Compiler Construction

(final version)

A 16-lecture course

David J Greaves

(Notes and slides thanks to Prof Alan Mycroft)

Computer Laboratory, University of Cambridge

`http://www.cl.cam.ac.uk/users/djg11`

2011–2012: Lent Term

Course Plan

Part A : intro/background

Part B : a simple compiler for a simple language

Part C : implementing harder things, selected additional detail

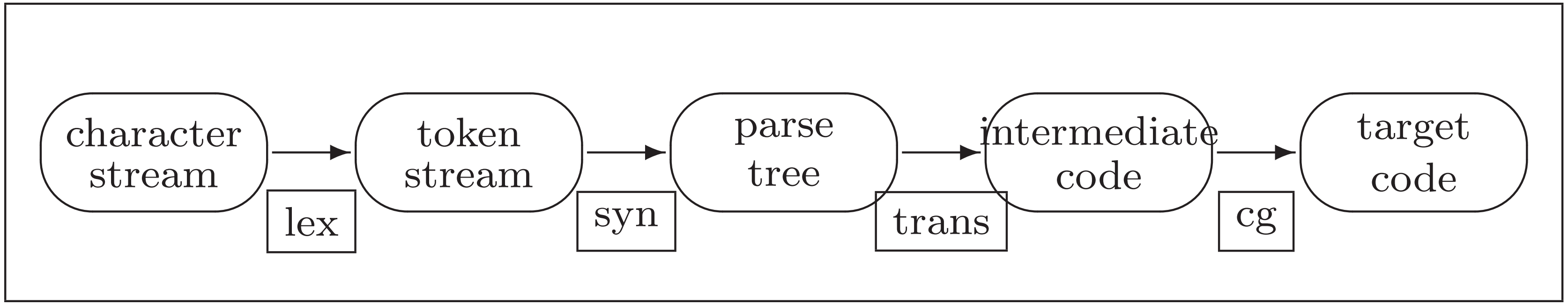
A compiler

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.
- Can appear a very hard program to write.
- How can one even start?
- It's just like juggling too many balls (picking instructions while determining whether this '+' is part of '++' or whether its right operand is just a variable or an expression . . .).

How to even start?

"When finding it hard to juggle 4 balls at once, juggle them each in turn instead . . . "



A *multi-pass compiler* does one 'simple' thing at once and passes its output to the next stage.

These are pretty standard stages, and indeed language and (e.g. JVM) system design has co-evolved around them.

Compilers can be big and hard to understand

Compilers can be very large. In 2004 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".

But, if we choose a simple language to compile (we'll use the 'intersection' of C, Java and ML) and don't seek perfect code and perfect error messages then a couple thousand lines will suffice.

Overviews

lex (lexical analysis) Converts a stream of characters into a stream of tokens

syn (syntax analysis) Converts a stream of tokens into a parse tree—a.k.a. (abstract) syntax tree.

trans (translation/linearisation) Converts a tree into simple (linear) intermediate code—we'll use JVM code for this.

cg (target code generation) Translates intermediate code into target machine code— often as (text form) assembly code.

But text form does not run

- use an *assembler* to convert text form instructions into binary instructions (Linux: `.s` to `.o` file format; Windows: `.asm` to `.obj` file format).
- use a *linker* (‘*ld*’ on linux) to make an *executable* (`.exe` on Windows) including both user’s compiled code and necessary libraries (e.g. println).

And that’s all there is to do!

Overview of ‘lex’

Converts a stream of characters into a stream of tokens.

From (e.g.)

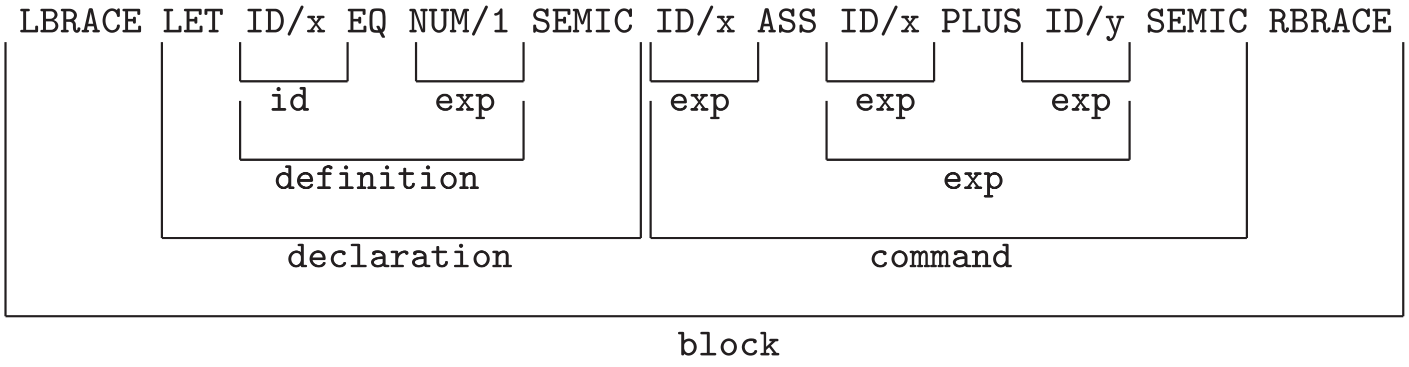
```
{ let x = 1;
  x := x + y;
}
```

to

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

Overview of ‘syn’

Converts the stream of tokens into a parse tree.

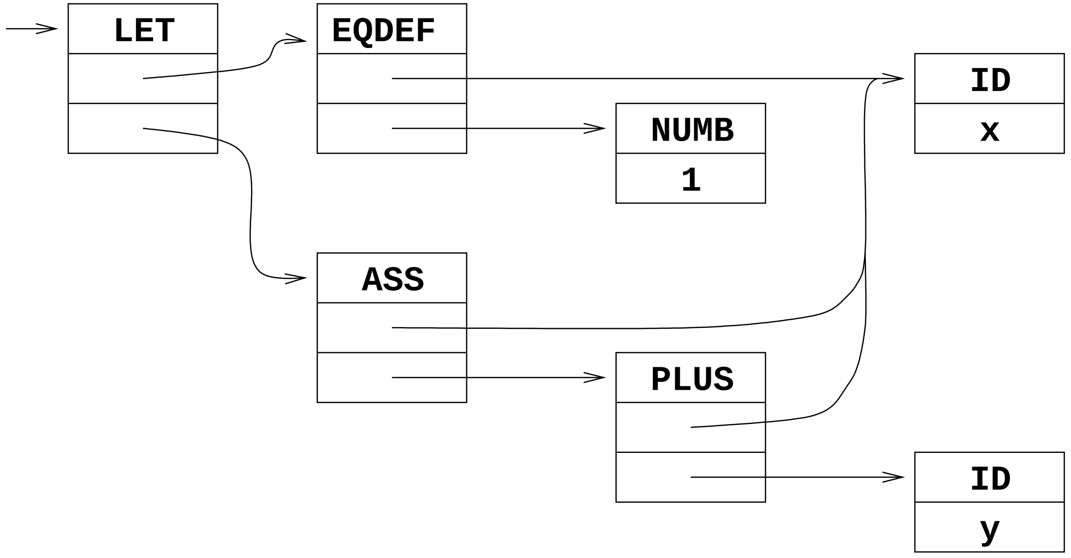

LBRACE LET ID/x EQ NUM/1 SEMIC ID/x ASS ID/x PLUS ID/y SEMIC RBRACE
id
exp
exp
exp
exp
definition
exp
declaration
command
block

Overview of 'syn' (2)

Want an an abstract syntax tree, not just concrete structure above:

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

might produce (repeated tree notes are shown *shared*)



Overview of ‘trans’

Converts a tree into simple (linear) intermediate code. Thus

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

might produce (using JVM as our intermediate code):

```
iload 4        load x (4th local 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)
```

Overview of ‘cg’

Translates intermediate code into target machine code.

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

can produce (simple if inefficient ‘blow-by-blow’) MIPS code:

```
     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)
```

Commercial justification for multi-pass compiler

Write n front-ends (lex/syn) and m back-ends (cg) and you get $n \times m$ compilers (lots of cash!) for compilers translating any of n languages into any of m target architectures.

Also, separate teams can work on separate passes. (‘passes’ are also called ‘phases’).

Machine Code

- Compilers typically translate a high-level language (e.g. Java) into machine instructions for some machine.
- This course doesn't care what machine we use, but examples will mainly use MIPS or x86 code.
- We only use the most common instructions so you don't need to be an expert on Part IB "Computer Design".
- So here's a very minimal subset we need to use:

MIPS Machine Code (1)

Instructions to:

- load a constant into a register, e.g. `0x12345678` by

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

- load/store local variable at offset `<nn>`

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

- load/store global variable at address `0x00be3f04`

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

MIPS Machine Code (2)

Instructions to:

- do basic arithmetic/logic/comparison

```
add     $a2,$a0,$a1
xor     $a2,$a0,$a1
slt     $a2,$a0,$a1          ; comparison
```

- function calling: complicated (and we're cheating a bit): *caller* pushes the arguments to a function on the stack (`$sp`) then uses `jal`; *callee* then makes a new *stack frame* by pushing the old value of `$fp` (and the return address—`pc` following caller) then sets `$fp` to `$sp` to form the new stack frame.
- function return is largely the opposite of function call; on the MIPS put result in `$v0` then return using `jr`.

JVM code subset (1)

We need only a small subset.

Arithmetic:

`iconst` $\langle n \rangle$ push integer n onto the stack.

`iload` $\langle k \rangle$ push the kth local variable onto the stack.

`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.

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

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

JVM code subset (2)

Branching:

`invokestatic` f 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` ℓ unconditional branch.

`label` ℓ not an instruction: just declares a label.

NB: apart from MIPS using registers and JVM using a stack the two subsets provided give very similar functionality.

How do I see these in use?

Reading assembly-level output is often really useful to aid understanding of how language features are implemented.

```
gcc -S foo.c        # option -O2 is often clearer
```

will write a file `foo.s` containing assembly instructions for your *current architecture*

Otherwise, use a *disassembler* to convert the object file back into assembler level form, e.g. in Java

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

Lecture 2

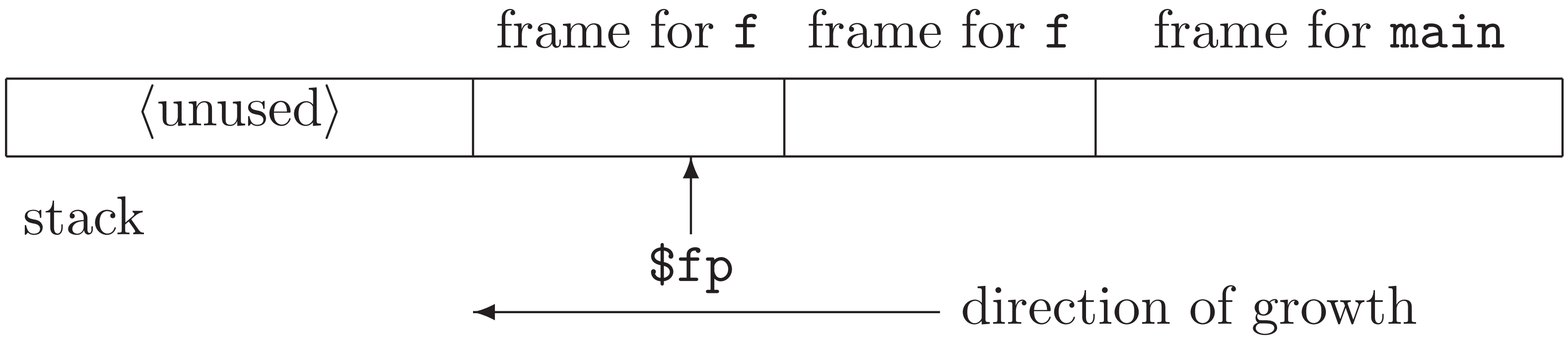
Stacks, Stack Frames, and the like . . .

