Overview of a Compiler

Copyright 2017, Pedro C. Diniz, all rights reserved.
Students enrolled in the Compilers class at the University of Southern California have explicit permission to make copies of these materials for their personal use.
Implications:

- Must recognize 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

Implications

• 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 (better code)

Typically, front end is $O(n)$ or $O(n \log n)$, while back end is NPC
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
Responsibilities

- Recognize legal (and 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 Front End

Scanner

- Maps character stream into words—the basic unit of syntax
- Produces pairs — a word & its part of speech
  - \( x = x + y \) becomes \(<\text{id},x> = <\text{id},x> + <\text{id},y>\);
  - \( \text{word} \equiv \text{lexeme}, \text{part of speech} \equiv \text{token type} \)
  - In casual speech, we call the pair a \textit{token}
- Typical tokens include \textit{number, identifier, +, –, new, while, if}
- Scanner eliminates white space (including comments)
- Speed is important
The Front End

Parser
- Recognizes 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
The Front End

Context-free syntax is specified with a grammar

\[ \text{SheepNoise} \rightarrow \text{SheepNoise} \ baa \]
\[ \quad | \ baa \]

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 \((P : N \rightarrow N \cup T)\)
The Front End

Context-free syntax can be put to better use

1. $goal \rightarrow expr$
2. $expr \rightarrow expr\ op\ term$
3. $\ |\ term$
4. $term \rightarrow number$
5. $\ |\ id$
6. $op \rightarrow +$
7. $\ |\ -$

- 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

$S = goal$

$T = \{ number, id, +, - \}$

$N = \{ goal, expr, term, op \}$

$P = \{ 1, 2, 3, 4, 5, 6, 7 \}$
The Front End

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 recognize a valid sentence in some CFG, we reverse this process and build up a *parse*
The Front End

A parse can be represented by a tree (parse tree or syntax tree)

\[ x + 2 - y \]

This contains a lot of unneeded information.
The Front End

Compilers often use an *abstract syntax tree*

```
+     <id,y>
|--   
|   <id,x>   <number,2>
```

This is much more concise

ASTs are one kind of *intermediate representation (IR)*
The Back End

Responsibilities

- 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
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

This was the problem of the future in 1978
  - Spurred by transition from PDP-11 to VAX-11
  - Orthogonality of RISC simplified this problem
The Back End

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
- Optimal allocation is NP-Complete

Compilers approximate solutions to NP-Complete problems
The Back End

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 **Optimization**)

- Analyzes 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
Typical Transformations

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

➤ Optimization of Subscript Expressions in Fortran

Address(A(I,J)) = address(A(0,0)) + J * (column size) + I

Does the user realize a multiplication is generated here?
Example

Optimization of Subscript Expressions in Fortran

Address(A(I,J)) = address(A(0,0)) + J * (column size) + I

Does the user realize a multiplication is generated here?

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

Optimization of Subscript Expressions in Fortran

Address(A(I,J)) = address(A(0,0)) + J * (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 addr(A(0,J))
DO I = 1, M
    add 1 to get addr(A(I,J))
    A(I,J) = A(I,J) + C
ENDDO
Modern Restructuring Compiler

Typical Restructuring Transformations:
- Blocking for Memory Hierarchy and Register Reuse
- Vectorization
- Parallelization
- All based on dependence
- Also full and partial inlining
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
1957: The FORTRAN Automatic Coding System

- Six passes in a fixed order
- Generated good code
  - Assumed unlimited index registers
  - Code motion out of loops, with ifs and gotos
  - Did flow analysis & register allocation
1969: IBM’s FORTRAN H Compiler

- Used low-level IR (quads), identified loops with dominators
- Focused on optimizing loops (“inside out” order)
  
  Passes are familiar today
- Simple front end, simple back end for IBM 370
Classic Compilers

1975: BLISS-11 compiler (Wulf et al., CMU)

- The great compiler for the PDP-11
- Seven passes in a fixed order
- Focused on code shape & instruction selection

LexSynFlo did preliminary flow analysis
Final included a grab-bag of peephole optimizations

Basis for early VAX & Tartan Labs compilers
Classic Compilers

1980: IBM’s PL.8 Compiler

- Many passes, 1 front end, several back ends
- Collection of 10 or more passes
  - Repeat some passes and analyses
  - Represent complex operations at 2 levels
  - Below machine-level IR

Dead code elimination
Global cse
Code motion
Constant folding
Strength reduction
Value numbering
Dead store elimination
Code straightening
Trap elimination
Algebraic reassociation
1980: IBM’s PL/8 Compiler

