Compiler Design

Lecture 2: The view from 35000 feet

Christophe Dubach
Winter 2022

Timestamp: 2022/01/03 14:39:17
First Compilers & Programming Languages
First “Compiler”: A-0 System. The term “compiler” was coined by Grace Hopper in the 1950s.

Automatic Coding for Digital Computers, Grace Hopper, 1955:

“Compiling [...] which withdraw sub-routines from a library and operate upon them, finally linking the pieces together to deliver, as output, a complete specific program.”

Actually more a sort of linker than what we call compiler today.
Fortran, 1957

- First “high-level” programming language.
- Fortran = Formula translation

Simple Fortran II program

```
C AREA OF A TRIANGLE — HERON’S FORMULA
C INPUT — CARD READER UNIT 5, INTEGER INPUT
C OUTPUT —
C INTEGER VARIABLES START WITH I, J, K, L, M OR N

READ(5,501) IA, IB, IC
501 FORMAT(3I5)
IF (IA.EQ.0 .OR. IB.EQ.0 .OR. IC.EQ.0) STOP 1
S = (IA + IB + IC) / 2.0
AREA = SQRT( S * (S - IA) * (S - IB) * (S - IC) )
WRITE(6,601) IA, IB, IC, AREA
601 FORMAT(4H A=, I5 , 5H B=, I5 , 5H C=, I5 ,
     8H AREA=,F10.2,13H SQUARE UNITS)
STOP
END
```


John Bakus,

IBM

source: Pierre.Lescanne, CC BY-SA 4.0, via Wikimedia Commons
Lisp, 1958

- Lisp = List processing language

Simple Lisp 1 program

```
((Y (LAMBDA (FN)
     (LAMBDA (X)
       (IF (ZEROP X) 1 (* X (FN (- X 1))))))))
```

Fortran and Lisp are the oldest, and most influential programming languages. Both are still in use today!

(Fortran) Imperative ↔ Functional (Lisp)
High-level view
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: 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)
Front End
Front End

PASSES
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
The Lexer

Lexical analysis

- Recognises words in a character stream
- Produces tokens (words) from lexeme
- Collect identifier information (e.g. variable names)
- Typical tokens include number, identifier, +, −, new, while, if
- Lexer eliminates white space (including comments)

Example: \( x = y+2; \)
becomes: \( \text{IDENTIFIER}(x) \text{ EQUAL IDENTIFIER}(y) \text{ PLUS CST}(2) \text{ SC} \)
The Parser

Parsing

- Recognises context-free syntax & reports errors
- Builds an AST (Abstract Syntax Tree)
- Hand-coded parsers are fairly easy to build
- Most books advocate using automatic parser generators

In the course project, you will build your own parser

- Will teach you more than using a generator!
- Once you know how to build a parser by hand, using a parser generator becomes easy
Semantic Analysis

- Guides context-sensitive ("semantic") analysis
- Checks variable and function declared before use
- Type checking

Type checking example:

```c
int foo(int a) = {...}
void main() {
    float f;
    f = foo(1,2); // type error
}
```
Generates the IR (Intermediate Representation) used by the rest of the compiler.

Sometimes the AST is the IR.
Front End

Representations
Simple Expression Grammar

S = goal
T = \{number, id, +, –\}
N = \{goal, expr, term, op\}
P = \{1, 2, 3, 4, 5, 6, 7\}

1. goal → expr
2. expr → expr op term
   | term
3. term → number
   | id
4. op → +
   | –

- 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
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 IR
Back end
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
Back end

Instruction Selection
Instruction Selection

- Produce fast, compact code
- Take advantage of target features such as addressing modes
- Usually viewed as a pattern matching problem

Example: \( d = a \times b + c \)

**Option 1**

- MUL rt, ra, rb
- ADD rd, rt, rc

**Option 2**

- MADD rd, ra, rb, rc
Back end

Register Allocation
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
Back end

Instruction Scheduling
- Avoid hardware stalls and interlocks
- Use all functional units productively
- Can increase lifetime of variables (changing the allocation)
- Optimality:
  - Optimal scheduling is NP-Complete in nearly all cases
  - Heuristic techniques are well developed
Optimiser
Compiler Optimization (or code improvement):

- Analyses IR and rewrites/transforms IR
- Primary goal is to reduce running time of the compiled code
  - May also improve code size, power consumption, ...
- Must preserve “meaning” of the code
  - Measured by values of named variables
- Subject of Compiler Optimisation course
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
- ...
Translate from high-level (HL) IR to low-level (LL) IR

- Blocking for memory hierarchy and data reuse
- Parallelisation (including vectorization)

All of above is based on data dependence analysis

- Also full and partial inlining

Compiler optimizations are not covered in this course
Role of the runtime system

- Memory management services
  - Allocate, in the heap or on the stack
  - 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 (real start of compiler course)
  - Decomposition of the input into a stream of tokens
  - Construction of scanners from regular expressions