Stacks and Stack Frames

- Static/Global variables: allocated to a fixed location in memory.
- Local variables: need multiple copies for recursion etc.—use a *stack*.
- A stack is a block of memory in which *stack frames* are allocated. Function call allocates a new stack frame; function return de-allocates it.
- MIPS register `$fp` points to stack frame of the currently active function. When a function returns, its stack frame is deallocated and `$fp` restored to point to the stack frame of the caller.
- Local variables: allocated to a fixed offset from `$fp`; 5th local variable typically at `-20($fp)`

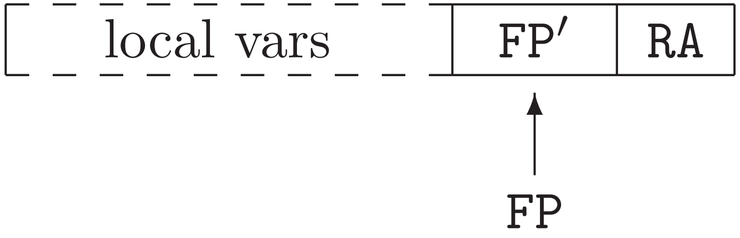
Stacks and Stack Frames (2)

A "downward-growing stack" exemplified for `main()` which calls `f()` which calls `f()`:



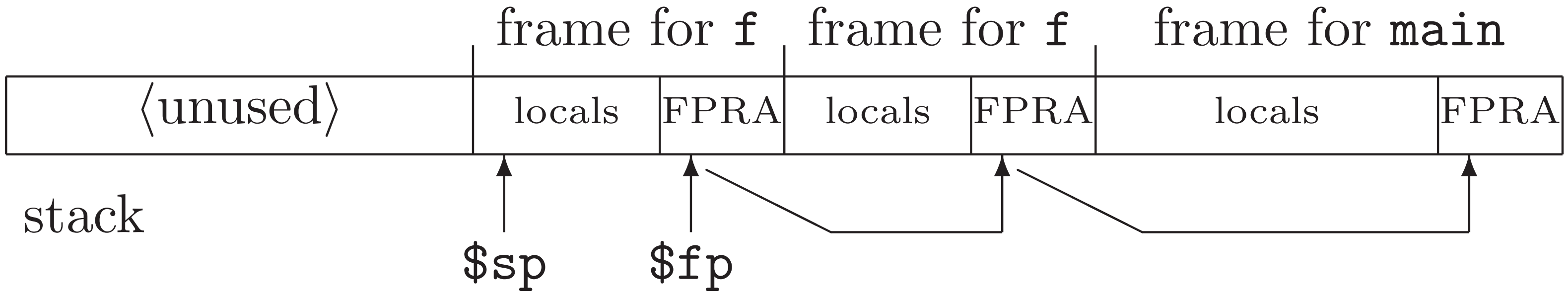
Stacks and Stack Frames (3)

Stack frame needs to save pointer to previous stack frame (FP′) and also return address (RA):



Stacks and Stack Frames (4)

A stack now looks like:



`$sp` points to the lowest used location in:

1. the stack as a whole; and
2. the currently active stack frame.

So, memory below `$sp` can be used for temporary work space (evaluation stack) and for preparing parameters for a callee.

Stacks and Stack Frames (5): parameter passing

We're cheating: the MIPS procedure standard uses registers (`$a0`–`$a3`) to communicate the first 4 arguments, and the stack for the rest (efficiency). We'll use the stack for all of them!

Treaty:

- the caller and callee agree that the parameters are left in memory cells at `$sp`, `$sp+4`, etc. at the instant of call.

Stacks and Stack Frames (6): parameter passing

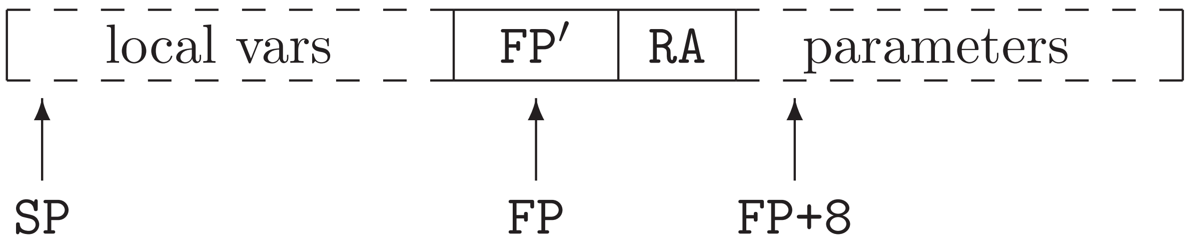
Done by:

- the caller evaluates each argument in turn pushing it onto `$sp`. I.e. `*--SP = arg;` in C.
- the callee first stores the linkage information (contiguous with the received parameters) and so parameters can be addressed as `$fp`+8, `$fp`+12, etc. (assuming 2-word linkage information pointed at by `$fp`).

So, the callee sees its parameters at positive offsets from `$fp` and its local variables at negative offsets from `$fp` with linkage info in between.

Stacks and Stack Frames (7): parameter passing

Better view of a stack frame:



Space below (to the left of) the stack frame is used to construct the argument list (possibly empty) of any called routines—the called routine then turns this into a 'proper' stack frame.

Typical code for procedure entry/return

Caller to `foo` does:

```
          addi  $sp,$sp,-4     ; make space for (single) argument
          sw    $a0,0($sp)     ; push argument
          jal   foo            ; do the call (puts r37 into $ra)
    r37:
```

At entry to callee:

```
    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
```

On return from callee (result in `$v0`):

```
    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
```

Who removes the arguments to a call?

One *subtlety* (below the level of examination) which I've omitted is:

> Who removes the arguments to a call? Caller or callee?

- On the MIPS, the caller does it (and doesn't happen very often on the real MIPS procedure calling standard because of the "first 4 arguments in registers" rule).
- On the JVM, the callee does it (see two slides on).
- On the x86 there are *two standards*—one of each.

Why? C, but not Java, offers support for '*vararg*' functions which take variable numbers of arguments.

Sample Java procedure calling code

Simpler—as expected—real machines have other trade-offs than "simple representation of Java" in the JVM design.

```
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:                ; <say meta data here: 2 args, 1 local>
  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
```

Sample Java procedure calling code (2)

Given

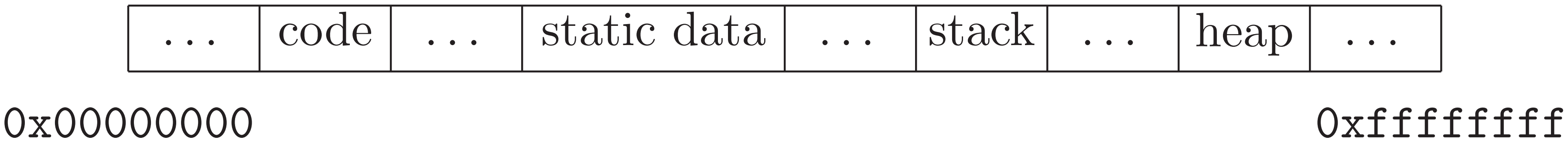
```
    public static void main(String args[]) {
        System.out.println("Hello World!" + f(f(1,2),f(3,4)));
        }
```

the series of calls in the `println` would be

```
      iconst 1
      iconst 2
      invokestatic f
      iconst 3
      iconst 4
      invokestatic f
      invokestatic f
```

Note how in the JVM a two-argument procedure call looks just like a binary operator (`iadd` etc.).

Address space map

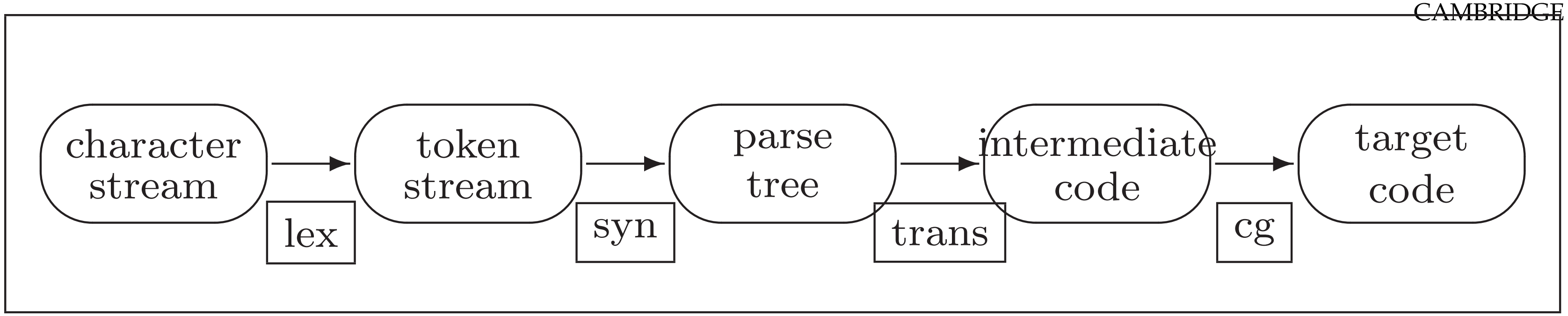


. . .	code	. . .	static data	. . .	stack	. . .	heap	. . .

`0x00000000` `0xffffffff`

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 this course.

What is "just in time compilation" (JIT)?



A classical compiler does all these on one machine. To distribute a system for multiple architectures we compile it once per architecture.

When running Java in a Browser, the JVM file is transported after the first 3 stages of compilation. The recipient browser may:

- Interpret the JVM code (see later).
- Do the last stage of compilation (CG) now the host architecture is known (this is called "just in time" compilation).

Our simple language

Use a language of your choice to *implement* the compiler.

The *source* language we use is in the 'intersection' of C/Java/ML!

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

Language syntax

```
<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>*
```

Plus various other restrictions (see notes).

Forms of Interpreter

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

token-stream form again this no longer makes sense, parsing is now so cheap that it can be done when a program is read; historically BBC Basic stored programs in tokenised form and re-parsed them on execution (probably for space reasons—only one form of the program was stored);

Forms of Interpreter

syntax-tree form this is a natural and simple form to interpret (also link to "operational semantics"). 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.

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 *emulators*) in the same manner as we might write a JVM interpreter.

Lecture 3

Interpreters

Interpreters

In general doing it makes sense to do as much work as possible before interpreting (or direct execution): "*Never put off till run-time what you can do at compile-time.*" [Gries].

Done once versus potentially done many times.

This particularly makes sense for *statically typed* languages.

BTW, not said in notes: systems people tend to call 'invented' machines "virtual machines"; theorists tend to call them "abstract machines", but *same concept*.

How to write a JVM interpreter

- read in a `.class` file
- put *code* into a byte array `imem[]` “byte code instructions”. make PC point to entry to `main`
- allocate a word array `dmem[]` “we only support integers”. Put *static data* at base of this; make SP and FP index top of it.
- (mumble about relocation/use of library routines)
- simulate the fetch/execute cycle until we hit a ‘halt’ instruction.

How to write a JVM interpreter (2)

```
void interpret()
{   byte [] imem;             // instruction memory
    int [] dmem;              // data memory
    int PC, SP, FP;           // JVM registers
    int T;                    // a temporary
    ...
    for (;;) switch (imem[PC++])
    {
/* special case opcodes for small values (smaller .class files): */
case OP_iconst_0:   dmem[--SP] = 0; break;
case OP_iconst_1:   dmem[--SP] = 1; break;
case OP_iconst_B:   dmem[--SP] = imem[PC++]; break;
case OP_iconst_W:   T = imem[PC++]; dmem[--SP] = T<<8 | imem[PC++]; break;

/* Note use of FP-k in the following -- downwards growing stack */
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;
```

How to write a JVM interpreter (3)

```
    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 */
        }
    }
}
```

How to write a JVM interpreter (4)

There's a worry here: the JVM opcodes just use contiguous offsets (to `iload` and `istore`) for arguments and locals—whereas previously we required a 2-word gap between them for linkage information.

- when *interpreting* it's simpler to have a (yet another) stack (or indeed two separate stacks) which just holds "return addresses" and "previous frame pointers"
- when *compiling* to a single-stack-segment solution (more flexible) such as MIPS, it's easy to insert a gap:

$$0 \mapsto +12; \quad 1 \mapsto +8; \quad 2 \mapsto -4; \quad 3 \mapsto -8; \quad \ldots$$

How to write a JVM interpreter (5)

```
    case OP_invokestatic:
                        T = <get callee start address from PC>
                        linkagestack[--LSP] = PC;
                        linkagestack[--LSP] = FP;
                        PC = T
                        FP = SP + <n_p>;
                        SP = SP - <n_v>; /////////// FIX

    case OP_ireturn:    ...
    /* etc etc etc */
        }
    }
}
```

And that *really is all*—it’s just coding.

How to write an emulator for another machine

Suppose we want to execute the output from a compiler which produces code for a machine we don't have (e.g. obsolete, or not yet manufactured).

Just write a JVM-style interpreter for its code.

This is traditionally called an *emulator* or *simulator*. If you're a hardware person you might want a *cycle-accurate* emulator which also tells you exactly how long the program would take to run on the real architecture.

If you're trying to sell your customers a new architecture and want to tell them their existing binary programs will still run you might want a "dynamic binary translator" (JIT translator looking like a fast emulator).

Syntax tree interpreter

We're going to cheat. The Expr/Cmd/Decl language is still a bit too big for lectures, so I'm going to ban Cmds:

- require function bodies to be of the form { `return` e`;` }
- re-allow limited local Decls by adding `let` x`=`e `in` e'

Syntax tree interpreter (2)

So get language (this is a subset of ML, but can be seen as Java or C too):

```
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;
```

Interpreters for expression-based languages are traditionally named `eval`...

Syntax tree interpreter (3)

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`:

```
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);
```

The evaluator takes an expression and an environment and returns its value.

Syntax tree interpreter (4)

```
(* 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
  | eval(Apply(s,el) r) = ...
```

Syntax tree interpreter (5)

We've not done 'Apply'. That's because it's harder (at least at first)!

When we apply a function we get a new lot of local variables (new environment) but keep the same set of global variables.

There's more sophistication later (Part C). But let's be naive for now.

Instead of one environment have two: `rl` (local) and `rg` global. Look a variable up locally and if that fails then look it up globally.

Syntax tree interpreter (6)

```
(* 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)
      in eval(body, rg,rlnew)
      end
```

(`zip` converts a pair of lists into a list of pairs.)

Syntax tree interpreter (7)

Writing an interpreter really focuses your mind on what a language does/means.

That's why theorists like 'semantics' (operational semantics are essentially an interpreter written in maths)—semantics give precise meanings to programs. From the interpreter you can see (e.g.)

- How one variable shadows the scope of another (assuming `lookup` is coded correctly).
- The difference between updating an existing variable (look it up with `lookup` and replace the value stored in the environment) and using `let` to create a new variable.
- How `let` x`=`e `in` e' is very similar to `f(`e`)` where `f(`x`)=`e' (inline expansion/beta-reduction).

