producing such tables from regular expressions.

4 Syntax analysis

Programming languages have evolved by-and-large to have syntax describable with a *context-free grammar* (introduced in the Part Ia course Regular Languages and Finite Automata).

A context-free grammar is a four-tuple (N, T, S, R) where T and N are disjoint sets of respectively *terminal* and *non-terminal* symbols[16], $S \in N$ is the *start* (or *sentence*) symbol, and R is a finite set of *productions*. The unqualified word symbol is used to mean any member of $N \cup T$. In a context-free grammar the productions are of the form

$$U \longrightarrow A_1\ A_2 \cdots\ A_k$$

where the U is a non-terminal and $A_1\ A_2 \cdots\ A_k$ ($k \geq 0$) are (terminal or non-terminal) symbols. We use letters like S, U, V, X, Y to range over non-terminal symbols and letters like A, B, C to range over (terminal and non-terminal) symbols.

A grammar *generates* a set of symbol strings called *sentential forms*:

- S is a sentential form;
- if $B_1\ \cdots\ B_m\ U\ C_1\ \cdots\ C_n$ is a sentential form then so is $B_1\ \cdots\ B_m\ A_1\ \cdots\ A_k\ C_1\ \cdots\ C_n$.

A *sentential form* is a *sentence* if it contains no non-terminal. The sequence of re-writes using production rules to produce a sentence is called its *derivation*. The *language* generated by a grammar is the set of its sentences.

Informally a grammar has additional symbols (non-terminals) which are not part of the language we wish to describe (whose phrases only contain terminals). The 'rule' for determining 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*; the language defined by the grammar is merely the set of strings containing no non-terminal symbols. For example suppose we have a grammar for Java expressions, with start symbol being the non-terminal `<expr>`, then the generated language would include `1, 2, 1+2, 1+1+1, (1+1)+1` but not `12+`, nor `<expr>+3` (the latter is only a sentential form, not a sentence).

In simple examples of grammars, a convention of "upper-case means non-terminal and everything else is terminal" suffices. For use in programming languages (which generally allow most ASCII symbols as tokens) it is important to have a clearer separation between non-terminals and terminals. Two important (and opposite) variations of this are:

- Backus-Naur Form (BNF): non-terminals are written between angle brackets (e.g. `<expr>`) while everything else stands for a terminal. (Informally: "non-terminals are quoted".) The text '`::=`' is used instead of '$\longrightarrow$'
- Parser generators like `yacc` assume the convention were all terminal symbols are written within quotes (") or otherwise named, and alphanumeric sequences are used for non-terminals. (Informally: "terminals are quoted".)

[16] In the context of this course terminal symbols are simply tokens resulting from lexing.

These differences are inessential to understanding, but vital to writing machine-readable grammars. Because context-free grammars typically have several productions having the same terminal on the left-hand side, the notation

$$U \longrightarrow A_1\ A_2 \cdots\ A_k \mid \cdots \mid B_1\ B_2 \cdots\ B_\ell$$

is used to abbreviate

$$\begin{array}{rcl} U & \longrightarrow & A_1\ A_2 \cdots\ A_k \\ & \cdots & \\ U & \longrightarrow & B_1\ B_2 \cdots\ B_\ell. \end{array}$$

But beware when counting: there are still multiple productions for U, not one.

Practical tools which read grammars often include additional meta-characters to represent shorthands in specifying grammars (such as the '*' used for repetition in in Section 2). For further information see BNF and EBNF ('extended') e.g. via
`http://en.wikipedia.org/wiki/Backus-Naur_form`.

While the definition of grammars focusses on *generating* sentences, a far more important use in compiler technology is that of *recognising* whether a given symbol string is a sentence of a given grammar. More importantly still, we wish to know what derivation (sequence of re-writes using production-rules) resulted in the string. It turns out that this corresponds to a parse tree of the string. This process of taking a symbol string and returning a parse term corresponding to it if it is a sentence of a given grammar and otherwise rejecting it is unsurprisingly called *syntax analysis* or *parsing*.

Before explaining how to make parsers from a grammar we first need to understand them a little more.

4.1 Grammar engineering

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 grammar is either ambiguous or it is not. However, it turns out that it is undecidable (not possible to write a program to determine) whether a given context-free grammar is ambiguous or not. It is surprisingly difficult for humans to tell whether a grammar is ambiguous. One example of this is that the productions in (c) above appeared in the original Algol 60 published specification.

It is very easy to write an ambiguous grammar. A good example is one with two production rules:

$$E \longrightarrow \texttt{1} \mid E\texttt{-}E$$

[Question: what are the terminals? Non-terminals? Start symbol?] Now consider the input `1-1-1`. Assuming this to be a sentence generated by the grammar, clearly it must match E`-`E as it does not match `1`. But which '`-`' should match? The answer is *either*—one derivation is

$$E \ \Rightarrow\ E\texttt{-}E \ \Rightarrow\ \texttt{1-}E \ \Rightarrow\ \texttt{1-}E\texttt{-}E \ \Rightarrow\ \texttt{1-1-}E \ \Rightarrow\ \texttt{1-1-1}$$

and another is

$$E \ \Rightarrow\ E\texttt{-}E \ \Rightarrow\ E\texttt{-1} \ \Rightarrow\ E\texttt{-}E\texttt{-1} \ \Rightarrow\ \texttt{1-}E\texttt{-1} \ \Rightarrow\ \texttt{1-1-1}$$

This may appear academic, but the derivation determines the grouping and the former corresponds to `1-(1-1)` while the latter corresponds to `(1-1)-1`. Note these have different values when interpreted as ordinary mathematics.

To avoid this we re-write the grammar to ensure there is only one derivation[17] for each sentence. Two possibilities are:

$$E \longrightarrow \texttt{1} \mid E\texttt{-1}$$

which yields a *left-associative* '`-`' operator (i.e. `1-1-1` parses as $(1-1)-1$); and

$$E \longrightarrow \texttt{1} \mid \texttt{1-}E$$

which yields a *right-associative* '`-`' operator (i.e. `1-1-1` parses as $1-(1-1)$).

Note in these cases the parentheses are used to show grouping in the parse tree; they are not themselves part of the grammar or the language yet.

Finally, if we wish to allow `1-1`, but disallow `1-1-1` then we might code the grammar as

$$E \longrightarrow \texttt{1} \mid \texttt{1-1}$$

thereby yielding a *non-associative* '`-`' operator (i.e. `1-1-1` is forbidden).

Additionally, grammars can conveniently express operator *precedence*, i.e. whether $2+3*4$ should be treated as $(2+3)*4$ or $2+(3*4)$ (note that this is independent of whether $+$ and $*$ are separately left- right- or non-associative. Assuming start symbol E, then the grammar

$$\begin{array}{lcl} E & \longrightarrow & E+F \mid F \\ F & \longrightarrow & F*P \mid P \\ P & \longrightarrow & 2 \mid 3 \mid 4 \end{array}$$

gives '`*`' higher precedence than '`+`', whereas

$$\begin{array}{lcl} E & \longrightarrow & E*F \mid F \\ F & \longrightarrow & F+P \mid P \\ P & \longrightarrow & 2 \mid 3 \mid 4 \end{array}$$

gives '`+`' higher precedence than '`*`'. *Exercise 2:* are $+$ and $*$ left- or right- associative in these examples? Can you change them individually?

4.2 Forms of parser

There are two main ways to write a parser. One is for a human to read the grammar, understand it and to encode the its requirements as program code. This naturally leads to having one procedure to read each non-terminal. These procedures tend to be mutually recursive, leading to the name *recursive descent parser*. As we will see, often there is a need for some ingenuity in writing code which corresponds to a given grammar; we consider this more in the next section.

The alternative is to avoid writing code by using an existing tool (common tools are: yacc, mlyacc, CUP). These take a grammar in text form as input and output a parser as program code. Typically they operate by encoding the grammar (which has to satisfy restrictions placed on its form by the tool—so again in principle ingenuity is needed to write the grammar in this form) as a table and then appending a fixed interpreter. The interpreter takes an input string together with this table and constructs a parse tree. Section 12.2 exemplifies their use and Section 14 explains how they work.

