Compiling Techniques
Lecture 2: The view from 35000 feet

Christophe Dubach

19 September 2017
Table of contents

1 High-level view

2 Front End
   • Passes
   • Representations

3 Back end
   • Instruction Selection
   • Register Allocation
   • Instruction Scheduling

4 Optimiser
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
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 two-pass compiler

- Can we build \( n \times m \) 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)
- Active research area (e.g. Graal, Truffle)
The Frontend

- 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
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: \( x = y + 2 \); becomes
  \[ \text{IDENTIFIER}(x) \ \text{EQUAL} \ \text{IDENTIFIER}(y) \ \text{PLUS} \ \text{CST}(2) \]
- Lexer eliminates white space (including comments)
The Parser

- Recognises context-free syntax & reports errors
- Hand-coded parsers are fairly easy to build
- Most books advocate using automatic parser generators
Guides context-sensitive (“semantic”) analysis

Checks variable and function declared before use

Type checking
Generates the IR used by the rest of the compiler.

Sometimes the AST is the IR.
Simple Expression Grammar

1. \( \text{goal} \rightarrow \text{expr} \)
2. \( \text{expr} \rightarrow \text{expr} \ \text{op} \ \text{term} \)
3. \( \text{term} \rightarrow \text{term} \)
4. \( \text{term} \rightarrow \text{number} \)
5. \( \text{term} \rightarrow \text{id} \)
6. \( \text{op} \rightarrow + \)
7. \( \text{op} \rightarrow − \)

- 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
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></td>
</tr>
<tr>
<td>1 expr</td>
<td></td>
</tr>
<tr>
<td>2 expr op term</td>
<td></td>
</tr>
<tr>
<td>5 expr op y</td>
<td></td>
</tr>
<tr>
<td>7 expr - y</td>
<td></td>
</tr>
<tr>
<td>2 expr op term - y</td>
<td></td>
</tr>
<tr>
<td>4 expr op 2 - y</td>
<td></td>
</tr>
<tr>
<td>6 expr + 2 - y</td>
<td></td>
</tr>
<tr>
<td>3 term + 2 - y</td>
<td></td>
</tr>
<tr>
<td>5 x + 2 - y</td>
<td></td>
</tr>
</tbody>
</table>

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

\[ x + 2 - y \]

This contains a lot of unnecessary information.
Abstract Syntax Tree (AST)

The AST summarises grammatical structure, without including detail about the derivation.

- 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
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
Modern optimisers are structured as a series of passes e.g. LLVM

- 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
Modern Restructuring Compiler

- Translate from high-level (HL) IR to low-level (LL) IR
- Blocking for memory hierarchy and register reuse
- Vectorisation
- Parallelisation
- All based on dependence
- Also full and partial inlining
- Not covered in this course
Role of the runtime 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 for parallelism (communication and synchronization)
Programs related to compilers

- **Pre-processor:**
  - Produces input to the compiler
  - Processes Macro/Directives (e.g. `#define`, `#include`)

- **Assembler:**
  - Translate assembly language to actual machine code (binary)
  - Performs actual allocation of variables

- **Linker:**
  - Links together various compiled files and/or libraries
  - Generate a full program that can be loaded and executed

- **Debugger:**
  - Tight integration with compiler
  - Uses meta-information from compiler (e.g. variable names)

- **Virtual Machines:**
  - Executes virtual assembly
  - Typically embedded a just-in-time (jit) compiler
Next lecture

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