Lecture 4

Lexical Analysis or Tokenisation.

Lexical Analysis

- Converts a character stream to a token stream (a.k.a. tokenisation).
- Tokens are things like “left shift symbol”, “integer constant” or “string”, or even “plus symbol” formed of a single character.
- Typically removes whitespace (including comments!) – whitespace might be needed to separate tokens but is not a token itself.
- Most common interface is procedural: `TokType lex();`. Compare the corresponding `int getchar();` in C which gives a character stream. Note `lex()` will probably need a 1-place buffer to tokenise things like “`abc+1`” as we only know the `abc` is complete after reading the `‘+’`.

Lexical Analysis (2)

Question: does one regard `System.out.println` as one token or five?

Answer: it depends on the language, but in Java it's most appropriate to think of it as five (and that's what the language definition says). A good reason is that the language requires things like `println` or even `x.println` for a suitable variable `x` to refer to the same name. (We don't want to be matching substrings during later phases, only subtrees.)

Lexical Analysis (3)

Languages have evolved to use *regular expressions* (over the alphabet of characters and possibly EOF, end-of-file) for tokens. First noted in Algol60 by Backus and Naur.

Recall “Regular Languages and Finite Automata”:

Regular Expression ⇔ regular language ⇔ Finite Automaton;

(warning: the notes tend to write “Finite State Automaton”).

We’ll come back to whether this is deterministic or not, but in the meantime also recall the “subset construction” which, given a NDFA, gives a DFA.

Lexical Analysis (4)

So, lexical analysis is easy:

- Write or download a regular expression description of the tokens of your language;
- Create a DFA which accepts this language, and turn it into C/Java/ML code which (beware see next slide) emits a token every time it hits an accepting state.
- Job done.

There are even automatic tools which read in the regular expression description, construct the DFA and write the code for you (see Lex and Yacc later in the course).

Now we look at the details.

Lexical Analysis – Details

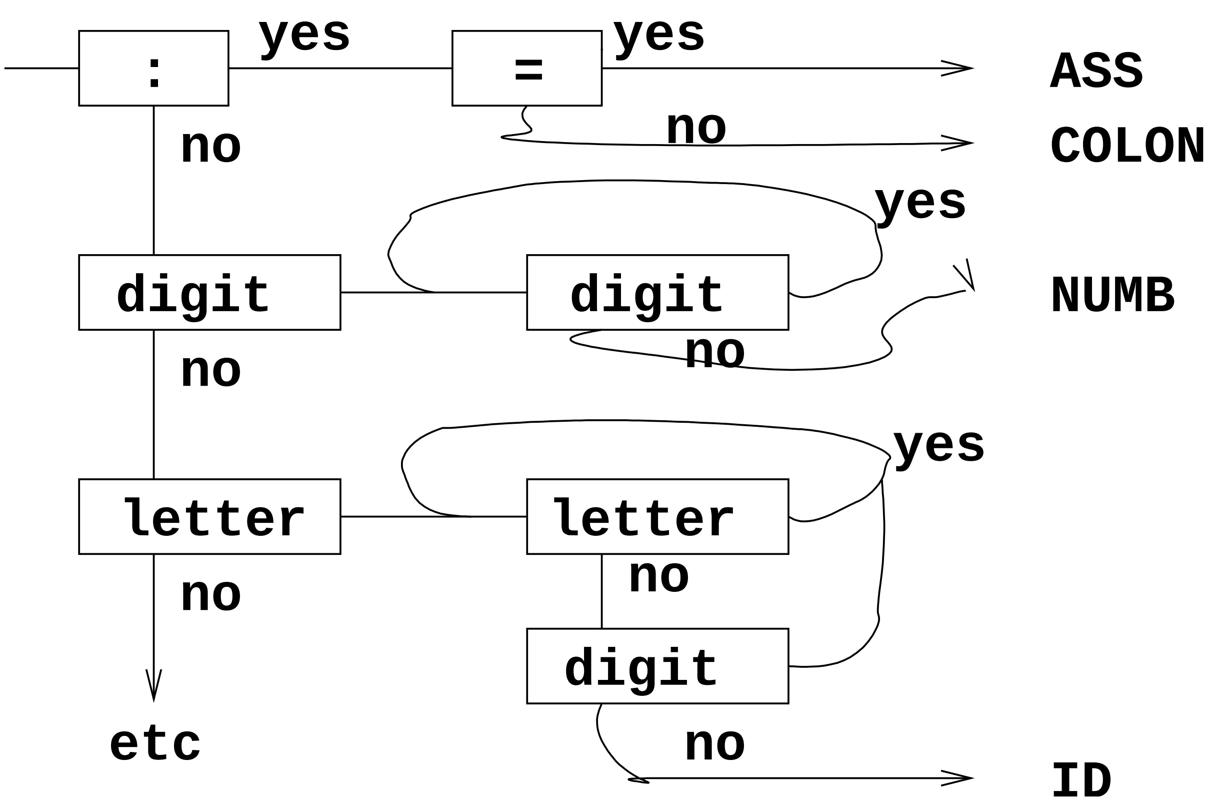
There's an implicit additional understanding in tokenisation beyond "accepting state" in DFAs. Consider input "`abc+1`" and the '*identifier*' token being defined by regular expression

$$(a \mid \cdots \mid z)(a \mid \cdots \mid z)^*$$

We don't want to accept `a`, then `b`, then `c` as three separate identifier tokens, even though '`a`', `ab` and `abc` all leave the DFA in an accepting state.

We want to accept the *longest* such string which remains in an accepting state, only emitting a token (for `abc`) when we see the '`+`'. Hence tokenisers generally have to read the character after the token and buffer it (or unread it), between calls.

Informal Example



Beware: while this picture is intuitive, the boxes represent transitions and the states are implicit; 'yes' consumes input and 'no' does not.

Example – floating point number syntax

Writing a *pure regular expression* for floating point numbers is hard in that it tends to be very large. Hence most formal notations for regular expressions have shortcuts, such as named intermediate definitions – just say d (digit) instead of writing out $0 \mid \cdots \mid 9$ lots of times.

So, let's define shorthand:

$$
\begin{array}{lllr}
s & = & + \mid - & \text{sign} \\
e & = & \texttt{E} & \text{exponent symbol} \\
p & = & . & \text{decimal point} \\
d & = & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 & \text{digit}
\end{array}
$$

Example – floating point number syntax (2)

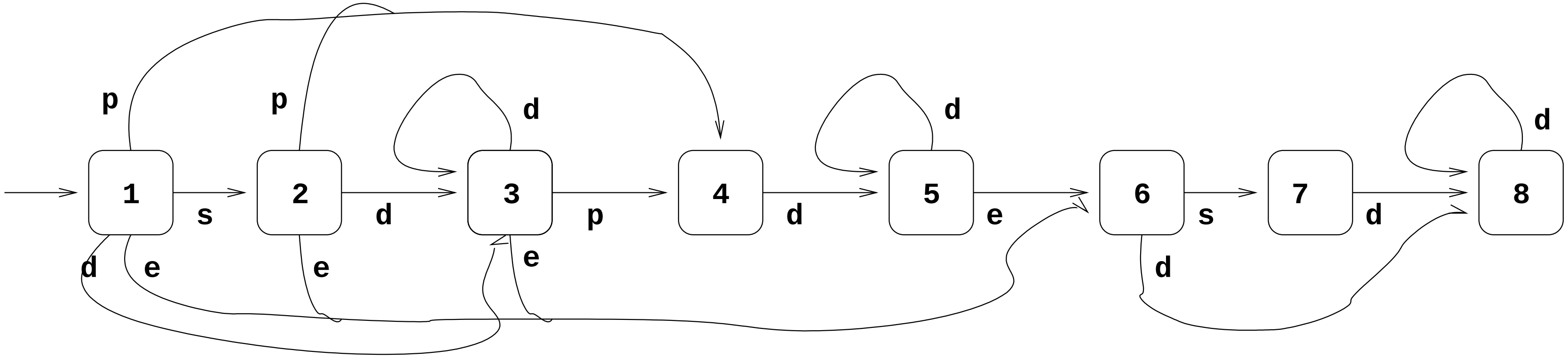
Now let's define a floating point number F step by step:

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: an digit string, a digit string preceded a point, or a point with digits either side, but disallows things like "`3.`". [You might pick a better/prettier definition.]

Example – floating point number syntax (3)

Can get the following DFA – but getting something so small requires careful state minimisation (RLFA or Hardware courses) possibly by a tool:



with states S3, S5 and S8 being accepting states (but see proviso earlier).

Mapping the DFA to code

One could represent the above DFA with lots of C labels and *gotos*, but it's simpler to represent it as a table—note that this also encodes "only accept at the first invalid character":

	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

Efficiency hack: I've also indexed by s, d etc. instead of characters!

One final problem with regular expressions

Suppose I say "`int` is a keyword, but identifiers are $[\texttt{a}-\texttt{z}]^*$" then these regular expressions overlap. It's really hard to write "sequences of `a` to `z` not including `int`" as a regular expression (try it!), so many notations and tools allow "first one (left-right) wins in case of a tie".

Moral: although tokens are just regular expressions, in practice these have lots of mathematical/programming short-hand to keep their size low and their expressivity high.

Lecture $4\frac{1}{2}$

Syntax Analysis or Parsing.

Parsing – big picture

- Regular Expressions/Finite Automata are too weak for this – e.g. they can't match brackets.
- We use the next strongest bit of theory "Context-free grammars".
- The syntax of programming languages is traditionally expressed using such grammars, often referred to as BNF, Backus-Naur Form.
- Logically we just repeat the progression of the previous lecture, but everything is much richer now, particularly we need typically to return a tree for a whole program, not just the next token.
- Need to learn a bit of theory before we can program a parser.

Grammars

A context-free grammar is a 4-tuple (T, N, S, R)

- T set of *terminal symbols* (things which occur in the source)
- N set of *non-terminal symbols* (names for syntactic elements)
- R set of *(production) rules*: $U \longrightarrow B_1\ B_2\ \cdots\ B_n$
- $S \in N$ is the *start symbol*

A *symbol* is either a T or an N.

We use U, V to range over N, and A, B to range over $N \cup T$

Sentences

Given a grammar (T, N, S, R)

- A *sentential form* is any sequence of symbols (in $N \cup T$) which can be produced from S by using a sequence of rules in R.
- A *sentence* is just a sentential form with all its symbols in T. (E.g. `1+2` but not `1+<expr>`).

Common notation (1)

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

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

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

There is various other shorthand, such as '$*$' for repetition, EBNF.

Common notation (2)

Alternatives:

- lower case for non-terminals, upper case for terminals (toy examples)
- `<expr>` etc for non-terminals, ordinary text for terminals (standards documents)
- ordinary identifiers for non-terminals, quoted text for terminals (input to `yacc` etc.)

Note that '$\longrightarrow$' is often written '`::=`'

Lecture 5

Syntax Analysis (continued)

Grammar Engineering

A grammar is *ambiguous* if a sentence can be produced in two different ways (using two different *derivations*—see later).

```
a)   S  ⟶  A B
     A  ⟶  a  |  a c
     B  ⟶  b  |  c b
          { a b,  a c b,  a c c b }

b)   C  ⟶  if E then C else C  |  if E then C
          if E then if E then C else C

c)   <sheepnoise> ⟶ "baa" | <sheepnoise> <sheepnoise>
          baa baa baa
```

This is a more serious version of the “overlapping token description” problem from last lecture.

Getting rid of ambiguity

Re-write grammar—usually to keep the same set of sentences, but where each sentence has a *unique* derivation. E.g.

```
(before) E ::= Num | E+E | (E)
(after: option 1) E ::= E + T | T  (left-associative)
                  T ::= Num | (E)
(after: option 2) E ::= T + E | T  (right-associative)
                  T ::= Num | (E)
(after: option 3) E ::= T + T | T  (non-associative)
                  T ::= Num | (E)
```

'Non-associative' disallows (say) `1+2+3`—forcing the user to parenthesise (and here we fortunately remembered to include parentheses in the syntax!).

Precedence

The grammar

```
E ::= E + T | E - T | E * T | E / T | E ^ T | T
T ::= Num | (E)
```

may be unambiguous, but it's probably not what you want—consider `2*3+4^5*6+7`. Want operators to have varying *precedence* (a.k.a. priority or binding power). E.g.

```
E ::= E + T | E - T | T   lowest prio, l-assoc
T ::= T * F | T / F | F   medium prio, l-assoc
F ::= P ^ F | P           highest prio, r-assoc
P ::= Num | (E)
```

Null productions

Note that we usually allow (e.g.)

```
E ::= X; E
E ::=
```

This encodes zero or more occurrences of "`X;`". The second rule is an *empty production* also written "`E` $\longrightarrow \epsilon$".

However, apart from particular uses such as the one above, empty productions can be hard to deal with when parsing ("there's an string of zero characters wherever one looks..."), and are often best avoided when possible.

Leftmost and Rightmost Derivations

I'll not make any real use of this, but it occurs in past exam questions.

A *leftmost derivation* (of a sentence from the start symbol) is when the sentence is generated by always taking the leftmost non-terminal and choosing a rule with which to re-write it. (The sequence of rules then exactly determines the string, at least for context free grammars). Given rules $S ::= A\texttt{+}A$ and $A ::= \texttt{1}$ we might have

$$S \longrightarrow A\texttt{+}A \longrightarrow \texttt{1+}A \longrightarrow \texttt{1+1}$$

A *rightmost derivation* is when the sentence is generated by always taking the rightmost non-terminal . . . .

