Compiler Optimisation

1 – Introductory Lecture

Hugh Leather
IF 1.18a
hleather@inf.ed.ac.uk

Institute for Computing Systems Architecture
School of Informatics
University of Edinburgh

2019

Textbooks

- **Engineering a Compiler** "EaC" by K. D. Cooper and L. Torczon. Published by Morgan Kaufmann 2003
- **Optimizing Compilers for Modern Architectures: A Dependence-based Approach** "CMA" 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)
- Plus research papers in last part of course

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

How to get the most out 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 dont understand - its probably my fault - so ask!

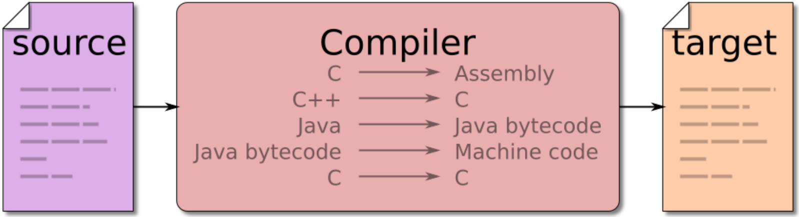
Course structure

- L1 Introduction and Recap
- L2 Course Work - again updated from last year
- 4-5 lectures on classical optimisation
  (Based on 🕮EaC)
- 5-6 lectures on high level/parallel
  (Based on 🕮CMA + papers)
- 4-5 lectures on adaptive compilation
  (Based on papers)
- Additional lectures on course work/ revision/ external talks/ research directions

Compilers review

What is a compiler?

- Translates a program from source language to target language
- Often target is assembly
- If target is a source language then “source-to-source” compiler



Compilers review

What is a compiler?

- Translates a program from source language to target language
- Often target is assembly
- If target is a source language then "source-to-source" compiler
- Compare this to an interpreter

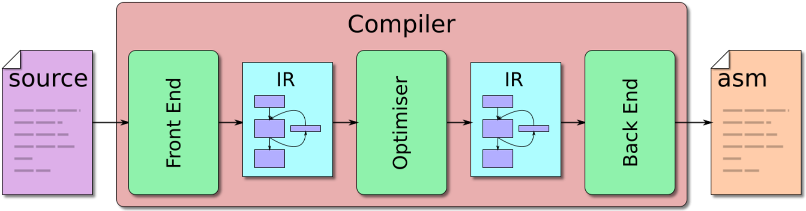
- Just translating not enough - must optimise!
- Not just performance - also *code size*, *power*, *energy*
- Generally *undecidable*, often *NP-complete*
- Gap between potential performance and actual widening
- Many architectural issues to think about
  - Exploiting parallelism: instruction, thread, multi-core, accelerators
  - Effective management of memory hierarchy registers,LI,L2,L3,Mem,Disk

Small architectural changes have big impact - hard to reason about

Program optimised for CPU with Random cache replacement.
What do you change for new machine with LRU?

Compilers review

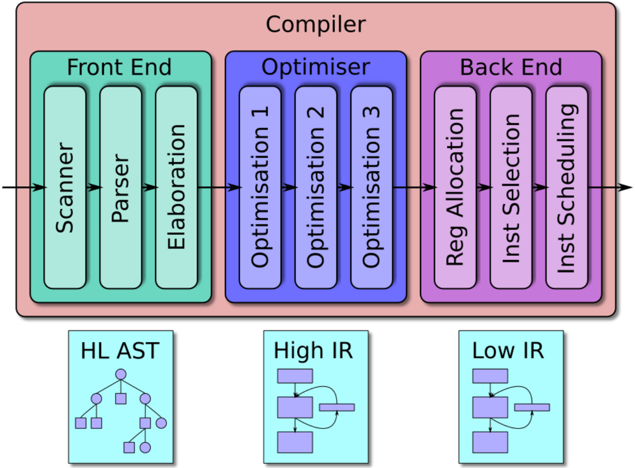
Typical compiler structure



- Front end takes string of characters into abstract syntax tree
- Optimiser does machine independent optimisations
- Back end does machine dependent optimisation and code generation

