The View from 35,000 Feet
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 (& 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 token
- Typical tokens include 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

\[ \text{SheepNoise} \rightarrow \text{baa SheepNoise} \]
\[ \mid \text{baa} \]

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 \( (P : N \rightarrow N \cup T) \)

(Example due to Dr. Scott K. Warren)
Context-free syntax can be put to better use

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

To recognize a valid sentence in some CFG, we reverse this process and build up a parse.
A parse can be represented by a tree (parse tree or syntax tree)

\[ x + 2 - y \]

This contains a lot of unneeded information.

1. goal → expr
2. expr → expr op term
3.  |  term
4.  |  number
5.  |  id
6. op → +
7.  |  -
The Front End

Compilers often use an abstract syntax tree

ASTs are one kind of intermediate representation (IR)

The AST summarizes grammatical structure, without including detail about the derivation

This is much more concise
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
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 (1 or $k$ registers)

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
Modern optimizers are structured as a series of passes

**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

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

Does the user realize a multiplication is generated here?
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**

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

  Does the user realize a multiplication is generated here?

  \[
  \begin{align*}
  &\text{DO } I = 1, M \\
  &\quad A(I,J) = A(I,J) + C \\
  &\text{ENDDO}
  \\
  &\text{compute addr}(A(0,J)) \\
  &\text{DO } I = 1, M \\
  &\quad \text{add 1 to get addr}(A(I,J)) \\
  &\quad A(I,J) = A(I,J) + C \\
  &\text{ENDDO}
  \end{align*}
  \]
Typical Restructuring Transformations:
• Blocking for memory hierarchy and register reuse
• Vectorization
• Parallelization
• All based on dependence
• Also full and partial inlining

Subject of CISC 673
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
Lab Zero

- Implement two COOL programs 100-200 lines each
  - Material on the web
    - Lab Assignment, Cool Manual
  - Specs for Lab 0 available on Web
    - Due in one week (9/16)
      - Speak to me after class if you will need more time
    - Practice with COOL and simulator available
    - Grading will be done by TA
      - You will meet with TA to deliver code
  - Next Class (Thursday)
    - Led by TA
    - Introduction to COOL, SVN, etc.
Next Week

- Introduction to Scanning (aka Lexical Analysis)
  - Material is in Chapter 2

- Specs for Lab 1 available next Tuesday (9/16)
Extra Slides Start Here
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
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
Classic Compilers

1980: IBM’s PL.8 Compiler

- Many passes, one 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

Dead code elimination
Global CSE
Code motion
Constant folding
Strength reduction
Value numbering
Dead store elimination
Code straightening
Trap elimination
Algebraic reassociation

*
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
1999: The SUIF Compiler System

Classic Compilers

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

**Classic Compilers**

SSA construction
Dead code elimination
Partial redundancy elimination
Constant propagation
Global value numbering
Strength reduction
Reassociation
Instruction scheduling
Register allocation
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

Front End
- Fortran 77
- C & C++
- Java

Middle End

Back End
- C/Fortran
- Alpha
- x86

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 UDEL ECE)

**Open source optimizing compiler for IA 64**

- 3 front ends, 1 back end
- Five-levels of IR
- Gradual lowering of abstraction level
Open source optimizing compiler for IA 64

- 3 front ends, 1 back end
- Five-levels of IR
- Gradual lowering of abstraction level
2000: The SGI Pro64 Compiler  (now Open64 from UDEL ECE)

Open source optimizing compiler for IA 64

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

Classic Compilers

Fortran
C & C++
Java

Interpr. Anal. & Optim’n
Loop Nest Optim’n
Global Optim’n
Code Gen.

Front End
Middle End
Back End

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 UDEL ECE)

Open source optimizing compiler for IA 64
- 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 UDEL ECE)

Open source optimizing compiler for IA 64
- 3 front ends, 1 back end
- Five-levels of IR
- Gradual lowering of abstraction level

Classic Compilers

Fortran
C & C++
Java

Front End  
Interpr. Anal. & Optim’n  
Loop Nest Optim’n  
Global Optim’n  
Code Gen.

Middle End  
Back End

Code Generation
If conversion & predication
Code motion
Scheduling (inc. sw pipelining)
Allocation
Peephole optimization
Classic Compilers

Even a 2007 Java JIT fits the mold, e.g., JIKES RVM (IBM)

Java Bytecodes → Class Loading, Verification, etc. → HIR → LIR → MIR → Executable

Front End: Several front end tasks are handled elsewhere
“Hot-spot” Optimizer
Avoid expensive analysis at first
Compilation must be profitable
Classic Compilers

Even a 2007 Java JIT fits the mold, e.g., JIKES RVM (IBM)

Java Bytecodes → Class Loading, Verification, etc. → HIR → LIR → MIR → Executable

- Front End
- Middle End
- Back End

- Several front end tasks are handled elsewhere
- “Hot-spot” Optimizer
  Avoid expensive analysis at first
  Compilation must be profitable

HIR Optimizations
Tail Recursion
Escape Analysis
Load Elimination
Loop Unrolling
Classic Compilers

Even a 2007 Java JIT fits the mold, e.g., JIKES RVM (IBM)

Java Bytecodes → Class Loading, Verification, etc. → HIR → LIR → MIR → Executable

Front End | Middle End | Back End

- Several front end tasks are handled elsewhere
- “Hot-spot” Optimizer
  Avoid expensive analysis at first
  Compilation must be profitable

LIR Optimizations
Constant Propagation
Copy Propagation
Constant Sub Elimination
Basic Block Reordering
Even a 2007 Java JIT fits the mold, e.g., JIKES RVM (IBM)

- Several front end tasks are handled elsewhere
- "Hot-spot" Optimizer
  - Avoid expensive analysis at first
  - Compilation must be profitable

MIR Optimizations
(Code Generation)
Live Analysis
Instruction Scheduling
Register Allocation