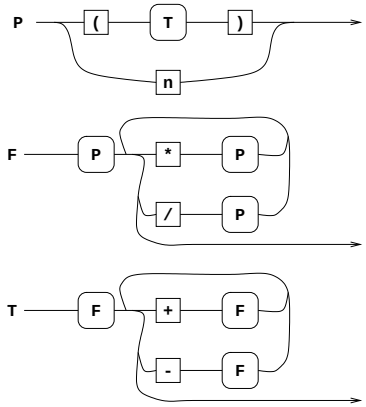
[17] Technically I need to say ignoring permuting independent uses of production rules here.

4.3 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 grammar has been modified to avoid left recursion[18] 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;
```

[18] By effectively 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.

```
    case '-': lex(); RdF(); continue;
    default:  return;
  }
}
```

4.4 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.
          struct { struct E *child; } monad;
            // selected by E_Paren.
          int num;
            // selected by E_Numb.
        } u;
};
```

In Java and C♯, where there is no union or variant record construct, a parent class must be multiply extended for each construct, resulting in a much more verbose defintion.

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 = (struct E*)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 2.

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 2: Recursive descent parser yielding a parse tree

4.5 Automated parser generation

Tools which generate parsers (and lexers) are covered later in the course: Section 12.2 explains their practical use and Section 14 explains how they work and how their tables are constructed.

In general, almost all modern *programming* languages have their syntax structured in a two-level fashion: characters are first lexed into tokens—defined by regular expressions—and then tokens are then parsed into trees—defined by context-free grammars. Sometimes there are warts on this structure, e.g. C's `typedef` can make an identifier behave like a keyword. Sticking rigidly to the two-level pattern is therefore over-constraining, and allowing `decl ::= <id> <id>;` instead of `decl ::= <type> <id>;` could introduce significant parsing difficulties (even for the advanced, mechanically-generated parsers introduced in §12.2). Therefore it is common to cheat, in that output from the syntax analyser that has noted a user-defined type, or similar construct, is fed backwards into the lexical analyser to alter the token returned next time that identifier is encountered. This works if the lexer is a subroutine of the syntax analyser so that is has not already run ahead in the input stream.

4.6 A note on XML

Markup languages (HTML, XML, etc. based on SGML) do not fall quite so simply into this two-level pattern. Firstly, there is no very obvious separation between lexical and syntactic structure apart from the lexical escape `&` and the lexical `<`, `>` tag delimiters. Secondly, the syntactic constraints required by XML (e.g. that a `<foo>` construct cannot be closed by a `</bar>`) are not simply expressible as context-free grammars—there are an unbounded number of bracketing symbols. Thirdly, in addition to the XML syntactic constraints, an input form is typically required to respect semantic constraints, e.g. that a `<zoo>...</zoo>` might have a `<content>elephant</content>` attribute, but only the latter might have a `<height>` attribute. These constraints are typically expressed by *XML schemas* (which generalise the original DTD—Document Type Definition—form in XML 1.0) and are best though of as a form of type checking: a schema processor takes in an XML document and an XML schema and either rejects it or normalises the XML document to fully conform to the scheme.

A more human-engineering point is that, in general, we are happy if a syntactically invalid program is rejected, but less happy when our browser says "refusing to display a web page due to markup bracket mismatch"—so being able to recover from an error is more important.

A consequence of all this (and the desirability of rapid XML reading) is that writing a good XML parser is non-trivial.

5 Type checking

Our simple language from Section 2 requires no type checking at all; we have arranged that all variables and expressions are of type `int` and arranged that variable name `<var>` and function names `<fnname>` are syntactically distinguished—at worst we just need to check the constraint that the number of arguments in a function call match its definition. As such the lex/syn/trans/cg phase structure made no provision for type-checking.

However, in a real compiler, type-checking generally has to take place after syntax analysis—consider the examples on page 7. For a typed intermediate code like JVM code which has separate `fadd` and `iadd` operations, then type information has to be resolved before translation to intermediate code. In Java code like

```
float g(int i, float f) { return (i+1)*(f+2); }
```

the two addition operations must compile into different operations; moreover, the integer result of `i+1` must be converted to a floating value with the `i2f` JVM opcode—the Java code has a implicit conversion which the compiler must make explicit as if the code were:

```
float g(int i, float f) { return ((float)(i+1))*(f+2); }
```

In a production compiler one would therefore expect to see a `typecheck` phase between `syn` and `trans` which takes an (abstract) syntax tree and either rejects it ("ill-typed") or produces an adjusted syntax tree (typically annotated with types and with implicit operations mapped into explicit tree nodes). Some compiler-writers prefer to see type-checking as re-writing the parse tree in-place (e.g. writing to fields left initialised when the tree was created), while others prefer to have separate data-types for "parse tree" and "typed parse tree" (this is largely a matter of taste). Note that, by consideration of the program above, type-checking must include determining which identifier *use* (e.g. `f` in `f+2`) is associated with each identifier *declaration* (e.g. `f` in `(int i, float f)` rather than any other declaration of the name `f`).

Nevertheless, under the minimalist philosophy of this course, these notes will incorporate the type-checking phase (including name resolution) phase as part of the translation phase, which we now turn to.

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 the intermediate code.

We choose to use the JVM instruction set as intermediate code for this course (and its structures have been discussed in Part A of these notes).

6.1 Interface to `trans`

Instructions, such the first few instructions of `f` from Part A

```
static int f(int a, int b) { int y = a+b; ... }
```

are generated by translation phase of the compiler invoking a series of calls such as:

```
gen2(OP_iload, 0);
gen2(OP_iload, 1);
gen1(OP_iadd);
gen2(OP_istore, 2);
```

The enumeration constants (`OP_iload` etc.) 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 Example tree form used in this section

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

```
datatype Expr = Num of int
              | Var of string
              | Neg of Expr         | Not of Expr
              | Add of Expr * Expr  | Sub of Expr * Expr
              | Mul of Expr * Expr  | Div of Expr * Expr
              | Eq of Expr * Expr   | Ne of Expr * Expr   (* etc *)
              | And of Expr * Expr  | Or of Expr * Expr   (* for &&/|| *)
              | Apply of string * (Expr list)
```

```
              | Cond of Expr * Expr * Expr;
datatype Cmd =  Assign of string * Expr
              | If3 of Expr * Cmd * Cmd
              | While of Expr * Cmd
              | Block of Decl list * Cmd list
              | Seq of Cmd * Cmd    (* redundant special case of Block *)
                                    (* (useful for exercises) *)
              | Return of Expr;
datatype Decl = Vardef of string * Expr
              | Fndef of string * (string list) * Cmd;
type Prog = Decl 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`(`$x_1, \ldots, x_k$`)` $\{c\}$.

We will translate programs in this language using routines

`void trexp(Expr e)`	translate an expression
`void trcmd(Cmd c)`	translate a command
`void trdecl(Decl d)`	translate a declaration

which have the side-effect of emitting JVM code which, when executed, causes the expression (or command, or declaration) to be computed.

6.3 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 term is sometimes is used 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 that 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 a simple language lacking heap-allocated variables, but how a translation might work will be covered in a later part of the course (§9.5).

Essentially, the routine `trdecl` will (*i*) save the current state of the symbol table and add the new declared names to the table, (*ii*) translate expressions and commands within this scope and (*iii*) finally restore old state of the symbol table (as any new variables are now going out of scope).

It is convenient to implement a routine `trname` which is used by `trexp` and `trcmd` and which consults the symbol table to determine the access path.[19] Because the access path is identical for loads and stores, it is convenient give *trname* the type

```
void trname(int op, String s)
```

where `op` is a flag indicating *load*, *store*, or *call* (or *loadaddress* if our language has pointers and one wants to take the addresses of variables and so on) because these are the only things one does with a name.

As an example for the above the table might contain

[19] For the purposes of these notes, this is just an offset in stack frame for a local variable, or the address of a `static` variable, but clearly further variations are needed when objects are introduced in part C.

```
"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 $(\texttt{x}, loc, 0)$ when `f`'s parameters (`x`) are scanned, and then by $(\texttt{y}, loc, 1)$ when the local definition of `y` is encountered.