- Many passes, 1 front end, several back ends
- Collection of 10 or more passes
  Repeat some passes and analyses
  Represent complex operations at 2 levels
  Below machine-level IR

Multi-level IR has become common wisdom
Classic Compilers

1986: HP’s PA-RISC Compiler

• Several front ends, an optimizer, and a back end
• Four fixed-order choices for optimization (9 passes)
• Coloring allocator, instruction scheduler, peephole optimizer
Classic Compilers

1999: The SUIF Compiler System

Another classically-built compiler
- 3 front ends, 3 back ends
- 18 passes, configurable order
- Two-level IR (High SUIF, Low SUIF)
- Intended as research infrastructure
1999: The SUIF Compiler System

Another classically-built compiler
- 3 front ends, 3 back ends
- 18 passes, configurable order
- Two-level IR (High SUIF, Low SUIF)
- Intended as research infrastructure

SSA construction
Dead code elimination
Partial redundancy elimination
Constant propagation
Global value numbering
Strength reduction
Reassociation
Instruction scheduling
Register allocation
Another classically-built compiler
• 3 front ends, 3 back ends
• 18 passes, configurable order
• Two-level IR (High SUIF, Low SUIF)
• Intended as research infrastructure

Data dependence analysis
Scalar & array privatization
Reduction recognition
Pointer analysis
Affine loop transformations
Blocking
Capturing object definitions
Virtual function call elimination
Garbage collection
2000: The SGI Pro64 Compiler  (now Open64 from Intel)

Open source optimizing compiler for IA 64

- 3 front ends, 1 back end
- Five-levels of IR
- Gradual lowering of abstraction level

**Interprocedural Classic Analysis**
- Inlining (user & library code)
- Cloning (constants & locality)
- Dead function elimination
- Dead variable elimination
2000: The SGI Pro64 Compiler (now Open64 from Intel)

- 3 front ends, 1 back end
- Five-levels of IR
- Gradual lowering of abstraction level

Open source optimizing compiler for IA 64

Loop Nest Optimization
- Dependence Analysis
- Parallelization
- Loop transformations (fission, fusion, interchange, peeling, tiling, unroll & jam)
- Array privatization
2000: The SGI Pro64 Compiler (now Open64 from Intel)

Open source optimizing compiler for IA64

- 3 front ends, 1 back end
- Five-levels of IR
- Gradual lowering of abstraction level

Global Optimization
SSA-based analysis & opt’n
Constant propagation, PRE, OSR+LFTR, DVNT, DCE
(also used by other phases)
2000: The SGI Pro64 Compiler (now Open64 from Intel)

Open source optimizing compiler for IA 64

- 3 front ends, 1 back end
- Five-levels of IR
- Gradual lowering of abstraction level

Code Generation
- If conversion & predication
- Code motion
- Scheduling (inc. sw pipelining)
- Allocation
- Peephole optimization
Even a 2000 JIT fits the mold, albeit with fewer passes

- Front end tasks are handled elsewhere
- Few (if any) optimizations
  - Avoid expensive analysis
  - Emphasis on generating native code
  - Compilation must be profitable
2014: Clang (LLVM) Compiler System – *Front End*

- Clang – C Language Front End for LLVM compiler infrastructure (default C/C++ compiler on Mac OS X and FreeBSD)
- Parser is *recursive descent* (top down), and most semantic analysis is completed at parse time
- *Lowering* process converts the higher-level IR (AST) into the lower-level IR (LLVM bitcode IR)
2014: Clang (LLVM) Compiler System – *Optimizer*

- LLVM includes ~100 optimization passes – this is (approximately) the default grouping of passes with the -O2 compiler flag
- Some passes are repeated multiple times (like simplify CFG)

- Basic local – memory promotion, dead arguments, combine instructions
- Call graph – Remove dead functions, inline functions, etc.
- Loops – Loop invariant code, loop unrolling, loop deletion
- Vectorization – Take advantage of SIMD processors
2014: Clang (LLVM) Compiler System – *Back End*

- Back end is designed to be retargetable to new platforms without *many* changes to the earlier code, but some (like function ABI) requires changes to Clang codebase

- SSA – LLVM IR uses *static single assignment* (covered later in the course)

- Some optimizations still happen in the back end
Summary

• Overview of a Compiler’s Tasks
  – Basic Translation from High-level to Instruction level

• Structure of a (Classical) Compiler
  – Traditional Three Phase Structure

• Classical Compilers
  – Static vs. Dynamic