Dual uses for grammars

- Describing languages—we've now just about done this.
- Parsing languages—how do I write a parser from a grammar?

Two answers to this question:

1. Just write it—i.e. encode the grammar as code.
2. Use a tool—this encodes the grammar as a table (data) along with a pre-implemented table interpreter.

We'll start with 1 and leave 2 to lecture 14.

Parsing by Recursive Descent

Easy in principle. For each non-terminal, `E` say, write a function `rdE()` which reads an `E`. We start by making rdE() return `void`—so this is a syntax checker—it says “OK” or “syntax error”.

So, given

```
F ::= P ^ F | P           highest prio, r-assoc
P ::= Num | (E)
```

Just write

```
int token;     // holds ‘current token’’ from lexing
void rdF() { rdP();
             if (token==’^’) { token=lex(); rdF(); }
           }
```

Parsing by Recursive Descent (2)

Similarly

```
P ::= Num | (E)
```

gives

```
void rdP() { if (token==Num) { token=lex(); }
             else if (token==’(’)
              { token=lex(); rdE();
                if (token==’)’) token=lex();
                else die("no ’)’");
              }
             else die("unexpected token");
           }
```

Parsing by Recursive Descent (3)

But what about:

```
   E ::= E + T | E - T | T   lowest prio, l-assoc
```

How do we know whether we are reading an `E` or a `T` first?

And do we really want to write the following?:

```
   void rdE() { rdE(); }
```

Answer: re-write to avoid *left recursion* in the grammar.

Lecture $6\frac{1}{2}$

Recursive Descent Continued; abstract syntax trees

Parsing by Recursive Descent (4)

What about:

```
E ::= E + T | E - T | T
```

How do we know whether we are reading an `E` or a `T` first?

Solution: find another (similar grammar) for the same language which (*a*) which only uses terminals to choose which way to parse and (*b*) has no left-recursion.

Note there's no general *algorithm* to do do this (indeed not always even possible), but *humans* can often do it (especially for common language cases).

Parsing by Recursive Descent (5)

Easy in this case

```
E ::= E + T | E - T | T
```

just means "any number of T's separated by '+' or '-' "; so re-write to

```
E′ ::= T + E′ | T - E′ | T      Cf. rule for  F
```

Bug: it associates wrongly—but this is not a problem for parse *checking* and we can fix the bug up later:

```
void rdE'() { rdT();
              if (token=='+') { token=lex(); rdE'(); }
              if (token=='-') { token=lex(); rdE'(); }
            }
```

Parsing by Recursive Descent (6)

A start on fixing the bug—rewrite

```
void rdE'() { rdT();
              if (token=='+') { token=lex(); rdE'(); }
              if (token=='-') { token=lex(); rdE'(); }
            }
```

as

```
void rdE'() { rdT();
              while (token=='+' || token=='-')
               { token=lex(); rdT(); }
            }
```

Abstract Syntax Trees

It's not much use just reporting yes/no whether a program matches a grammar—we want the derivation tree (which productions were used (backwards) to convert the string of terminals into the (non-terminal) sentence symbol.

If we've got an unambiguous grammar this is unique (unless the input is not a valid sentence).

The trouble is that we don't want all the incidental clutter of this—we don't want to know that the number 42 in a program is "a `Num` which is a `P` which is an `F` which is a `T` which is an `E`"

We want a tree showing the parsed expression's *abstract syntax*.

Abstract Syntax

Grammar for concrete syntax:

```
E ::= E + T | E - T | T
T ::= T * F | T / F | F
F ::= P ^ F | P
P ::= Num | (E)
```

Abstract syntax:

```
E ::= E + E | E - E | E * E | E / E | E ^ E | Num
        NB probably not (E)
```

Isn't this ambiguous? Yes—if we see it as a grammar on *strings*, but not if we see it as a specification of a datatype ("a tree grammar"). [That's why (for most languages) we can leave out `(E)`.]

Abstract Syntax (2)

What data structure represents such trees?

```
E ::= E + E | E - E | E * E | E / E | E ^ E | Num
```

In ML:

```
datatype E = Add of E * E | Sub of E * E |
             Mul of E * E | Div of E * E |
             Pow of E * E | Paren of E  |  Num of int;
```

In C: (over)

Abstract Syntax (3)

```
E ::= E + E | E - E | E * E | E / E | E ^ E | Num
```

In C:

```
typedef struct E E;             // In C allows to use E as a type name.
struct E {
  enum { E_Add, E_Sub, E_Mult, E_Div, E_Pow, 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;
};
```

Abstract Syntax (4)

```
E ::= E + E | E - E | E * E | E / E | E ^ E | Num
```

In Java, you can either simulate the C (considered bad O-O style) or write:

```
class E {}
class E_num extends E { int num; }
class E_paren extends E { E child; }
class E_add extends E { E left, right; }
class E_sub extends E { E left, right; }
```

Abstract Syntax Constructors

These are free in ML, but in C (or Java) we'd have to write them explicitly:

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

Reprise: Parsing by Recursive Descent (2)

For syntax *checking* we had

```
void rdP() { if (token==Num) { token=lex(); }
             else if (token==’(’)
              { token=lex(); rdE();
                if (token==’)’) token=lex();
                else die("no ’)’);
              }
             else die("unexpected token");
           }
```

A practical parser

For ease of reading/size I have cheated slightly by assuming the lexer returns single characters encoding the token it has just read (including ’n’ as a hack for Num):

```
E *RdP()
{   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;
        case ’i’: a = mkE_Name(lex_aux_string); lex(); return a;
        default:  error("unexpected token");
    }
}
```

Note the common hack whereby lex_aux_... returns additional details for a token with sub-structure.

Reprise: Parsing by Recursive Descent

```
  void rdF() { rdP();
               if (token=='^') { token=lex(); rdF(); }
             }
```

and, mutatis mutandis, `rdT()` (was `rdE'()`):

```
  void rdT() { rdF();
               while (token=='*' || token=='/')
                { token=lex(); rdF(); }
             }
```

A practical parser (2)

```
E *RdF()         r-assoc
{   E *a = RdP();
    switch (token)
    {   case '^': lex(); a = mkE_Pow(a, RdF()); return a;
        default:  return a;
    } }
E *RdT()         l-assoc
{   E *a = RdF();
    for (;;) switch (token)
    {   case '*': lex(); a = mkE_Mult(a, RdF()); continue;
        case '/': lex(); a = mkE_Div(a, RdF()); continue;
        default:  return a;
    } }
```

Remarks

- Recursive Descent performs *leftmost derivations* and so recursive descent parsers are often called LL-parsers (details not on course, see Wikipedia).
- Grammars in a form suitable for LL parsing are called LL(k) grammars.
- The tool *antlr* can automatically generates LL(k) parsers from a grammar.

Also, note that we would not have just one type for an abstract syntax tree in a real languages—we might only have one for *expressions*, but others for (say) declarations, commands etc. See `Expr`, `Cmd`, `Decl` in the introduction.

Lecture 7

Type Checking and Translating a parse tree into stack-based intermediate code.

Type Checking

Our language requires no type checking; all variables and expressions are of type `int` and variable name `<var>` and function names `<fnname>` are syntactically distinguished.

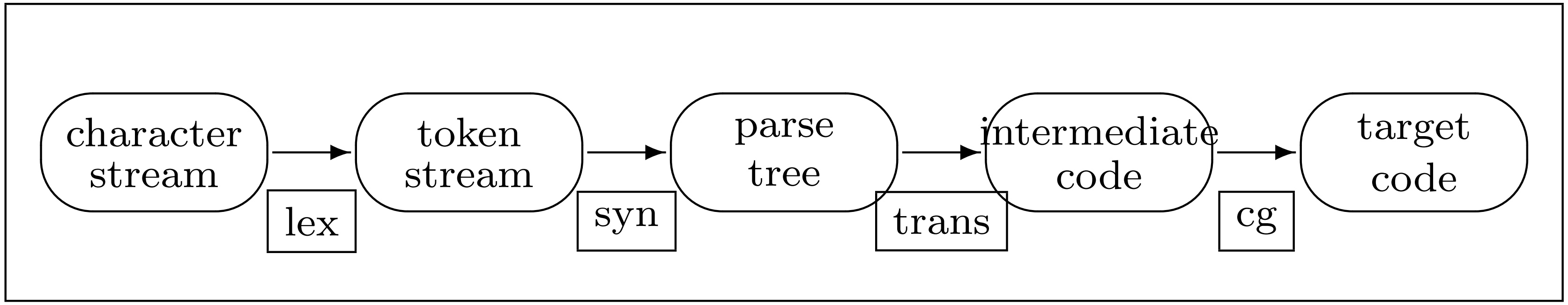
Real compilers (e.g. ML, Java) need type-checking generally to happen after syntax analysis. JVM code has separate `fadd` and `iadd` operations, so type information has to be resolved before or during translation to intermediate code. Java code like

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

must be compiled as if it were:

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

Type Checking

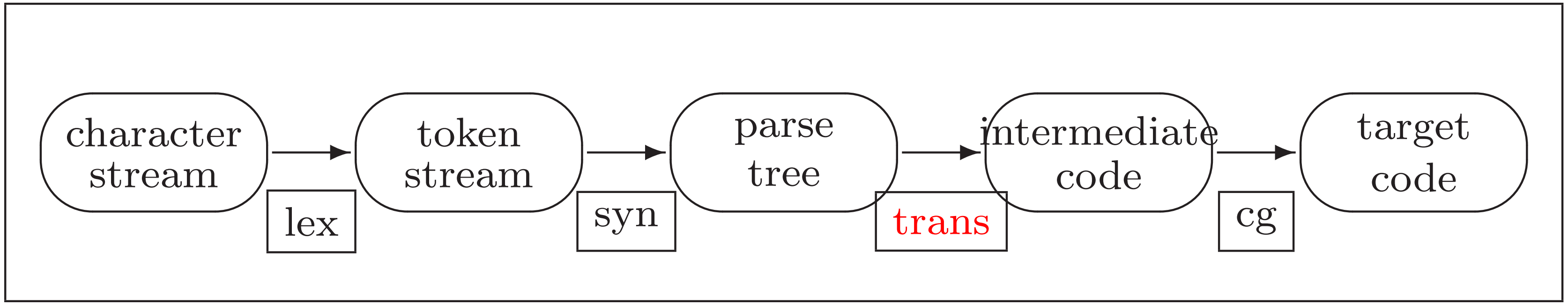


Type checking often not mentioned explicitly. Here you can think of it as being an arrow from *parse tree* to *parse tree* which checks types (rejecting ill-typed programs) and fixes up the parse tree.

We'll cheat and do it 'on-the-fly' during *trans*. . .

Trans: what do we have to do?

Convert the abstract syntax tree representation of a program into intermediate object code (here JVM code).



What do we have to do (2)?

The translation phase deals with

- the scope and allocation of variables,
- determining the type of all expressions,
- the selection of overloaded operators (type-based!), and
- generating the intermediate code.

What we want to happen:

Given, for example,

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

we want the translation phase to issue a series of calls of the following form for the declaration and initialisation of `y`:

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

We'll assume (1) `OP_xxx` above are enumeration constants representing opcodes and (2) `gen1()` and `gen2()` write the intermediate code instructions to a file or append them to some other data structure.

Reminder—flattening a tree in ML

```
datatype tree = Leaf of int | Branch of tree*tree;

fun flatten(Leaf n) = [n]
  | flatten(Branch(t,t')) = flatten t @ flatten t';

val test = Branch(Branch(Leaf 1, Leaf 2),
                  Branch(Leaf 3, Leaf 4));
flatten(test);
```

gives:

```
val it = [1,2,3,4] : int list
```

Reminder—flattening a tree (alternative)

```
datatype tree = Leaf of int | Branch of tree*tree;

fun walk(Leaf n) = (print (Int.toString n);
                    print ";")
  | walk(Branch(t,t')) = (walk t;
                           walk t');

val test = Branch(Branch(Leaf 1, Leaf 2),
                  Branch(Leaf 3, Leaf 4));
walk(test);
```

instead of making a list, this just prints the values in the leaves of the tree:

```
1;2;3;4;
```

Adjusting things a bit

```
datatype sourceop = Add | Mul;
datatype tree = Num of int | Diad of sourceop * tree * tree;

datatype jvmop = Iconst of int | Iadd | Imul;
fun trnop(Add) = Iadd
  | trnop(Mul) = Imul;

fun flatten(Num n) = [Iconst n]
  | flatten(Diad(binop,t,t')) = flatten t @ flatten t' @ [trnop binop];

val test = Diad(Add, Diad(Mul, Num 1, Num 2),
                     Diad(Mul, Num 3, Num 4));
flatten(test);
```

gives:

```
val it = [Iconst 1,Iconst 2,Imul,Iconst 3,Iconst 4,Imul,Iadd] : jvmop list
```

A postorder tree walk is pretty exactly a compiler from syntax trees to JVM code!

Tree walking

Essentially need one tree-walker for each type in the abstract syntax tree:

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

Here we'll mainly consider `trexp()` but the others are similar.

