Compiling Techniques
Lecture 2: The View from 35000 Feet
Christophe Dubach
Overview

- High-Level View of a Compiler
- The Front End
- The Back End
- The Optimiser
Tutorials

- Monday 1:10pm - AT 4.07 (Christophe Dubach)
- Monday 1:10pm - AT 4.14A (Björn Franke)
- Thursday 1:10pm - AT 4.07 (Christophe Dubach)
- Tutorials start in week 2 (next week)
- Group allocation on course website
High-Level View of a Compiler

- Must recognise legal (and illegal) programs
- Must generate correct code
- Must manage storage of all variables (and code)
- Must agree with OS & linker on format for object code
- Big step up from assembly language—use higher level notations
Traditional Two-Pass Compiler

- Use an intermediate representation (IR)
- Front end maps legal source code into IR
- Back end maps IR into target machine code
- Admits multiple front ends & multiple passes
- Typically, front end is $O(n)$ or $O(n \log n)$, while back end is NPC (NP-complete)
A Common Fallacy

- Can we build n x m compilers with n+m components?
- Must encode all language specific knowledge in each front end
- Must encode all features in a single IR
- Must encode all target specific knowledge in each back end
- Limited success in systems with very low-level IRs (e.g. LLVM)
The Front End

- Recognise legal (& illegal) programs
- Report errors in a useful way
- Produce IR & preliminary storage map
- Shape the code for the back end
- Much of front end construction can be automated
Scanner / Lexer

- Lexical analysis
  - Recognises words in a character stream
  - Produces tokens (words) from lexeme
  - Collect identifier information
  - Typical tokens include number, identifier, +, -, new, while, if

- Example:
  - \( x=x+y; \) becomes
  - `IDENTIFIER(x)` EQUAL `IDENTIFIER(x)` PLUS `IDENTIFIER(y)`

- Scanner eliminates white space (including comments)
Parser

- Recognises context-free syntax & reports errors
- Guides context-sensitive ("semantic") analysis (type checking)
- Builds IR for source program
- Hand-coded parsers are fairly easy to build
- Most books advocate using automatic parser generators
Context-Free Syntax

- Context-free syntax is specified with a grammar
  - SheepNoise → SheepNoise baa | baa
  - This grammar defines the set of noises that a sheep makes under normal circumstances
- It is written in a variant of Backus-Naur Form (BNF)
- Formally, a grammar $G = (S,N,T,P)$
  - $S$ is the start symbol
  - $N$ is a set of non-terminal symbols
  - $T$ is a set of terminal symbols or words
  - $P$ is a set of productions or rewrite rules ($P:N \rightarrow NuT$)
This grammar defines simple expressions with addition & subtraction over “number” and “id”

This grammar, like many, falls in a class called “context-free grammars”, abbreviated CFG
Derivations

- Given a CFG, we can derive sentences by repeated substitution

<table>
<thead>
<tr>
<th>Production</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>goal</td>
<td>expr</td>
</tr>
<tr>
<td>expr</td>
<td>op term</td>
</tr>
<tr>
<td>expr</td>
<td>op y</td>
</tr>
<tr>
<td>expr</td>
<td>- y</td>
</tr>
<tr>
<td>expr</td>
<td>op term - y</td>
</tr>
<tr>
<td>expr</td>
<td>op 2 - y</td>
</tr>
<tr>
<td>expr</td>
<td>+ 2 - y</td>
</tr>
<tr>
<td>term</td>
<td>+ 2 - y</td>
</tr>
<tr>
<td>x</td>
<td>+ 2 - y</td>
</tr>
</tbody>
</table>

- To recognise a valid sentence in some CFG, we reverse this process and build up a parse tree
Parse Trees

\[ x + 2 - y \]

This contains a lot of unneeded information.
Abstract Syntax Trees

- Compilers often use an abstract syntax tree
- This is much more concise
- ASTs are one kind of intermediate representation (IR)
The Back End

- Translate IR into target machine code
- Choose instructions to implement each IR operation
- Decide which value to keep in registers
- Ensure conformance with system interfaces
- Automation has been less successful in the back end
Instruction Selection

- Produce fast, compact code
- Take advantage of target features such as addressing modes
- Usually viewed as a pattern matching problem
  - ad hoc methods, pattern matching, dynamic programming
- Example: madd instruction
Register Allocation

- Have each value in a register when it is used
- Manage a limited set of resources
- Can change instruction choices & insert LOADs & STOREs (spilling)

- Optimal allocation is NP-Complete (1 or k registers)
  - Graph colouring problem
- Compilers approximate solutions to NP-Complete problems
Instruction Scheduling

- Avoid hardware stalls and interlocks
- Use all functional units productively
- Can increase lifetime of variables (changing the allocation)
- Optimal scheduling is NP-Complete in nearly all cases
- Heuristic techniques are well developed
Traditional Three-Pass Compiler

- Code Improvement (or Optimisation)
- Analyses IR and rewrites (or transforms) IR
- Primary goal is to reduce running time of the compiled code
  - May also improve space, power consumption, ...
- Must preserve “meaning” of the code
  - Measured by values of named variables
- Subject of UG4 Compiler Optimisation
The Optimiser

- Discover & propagate some constant value
- Move a computation to a less frequently executed place
- Specialise some computation based on context
- Discover a redundant computation & remove it
- Remove useless or unreachable code
- Encode an idiom in some particularly efficient form
Optimisation of Subscript Expressions

\[ \text{Address}(A(I,J)) = \text{address}(A(0,0)) + J \times \text{(column size)} + I \]

Does the user realize a multiplication is generated here?

DO I = 1, M
A(I,J) = A(I,J) + C
ENDDO

compute \text{addr}(A(0,J))
DO I = 1, M
add 1 to get \text{addr}(A(I,J))
A(I,J) = A(I,J) + C
ENDDO
Modern Restructuring Compiler

- Blocking for memory hierarchy and register reuse
- Vectorisation
- Parallelisation
- All based on dependence
- Also full and partial inlining
- Subject of UG4 Compiler Optimisation
Role of the Run-time System

- Memory management services
  - Allocate
    - In the heap or in an activation record (stack frame)
  - Deallocate
  - Collect garbage
- Run-time type checking
- Error processing
- Interface to the operating system
  - Input and output
- Support of parallelism
  - Parallel thread initiation
  - Communication and synchronization
Preview

- Introduction to Lexical Analysis
- Decomposition of the input into a stream of tokens
- Construction of scanners from regular expressions