6.4 Translation of expressions

As mentioned above we use `trexp` to translate expressions; its argument is the tree for the expression being translated; it returns no result,[20] instead merely outputting the generated code to a data structure (or file) for later use. The routine `gen1` is used to output a JVM opcode without an operand, and `gen2` to output one with a single operand.

An outline of its definition is as follows:[21]

```
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))
  | trexp(Mul(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 (16 lines on)
                  trname(OP_invokestatic, f)) // Compile call to f
  | trexp(Cond(b,x,y)) =
                let val p = ++label;       // Allocate two labels
                    val q = ++label in
                  trexp(b);                // eval the test
                  gen2(OP_iconst, 0);      // put zero on stack...
                  gen2(OP_if_icmpeq, p);   // ... so branch if b=0 (ie false)
                  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));
```

6.5 Translation of short-circuit and other 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

[20] If type checking is integrated into this phase/pass then `trexp` may return the type of the expression is has just manipulated. If the optimisations mentioned at the end of §7 are implemented it may also return the storage class of the expression (e.g. compile-time constant, in a register, in run-time memory, etc.).

[21] 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`.

```
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. Therefore we *cannot* extend the above code for `trexp` for `And` and `Or` by

```
| trexp(Or(x,y))     = (trexp(x); trexp(y); gen1(<something>))
| trexp(And(x,y))    = (trexp(x); trexp(y); gen1(<something>))
```

Instead we have to treat e`||`e' as e`?1:(`e'`?1:0)` and e`&&`e' as e`?(`e'`?1:0):0`. One lazy way to do this is just to call *trexp* recursively with the equivalent code above (which does not use `And` and `Or`):

```
| trexp(Or(x,y))     = trexp(Cond(x, Num(1), Cond(y,Num(1),Num(0))))
| trexp(And(x,y))    = trexp(Cond(x, Cond(y,Num(1),Num(0)), Num(0)))
```

Translation of relational operators (`Eq`, `Ne`, `Lt`, `Gt`, `Le`, `Ge`) is also slightly tricky. The 'obvious' and otherwise correct code (and this *is* acceptable for exams on this course) is just to do:

```
| trexp(Eq(x,y))     = (trexp(x); trexp(y); gen1(OP_EQ))
```

etc. But the JVM does not have an `eq` operation—instead it has conditional branches `if_icmpeq` and the like. So similar code to that for `Cond` in `trexp` is needed:

```
// note we reverse the sense e.g. Eq -> CmpNe etc for trboolop:
| trexp(Eq(x,y))     = trboolop(OP_if_icmpne, x, y)
| trexp(Ne(x,y))     = trboolop(OP_if_icmpeq, x, y)
| trexp(Le(x,y))     = trboolop(OP_if_icmpgt, x, y)
| trexp(Lt(x,y))     = trboolop(OP_if_icmpge, x, y)
| trexp(Ge(x,y))     = trboolop(OP_if_icmplt, x, y)
| trexp(Gt(x,y))     = trboolop(OP_if_icmple, x, y);

fun trboolop(brop,x,y) =
                let val p = ++label;       // Allocate two labels
                    val q = ++label in
                  trexp(x);                // load operand 1
                  trexp(y);                // load operand 2
                  gen2(brop, p);           // do conditional branch
                  trexp(Num(1));           // code to put true on stack
                  gen2(OP_goto,q);         // jump to common point
                  gen2(OP_Lab,p);
                  trexp(Num(0));           // code to put false on stack
                  gen2(OP_Lab,q)           // common point; result on stack
                end;
```

The interested reader might note that this technique produces dreadful (but correct) code for:

```
if (x>0 && y<9) foo();
```

because it computes booleans corresponding to `x>0` and `y<9`, then computes a boolean representing whether both are true, and finally compares that with zero. To do things better, it would help if the `Cond` case of `trexp` peeked inside its operand to see if it was a case like `And`, `Or`, `Eq`, ... `Ge` and generated special case code if so. This is very easy to implement using the pattern-matching constructs of ML, but cumbersome in other languages. It is left as an exercise to the interested reader (beyond the scope of the taught and examinable) course).

6.6 Translation of declarations and commands

Again this is left as an exercise, but one which you are encouraged to sketch out, as it uses simple variants of ideas occurring in `trexp` and is therefore not necessarily beyond the scope of examinations.

Hint: start with

```
fun trcmd(Assign(s,e))  = (trexp(e); trname(OP_istore,s))
 |  trcmd(Return e)     = (trexp(e); gen1(OP_ireturn))
 |  trcmd(Seq(c,c'))    = (trcmd(c); trcmd(c'))
 |  trcmd(If3(e,c,c'')) = ...
```

6.7 Assembly:converting labels to addresses

In the above explanation, given a Java procedure

```
static int f(int x, int y) { return x<y ? 1:0; }
```

I have happily generated 'JVM' code like

```
   iload 0
   iload 1
   if_icmpge label6
   iconst 1
   goto label7
label6:                  //  this was written "Lab 6" earlier
   iconst 0
label7:                  //  this was written "Lab 7" earlier
   ireturn
```

but when looking at the JVM code using `javap -c` I get

```
   0:   iload_0
   1:   iload_1
   2:   if_icmpge 9
   5:   iconst_1
   6:   goto      10
   9:   iconst_0
   10:  ireturn
```

So, my explanation was wrong? No, but I did elide a few details. The actual JVM code has *numeric* addresses for instructions (printed to the left by `javap -c`) and `if_icmpge` and `goto` use the *address* of destination instructions as their operands instead of a label. A separate pass of the compiler determines the size of each JVM instruction—to calculate the address of each instruction (relative to the start of the procedure) which then determines the numeric address for each of the labels. Each use of a label in a `if_icmpge` and `goto` instruction can now be substituted by a numeric offset and the labels deleted.

This process (of converting symbolic JVM code to binary JVM code) is called `assembly` and the program which does it an *assembler*. A corresponding pass (either as part of the compiler or as a stand-along assembler program) is used to convert data structures (often strings) representing target instructions into their binary form in an object file.

Assembly needs to be performed at the end of compilation—if we are intending to use the JVM code to generate further code (as we do in subsequent sections below) then we will want to keep the symbolic 'label_*nnn*' form. Indeed, if we download a Java `.class` file which contains binary JVM code with the intention of JIT'ing it (compiling it to native binary code), the first thing we need to do is to identify all binary branch offsets and turn them to symbolic labels.

6.8 Type checking revisited

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 ML, type checking is performed by a large-scale unification algorithm that infers a most general type for the subroutine at large. In most other languages, types are inferred locally and the type of an expression is determined compositionally from the types of its sub-expressions. Current thinking is that it is most convenient if typing is fully-automatic at a fine grain, such as within expressions or sequences of statements/commands, but explict at large module boundaries.

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 might 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 an `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

Note: The part II course, 'Optimising Compilers', will cover code generation in an alternative manner dispensing with the stack-based intermediate code. However, let us note that translating JVM code to native code is exactly what a browser using JIT compilation technology does when an applet `.class` *file is downloaded.*

In fitting with our cheap-and-cheerful approach to compilation let us for now merely observe that each intermediate instruction listed above can be mapped into a small number of MIPS, ARM or x86 instructions, essentially treating JVM instructions as a `macro` for a sequence of x86 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 the following x86[22] code (note that x86 conventionally uses `%ebp` for FP):

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

[22] It is part of the general philosophy of this course that the exact language and exact target machine do not matter for presentation; a good exercise for the reader is to rephrase this in MIPS machine code—this is not a difficult task, requiring little more than replacing `%eax` and `%ebx` with `$a0` and `$a1` and adjusting opcode names.

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; but in general we would not 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:

```
                          ;            stackpend=[],     regmem=[]
    movl  %eax,-4-16(%ebp) ; iload 4    stackpend=[%eax], regmem=[%eax=local4]
                          ; iconst 3   stackpend=[%eax,3], regmem=[%eax=local4]
    cmpl  %eax,#3         ; if_icmpgt  stackpend=[],     regmem=[%eax=local4]
    bgt   L36             ; if_icmpgt  stackpend=[],     regmem=[%eax=local4]
                          ; iload 4    stackpend=[%eax], regmem=[%eax=local4]
    negl  %eax            ; ineg       stackpend=[%eax], regmem=[]
    b     L37             ; goto       stackpend=[%eax], regmem=[]
L36:                      ; (label)    stackpend=[],     regmem=[%eax=local4]
                          ; iload 4    stackpend=[%eax], regmem=[%eax=local4]
L37:                      ; (label)    stackpend=[%eax], regmem=[]
    movl  -4-28(%ebp),%eax ; istore 7  stackpend=[],     regmem=[%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.[23]

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. There are very many instruction sequences that achieve the same effect. Choosing the optimal (provably shortest) sequence for any basic block would involve an excessively-expensive search algorithm that is only used for specialist applications. 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 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.