Dealing with names (and hence scoping)

```
class A {
  static int g;
  int n,m;     /* non-static members just for illustration */
  static int f(int x) { int y = x+1; return foo(g,n,m,x,y); }
}
```

Use a compile-time data structure to remember the names in scope—the *symbol table*. At the **`return`** this might be:

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

Symbol table

Symbol table is just an abstract data type.

Decl's and scope exit call methods to add/remove items from the symbol table, and we'll assume `trname()` looks up things in the table:

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

Rather sloppily for this year I'll assume it not only looks up the offset of name `s` but also emits it along with `op` using `gen2()`.

Translation of Expressions

```
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
        trname(OP_invokestatic, f)) // Compile call to f
  | ...
fun trexplist[] = ()
  | trexplist(e::es) = (trexp(e); trexplist(es));
```

Translation of Expressions (2)

Note the invariant: a call to `trexp()` emits code which when executed has the net result of pushing one item to the stack.

(Prove by induction assuming the result for sub-expressions.)

Lecture 7

Type Checking and Translating a parse tree into stack-based intermediate code (continued)

Translation of Conditional Expressions

```
fun trexp(Num(k))      = gen2(OP_iconst, k);
  | 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); // ... branch if b 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;
  | ...
```

Short-circuit boolean operations

Can't translate Java `&&` and `||` in the way we translate `+` etc. E.g. this is bad:

```
| trexp(Or(x,y))  = (trexp(x); trexp(y); gen1(...))
| trexp(And(x,y)) = (trexp(x); trexp(y); gen1(...))
```

Must 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)))
```

Relational Operators

Logically, code for Java relational operators (`Eq`, `Ne`, `Lt`, `Gt`, `Le`, `Ge`) is simply done by (e.g.):

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

and this is OK for exams. Sadly in reality JVM does not have such operations which push a boolean onto the stack, so we instead generate a branch around code which puts zero/one on the stack (just like `&&` and `||`):

```
// note the mapping for branch-false: Eq -> CmpNe etc.
| trexp(Eq(x,y))     = trboolop(OP_if_icmpne, x, y)
| ...
| trexp(Gt(x,y))     = trboolop(OP_if_icmple, x, y);
```

Relational Operators (2)

```
fun trboolop(brop,x,y) =
    let val p = ++label; 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;
```

This gives ugly code for `a>b?a:b` (first we branch to make 0/1 then we compare it with zero and branch again), but hey, it works. (It's the JVM's fault, and we could fix it up with a bit more work.)

Translation of declarations and commands

Rather 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’’)) = ...
```

Think also how variable declarations call methods to add names to the symbol table and also increment the compiler’s knowledge of the offset from FP of where to allocate the next local variable . . .

Labels vs addresses – the assembler

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:                 //  written "Lab 6" earlier
   iconst 0
label7:                 //  written "Lab 7" earlier
   ireturn
```

Labels vs addresses – the assembler (2)

But, given

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

and 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
```

Did I cheat? Only a little...

Labels vs addresses – the assembler (3)

The actual JVM *binary* 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 [or other] code to binary JVM [or other] code) is called `assembly` and the program which does it an *assembler*.

Labels vs addresses – the assembler (4)

While being a vital system component (and additional pass in compilation), assemblers are often disregarded in a simple explanation because they merely map text-form instructions to binary-form in a 1–1 manner.

One final remark: this assembly process is only done at the end of compilation—if we are intending to use the JVM code to generate further code 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 (disassemble it).

Type Checking

We've used `int` for everything so far. While types are better treated in part C of this course. What would happen if we also had type `float` (Java/C-style in which every variable is given a type when declared)?

We have additional JVM ops `fload`, `fstore` `fadd` etc.

So: put the type in the symbol table (along with global/local etc).

But how does $e + e'$ work? E.g. Java says that $e + e'$ has type `float` if e has type `int` and e' has type `float`.

Type Checking (2)

Have a data type representing language types with at least: `T_float` and `T_int`. Then write:

```
fun typeof(Num(k))       = T_int
  | typeof(Float(f))     = T_float
  | typeof(Var(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");
```

Type Checking (3)

When the type of an operand does not match the type required, then we insert a coercion: e.g. given `int x; float y;` then treat `x+y` as `((float)x)+y`. There is a JVM instruction `i2f`.

So `float f(int x, float y) { return x+y; }` generates

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

Type Checking (4)

Can either see type-checking as part of the translation phase, or as a separate phase which turns an abstract syntax tree (AST) into a type-decorated AST.

Note however, type-checking has to be done after scope-determination of variables, and the two phases would be

- scope resolution + type checking + coercion insertion
- translate typed (and scope-resolved) tree to intermediate code.

Lecture 8

Code Generation for target machine.

Code Generation for Target Machine

We'll do a cheap and cheerful blow-by-blow translation (see next year's course on how to do it better). First recall:

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

gives JVM code

```
iload 4        load x (4th load variable)
iconst 3       load 3
if_icmpgt L36  if greater 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)
```

Now can translate one at a time...

Code Generation for Target Machine (2)

```
movl    %eax,-4-16(%ebp) ; iload 4
pushl   %eax             ; ⟨ditto⟩
movl    %eax,#3          ; iconst 3
pushl   %eax             ; ⟨ditto⟩
popl    %ebx             ; if_icmpgt
popl    %eax             ; ⟨ditto⟩
cmpl    %eax,%ebx        ; ⟨ditto⟩
bgt     L36              ; ⟨ditto⟩
movl    %eax,-4-16(%ebp) ; iload 4
...
```

Oh yuk! (But it works.)

Code Generation for Target Machine (3)

What's wrong? We first load things into registers (OK), then push them (OFTEN WASTE), pop them back (OFTEN WASTE), and then operate on the registers (OK).

So use a compile-time data structure `stackcache` holding registers which should have been pushed but haven't …

[This can alternatively be seen as form of *peephole optimisation*: emit target machine instructions one-by-one but watch over finite-size window in the target-code replacing short sequences of instructions with simpler ones.]

Code Generation for Target Machine (4)

generated code		JVM op	scache
movl	%eax,-4-16(%ebp)	iload 4	[%eax]
movl	%ebx,#3	iconst 3	[%eax,%ebx]
cmpl	%eax,%ebx	if_icmpgt	[]
bgt	L36	⟨ditto⟩	[]
movl	%eax,-4-16(%ebp)	iload 4	[%eax]
negl	%eax	ineg	[%eax]
pushl	%eax	(flush/goto)	[]
b	L37	goto	[]
L36:			
movl	%eax,-4-16(%ebp)	iload 4	[%eax]
pushl	%eax	(flush/label)	[]
L37:			
popl	%eax	istore 7	[]
movl	-4-28(%ebp),%eax	⟨ditto⟩	[]

Code Generation for Target Machine (5)

What else could we do?

1. Allowing `stackcache[]` to remember integers as well as registers (quite easy)
2. Arrange that local variables are not repeatedly loaded by remembering (`regmem[]`) when they are in a register (quite easy)
3. Doing 2. over branches and labels (significantly harder)

Code Generation for Target Machine (6)

generated code		JVM op	scache	regmem
movl	%eax,-4-16(%ebp)	iload 4	[%eax]	[%eax=local4]
		iconst 3	[%eax,3]	[%eax=local4]
cmpl	%eax,#3	if_icmpgt	[]	[%eax=local4]
bgt	L36	if_icmpgt	[]	[%eax=local4]
negl	%eax	ineg	[%eax]	[]
b	L37	goto	[%eax]	[]
L36:		(label)	[]	[%eax=local4]
		iload 4	[%eax]	[%eax=local4]
L37:		(label)	[%eax]	[] (NB here)
movl	-4-28(%ebp),%eax	istore 7	[]	[%eax=local7]

A table-driven code-generator?

If we can generate *parsers* automatically from grammars, can't get generate code-generators directly from an instruction-set specification?

Harder.

- We don't worry about efficiency in parsing, but we do care about bad instruction sequences.
- Peephole optimisation harder to specify.
- Lots of special case tricks, e.g. `(x<=0)` can be generated into `(x>>>31)`.

Lecture 9

Object modules, linking etc.

Why do we need a linker?

When we compile e.g. C program

```
extern int printf(char *format, ...);
int main() { printf("Hello world\n"); return 0; }
```

We can generate code for everything except the call to `printf`. We can even generate the `call` (x86), or `jal` (MIPS) instruction but not the address to be branched to because we don't know it yet!

So, we generate an instruction like

```
jal   0
```

`or`

```
jal   .
```

and ask someone else (the linker) to finish off the job . . .

What is the role of object files (.o/.obj)?

- Holds binary output from compiler
- ELF is typical – and easy to understand in principle.
- A compiler or assembler can easily produce ELF as output.
- ELF is input to linker, along with libraries of object libraries.
- Output from linker is (usually) an executable file (.EXE on Microsoft Windows)
- ELF is sufficiently general that executables can also be represented, so an ELF linker takes ELF as user-inputs and library format – and also produces ELF as executable output (only one format to learn).

What makes an executable?

In ELF, to first approximation, an executable file is just one which has no remaining "undefined" symbols in its `.symtab`.

Yes, one of the object files has provided a "start address", often offset zero in the `.text segment`.

So, to run an executable, the operating system just reads in `.text` and `.data` (or maps the file via virtual memory) and branches to its start address.

ELF details

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 giving: (i) offset within `.text` to be relocated; and (iii) by which symbol
`.rela.data` data segment relocation table: list of (offset,symbol) pairs giving: (i) offset within `.data` to be relocated; and (iii) by which symbol
. . .

But how is a ‘symbol’ specified?

- A string? Too clumsy – multiple references to the same symbol?
- And how do we say a symbol is *defined here* as opposed to *missing and defined elsewhere*?

Answer:

- Use indexes into `.symtab` – a list of external symbols each specified as “undefined”, “defined as a code segment symbol” or “defined as a data segment symbol”.
- But, to keep these table entries of the same size we’ll store the strings in yet another table `.strtab`

The fine details of `symtab`/`strtab` are not examinable, but the principle of a symbol being defined here or referenced and defined elsewhere is!

ELF details (2)

. . .

`.symtab` symbol table:

List of external symbols (as triples) used by the module.

Each is (attribute, offset, symname) with attribute:
1. undef: externally defined, offset is ignored;
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

Phew!

The linker

What does a linker do?

- takes some object files as input, noting all undefined symbols.
- recursively searches libraries adding ELF files which define such symbols until all names defined ("library search").
- whinges if any symbol is undefined or multiply defined.

Then what?

- concatenates all code segments (forming the output code segment).
- concatenates all data segments.
- performs relocations (updates code/data segments at specified offsets) now all symbols are known.

Static versus Dynamic Linking

There are two approaches to linking:

Static linking (already done). Problem: a simple “hello world” program may give a 10MB executable if it refers to a big graphics or other library.

Dynamic linking Don’t incorporate big libraries as part of the executable, but load them into memory on demand. Such libraries are held as “.DLL” (Windows) or ”.so” (Linux) files.

Static versus Dynamic Linking

Pros and Cons of dynamic linking:

- Executables are smaller (and your disc doesn't have 100 copies of a graphics library, one in each executable).
- Bug fixes to a library don't require re-linking as the new version is automatically demand-loaded every time the program is run.
- Non-compatible changes to a library wreck previously working programs "DLL hell".

Dynamic Linking (mechanism)

Here's one mechanism, not quite what's used, but gives the idea: suppose "`sin()`" is to be dynamically loaded. Instead of linking in `sin()` we link in a 'stub' of the form:

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

Part C—how to compile other things

- Rvalues, Lvalues, aliasing
- Non-local non-global variables
- Binding/Scoping models (λ/OO); dynamic binding
- Exceptions
- Storage allocation, new, garbage collection
- OO inheritance (class members and methods)
- various type models
- misc, e.g. debugging tables.

Rvalues, Lvalues, aliasing

[Material taken from the notes.]

Copying (taking a snapshot) versus using the original variable.

An example: Java inner classes

```
class A {
  void f(int x) {
    class B {
      int get() { return x; }
      // void inc() { x++; }   // allowed? or not?
    }
    B p = new(B);
    x++;
    B q = new(B);
    if (p.get() != q.get()) println("x != x??");
};
```

Is ‘x’ copied or accessed in place? Language choice!

Lecture 10

A lambda-calculus evaluator

The power of Lambda

Lambda subsumes ML `let`, function definitions and even recursion:

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

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)
```

The power of Lambda (2)

This translation cannot immediately do ‘`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.

The power of Lambda (3)

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) ...   // NB: no f in body
in f(f, 5)
```

Here the call `g(g,n-1)` makes a recursive call of (non-recursive) `f` . . .

And this trick can be extended – giving the fixed point combinator Y.

