Compiler Optimisation

Michael O'Boyle
mob@inf.ed.ac.uk
Room 1.06

January, 2014

School of informatics

Recommended texts

Two recommended books for the course

- *Engineering a Compiler Engineering a Compiler* by K. D. Cooper and L. Torczon. Published by Morgan Kaufmann 2003
- *Optimizing Compilers for Modern Architectures: A Dependence-based Approach* by R. Allen and K. Kennedy. Published Morgan Kaufmann 2001
- *Advanced Compiler Design and Implementation* by Steven S. Muchnick, published by Morgan Kaufmann. (extra reading - not required)

Additional papers especially for the later part of course - beyond books

Note Slides do not replace books. Provide motivation, concepts and examples not details.

How to get the most of the course

- Read ahead including exam questions and use lectures to ask questions
- L1 is a recap and sets the stage. Check you are comfortable
- Take notes.
- Do the course work and write well. Straightforward - schedule smartly.
- Exam results tend to be highly bi-modal
- If you are struggling, ask earlier rather than later
- If you don't understand - it's probably my fault - so ask!

Course Structure

- L1 Introduction and Recap
- L2 Course Work - again updated from last year
- 4-5 lectures on classical optimisation (Based on Engineering a Compiler)
- 5-6 lectures on high level/parallel (Based on Kennedy's book + papers)
- 4-5 lectures on adaptive compilation (Based on papers)
- Additional lectures on course work/ revision/ external talks/ research directions

Overview - Recap

- Compilation as translation and optimisation
- Compiler structure
- Phase order lexical, syntactic, semantic analysis
- Naive code generation and optimisation
- Next lecture looks at coursework and then focus on scalar optimisation -middle end

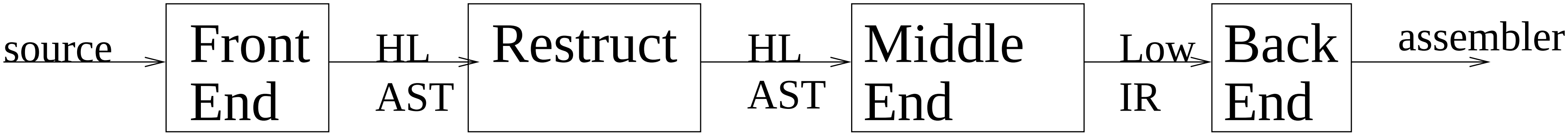
Compilation

- Compilers : map user programs to hardware. Translation - must be correct
- Hide underlying complexity. Machines are not Von Neumann
- Current focus : Optimisation go faster, smaller, cooler.
- 40+ years. In general undecidable, sub-problems at least NP-complete
- Try to solve undecidable problem in less time than execution!
- Tackling a universal systems problem: Java to x86, VHDL to netlists etc.
- Gap between potential performance and actual widening - compilers help?

Compilation as translation vs optimisation

- Modern focus is on exploiting architecture features
- Exploiting parallelism: instruction, thread, multi-core, accelerators
- Effective management of memory hierarchy registers,LI,L2,L3,Mem,Disk
- Small architectural changes have big impact
- Compilers have to be architecture aware -codesign e.g. RISC
- Optimisation at many levels source, internal formats, assembler

Compiler structure



- Front end translates “strings of characters” into a structured abstract syntax tree
- Middle end attempt machine independent optimisation. Can also include “source to source” transformations - restructurer - outputs a lower level intermediate format
- Many choices for IRs. Affect form and strength of later analysis or optimisation
- Backend: code generation, instruction scheduling and register allocation

Phase Order

- Lexical Analysis: Finds and verifies basic syntactic items - lexemes, tokens using finite state automata
- Syntax Analysis: Checks tokens follow a grammar based on a context free grammar and builds an Abstract Syntax Tree (AST)
- Semantic Analysis: Checks all names are consistently used. Various type checking schemes employed. Attribute grammar to Milner type inference. Builds a symbol table
- Optimisation + Code generation - later lectures

Lexical Analysis

Tokens include keywords *int*, identifiers *main_update* and constants *10E6*

Tokens defined using regular expression (RE), alphabet $\Sigma, |, *, \epsilon$

Input to scanner generators translated to NFA and simplified to DFA

Number of states = size of table. No impact on scan time complexity

Modern languages use white space as separators. DO i = 1 . 16 !