[23] 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 converted 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 Windows). These object files are linked (together with other object files from program libraries) to produce an *executable file* (`.EXE` on Windows) 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* (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 3. This is for the format ELF (we only summarise the essential features of ELF).

8.1 The linker

Using a comprehensive object module format such as ELF, the job of the linker is relatively straight-forward. 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.)

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.[24]) The result is a file which can be immediately executed

[24] On some systems, certain data is read-only, such as when it is placed in ROM or if the high-level language does

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 3: Basic Structure of a common object file in ELF format.

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. A third advantage is that occupancy of physical RAM on a multi-tasking operating system can be minimised, since (the text segment of) shared libraries can be only loaded once and shared over active programs.

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 */
```

not allow it to be changed, such as character strings in some versions C/C++. Read only data can be mapped to its own segment. On Unix systems, static variables that are initialised to zero are also placed in their own segment as a form of file compression.

```
            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—given 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! Linux gives even less helpful messages and the `ldd` command must be used to find the name of the missing library. 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 C: Implementing Language Features

In earlier sections 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 part of the course we will try to crystallise 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.

Although everyone is familiar with ML and Java (and learning C/C++), it helps first to explore the idea of assignment and aliasing in a language-neutral manner.

Given an assignment $e = e'$ we:

1. evaluate e to give an address; [see below re ordering here]
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, both Lvalues and Rvalues require work to be done when they are being evaluated, consider `A[f()] = B[g()];`.

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 ML) and better for maintenance to break down a single complicated expression by using assignments, executed in sequence, to temporary variables.

9.1 Aliasing

We see situations where multiple names refer to the *same* variable or storage location—this is called *aliasing*. In C and C++, there are obvious sources of aliases arising from the support of unrestricted pointers and unions. In a language with call-by-reference, aliasing commonly occurs when the same variable is passed twice into the same function.

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/C++, definition by reference is written:

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

whereas 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;
```

One reason for making such a fuss about this is to be able to discuss the effects of Java (and C♯) inner classes.

```
class A {
  void f(int x) {
    class B {
      int get() { return x; }
      // void inc() { x++; }   // should this be allowed?
    }
    B p = new(B);
    x++;
    B q = new(B);
    if (p.get() != q.get()) println("x != x??");
};
```

The question is whether `x` is copied or a whether a single storage cell is used for it is often best thought of as whether `x`'s Lvalue or Rvalue is used in `class B`. The choice is a matter of language design. The update in place looks, at first sight, more natural, until you consider that the B's are heap-allocated object that we expect to be independent and returnable after the stack frame for `f` has been deallocated.

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

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

One adds a new entry to the environment while the other modifies a cell already present in the environment.

9.2 Lambda calculus

The main concept of the lambda calculus is embodied in the anonymous functions of ML, denoted with `fn <bv> => <body>`.

We revisit this briefly for two reasons: firstly it exemplifies the idea of source-to-source translation whereby a larger language (here ML) can be explained in terms of a subset (here the lambda-calculus subset of ML); The second, related reason, is that it provides the archetypical notion of variable binding—which we need to understand well to write an interpreter for a language like ML.

Let's first recall that the ML `let` and `fun` (non-recursive—see below for how to fix this) forms are expressible just using lambda.

$$\begin{array}{rcl} \texttt{let}\ f\ x = e & \Rightarrow & \texttt{let}\ f = \lambda x.e \\ \texttt{let}\ y = e\ \texttt{in}\ e' & \Rightarrow & (\lambda y.e')\, e \end{array}$$

So, for example,

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

can be simplified to

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

This translation does not encode recursion (ML's `fun f x = e` implicitly expands to `let rec f = λx.e`) and the above simplifying translation does not say how to represent `rec`. For example, without `rec`

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

translates to

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

in which the right-most use of `f` is unbound rather than recursive.

One might think that recursion must inevitably require an additional keyword, but note that it is possible to call a function recursively without defining it recursively:

```
let f(g,n) = ... g(g,n-1) ...   // f does not appear in the body of f
in f(f, 5)
```

Here the call `g(g,n-1)` makes a recursive call of (non-recursive) `f`!
Exercise 3: complete the body of `f` so that the call yields 5! = 120.

By generalising this idea it is possible to represent a recursive definition `let rec f = e` as the non-recursive form `let f = Y (λf.e)` which at least binds all the variables to the right places. The question is whether `Y` needs to be a piece of magic built in to the system or whether it can itself be a term (expression) of the lambda-calculus. For instance, in computation theory, one learns that the primitive recursive functions **need** to be extended with the μ operator to become Turing powerful.

The remarkable answer is that `Y` can be expressed purely in lambda-calculus. It is called the *fixed-point combinator*.

One can write[25]

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

[25] The alternate 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 in the course supplemental material.

(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".

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 9.4. This is also useful since neither form for `Y` (nor indeed the `f(f,5)` trick) passes the ML type system even though they are well formed in most other senses.

9.3 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 of 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?
}
```

For years, the lambda-calculus provided a neat understanding of scoping that 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 qualifiers 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 ISO C++ checks accessibility after determining scoping.)

We will later return to implementation of objects and methods in terms of data and procedures using source-to-source translations.

9.4 Mechanical evaluation of lambda expressions

We will now describe a simple way in which lambda expressions may be evaluated in a computer.[26] The reason for exploring lambda-calculus evaluation here is to better understand references to non-local/non-global variables. It is also useful for exploring untyped languages where we do not know whether an expression returns an integer or a function.

[26] We do not do the evaluation by textual re-writing (as in Part 1B Operational Semantics course) because we later wish 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.

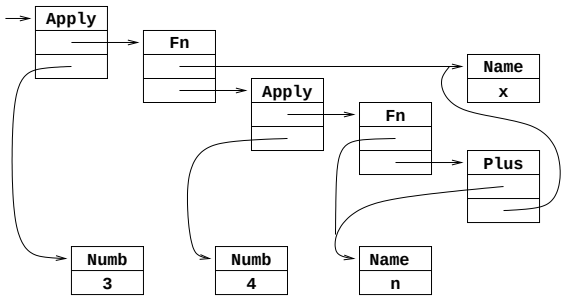
We will represent the expression as a parse tree and evaluate it in an environment that is initially empty. As above there will be tree nodes representing variables, constants, addition, function abstraction and function application. In ML this can be written concisely:

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

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

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

and be represented as follows:



When we evaluate such an `Expr` we expect to get a value which is either an integer or a function (note this is an example of *dynamic typing*—see Section 10.13). In ML we write this concisely 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). In languages without disjoint union types (such as Java) we have again to write it cumbersomely as an abstract parent class with two instantiable extensions.

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[27] 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:

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

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

The immediate action of `eval` depends on the leading operator of the expression it is evaluating. If it is `Name`, the bound variable is looked up in the current environment using the function `lookup`. If it is `Numb`, the value can be obtained directly from the node (and tagged as an `IntVal`). If it is `Plus`, the two operands are evaluated by (recursive) calls of `eval` using the current environment and their values summed (note the slightly tedious code to check both values correspond to numbers else to report an error). The value of a lambda expression (tagged as a `FnVal`) is called a *closure* and consists of three parts: the bound variable, the body and the current environment. These three components are all needed at the time the closure is eventually applied to an argument. To evaluate a function application we first evaluate both operands in the current environment to produce (hopefully) a closure (`FnVal(bv, body, r_fromdef)`) and a suitable argument value (`arg`). Finally, the body is evaluated in an environment composed of the environment held in the closure (`r_fromdef`) augmented by `(bv, arg)`, the bound variable and the argument of the call.