The power of Lambda (4)

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)
```

(NB: this at least binds all the variables to the right places.)

Surprisingly at first, this `Y` can even be expressed directly in the lambda-calculus.

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

(Experts beware: this is the form for the call-by-value lambda calculus as befits the following interpreter.)

A lambda-calculus evaluator

Why do this? (In 2009–10 it is also be covered in “Computation Theory”.)

It is a simple language which directly models:

- nested function definitions e.g. $\lambda x.\lambda y.x + y$ and the nature of function values.
- dynamic types (the identity function can first be applied to an integer and then to another function).

It extends the simple interpreter in Part A of the notes.

A lambda-interpreter in ML

Syntax of the λ-calculus with constants in ML as

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

Values are of *either* integers or functions (closures):

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

A lambda-interpreter in ML (2)

Environments are just a list of (name,value) pairs as before:

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

and name lookup is natural:

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

A lambda-interpreter in ML (3)

The main code of the interpreter is as follows:

```
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;
```

A lambda-interpreter in ML (4)

Note particularly the way in which dynamic typing is handled (`Plus` and `Apply` have to check the type of arguments and make appropriate results). Also note the two different environments (`r`, `r_fromdef`) being used when a function is being called.

A fuller version of this code (with test examples and with the "tying the knot" version of Y) appears on the course web page.

Static Scoping and Dynamic Scoping

Modern programming languages normally look up free variables in the environment where the function was *defined* rather than when it 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 many dialects of Lisp. The difference is most easily seen in the following example:

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

Replacing `r_fromdef` with `r` in the interpreter moves from static to dynamic scoping!

Implementing the environment

Searching `lookup()` for names of variables is inefficient.

Before running the program we know, given a particular variable access, how many iterations lookup `lookup()` will take.

It's the number of variables declared (and still in scope) between the variable being looked up and where we are now. So we could use '*(de Bruijn) indices*' instead (translating with an additional compiler phase).

`Lam("x",Name("x"))` becomes `Lam("x",NameIndex(1))`

And we don't even need the names anymore:

`Lam("x",Name("x"))` ⟶ `Lam(NameIndex(1))`

`Lam("x",Lam("y",Name("x")))` ⟶ `Lam(Lam(NameIndex(2)))`

Implementing the environment (2)

This is *still* inefficient.

Accessing the nth element of a list by index is $O(n)$ – just like searching for the nth element by name!

What about using an array for the environment to get $O(1)$ access? Yes, but scope entry and scope exit then costs $O(n)$ with n variables in scope.

Practical idea: group variables in a single function scope putting their values in an array(*), and use a list of arrays for the environment. Scope entry and exit is just a `cons` or `tl`. Lookup costs $O(k)$ where k is the maximum procedure nesting.

(*) think of this array as a stack frame.

Implementing the environment (3)

BEWARE: this list of arrays/stack frames is not the same as the stack frames encountered by following the “Old FP” stored in the linkage information – it’s the static nesting structure.

Another point: De Bruijn indices become not a single integer but a pair (i, j) – meaning access the jth variable in the ith array.

Implementing the environment (4)

```
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)
```

Using ρ_1, ρ_2, ρ_3, ρ_4 for environments at the start of `f`, `g`, `h`, `k` (and ignoring function names themselves) gives scopes:

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`

Implementing the environment (5)

We put these entries in the symbol table.

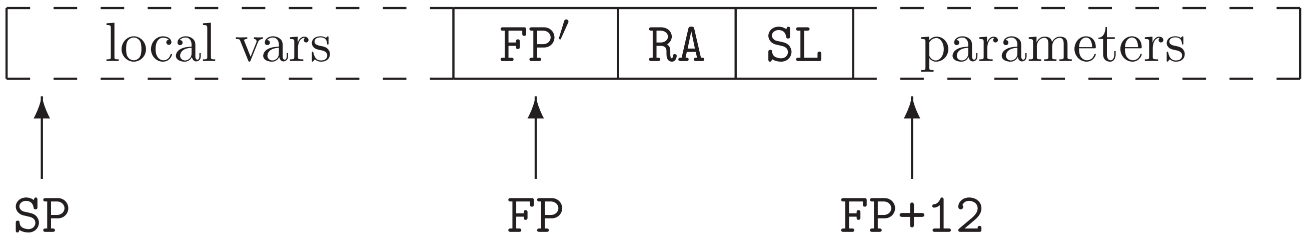
Now, given an access to a variable x (with 2D address (i, j) from a point at function nesting level d, instead of accessing x by name we can instead use 2D index (relative address) of $(d - i, j)$. For example, access to `c` (whose 2D address $(1, 3)$) is $(2, 3)$ in E (in environment ρ_3 of depth 3)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

Lecture 11

Static link method, ML free variables, etc.

Static Link

Add a pointer to stack of caller to linkage information:



SL is the 'static link'—a pointer to the frame of the *definer*

Note that FP′ is a pointer to the frame of the *caller*.

Talk in lectures about how these may not coincide.

But the static link does not always work

- It works *provided that* no function value is ever returned from a function (either explicitly or implicitly by being stored in a more global variable). This is enforced in many languages (particularly the Algol family—e.g. functions can be arguments but not result values).
- Remember function values need to be pairs (a *closure*) of function text (here a pointer to code), and some representation of the definer's environment (here its stack frame).
- So by returning a function we might be returning a pointer to a deallocated stack frame.

Why the static link method can fail

Consider a C stupidity:

```
int *nu(int x) { int a = x
                 return &a;
               }
int main() { int *p = nu(1);
             int *q = nu(2);
             foo(p,q);
           }
```

Why does this fail: because we return a pointer `&a` to a variable allocated in a stack which is deallocated on return from `nu()`. Probably `p` and `q` will point to the same location (which can't be both 1 and 2!). This location is also likely to be allocated *for some other purpose* in `main()`.

Why the static link method can fail (2)

Now consider a variant of this:

```
let f(x) = { let g(t) = x+t     // i.e. f(x) = λt.x+t
             in g }
let add1 = f(1)
let add2 = f(2)
...
```

Here the (presumed outer) `main()` calls `f` which has local variable `x` and creates function `g`—but the value of `g` is a closure which contains a pointer to the stack frame for `f`.

So, when `f` returns, its returned closure becomes invalid (dangling pointer to de-allocated frame containing `x`).

Again, `add1` and `add2` are likely to be identical values (BUG!).

Why the static link method can fail (3)

The core problem in both the above examples is that we want an allocation (either `&a` or a stack frame in a closure) to live longer than the call-return stack allows it too.

Solution: allocate such values in a separate area called a *heap* and use a separate de-allocation strategy on this—typically *garbage collection*. (Note that allowing functions to return functions therefore has hidden costs.)

It's possible (but rather drastic) to avoid deallocating stack frames on function exit, and allow a garbage collector to reclaim unused frames, in which the static link solution works fine again ("spaghetti stack").

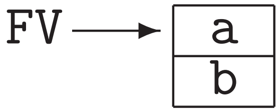
An alternative solution (Strachey)

So, if we want to keep a stack for function call/return we need to do better than storing pointers to stack frames in closures when we have function results.

One way to implement ML free variables to have an extra register `FV` (in addition to `SP` and `FP`) which points to the a heap-allocated vector of *values* of variables free to the current function:

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

Gives (inside `f`):



An alternative solution for ML

For reasons to do with polymorphism, ML likes all values to be (say) 32 bits wide.

A neat trick is to make a closure value not to be a pair of pointers (to code and to such a Free Variable List), but to be simply a pointer to the Free Variable List. We then store a pointer to the function code in offset 0 of the free variable list as if it were the first free variable.

NB. Note that this solution copies free variable values (and thus incorporate them as their current rvalues rather than their lvalues). We need to work harder if we want to update free variables by assignment (in ML the language helps us because no variable is every updated—only `ref` cells which are separately heap-allocated).

Parameter passing mechanisms

In C/Java/ML arguments are passed by *value*—i.e. they are copied (rvalue is transferred). (Mumble: Java class values have an implicit pointer compared to C.)

But many languages (e.g. Pascal, Ada, C#) 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) = ...
```

could pass an lvalue, thereby creating an alias rather than a copy.

Lecture 12

Parameter passing by source-to-source translation; Exceptions; Object-Orientation.

Implementing parameter passing

Instead of giving an explanation at the machine-code level, it's often simple (as here) to explain it in terms of 'source-to-source' translation (although this is in practice implemented as a tree-to-tree translation).

For example, we can explain C++ call-by-reference in terms of simple call-by-value in C:

```
int f(int &x) { ... x ... x ... }
main() { ... f(e) ... }
```

maps to

```
int f'(int *x) { ... *x ... *x ... }
main() { ... f'(&e) ... }
```

Implementing parameter passing (2)

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

implement as (all using C-style call-by-value):

```
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) ...
```

Labels and Jumps

Many languages provide `goto` or equivalent forms (`break`, `continue` etc.).

These normally turn into `goto` instructions in JVM or unconditional branches in assembly code—as we saw:

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

gave

```
    iload 4         load x (4th local 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)
```

Long Jumps

But what about:

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

If permitted, such jumps may exit a procedure, and so cannot just be implemented as an unconditional branch. They need to reset FP too (so that at the destination accesses to local variables access the correct frame).

Solution: implement such label values as a pair of pointers—one the code address of the destination label and the other the frame pointer of the destination— a *label closure*.

Long Jumps (2)

Such a *goto* is implemented as:

1. load the label value
2. load FP from the frame part of the label value
3. transfer control (load PC from the code pointer part of the label value)

Note: SP must also be updated (if not optimisied away) from saved or calculated value.

Note: as in accessing variables via static link, we can't use this method to jump back into procedures which have previously been exited (because the stack pointer part of the label value will have become invalid).

Why such esoteric stuff...?

Exceptions

For example given `exception foo;` we could implement

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

as (using `H` as a stack of active exception labels)

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

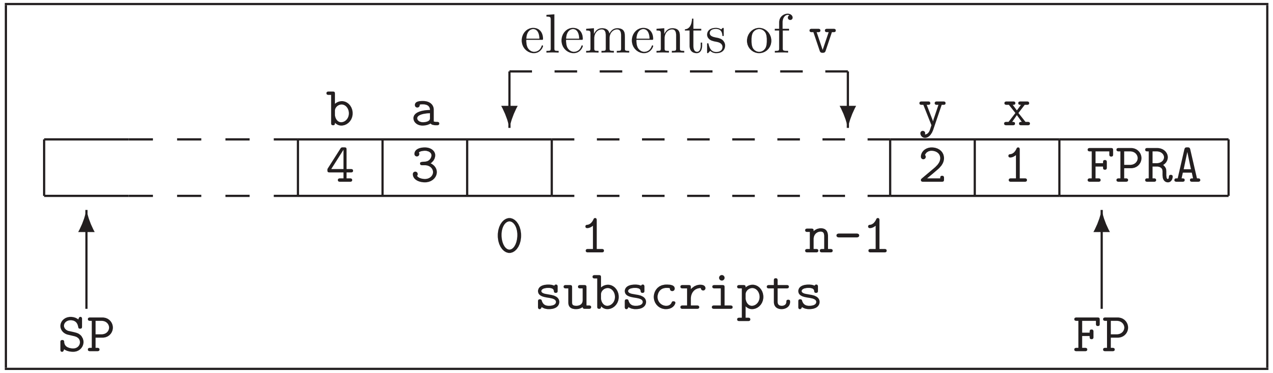
and the `doraise()` function looks like

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

Arrays

C-like arrays are typically allocated within a stack frame (array of 10 ints is just like 10 int variables allocated contiguously within a stack frame). [Java arrays are defined to be objects, and hence heap allocated—see later.]

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



Lecture 13

Objects, methods, inheritance.

NB: This material, including the run-time system slides, may extend into lecture 14 with the Tombstone diagrams only being covered briefly.

Class variables and access via ‘this’

A program such as

```
class C {
  int a;
  static int b;
  int f(int x) { return a+b+x;}
};
C exampl;
main() { ... exampl.f(3) ... }
```

can be mapped to:

```
int unique_name_for_b_of_C;
class C {
  int a;
};
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
};
main() { ... unique_name_for_f_of_C(exampl,3); ... }
```

Class variables and access via ‘this’ (2)

Using `this` (a pointer—provides lvalue of class instance):

```
class C {
  int a;
  static int b;
  int f(int x) { return a+b+x;}
};
C exampl;
main() { ... exampl.f(3) ... }
```

is mapped to:

```
int unique_name_for_b_of_C;
class C {
  int a;
};
int unique_name_for_f_of_C(C *this, int x)
{   return this->a                   // fixed offset within ‘this‘
         + unique_name_for_b_of_C  // global variable
         + x;                      // argument
};
main() { ... unique_name_for_f_of_C(&exampl,3); ... }
```

How does member inheritance work?

Missing slide – roughly...

Make sure at in

```
    class A { ... };
    class B:A { ... };
```

that the layout `B` is such that its initial segment is laid out as `A`.

This is necessary so that a pointer to a `B` may be validly be passed to a function expecting a pointer to an `A`.

But how does method inheritance work?

```
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?
       }
```

Java says ‘B’, but C (and our translation) says ‘A’!

To get the Java behaviour in C we must write `virtual`, i.e.

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

But how does method inheritance work? (2)

So, how do we implement `virtual` methods?

We need to use the *run-time* type of the argument of `g()` rather than the compiler-time type. So, values of type `A` and `B` must now contain some indication of what type they are (previously unnecessary). E.g. by translating to C of the form:

```
void f_A(struct A *this) { printf("I am an A"); }
void f_B(struct A *this) { printf("I am a B"); }
struct A { void (*f)(struct A *); } x = { f_A };
struct B { void (*f)(struct A *); } y = { f_B };
void g(A p) { p.f(&p); }
```

The use of a function pointer `g()` invokes the version of `f()` determined by the value of ‘`p`’ rather than its type.

Downcasts and upcasts

Consider Java-ish

```
class A { ...};
class B extends A { ... };
main()
 { A x = ...;
   B y = ...;
   x = (A)y;   // upcasting is always OK
   y = (B)x;   // only safe if x’s value is an instance of B.
 }
```

If you want downcasting (from a base class to a derived class) to be safe, then it needs to compile code which looks at the type of the value stored in `x` and raise an exception if this is not an instance of `B`. This means that Java class values must hold some indication of the type given to `new()` when they were created.