Compilers review

Typical compiler structure



- Work broken into small passes or phases
- Different IRs used - choice affects later analysis/optimisation

Front end stages

Lexical Analysis - Scanner

Finds and verifies basic syntactic items - lexemes, tokens using finite state automata

Syntax Analysis - Parser

Checks tokens follow a grammar based on a context free grammar and builds an Abstract Syntax Tree (AST)

Semantic Analysis - Parser

Checks all names are consistently used. Various type checking schemes employed. Attribute grammar to Milner type inference. Builds a symbol table

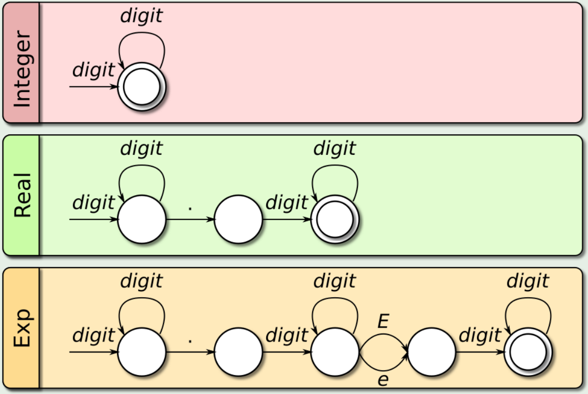
- Find keywords, identifiers, constants, etc. - these are tokens
- A set of rules are expressed as **regular expressions** (RE)
- Scanner automatically generated from rules [1]
- Transform RE $\rightarrow$ NFA $\rightarrow$ DFA $\rightarrow$ Scanner table

Example scanner rules

$$\ell \rightarrow (\text{`a'}|\text{`b'}|\ldots|\text{`z'}|\text{`A'}|\text{`B'}|\ldots|\text{`Z'})$$
$$\mathit{digit} \rightarrow (\text{`0'}|\text{`1'}|\ldots|\text{`9'})$$
$$\mathit{integer} \rightarrow \mathit{digit}\ \mathit{digit}^*$$
$$\mathit{real} \rightarrow \mathit{digit}\ \mathit{digit}^*\ \text{`.'}\ \mathit{digit}\ \mathit{digit}^*$$
$$\mathit{exp} \rightarrow \mathit{digit}\ \mathit{digit}^*\ \text{`.'}\ \mathit{digit}\ \mathit{digit}^*\ (\ \text{`e'}\ |\ \text{`E'}\ )\ \mathit{digit}\ \mathit{digit}^*$$

[1] Except in practically every real compiler, where all of this is hand coded

Token scanning example



How are the following classified?

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

- Each token has at least:
  - Type (Keyword, LBracket, RBracket, Number, Identifier, String, etc.)
  - Text value (and number value etc.)
  - Source file, line number, position
- White space and comments are typically stripped out
- Error tokens may be returned

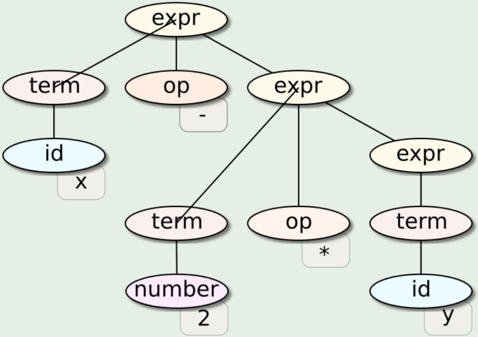
- REs not powerful enough
  (matched parentheses, operator precedence, etc)
- Syntax parser described by context free grammar (often BNF)
- Care must be taken to avoid ambiguity
  Generators (YACC, BISON, ANTLR) will complain

Example grammar

$$\begin{aligned} expr &\rightarrow term\ op\ expr \mid term \\ term &\rightarrow number \mid id \\ op &\rightarrow * \mid + \mid - \end{aligned}$$