$$\ell = (a|b|...|z|A|B...|Z)$$

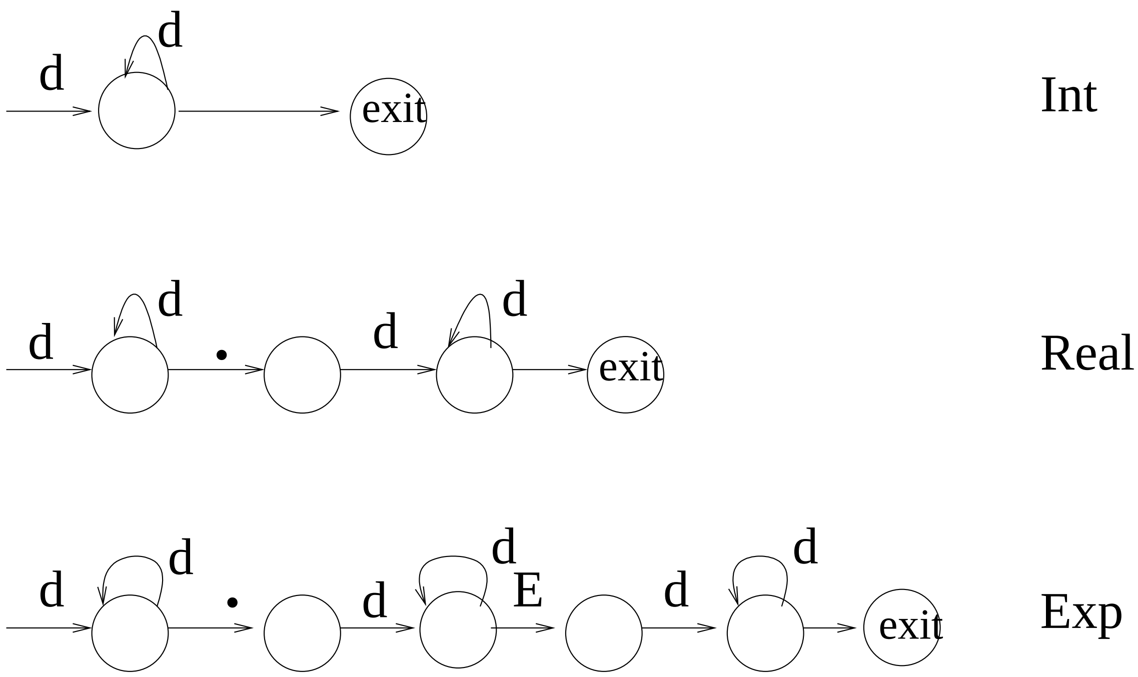
$$d = (0|1|..|9)$$

$$\text{integer} = dd*$$

$$\text{real} = dd * .dd*$$

$$\text{exp} = dd * .dd*\text{E}dd*$$

Lexical Analysis as deterministic finite automata



How are the following classified?

0, 01, 2.6, 2., 2.6E2 and 2E20

Syntax Analysis

Tokens form the words or terminals for the grammar.

RE not powerful enough. Use context free grammar (CFG) based on BNF variants

Next strip out syntax sugar and builds AST

Form of CFG determines type of language and parser family.