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 closure itself (under name `f`).

Note that, in ML, the assignment to `fv` requires it to actually be a ref cell or for some other mechanism to be really used. A more detailed working evaluator including Y and *let*) can be found in the supplemental material.

9.5 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 are looked up in the environment 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* (or *dynamic scoping*) and was used in old dialects of Lisp. It is still used (unfortunately) in (YYY?). The difference is most easily seen in the following example:

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

Exercise 4: 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.

9.6 A more efficient implementation of the environment

The previous lambda evaluator (we'll use the word interpreter interchangeably) is particularly inefficient in its treatment of names since it searches a potentially long environment chain every time a name is used.

There are two reasons the inefficiency. Firstly, the `lookup` function *searches* for a given name, when for at a given point in the program and for a given name we can see statically how many names have been bound since the given name, and hence how many steps down the list-form environment we need to make before finding the name. Hence, instead of representing names in the form `Name(`s`)` where s is a `string`, we could first translate the program into one in which `Name(`s`)` constructs have been replaced by `NameIndex(`k`)` where k is an `int`; `Lam("x",Name("x"))` becomes `Lam("x",NameIndex(1))` or even `Lam(NameIndex(1))` as the `"x"` is no longer needed for the `Lam` construct either. This transformation—which could be thought of as a compiler phase—logically encodes the binding of variable references to the bindings of those variables ("never put off till run-time what you can do at compile-time") and formally goes under the name "De Bruijn indices"—see `http://en.wikipedia.org/wiki/De_Bruijn_index` for a wider story. Note that replacing variables by indices can only be done with static binding—Lisp-like dynamic binding needs a run-time search.

However, replacing list search by list indexing make no asymptotic difference to cost; stepping n steps down a list takes $O(n)$ time as does finding an item n steps down a list. Alternatively we could use an array instead of lists to represent environments, but then while variable access would be $O(1)$, the cost of environment manipulation would be excessive (adding to the environment would involve copying an array, whereas it only involves a CONS when using lists). So, we need a better representation of environment which reduces the sum of these costs.

Dijkstra observed (in the context of Algol60 implementation) essentially that while environments can be long, they interleave variables bound by lambda (function arguments) and those bound by let (local variables). We can arrange (as in the JVM) that local variable live next to each other as if in an array. Hence he proposed a 2D approach to variable access which goes under the name "Dijkstra display". We see an environment as being an array of arrays: a variable access is replaced by a 2D (i, j) co-ordinate meaning the jth variable in the ith array. Array 0 holds values of variables inside the currently active procedure[28]; array 1 holds values of variable one

[28] You might conveniently think of this as its stack frame and this works for many languages, but environment allocation and deallocation does not follow a stack discipline for languages like ML, so to make the lambda-evaluator work these need to be heap-allocated—see later.

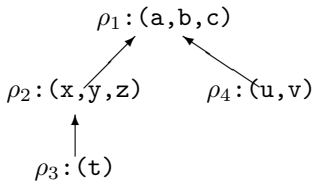
lexical nesting level out and so on. This gives $O(1)$ access time with the cost of a call (establishing a new environment) being that of copying an array of size of the number of outer-nested functions to the one being entered.

Nowadays (see later), rather than using an array of pointers to stack frames, we chain through the stack frames themselves, the so-called *static link method* and use absolute addressing to refer to top-level variables (which would otherwise involve $O(d)$ access time for a procedure at nesting depth d). [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.]

For an example of this 2D addressing, consider variables bound in the following contrived program (we ignore function names `f`, `g`, `h`, `k` for simplicity):

```
let f(a,b,c) =
  ( let g(x,y,z) = (let h(t) = E in ...)
    in g((let k(u,v) = E' in ...), 12, 63)
  )
in f(1,6,3)
```

and suppose the environments (here just variable names, we will not know values until run-time) at the start of the bodies of `f`, `g`, `h`, `k` are respectively ρ_1, ρ_2, ρ_3, ρ_4. The environment structure can be represented as a tree as follows (note that here the tree is in some sense backwards from usual in that each node has a single edge to its parent, rather than each node having an edge to its children):



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

ρ_1	`a:(1,1)`	`b:(1,2)`	`c:(1,3)`	`level 1`
ρ_2	`x:(2,1)`	`y:(2,2)`	`z:(2,3)`	`level 2`
ρ_3	`t:(3,1)`			`level 3`
ρ_4	`u:(2,1)`	`v:(2,2)`		`also level 2`

Note that from E only the first lines' variables are visible, while from E' only the first and last lines' variable are visible—hence the fact that (e.g.) `x` and `u` have the same 2D-address is unimportant.

Now, given an access to a variable x (with 2D address (i, j) from a point at function nesting level d, instead of accessing x by name we can instead use 2D index (relative address) of $(d-i, j)$. For example, access to `c` (whose 2D address $(1, 3)$) is $(2, 3)$ in E (in environment ρ_3 of depth 3) is $(2, 3)$, whereas access to the same variable in E' (in ρ_4 of depth 2) is $(1, 3)$.

Exercise 5: replace all variables in the above sample code by their indices (relative 2D addresses).

Logically, the move from environments as a list of name-value pairs to environments as chained arrays is just a matter of changing the abstract data-type for environments. It is a good exercise to code this for the lambda evaluator; however the full benefit is only achieved when the tree is more fully adjusted to make a single allocation site (at procedure entry) for all the `let` bindings within a procedure rather than an interpreter coming across `let` statements one at a time.

Note particularly that the idea of "function values are closures" is unaffected by this change of representation of environment; a compiled form of function value will be a pair consisting of a pointer to the code for the function together with an environment (a pointer to array of arrays holding variable values or a pointer to the stack frame of its definer).

9.7 Closure Conversion

Closure Conversion, also known as lambda-lifting, is an alternative way of implementing free variables—we do a source-to-source translation on the program to turn them into additional parameters. For example (say more):

```
fun f(x) {
   let a = ...;
   fun h(y) {
     let b = ...;
     fun g(w) {
        let c = ...;
        if ...
        then return a;
        else return h(c)
     }
     return b + g(y);
   }
   return x + h(a);
}
fun main() { return f(17); }
```

is transformed into

```
fun g'(w, x, a, y, b) {
   let c = ...;
   if ...
   then return a;
   else return h'(c, x, a )
}
fun h'(y, x, a) {
   let b = ...;
   return b + g'(y, x, a, y, b);
}

fun f'(x) {
   let a = ...;
   return x + h(a, x, a);
}

fun main() { return f'(17); }
```

This example only shows the case where functions are called directly; representing function variables which may point to one of several functions requires more work.

Another issue is that simple closure conversion replaces references to variables with variable (Rvalue) copies; if the source language supports variable update then the copies should instead be aliases (i.e. Lvalue copies).

Exercise 6: update the above example so that (e.g.) instead of having three separate variables named '`a`' all alias the original one, for example by use of C's address-of and dereference operators.

9.8 Landin's Principle of Correspondence *Non-examinable 10/11*

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. MIPS, ARM or x86.

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, and Java inner classes (a relatively recent addition) *do* allow a method to be nested inside a class inside a method which produces a variant of the problem.

The usual answer is to extend the *linkage* information so that in addition to holding the return address `RA` 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*[29] 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. In the example in Section 9.6, the environment for E was ρ_3 with accessible variables as follows:

ρ_1	`a:(1,1)`	`b:(1,2)`	`c:(1,3)`	`level 1`
ρ_2	`x:(2,1)`	`y:(2,2)`	`z:(2,3)`	`level 2`
ρ_3	`t:(3,1)`			`level 3`

Thus `t` is accessed relative to `FP` in one instruction, access to variables in ρ_2 first load the `S` field from the linkage information and then access `x`, `y` or `z` from that (two instructions), and variables in ρ_1 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

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

environment pointer from the closure to the S field in the linkage information in addition to the work detailed for the JVM.