Parse $x - 2 * y$

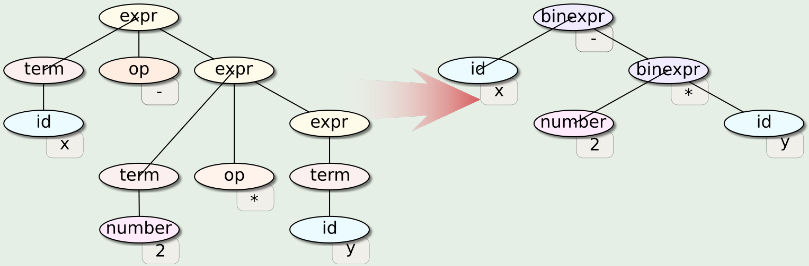
Parse tree for $x - 2 * y$



Notice this is parsed as $x - (2 * y)$
What about $x * 2 - y$?

- Parse trees have irrelevant intermediate nodes
- Removing them gives AST

Simplified parse tree for $x - 2 * y$



- Arbitrary CFGs can be expensive to parse
  Simple dynamic programming $T(n) = O(n^3)$
- Restricted classes of CFG with more efficient parsers

CFG classes

LR(1) — **L**eft to right scan, **R**ightmost derivation with **1** symbol lookahead

LL(1) — **L**eft to right scan, **L**eftmost derivation with **1** symbol lookahead; cannot handle left-recursive grammars

Others[a] — LR(k), LL(k), SLR(k), LALR(k), LR(k), IELR(k), GLR(k), LL(*), etc

[a] Some represent the same langauges

- Syntactic analysis produces **abstract** syntax tree
  Program may still be invalid
- Semantic analysis checks correct meaning and decorates AST
- Symbol tables record what names refer to at different scopes
- Semantic actions embedded in grammar allow arbitrary code during parsing
- Attribute grammars propagate information around AST

- Symbol tables provide two operations
  - lookup(name) retrieve record associated with name
  - insert(name, record) associate record with name
- Stack of symbol tables manages lexical scopes
- Lookup searches stack recursively for name

Scope example

```
(0) char* n = "N";
(0) char* fmt = "%d";
(0) void foo() {
(1)   int n = 10;
(2)   for( int i = 0; i < n; ++i ) {
(3)     printf(fmt, n);
(2)   }
(0) }
```

- Semantic actions allow arbitrary code to be executed during parsing
- Action executed only on successful parse of rule or
- Action provides conditional check to help parser choose between rules
- Side effects can cause trouble with back tracking

Semantic actions

$$\mathit{decl} \rightarrow \mathit{var}\ \mathit{id}\ =\ \mathit{expr} \qquad \texttt{\{symtab.insert(id.name)\}}$$

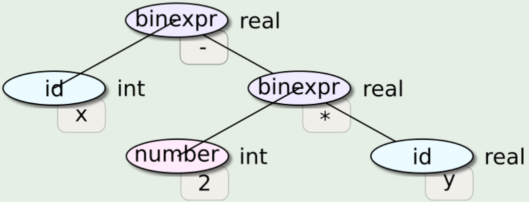
$$\mathit{expr} \rightarrow \mathit{number} \mid \mathit{id} \qquad \texttt{\{assert(symtab.exists(id.name)\}}$$

- Attribute grammar is a CFG with:
  - Attributes associated with each symbol
  - Semantic rules per production to move attributes
- Attributes can be inherited or synthesised
- Semantic rules can access global data structures, such as a symbol table

Attribute grammar example - types

$expr \rightarrow term\ op\ expr$	$expr.type = F_{op}(\ term.type,\ expr.type\ )$
$term \rightarrow num \mid id$	$term.type = num.type \mid id.type$
$op \rightarrow * \mid + \mid -$	$F_{op} = F_{*} \mid F_{+} \mid F_{-}$