Top down vs Bottom up. Automation, error handling. Grammar rewriting

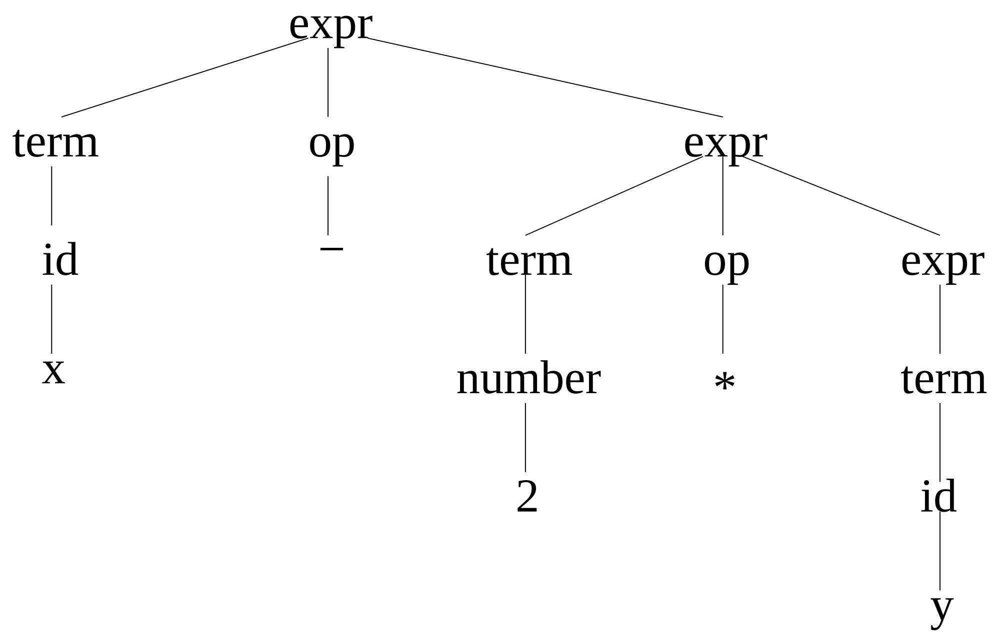
$$\text{expr} = \text{term (op expr)}$$

$$\text{term} = \text{number} \mid \text{id}$$

$$\text{op} = *| + |-$$

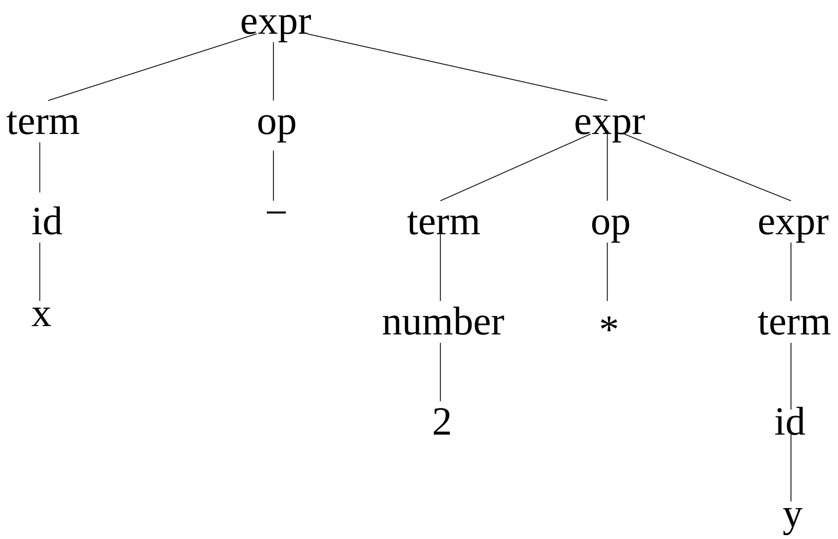
Example: parse $x - 2 * y$

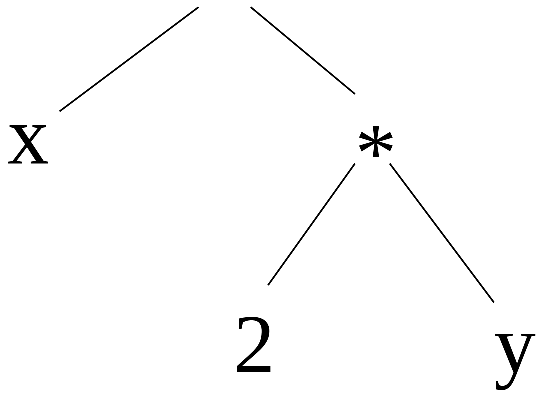
Syntax Analysis $x - 2 * y$



Impact on binding of operators. $x - 2 * y$ is parsed as $x - (2 * y)$.
What about $x * 2 - y$?

The Abstract Syntax Tree





The straightforward parse tree has many intermediate steps that can be eliminated

This cutdown tree is known as the abstract syntax tree and is a central data structure used by compilers

Semantic Analysis

- One name can be used for different vars depending on scope. Symbol table
- Type checking. Attribute grammars augment BNF rules with type rules

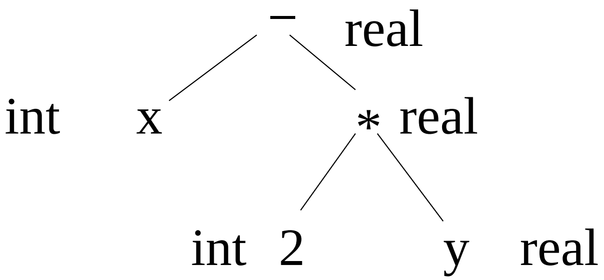
$$\begin{array}{ll} \text{expr} = \text{term (op expr)} & \text{expr.type} = \text{term.type } (F_{op} \text{ expr.type}) \\ \text{term} = \text{num} \mid \text{id} & \text{term.type} = \text{num.type} \mid \text{id.type} \\ \text{op} = *| + |- & F_{op} = F_*|F_+|F_- \end{array}$$