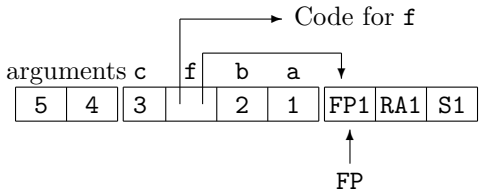
Exercise 7: give a simple example in which S and FP' pointers differ.

An example

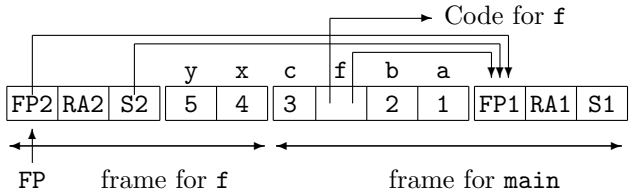
Consider the following function (we wrap all example code in this section within a main function to ensure variables are not seen as top-level variables which use absolute addressing):

```
let main() =
( 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 stack frame for main is as follows (in this and subsequent diagrams I have labelled the linkage information FP1 FP2 etc. instead of confusingly using FP' several times):



At the moment just after f has been entered (when a*x+b*y is about to be evaluated) the state is as follows (SP points to the same location as FP as f has not (yet) allocated any local variables):



We see that f can now access x and y from FP (at offsets +4 and +3, recall parameters appear higher in memory than a frame, and locals below a frame), and a and b from the definer's stack frame (offsets −1 and −2) 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 main (or even from a further function to which it was passed as a parameter) and so the stack frame for main() may or may not be contiguous with x as it is in the example. Similarly, it is vital to follow the static chain (S2 here) rather than the dynamic chain (FP2 here) even though in the example these point to the same place; a general explanation was given earlier, but here just consider what happens if f calls itself recursively—the FP3, FP4 (etc.) pointers chain down the stack each to the next previous frame, while the S3 and S4 (etc.) pointers all point to the frame for f.

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:

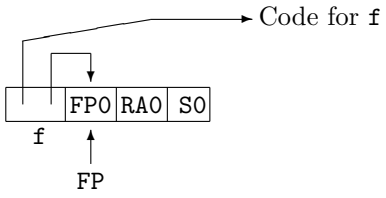
```
let main() =
(   let f() = { let a = 0
                in &a
              }
    let p = f()
    ...
)
```

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.

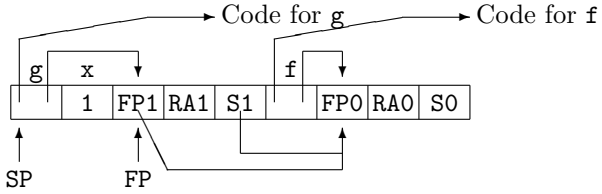
Some other objects such as functions and arrays contain implicit pointers to the stack and so have to be restricted if a stack implementation is to work. Consider:

```
let main() = {
    let f(x) = let g(t) = x+t // i.e. g = λt.x+t
               in g
    let add1 = f(1)
    ...
}
```

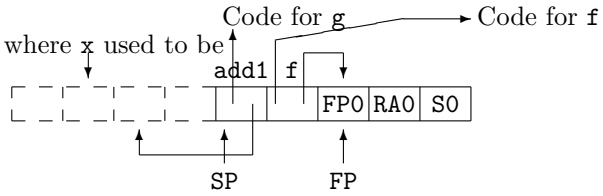
The result of `f(1)` should be a function which will add one to its argument. Thus one might hope that `add1(23)` would yield `24`. It would, however, fail if implemented using a simple stack. We can demonstrate this by giving the state of the stack at various stages of the evaluation. Just after the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



At the time when g has just been declared in the evaluation of `f(1)` the stack (for `main` and `f`) is as follows (SLIGHT ERROR: In the next two diagrams I've put `x`, an argument to `f`, on the 'wrong' side of `FP1`,`RA1`,`S1` as if `x` were a local variable of `f`—this is not quite consistent with the JVM model I presented, but the story is unchanged, so don't worry):



After the deallocation of the frame for `f` and declaration of `add1` in the stack frame of `main` 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 (note its dangling pointer to a de-allocated frame). 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 (and has been in this example–by `add1`).

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 (functions, arrays or labels) may be assigned or returned as the result of a procedure call. Algol-60 first coined these restrictions (functions, stack-allocated arrays etc. can be passed into functions but not returned from them) as enabling a stack-based implementation to work and they still echo in many current languages. Languages like Java avoid this issue at the cost of heap-allocating all objects.

ML clearly does allow function values to be returned from function 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.[30] 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 might seem excessive (but see Appel's "Compiling with Continuations" book) and we now turn to a common, efficient implementation.

10.3 Implementing ML free variables

In ML programs like

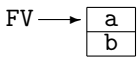
```
val a = 1;
fun g(b) = (let fun f(x) = x + a + b in f end);
val p = g 2;
val q = g 3;
```

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

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

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: because of standard ML's use of `ref`, there are no updatable variables in ML—only updateable `ref` cells. Therefore the values (ordinary values or (pointers to) `ref` cells can be copied without unaliasing anything.)

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.[31]

Thus if a simple variable may be defined (see Section 9.1) 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 a = 10
r(a)
// a now equals 11
```

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

10.5 Algol call-by-name and laziness

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.[32]

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.

[31] 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!

[32] 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 the most appropriate calling mechanism for each definition in such languages.

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

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

1. Declare the parameter as a parameterless function (a 'thunk').
2. Replace all occurrences of it in the body by parameterless calls.
3. Replace the actual parameter expression by a parameterless function whose body is that expression.

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

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

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

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 and C♯'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.

Incidentally while call-by-value-result (IN OUT) parameter passing fell out of favour after Ada, it has significant advantages for a concurrent call to another processor on a shared-memory multi-core processor compared to using aliasing. Each write to a common location from a given processor invalidates the corresponding cache line in other processors, hence taking a copy and writing back can be significantly more efficient.

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 operand 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.

Such a jump is called a *long jump*.

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![33])

[33] When compiling Prolog (beyond this course), note that backtracking has a good deal in common with branching back into a previously exited function!

10.8 Exceptions

ML and Java exceptions and their handlers are a form of long jump where the destination depends on the program dynamics and that can pass 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.

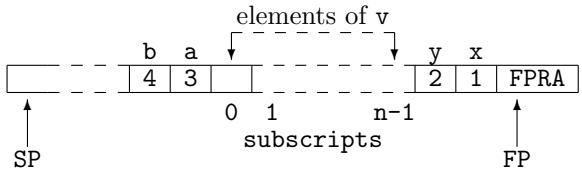
Finally, sadly ISO C labels cannot be used as values as indicated above, and so code shown above would have to be implemented using the library function `setjmp()` instead.

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 function with 'obvious' storage layout on a descending stack:

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

Within its body its stack frame might look like the following:



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,[34] 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.[35]

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 '`:`'.]

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 type `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:

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

[35] 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.

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

Note that `hidden` is usually called `this` (modulo the fact that my `hidden` is a value of type `C` while `this` is a pointer to a `C`)—the idea here is merely to emphasis its implicit nature as a 'hidden parameter'.

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 record containing a storage cell containing the address of the function to be called. Using C as a representation language, objects `a` of `class A` and `b` of `class B` would be represented by:

```
void f_A(struct A *hidden) { printf("I am an A"); }
void f_B(struct A *hidden) { printf("I am a B"); }
struct A { void (*f()); } a = { f_A };
struct B { void (*f()); } b = { f_B };
```

(Aside: in practice, since there may be many virtual functions, in practice a *virtual function table* is often 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 when methods of this object are invoked. This can be shared among all objects declared of 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 Stroustrup "The C++ Programming Language (3rd Edition)" or to the C++ standard (ISO/IEC 14882:2003 "Programming languages—C++").

10.11 C++ multiple inheritance

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

```
class A { int a1, a2; };
class B : A { int b; };
class C : A { int c; };
class D : B,C { int d; };
```

(Example, a car and a boat both inherit from class vehicle, so think about an amphibious craft.)

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` still can be accessed at byte offset 8 in the received `C`. (This assumes that `B` is stored at offset zero in `D`, and `C` follows contiguously.)