Attribute grammar example - $x - 2 * y$ **x:int, y:real, int $<$ real**



F	int	real
int	int	real
real	real	real

Type matrices can encode errors

Example

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

Translate AST in to assembler - walk through the tree and emit code based on node type

ILOC instruction set[2,3]

Load constant 2 into r_2	
loadI 2 $\rightarrow r_2$	

Load value x into r_1	
loadI @x $\rightarrow r_1$	@x is offset of x
loadA0 $r_0, r_1 \rightarrow r_1$	Mem$[r_0 + r_1] \rightarrow r_1$

Add integers $r_1 = r_2 + r_3$
add $r_2, r_3 \rightarrow r_1$

[3] EaC Appendix A

[3] Assume activation record pointer in r_0

Typical top down generator - left to right - for simple expressions

Assume activation record pointer in register r_0

```
function gen( node ) :  Register
```

Typical top down generator - left to right - for simple expressions

Assume activation record pointer in register r_0

```
function gen( node ) :  Register
  case num
    r = nextreg()
    emit(loadI value( node ) → r)
    return r
```

Typical top down generator - left to right - for simple expressions

Assume activation record pointer in register r_0

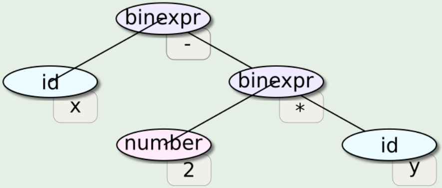
```
function gen( node ) :  Register
  case num
    r = nextreg()
    emit(loadI value( node ) → r)
    return r
  case id
    r = nextreg()
    emit( loadI offset( node ) → r)
    emit( loadA r0, r → r)
    return r
```

Typical top down generator - left to right - for simple expressions

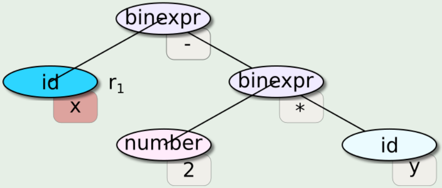
Assume activation record pointer in register r_0

```
function gen( node ) :  Register
  case num
    r = nextreg()
    emit(loadI value( node ) → r)
    return r
  case id
    r = nextreg()
    emit( loadI offset( node ) → r)
    emit( loadA r0, r → r)
    return r
  case binop( left, +, right )
    rL = gen( left ); rR = gen( right )
    emit( add rL, rR → rR )
    return rR
```