Practical twist: virtual function tables

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 at that type, although each type inheriting the given type will in general need its own table. (This cuts the per-instance storage overhead required for a class with 40 virtual methods from 160 bytes to 4 bytes at a cost of slower virtual method call.)

Virtual method tables can also have a special element holding the type of the value of instances; this means that Java-style safe-downcasts do not require additional per-instance storage.

C++ multiple inheritance

Looks attractive, but troublesome in practice...

Multiple inheritance (as in C++) so allows one to 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.)

Sounds neat, but...

C++ multiple inheritance (2)

Issues:

- How to pass a pointer to a `D` to a routine expecting a `C`? A `D` can't contain both a `B` and a `C` at offset zero. Run-time cost is an addition (guarded by a non-NULL test).
- Worse: what are `D`'s elements? We all agree with `b`, `c` or `d`. But are their one or two `a1` and `a2` fields? Amphibious craft: has only one weight, but maybe two number-plates!

C++ multiple inheritance (3)

C++ provides `virtual` keyword for bases. Non-virtual means duplicate; virtual means share.

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

Main use of `virtual` is to cause method selection by value instead of by type.

C++ multiple inheritance (4)

But this sharing is also expensive (additional pointers)—as C:

```
struct D { 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.

Yuk?

Run-time System

Run-time system: a specific library required to run the compiled code on a given operating system.

Provides resources that the compiler expects to find on the target machine which are not provided by the operating system.

Very short for the C language, hence C popular for embedded systems and OS kernels.

Typical components:

- start & exit routines,
- heap memory allocator,
- support for threads and exceptions,
- common functions missing from ISA (e.g. 64-bit divide),
- and perhaps a minimal console (teletype).

Run-time System: Start and Exit Routines

When the binary machine code is loaded by the OS, the OS transfers execution to the start routine. The offset of the start routine is either virtual address zero or given by a header in the machine code file (e.g. an ELF header).

```
start:   LDR sp, #0xFFFFF0000  ; Set up stack pointer
         LDR fp, sp            ; Same value in frame base
         LDR r0, #0x800000000  ; Start of the heap
         STR r0, heapptr
         JSR main              ; Call user's main
exit:    LDR r0, #TrapCode_exit
         TRAP #0               ; Exit to OS
```

When `main()` returns, the run-time system may call exit functions registered by any loaded libraries (not shown above) and then executes an exit system call, which never returns.

Run-Time System: System Calls

Compilers do not normally emit the system call instruction.

Every OS has a syscall convention (what arguments are in which registers).

```
; int fd = open("/dev/console", O_WRITE);
     LDR r0, #TrapCode_open
     LDR r1, #L1
     LDR r2, #2   ; hash sign denotes immediate addressing
     TRAP #0
     STR r0, [fp, -16]

; write(fd, "Hello World DJG", strlen("Hello World DJG"));
     LDR r0, #TrapCode_write
     LDR r1, [fp, -16]
     LDR r2, #L2
     LDR r3, #14
     TRAP #0

L1:  defb "/dev/console", 0
L2:  defb "Hello World DJG", 0
```

Run-Time System: System Calls Continued

More realistically, each system call is wrapped in an assembler stub and so they looks more like this:

```
open:   LDR r0, #TrapCode_open
        LDR r1, [fp, -4]
        LDR r2, [fp, -8]
        TRAP #0
        RET

write:  LDR r0, #TrapCode_write
        LDR r1, [fp, -4]
        LDR r2, [fp, -8]
        LDR r3, [fp, -12]
        TRAP #0
        RET
```

Run-time System: Heap allocation and `new`

The heap is a storage area (separate from code, static data, stack). Allocations are done by `new` (C: `malloc`); deallocations by `delete` (C++, C uses `free`).

In Java deallocations are implicit (done by *garbage collector*).

A simple C version of `malloc` (with various infelicities):

```
char heap[1000000], *heapptr = &heap[0];
void *malloc(int n)
{   char *r = heapptr;
    if (heapptr+n >= &heap[1000000]) return 0;
    heapptr += n;
    return r;
}
void free(void *p) {}
```

Better implementations make `free` maintain a list of unused locations (a ‘free-list’); `malloc` tries these first.

Garbage collection: implicit `free`

Simple strategy:

- `malloc` allocates from within its free-list (now it's simpler to initialise the free-list to be the whole heap); when an allocation fails call the garbage collector.
- the garbage collector first: scans 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'.
- the garbage collector second: (logically) calls the heap de-allocation function on these before returning. If garbage collection did not free any storage then you are out of memory!

Garbage collection: issues

- stops the system (bad for real-time). There are *concurrent garbage collectors*
- as presented this is a *conservative* garbage collector: nothing is moved. Therefore memory can be repeatedly allocated with (say) only every second allocation having a pointer to it. Even after GC a request for a larger allocation may fail. 'Fragmentation'.
- conservative garbage collectors don't need to worry about types (if you treat an integer as a possible pointer then no harm is done).
- There are also *compacting* garbage collectors. E.g. copy all of the reachable objects from the old heap into a new heap and then swap the roles. Need to know type information for every object to know which fields are pointers. (cf. 'defragmentation'.)

Lecture 14

Correctness, Types: static and dynamic checking, type safety.

Need for semantics

You can't compile a language unless you know exactly what it means ('semantics' is a synonym for 'meaning').

An example:

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

Should there be one counter for each `Elt` type, or just one counter of all conses? Entertainingly the languages Java and C$^\sharp$ differ on this.

Compiler Correctness

Suppose S, T are source and target languages of a compiler f.

See f as a function $f : S \to T$ (in practice f is implemented by a compiler program C written in an implementation language L)

Now we need semantics of S and T, written as $[\![\cdot]\!]_S : S \to M$ and $[\![\cdot]\!]_T : T \to M$ for some set of meanings M.

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

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

Tombstone diagrams

As above, let L, S, T, U be languages.

Write $f : S \to T$ for a function from S to T.
Similarly write $C : S \overset{L}{\rightsquigarrow} T$ for C a compiler from S to T written in language L.

Functions $f : S \to T$ and $g : T \to U$ can be composed to give $S \to U$.

So can compilers (output of one as input to the other):

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

Tombstone diagrams (2)

A semantics for L now turns a program in L into a function, in particular it now also has type $[\![\cdot]\!]_L : (S \overset{L}{\rightsquigarrow} T) \to (S \to T)$.

Let H be our host architecture.

Then $[\![\cdot]\!]_H$ means “execute a program in language H”.

The only useful compilers are ones of type $S \overset{H}{\rightsquigarrow} H$.

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 $S \overset{H}{\rightsquigarrow} H$.

Bootstrapping

We've designed a great new language U, and write a compiler C for U in U, i.e. $C : U \overset{U}{\rightsquigarrow} H$.

How do we make it useful, i.e. $U \overset{H}{\rightsquigarrow} H$?

Write a quick-and-nasty prototype compiler $U \overset{H}{\rightsquigarrow} H$ and use that to compile C to get a better compiler $U \overset{H}{\rightsquigarrow} H$.

This is called *bootstrapping* ("lift oneself up by one's own bootlaces").

Does this always work? Does it terminate? Is it unique?

Trojan compilers

It's not unique. Adjust C to make C' such that:

1. it miscompiles (say) the `login` program
2. it miscompiles the compiler so that when it compiles something which looks like the compiler then re-introduces bugs 1. and 2. if they have been removed from the source.

Now C has no visible bugs in the source code, but whenever it is compiled on a descendent of C' then the bug is propagated.

Source code audits don't find all security bugs (Ken Thompson's 1984 Turing Award paper)!

Type safety

Type safety (sometimes called “strong typing” – but this word has multiple meanings) means that you can’t cheat on the type system. This is often, but not always, dangerous.

- E.g. C: `float x = 3.14; int y = *(int *)&x;`
- E.g. C: `union { int i, int *p; } u; u.i=3; *u.p=4;`
- E.g. C++ unchecked downcasts
- Java and ML are type-safe.

Can be achieved by run-time or compile-time type checking.

See also: `http://en.wikipedia.org/wiki/Type_safety` and `http://en.wikipedia.org/wiki/Strong_typing`.

Dynamic types, Static types

- Dynamic: check types at run-type — like `eval()` earlier in the notes, or Lisp or Python. Get type errors/exceptions at run-time. Note run-time cost of having a “type tag” as part of every value.
- Static: check types at compile time and eliminate them at run-time.
  E.g. ML model: infer types at compile time, remove them at run-time and then glue them back on the result for top-level interaction.

Static types sometimes stop you doing things which would run OK with dynamic types.

```
if true then "abc" else 42
```

Untyped language

BCPL (precursor of C) provided an entertaining, maximally unsafe, type system with the efficiency of static types. There was one type (say 32-bit) *word*. A word was interpreted as required by context, e.g.

```
    let f(x) = x&5 -> x(9), x!5
```

('!' means subscripting or indirection, and e_1`->`e_2,e_3 is conditional.)

Arrays and structs become conflated too.

Static/Dynamic not enough

Static can be very inflexible (e.g. Pascal: have to write separate `length` functions for each list type even though they all generate the same code).

Dynamic gives hard-to-eliminate run-time errors.

Resolve this by polymorphism—either ML-style ('parametric polymorphism') OO-style ('subtype polymorphism')—this gives more flexibility while retaining, by-and-large, static type safety.

Static/Dynamic not enough (2)

- Parametric polymorphism can be implemented (as in most MLs) by generating just one version of (say) $I = \lambda x.x : \alpha \to \alpha$ even though its argument type can vary, e.g. $((II)7)$.
  Implementation requirement: all values must occupy the same space.

- Sub-type polymorphism: e.g. Java

  ```
  class A { ... } x;
  class B extends A { ... } y;
  ```

  Assigning from a subtype to a supertype (`x=y`) is OK. Allowing downcast requires run-time value checking (a limited form of dynamic typing).
  Note that a variable `x` above is of compile-time type `A`, but at run-time can hold a value of type `A`, `B`, or any other subtype.

Static/Dynamic not enough (3)

Overloading (two or more definitions of a function) is often called ‘ad-hoc’ polymorphism.

With dynamic typing this is a run-time test; with static typing operations like `+` can be resolved into `iadd` or `fadd` at compile time.

Source-to-source translation

(Said earlier this year.)

Many high-level constructs can be explaining in terms of other high-level (or medium-level) constructs rather than explaining them directly at machine code level.

E.g. my explanation of C++/Java in terms of C structs.

E.g. the

`while` e `do` e'

construct in Standard ML as shorthand (syntactic sugar) for

`let fun f() = if` e `then (`e'`; f()) else () in f() end`

Interpreters versus Compilers

Really a spectrum...

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.

A language is '*more compiled*' if less work is done at run-time.

Parts of a program that depend on input data cannot be evaluated at compile time.

The Debugging Illusion

It's easy to implement source-level debugging if we have a source-level interpreter.

It gets harder as we do more work at compile time (and have less information at run-time).

One solution: debug tables (part of ELF), often in 'DWARF' format, which enables a run-time debugger find out source corresponding to a code location or a variable.

Lecture $14\frac{1}{2}$

Parsing Theory and Practice

Phrase-Structured Grammars

A grammar is a 4-tuple (T, N, S, R)

- T set of *terminal symbols* (things which occur in the source)
- N set of *non-terminal symbols* (names for syntactic elements)
- R set of *(production) rules*: $A_1\ A_2\ \cdots\ A_m \longrightarrow B_1\ B_2\ \cdots\ B_n$ (there must be at least one N within the A_i)
- $S \in N$ is the *start symbol*

The only change from context-free grammars is the more permissive format of production rules; all other concepts are unchanged.

Chomsky Hierarchy (1)

So far we've seen one special case: the so-called "context-free grammars", or "type 2 grammars" in the Chomsky Hierarchy.

These have the LHS of every production just being a *single non-terminal.*

Chomsky Hierarchy (2)

type 0: no restrictions on rules. Turing-powerful.

type 1: ('context-sensitive grammar'). Rules are of form:

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

where A is a single non-terminal symbol and $n \neq 0$.

type 2: ('context-free grammar'). Most modern languages so specified (hence context-sensitive things—e.g. in-scope variables, e.g. C's typedef—are done separately).

type 3: ('regular grammar') Rules of form $A \longrightarrow a$ or $A \longrightarrow aB$ where a is a terminal and B a non-terminal.

`http://en.wikipedia.org/wiki/Chomsky_hierarchy`

Lecture 15

Parser Generators – table driven parsers

Automated tools (here: lex and yacc)

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

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

See `calc.l` and `calc.y` on course web-site for examples.

Lex

Example source `calc.l`

```
%%
[ \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 */
```

These rules become fragments of function `lex()`. Note how the chars in the token get assembled into `yytext`; `yylval` is what we called `lex_aux_int` earlier.

Lex (2)

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.

Lex (3)

One may also make regular expression definitions in this section, for instance:

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

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

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 outputs 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
```

Yacc input for calculator (1 of 3)

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

%token NUMBER

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

%%
```

Don't worry about the fine details!

Yacc input for calculator (2 of 3)

```
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
    ;

%%
```

Don’t worry about the fine details!

Yacc input for calculator (3 of 3)

```
#include "lex.yy.c"          /* lexer code */

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

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

Don’t worry about the fine details!

This example code is on the course web-site—just download it and say ‘make’.