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 in the language C as

```
struct { A *__p, int b;    // object of class B
         A *__q, int c;    // object of class C
         int d;
         A x;              // the shared object in class A
       } s =
            { &s.x, 0,   // the B object shares a pointer ...
              &s.x, 0,   // with the C object to the A base object
              0,         // the d
              { 0, 0 }   // initialise A's fields to zero.
            };
```

I.e. there is a single `A` object (stored as 'x' above) and both the `__p` field of the logical `B` object (containing `__p` and `b`) and the `__q` field of the logical `C` object (containing `__q` and `c`) point to 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 somewhat similar facilities, but components of the interface must be freshly implemented, rather than being inherited from a parent.

10.12 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[36] is a storage area from which storage blocks can be allocated for dynamic data where the order of freeing need not be known in advance. Hence it is separate from the stack and from statically allocated global variables. Dynamic storage is used when the amount of date to be stored will depend on the runtime input to the 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 system that records which blocks on disc are used and which are available for allocation for new files.

[36] Note this use of the word 'heap' is complete distinct from the meaning "implementation of a priority queue within an array".

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).

10.13 Data types

With a small exception in Section 5, 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 Python, 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 needs to check whether the value stored in the cell is of the correct type. (This manifested itself in the lambda calculus evaluator in Section 9.4 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 likely 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[37] 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).

Finally, sometimes languages are described as *typeless*. BCPL (a forerunner of C) is an example. The idea here is that we have a single data type, the word (e.g. 32-bit bit-pattern), within which all values are represented, be they integers, pointers or function entry points. Each value is treated as required by the operator which is applied to it. E.g. in `f(x+1,y[z])` we treat the values in `f`, `x`, `y`, `z` as function entry point, integer, pointer to array of words, and integer respectively. Although such languages are not common today, one can see them as being in the

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

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).

10.14 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 Correctness

These notes have just presented compilation as a program which takes source code and produces target code justified by "it looks plausible and the lecturer says it's OK". But for more complicated languages, and more sophisticated translation techniques, we can be left thinking "I wonder if that trick works when features X and Y interact...". A nice example is static members in generic classes: when given

```
class List<Elt>
{  List <Elt> Cons(Elt x) { number_of_conses++; ... }
   static int number_of_conses = 0;
}
```

then should there be one counter for each `Elt` type, or just one counter of all conses? Entertainingly the languages Java and C♯ differ on this. So writing a correct compiler demands complete understanding of source and target languages.

Semantics courses provide a proper formal answer to the question of compiler correctness. Suppose S, T are source and target languages of a compiler f. For simplicity we assume f is a mathematical function $f : S \to T$ (in practice f is implemented by a compiler program C written in an implementation language L, but in order to avoid worrying recursively about whether the implementation of L is correct and whether C terminates, we will just think of f as a function).

Now we need semantics of S and T (semantics are just precise language specifications which of course include explanation of whether things like `number_of_conses` are one-per-system or one-per-class). Let's write these semantics as $[\![\cdot]\!]_S : S \to M$ and $[\![\cdot]\!]_T : T \to M$ for some set of meanings M. To avoid nasty issues we will assume the semantics give the "final value", or output, of a program (of course we expect internal behaviour of evaluation of terms in S and T to differ) and moreover M is something simple like an integer or string (to facilitate comparison).

We can now say that f is a *correct compiler* provided that

$$(\forall s \in S)[\![f(s)]\!]_T = [\![s]\!]_S.$$

There are two subtleties. One concerns non-determinism: if S has non-deterministic features (like races), then is it acceptable for a compiler to reduce this (making a run-time choice at compile time)? This course has only considered deterministic languages, so we leave this issue to others. A second is termination: either we need to see $[\![\cdot]\!]_S$ and $[\![\cdot]\!]_T$ as partial functions (and equality to be "both yield the same value or both are undefined"), or we need to introduce a distinguished element $\perp$ to the set M which explicitly represents non-termination (and let equality be the natural operation on this larger set). The semantics courses here explore some of these issues more.

11.2 Compiler composition, bootstrapping and Trojan compilers

Non-examinable—just for fun (and only part-written).

As above, let L, S, T, U be languages. As we know, functions $f : S \to T$ and $g : T \to U$ can be composed. But, exploring the difference between f and C in the previous section more, we can express a compiler C from S to T written in language L as $C : S \overset{L}{\rightsquigarrow} T$. Note that juxtaposing two compilers written in the same language (putting the output of one as the input of another) gives a form of composition having type:

$$(S \overset{L}{\rightsquigarrow} T) \times (T \overset{L}{\rightsquigarrow} U) \to (S \overset{L}{\rightsquigarrow} U).$$

A semantics for L now turns a program in L into a function, in particular it now also has type $[\![\cdot]\!]_L : (S \overset{L}{\rightsquigarrow} T) \to (S \to T)$. We are now going to consider the machine code of our host architecture as a language H and, by some extent abuse of concepts, regard $[\![\cdot]\!]_H$ as meaning "execute a program in language H".

Clearly the only practically useful compilers are of the form $S \overset{H}{\rightsquigarrow} H$, since we need them to be both executable and to produce executable code. But we can use composition to produce other compilers: we've already seen how a compiler $S \overset{H}{\rightsquigarrow} T$ can be composed with one $T \overset{H}{\rightsquigarrow} H$ to give a usable compiler $S \overset{H}{\rightsquigarrow} H$. But these 'compilation types' can also be 'vertically' composed: use a $L \overset{H}{\rightsquigarrow} H$ compiler to compile a $S \overset{L}{\rightsquigarrow} H$ one to yield a usable compiler $S \overset{H}{\rightsquigarrow} H$

One problem is that people typically write a compiler for a new language U in the language itself (because it's a great new perfect language!). Suppose they are kind enough to make it generate H code, so we have a compiler $U \overset{U}{\rightsquigarrow} H$, but we still have the so-called 'bootstrapping problem' that until we have a compiler or other execution mechanism for U we have no composition to make the compiler useful (i.e. of the form $U \overset{H}{\rightsquigarrow} H$). The trick is to somehow make some sort of prototype compiler of the form $U \overset{H}{\rightsquigarrow} H$ and then use that to compile the $U \overset{U}{\rightsquigarrow} H$ compiler to produce (hopefully a better) compiler $U \overset{H}{\rightsquigarrow} H$. This bootstrapping processes can be repeated.

But, one might ask various interesting questions, such as: do the sequence compilers eventually become equal? Can the original prototype compiler leave some form of footprint? In particular, if the $U \overset{U}{\rightsquigarrow} H$ compiler is correct, does that mean that the bootstrapped compiler is correct?

The latter question is answered in the negative, and has entertaining security implications. It is possible to write a compiler which miscompiles one piece of code (e.g. the login program) but which correctly compiles every other piece of code *apart from itself, because it compiles this to a compiler which miscompiles the login program and itself.* Moreover this compiler is correct when seen as a program $U \overset{U}{\rightsquigarrow} H$ (it compiles the login program correctly) but (of course) incorrect as is the program in $U \overset{H}{\rightsquigarrow} H$ resulting from bootstrapping. So the security attack cannot be discovered by source code inspection. See `http://cm.bell-labs.com/who/ken/trust.html` for details (the 1984 Turing Award classic by Ken Thompson).

11.3 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 by a C program. One comparatively-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 input data is available and execution starts) and how much is done during execution.

In typical implementations of Python and PHP, and 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.

11.4 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.

The design of debuggers involves a difficult compromise between space efficiency, execution efficiency and the effectiveness of the debugging aids. The unix utility `strip` and and the gcc option `-g` and `-fomit-frame-pointer` reflect this. *Exercise 8:* Find out why.

11.5 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?).

```
%%
[ \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 4: `calc.l`

12 Automated tools to write compilers

Automated tools to write compilers 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). Automation has largely been applied to parts of a compiler not directly related to run-time efficiency, i.e. lexing and parsing, but there is increasing, if not yet common, interest in automated generation of code generators from machine specifications.

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.

12.1 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 4. 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 a synthesised function called `lex`, and thus a `return` statement will cause a return from this function with a specified value. Certain 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 by 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 translation 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.

12.2 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 5.

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 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.