$x - 2 * y$ int:x, real:y, int <real

Difficult to add non-local knowledge : Ad-hoc syntax approaches, yacc

Higher order functional languages and dynamic typing make things interesting

Semantic Analysis $x - 2 * y$ int:x, real:y, int $<$real


− real

int x

∗ real

int 2

y real

F	int	real
int	int	real
real	real	real

Can be used for type inconsistencies/errors

F	int	real	double
int	int	real	double
real	real	real	$\perp$
double	double	$\perp$	double

Basic Code Generation

- Translate AST in to assembler. Walk through the tree and emit code based on node type
- Handle procedure calls and storage layouts. Assume activation record pointer in register r_0
- Loading value x into register r_3 - ILOC instruction set (EaC)

loadI @x $->r_1$	@x$->r_1$ (Not a mem op) Load address offset
loadA0 r_0, $r_1 -> r_3$	Mem$[r_0+r_1]->r_3$

Code Generation

Typical top down generator - left to right

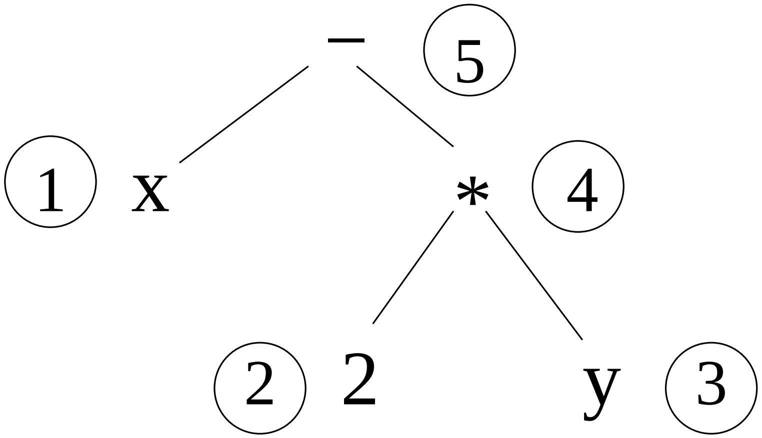
$$x - 2 * y$$

```
case op
 gen(left(node), right(node), op(node))
case identifier
 reg = nextreg()
 gen( loadI, offset(node),reg)
 gen( loadA0, r0,reg,reg)
case num
 gen(loadI, val(node),nextreg())
```

Optimisations include elimination of redundancy. Unnecessary loads

This scheme assumes unbounded registers - nextreg()

Code Generation



```
loadI @x -> r1         1
loadA0 r0,r1 -> r1     1
loadI 2 -> r2          2
loadI @y -> r3         3
loadA0 r0,r3 ->r3      3
mult r2,r3 -> r3       4
sub r1,r3->r3          5
```

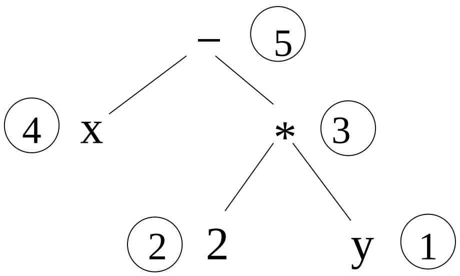
3 registers used

Optimisation

Generate more efficient code -eliminate redundancy

a = b*c +d	t = b*c
e = 2-b*c	a = t +d
	e = 2- t

Different traversal - less registers



```
loadI @y -> r1
loadA0 r0,r1 -> r1
loadI 2 -> r2
mult r2,r1 -> r1
loadI @x -> r2
loadA0 r0,r2->r2
sub r2,r1->r2
```

Machine models/ Optimisation goals

In first part of course

- Assume uni-processor with instruction level parallelism, registers and memory
- Generated assembler should not perform any redundant computation
- Should utilise all available functional units and minimise impact of latency
- Register access is fast compared to memory but limited in number . Use wisely
- Two flavours considered superscalar out-of-order vs VLIW: Dynamic vs static scheduling

Later consider multi-core architecture

Summary

- Compilation as translation and optimisation
- Compiler structure
- Phase order lexical, syntactic, semantic analysis
- Naive code generation and optimisation
- Next lecture course work
- **Monday next week Jan 20 lecture postponed**
- Then scalar optimisation - middle end