Yacc and parse trees

To get a parse tree change the semantic actions from

```
expr: '(' expr ')'    { $$ = $2; }
    | expr '+' expr   { $$ = $1 + $3; }
    | NUMBER          // (implicit) $$ = $1;
    ;
%%
```

to

```
expr: '(' expr ')'    { $$ = $2; }
    | expr '+' expr   { $$ = mk_add($1,$3); }
    | NUMBER          { $$ = mk_intconst($1); }
    ;
%%
```

Parsing (info rather than examination)

- Recursive descent parsers are LL parsers (they read the source left-to-right and perform leftmost-derivations). In this course we made one by hand, but there are automated tools such as *antlr* (these make table-driven parsers for LL grammars which logically operate identically to recursive descent).
- Another form of grammar is the so-called LR grammars (they perform rightmost-derivations). These are harder to build by hand; but historically have been the most common way to make a parser with an automated tool. In this course we show how LR parsing is done.

But in principle, you can write both LL and LR parsers either by hand (encode the grammar as code), or generate them by a tool (tends to encode the grammar as data for an interpreter).

LR grammars

An LR parser is a parser for context-free grammars that reads input from **L**eft to right and produces a **R**ightmost derivation.

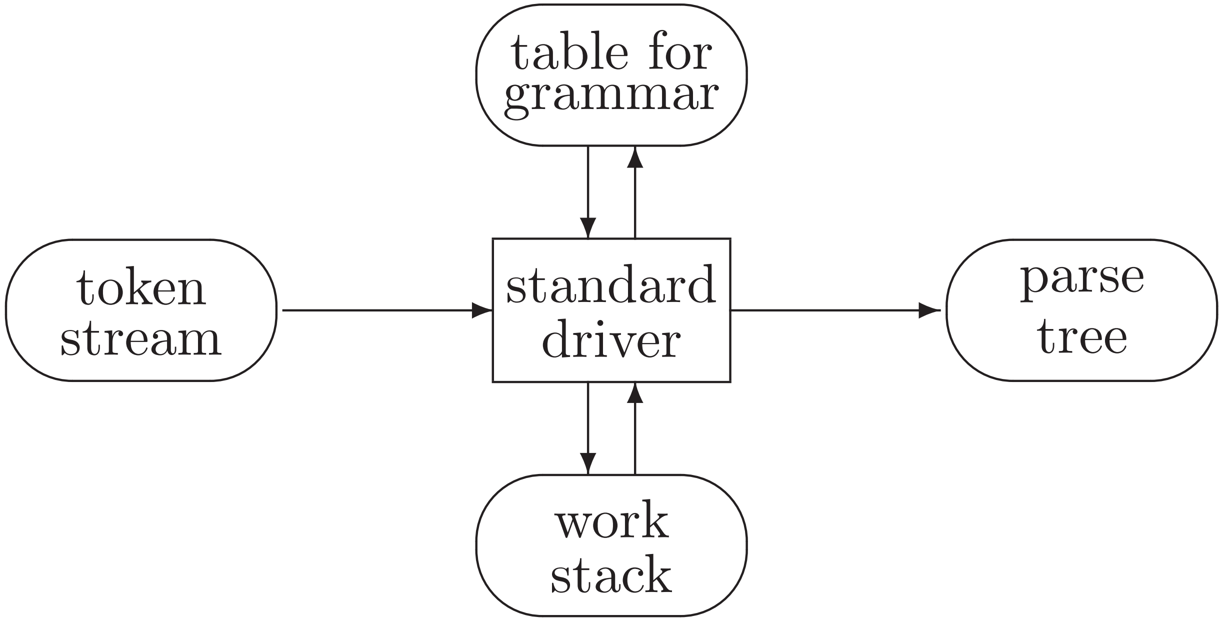
The term LR(k) parser is also used; k is the number of unconsumed "look ahead" input symbols used to make parsing decisions. Usually k is 1 and is often omitted. A context-free grammar is called LR(k) if there exists an LR(k) parser for it.

There are several variants (LR, SLR, LALR) which all use the same driver program; they differ only in the size of the table produced and the exact grammars accepted. We'll ignore these differences (for concreteness we'll use SLR(k)—Simple LR).

(See also `http://en.wikipedia.org/wiki/LR_parser` and `http://en.wikipedia.org/wiki/Simple_LR_parser`)

Table driven parsers

General idea:



In LR parsing, the table represents the *characteristic finite state machine* (CFSM) for the grammar; the standard driver (grammar independent) merely interprets this.

SLR parsing

So, we only have to learn:

- how do we construct the CFSM?
- what's the driver program?

SLR parsing – example grammar

To exemplify this style of syntax analysis, consider the following grammar (here E, T, P abbreviate 'expression', 'term' and 'primary'—a sub-grammar of our previous grammar):

#0	S	⟶	E	$\boxed{\texttt{eof}}$			
#1	E	⟶	E	+	T	l-assoc	+
#2	E	⟶	T				
#3	T	⟶	P	**	T	r-assoc	**
#4	T	⟶	P				
#5	P	⟶	i				
#6	P	⟶	(	E	)		

The form of production #0 defining the sentence symbol S is important. Its RHS is a single non-terminal followed by the special terminal symbol $\boxed{\texttt{eof}}$ (which occurs nowhere else in the grammar).

SLR parsing – items and states

An *item* is a production with a position marker (represented by .) marking some position on its right-hand side. There are four possible items involving production #1:

```
E  ⟶  .E   +   T
E  ⟶   E  .+   T
E  ⟶   E   +  .T
E  ⟶   E   +   T  .
```

So around 20 items altogether (there are 13 symbols on the RHS of 7 productions, and the marker can precede or follow each one). Think of the marker as a *progress indicator*.

A state (in the CFSM) is just a set of items (but not just any set ...).

SLR parsing – items and states (2)

- 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.

SLR parsing – items and states (3)

The first item set is formed by taking the initial item for the production defining the sentence symbol ($\texttt{S} \longrightarrow \texttt{.E}\ \boxed{\texttt{eof}}$) and then performing the closure operation, giving the item set:

$$\begin{array}{llllll}
\texttt{1: \{} & \texttt{S} & \longrightarrow & \texttt{.E} & \boxed{\texttt{eof}} & \\
 & \texttt{E} & \longrightarrow & \texttt{.E} & \texttt{+} & \texttt{T} \\
 & \texttt{E} & \longrightarrow & \texttt{.T} & & \\
 & \texttt{T} & \longrightarrow & \texttt{.P} & \texttt{**} & \texttt{T} \\
 & \texttt{T} & \longrightarrow & \texttt{.P} & & \\
 & \texttt{P} & \longrightarrow & \texttt{.i} & & \\
 & \texttt{P} & \longrightarrow & \texttt{.(} & \texttt{E} & \texttt{)} \\
\quad\texttt{\}} & & & & &
\end{array}$$

(Remember: item sets are the states of the CFSM.)

SLR parsing – items and states (4)

OK, so that's the first state, what are the rest?

- I tell you the transitions which gives new items; you then turn these into a state by forming the closure again.

SLR parsing – items and states (5)

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:

$$
\begin{array}{llll}
\mathtt{2:}\ \{ & \mathtt{S} & \longrightarrow & \mathtt{E}\ .\boxed{\mathtt{eof}} \\
 & \mathtt{E} & \longrightarrow & \mathtt{E}\ .+\ \ \mathtt{T} \\
\ \ \ \ \} & & &
\end{array}
$$

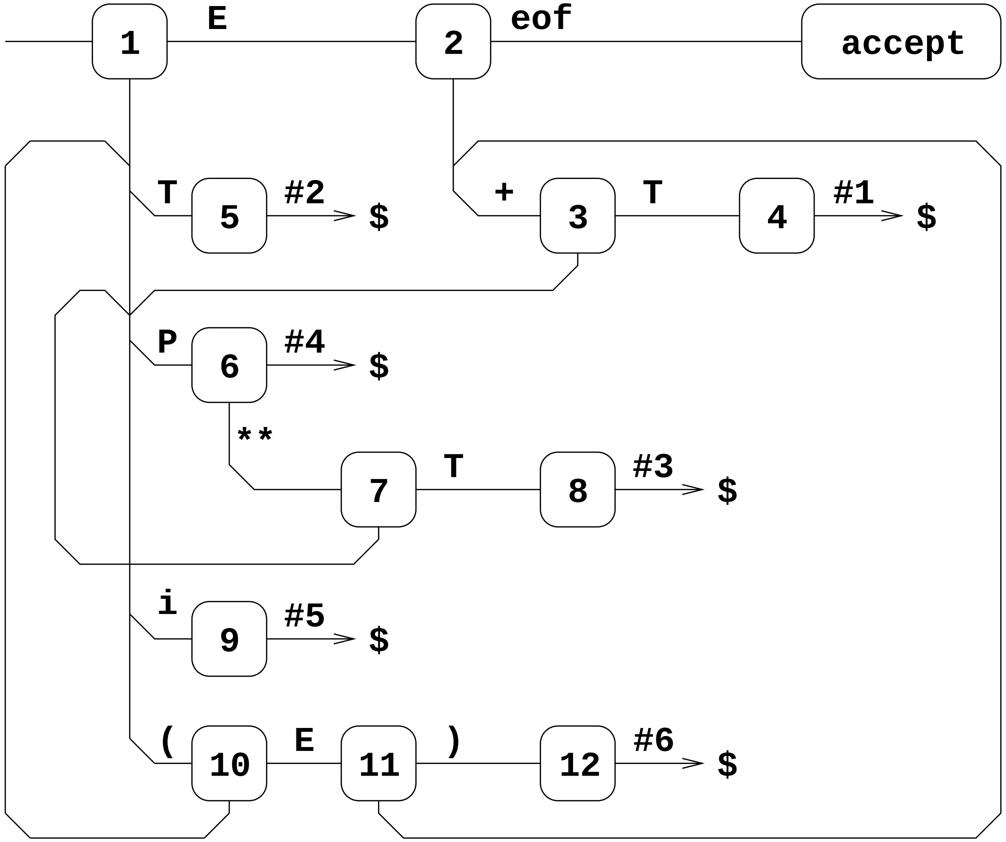
The other successor states are defined similarly, except that the successor of $\boxed{\mathtt{eof}}$ is always the special state `accept`. If a new item set is identical to an already existing set then the existing set is used.

SLR parsing – completed items and states

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.

CFSM for the grammar



Start to think what happens when I feed this `1**2+3**3`.

SLR(0) parser

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

1. If there is a transition from state i to state j under the terminal symbol k, then set `action`$[i, k]$ to `S`j.

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

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

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

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

Is our grammar SLR(0)?

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

state	action						goto		
	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	-	-	-

Is our grammar SLR(0)?

No: 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.

So: use lookahead—the construction then succeeds, so our grammar is SLR(1) but not SLR(0).

Lecture 16

SLR(1) grammars and LR driver code

LR and look-ahead

Key observation (for this grammar) is: after reading a `P`, the only possible sentential forms continue with

- the token `**` as part of `P ** T` (rule `#3`)
- the token '`+`' or '`)`' or '$\boxed{\texttt{eof}}$' as part of a surrounding `E` or `P` or `S` (respectively).

So a *shift* (rule `#3`) transition is always appropriate for lookahead being `**`; and a *reduce* (rule `#4`) transition is always appropriate for lookahead being '`+`' or '`)`' or '$\boxed{\texttt{eof}}$'.

In general: construct sets `FOLLOW`(U) for all non-terminal symbols U. To do this it helps to start by constructing `Left`(U).

Left sets

`Left`(U) is the set of symbols (terminal and non-terminal) which can appear at the start of a sentential form generated from the non-terminal symbol U.

Algorithm for `Left`(U):

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.

Left sets continued

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

Follow sets

Algorithm for `FOLLOW`(U):

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

Follow sets continued

For our example grammar, the `FOLLOW` sets are as follows:

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

SLR(1) table construction

Form 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).

Is our grammar SLR(1)?

Yes—SLR(1) is sufficient for our example grammar.

	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	-	-	-

Note now SLR(1) has no clashes (in SLR(0) S6/** clashed).

LR parser run-time code

This is the '*standard driver*' from last lecture.

We use a stack that contains alternately state numbers and symbols from the grammar, and a list of input terminal symbols terminated by $\boxed{\texttt{eof}}$. A typical situation:

```
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.

LR parser run-time code (2)

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

$$\texttt{1} \quad \texttt{|} \quad \langle \text{the whole program upto } \boxed{\texttt{eof}} \rangle$$

and then repeatedly applies the following rules until either a syntactic error is found or the parse is complete.

LR parser run-time code (3)

shift transition If `action[f, u] = Si`, then transform

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

to

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

reduce transition If `action[f, u] = #p`, and production `#p` is of length 3, say, necessarily `P` ⟶ `C D E` where `C D E` exactly matches the top three symbols on the stack. Then transform

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

to (assuming `goto[c, P] = g`)

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

LR parser run-time code (4)

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

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

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

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

LR parser sample execution

Example—parsing `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)

Why is this LR-parsing?

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

$$\texttt{E} \longrightarrow \texttt{E+T} \longrightarrow \texttt{E+P} \longrightarrow \texttt{E+i} \longrightarrow \texttt{T+i} \longrightarrow \texttt{P+i} \longrightarrow \texttt{i+i}$$

i.e. a *rightmost derivation*.

What about the parse tree?

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`.

(Exactly what we want!).

The end

Come along to “Optimising Compilers” in Part II if you want to know how to do things better.