```
%{
#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 5: `calc.y`

13 Phrase-structured grammars

The concept of the phrase-structured grammar was formalised by Chomsky.

We start with an *alphabet*, Σ, of symbols (think of these as input characters to lexing or tokens resulting from lexing that are input to the syntax analyser). 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 defined 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 defined subset of Σ^*, i.e. $\mathcal{L} \subseteq \Sigma^*$.

To make the definition we need a rule (or rules) that defines 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 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* 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 '$\longrightarrow$'). 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 is the Chomsky Type 2 grammar (commonly known as a context-free grammar for reasons which will become clear below).

13.1 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
```

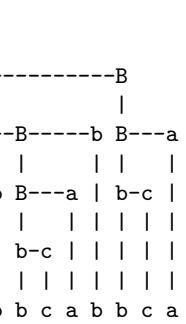
A slightly more convenient way of writing the above grammar is:

```
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:



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

13.2 Type 0 grammars

In a type 0 grammar 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.

13.3 Type 1 grammars

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

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

where U 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 U can only be replaced by $B_1 \cdots B_n$ if it occurs in a suitable context (the L_i form the left context and the R_i the right context).

13.4 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:

$$\begin{aligned} U &\longrightarrow a \\ U &\longrightarrow a\ V \end{aligned}$$

That is, the right hand side must consist of a single terminal symbol possibly followed by a single non-terminal.

Type 3 grammars can clearly be parsed using a finite state recogniser, and for this reason they are often called *regular grammars*. [To get precise correspondence to regular languages it is necessary also to allow the empty production S ⟶ϵ otherwise the 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; more information on topics like this can be found on Wikipedia, e.g. http://en.wikipedia.org/wiki/Regular_grammar

13.5 Grammar inclusions

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.

14 How parser generators work

As mentioned earlier, given a relatively large grammar, it is more convenient to supply the grammar to a tool (a 'parser generator') and allow it to generate a parser rather than hand-writing a parser. Such tools tend not to generate a parser in code form; rather they derive a table from the grammar and attach a fixed table-driven parsing algorithm to it. Parser generators do not accept an arbitrary context-free grammar, but instead accept a restricted form, of which the most common form for practical tools (e.g. yacc, mlyacc, CUP) is the LALR(1) grammar (although antlr uses LL(k) parsing).

Rather than explain how a LALR(1) parser generator works, I will explain so-called SLR(k) parsers work; these use the the same underlying parsing algorithm, but generate less compact (but easier to understand) tables.

The basic idea is that the currently-consumed input could potentially be parsed in various ways and we keep all feasible possibilities alive using a non-deterministic automaton. However, we convert the NDA to a deterministic one at compile time (inside the yacc tool or whatever) (using the *subset construction*) so we only need to run the deterministic machine at run time.

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

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

The form of production `#0` is important. It defines the sentence symbol `S` and its RHS consists of a single non-terminal followed by the special terminal symbol `eof` which must not occur anywhere else in the grammar. (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. Items are also commonly known as *configurations*. 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 then the item is called a *completed* item. In forming item sets a *closure* operation must be performed to ensure that whenever the marker in an item of a set precedes a non-terminal, `E` say, then initial items must be included in the set for all productions with `E` on the left hand side.

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

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

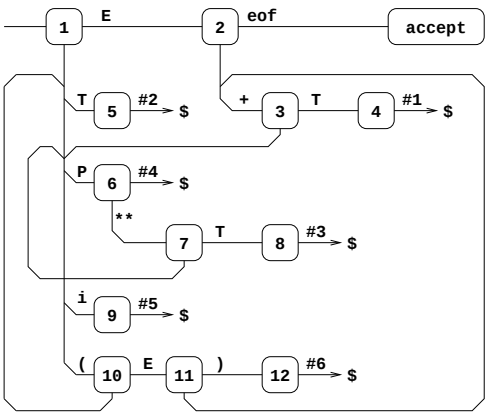
States have *successor* states formed by advancing the marker over the symbol it precedes. For state 1 there are successor states reached by advancing the marker over the symbols '`E`', '`T`', '`P`', '`i`' or '`(`'. Consider, first, the `E` successor (state 2), it contains two items derived from state 1 and the closure operation adds no more (since neither marker precedes a non terminal). State 2 is thus:

```
2: {  S  ⟶  E .eof
      E  ⟶  E .+  T
   }
```

The other successor states are defined similarly, except that the successor of `eof` is always the special state `accept`. If a new item set is identical to an already existing set then the existing set is used. The successor of a completed item is a special state represented by `$` and the transition is 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 6. Note that for completed items the successor state is reached via the application of a production (whose number is given in the diagram).

The CFSM can be represented diagrammatically as follows:



```
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 6: CFSM item sets (please excuse ascii art)

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

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

The example grammar gives matrices (using dash (`-`) to mark blank entries);

	action						goto		
state	eof	(	i	)	+	**	P	T	E
S1	-	S10	S9	-	-	-	S6	S5	S2
S2	acc	-	-	-	S3	-	-	-	-
S3	-	S10	S9	-	-	-	S6	S4	-
S4	#1	#1	#1	#1	#1	#1	-	-	-
S5	#2	#2	#2	#2	#2	#2	-	-	-
S6	#4	#4	#4	#4	#4	XXX	-	-	-
S7	-	S10	S9	-	-	-	S6	S8	-
S8	#3	#3	#3	#3	#3	#3	-	-	-
S9	#5	#5	#5	#5	#5	#5	-	-	-
S10	-	S10	S9	-	-	-	S6	S5	S11
S11	-	-	-	S12	S3	-	-	-	-
S12	#6	#6	#6	#6	#6	#6	-	-	-

and so therefore is not SLR(0)—because (state `S6`, symbol '$**$') is marked '`XXX`' to indicate that it admits both a shift transition (`S7`) and a reduce transition (`#4`) 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.

14.2 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 U 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.[38]

1. If there is a production of the form $U \longrightarrow \cdots VB \cdots$ put B into `FOLLOW`(V).
2. If, moreover, B is a non-terminal then also put all symbols in `Left`(B) into `FOLLOW`(V).
3. If there is a production of the form $U \longrightarrow \cdots V$ put all symbols in `FOLLOW`(U) into `FOLLOW`(V).

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	eof +)
T	eof +)
P	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	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	-	-	-

14.3 SLR parser runtime code

The parsing algorithm used for all LR methods uses a stack that contains alternately state numbers and symbols from the grammar, and a list of input terminal symbols terminated by eof.[39] A typical situation is represented below:

[38] Apology: because of the structure of the example language (which does not contain two adjacent non-terminals in any production) case 2 of this construction is never exercised—thanks to Tom Stuart for pointing this out. A production such as $U \longrightarrow xVWy$ with x and y terminals and U, V and W non-terminals would exercise this case.

[39] The stack can also be coded as a stack of pairs with minor changes. This is more convenient when the generator is implemented in modern, strongly-typed languages. The pairs are instead triples when a parse tree is being generated, which is the normal case.

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

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

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

will be a sentential form.

The parsing algorithm starts in state `S1` with the whole program, i.e. configuration

`1 | ⟨the whole program upto 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 eof
   ```

   to

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

   will transform to

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

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

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

   ```
   a Q f  |  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 `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 eof	
1 i 9	+ i eof	P ⟶ i
1 P 6	+ i eof	T ⟶ P
1 T 5	+ i eof	E ⟶ T
1 E 2	+ i eof	
1 E 2 + 3	i eof	
1 E 2 + 3 i 9	eof	P ⟶ i
1 E 2 + 3 P 6	eof	T ⟶ P
1 E 2 + 3 T 4	eof	E ⟶ E + T
1 E 2	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.

Note that the above parse *is* an LR-parse: look at the productions used (backwards, starting at the bottom of the page since we are parsing, not deriving strings from the start symbol).

We see

E ⟶ E+T ⟶ E+P ⟶ E+i ⟶ T+i ⟶ P+i ⟶ i+i

i.e. a *rightmost derivation*.

14.4 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.

[The end]