Generate code for $x - 2 * y$



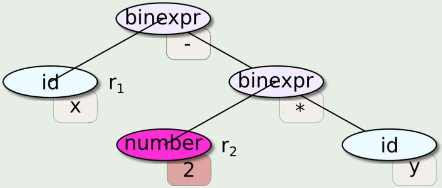
Generate code for $x - 2 * y$



loadI @x $\rightarrow r_1$

loadA0 $r_0, r_1 \rightarrow r_1$

Generate code for $x - 2 * y$

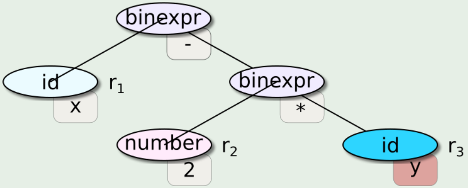


loadI @x $\rightarrow r_1$

loadA0 $r_0, r_1 \rightarrow r_1$

loadI 2 $\rightarrow r_2$

Generate code for $x - 2 * y$



loadI @x $\rightarrow r_1$

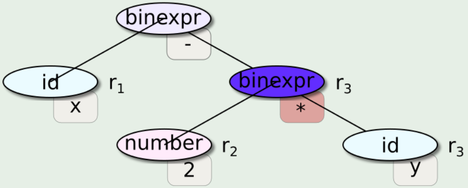
loadA0 $r_0, r_1 \rightarrow r_1$

loadI 2 $\rightarrow r_2$

loadI @y $\rightarrow r_3$

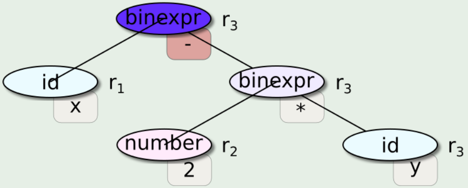
loadA0 $r_0, r_3 \rightarrow r_3$

Generate code for $x - 2 * y$



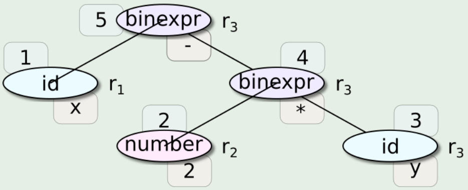
loadI @x $\rightarrow r_1$
loadA0 $r_0, r_1 \rightarrow r_1$
loadI 2 $\rightarrow r_2$
loadI @y $\rightarrow r_3$
loadA0 $r_0, r_3 \rightarrow r_3$
mult $r_2, r_3 \rightarrow r_3$

Generate code for $x - 2 * y$



loadI @x $\rightarrow r_1$
loadA0 $r_0, r_1 \rightarrow r_1$
loadI 2 $\rightarrow r_2$
loadI @y $\rightarrow r_3$
loadA0 $r_0, r_3 \rightarrow r_3$
mult $r_2, r_3 \rightarrow r_3$
sub $r_1, r_3 \rightarrow r_3$

Generate code for $x - 2 * y$

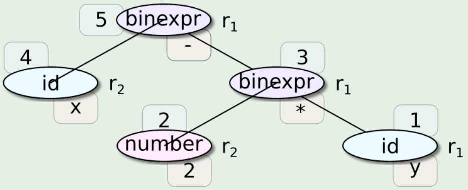


$\text{loadI } @x \rightarrow r_1$
$\text{loadA0 } r_0, r_1 \rightarrow r_1$
$\text{loadI } 2 \rightarrow r_2$
$\text{loadI } @y \rightarrow r_3$
$\text{loadA0 } r_0, r_3 \rightarrow r_3$
$\text{mult } r_2, r_3 \rightarrow r_3$
$\text{sub } r_1, r_3 \rightarrow r_3$

3 registers used

- Reducing number of registers used *usually* good
- Current traversal order left to right
  (r_L = gen(left); r_R = gen(right))
- Instead traverse child needing most registers first
- `nextreg()` must know which regs unused

Most registers first traversal order



loadI @y $\rightarrow r_1$
loadA0 $r_0, r_1 \rightarrow r_1$
loadI 2 $\rightarrow r_2$
mult $r_2, r_1 \rightarrow r_1$
loadI @x $\rightarrow r_2$
loadA0 $r_0, r_2 \rightarrow r_2$
sub $r_2, r_1 \rightarrow r_2$

2 registers used

- Expression, $x - 2 * y$ will have context
- Subtrees of expression already evaluated?

Common subexpression elimination

$a = \mathbf{2} * \mathbf{y} * z$	$\rightarrow$	$t = 2 * y$
$b = x - \mathbf{2} * \mathbf{y}$		$a = t * z$
		$b = x - t$

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
- Then scalar optimisation - middle end

PPar CDT Advert

EPSRC Centre for Doctoral Training in
Pervasive Parallelism

- 4-year programme:
  MSc by Research + PhD
- Research-focused:
  Work on your thesis topic from the start
- Collaboration between:
  - University of Edinburgh's School of Informatics
    - Ranked top in the UK by 2014 REF
  - Edinburgh Parallel Computing Centre
    - UK's largest supercomputing centre
- Research topics in software, hardware, theory and application of:
  - Parallelism
  - Concurrency
  - Distribution
- Full funding available
- Industrial engagement programme includes internships at leading companies

The biggest revolution in the technological landscape for fifty years

Now accepting applications!
Find out more and apply at:
pervasiveparallelism.inf.ed.ac.uk

THE UNIVERSITY *of* EDINBURGH
informatics

EPSRC
Engineering and Physical Sciences Research Council