Computer Science 160
Translation of Programming Languages

Instructor: Christopher Kruegel
Overview of Compilers
Compilers

A) Why do we need a compiler?
B) What steps do we need to take to realize a compiler?
C) How is a compiler put together?
Compilers

• What is a compiler?
  – A program that translates a program in one language (source language) into an equivalent program in another language (target language), and it reports errors in the source program

• A compiler typically lowers the level of abstraction of the program
  C -> assembly code for Intel x86
  Java -> Java bytecode

• What is an interpreter?
  – A program that reads an executable program (one instruction at a time) and produces the results of executing these instructions

• C is typically compiled
• Script languages (Python, Javascript) are typically interpreted
• Java is compiled to bytecode, which is then interpreted
Why Build Compilers?

• Compilers provide an essential interface between applications and architectures

• High level programming languages:
  – Increase programmer productivity
  – Better maintenance
  – Portable

• Low level machine details:
  – Instruction selection
  – Addressing modes
  – Pipelines
  – Registers and cache

• Compilers efficiently bridge the gap and shield the application developers from low level machine details
Effectiveness of A Compiler

- Performance of a matrix multiplication kernel (with n = 4,096) on Intel Xeon E5-2666 v3E, with **mostly just** compiler software optimization:

<table>
<thead>
<tr>
<th>Version</th>
<th>Implementation</th>
<th>Running time (s)</th>
<th>Relative speedup</th>
<th>Absolute Speedup</th>
<th>GFLOPS</th>
<th>Percent of peak</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Python</td>
<td>21041.67</td>
<td>1.00</td>
<td>1</td>
<td>0.006</td>
<td>0.001</td>
</tr>
<tr>
<td>2</td>
<td>Java</td>
<td>2387.32</td>
<td>8.81</td>
<td>9</td>
<td>0.058</td>
<td>0.007</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>1155.77</td>
<td>2.07</td>
<td>18</td>
<td>0.118</td>
<td>0.014</td>
</tr>
<tr>
<td>4</td>
<td>+ interchange loops</td>
<td>177.68</td>
<td>6.50</td>
<td>118</td>
<td>0.774</td>
<td>0.093</td>
</tr>
<tr>
<td>5</td>
<td>+ optimization flags</td>
<td>54.63</td>
<td>3.25</td>
<td>385</td>
<td>2.516</td>
<td>0.301</td>
</tr>
<tr>
<td>6</td>
<td>Parallel loops</td>
<td>3.04</td>
<td>17.97</td>
<td>6,921</td>
<td>45.211</td>
<td>5.408</td>
</tr>
<tr>
<td>7</td>
<td>+ tiling</td>
<td>1.79</td>
<td>1.70</td>
<td>11,772</td>
<td>76.782</td>
<td>9.184</td>
</tr>
<tr>
<td>8</td>
<td>Parallel divide-and-conquer</td>
<td>1.30</td>
<td>1.38</td>
<td>16,197</td>
<td>105.722</td>
<td>12.646</td>
</tr>
<tr>
<td>9</td>
<td>+ compiler vectorization</td>
<td>0.70</td>
<td>1.87</td>
<td>30,272</td>
<td>196.341</td>
<td>23.486</td>
</tr>
<tr>
<td>10</td>
<td>+ AVX intrinsics</td>
<td>0.39</td>
<td>1.76</td>
<td>53,292</td>
<td>352.408</td>
<td>41.677</td>
</tr>
</tbody>
</table>

53,292X performance difference!

[Charles Leiserson, MIT 6.172]
Desirable Properties of Compilers

• Compiler must generate a correct executable
  – The input program and the output program must be equivalent; the compiler must preserve the meaning (semantics) of the input program

• Output program should run fast
  – We expect the output program to be more efficient than the input program

• Compiler itself should be fast

• Compiler should provide good diagnostics for programming errors

• Compiler should support separate compilation (modules, object files)

• Compiler should work well with debuggers

• Compiled code should be small

• Optimizations should be consistent and predictable

• Compile time should be proportional to code size
Source code
- Written in a high-level programming language

// simple example
while (sum < total) {
    sum = sum + x*10;
}

Target code
- Assembly language, which in turn is translated to machine code

L1: MOV total,R0
    CMP sum,R0
    JL L2
    GOTO L3

L2: MOV #10,R0
    MUL x,R0
    ADD sum,R0
    MOV R0,sum
    GOTO L1

L3: first instruction following the while statement
A) Why do we need a compiler?
B) What steps do we need to take to realize a compiler?
C) How is a compiler put together?
What is the Input?

• Input to the compiler is not

```c
//simple example
while (sum < total)
{
    sum = sum + x*10;
}
```

• Input to the compiler is

```c
//simple\bexample\bwhile\b(sum\b<\btotal)\b{\n    tsum\b=\n    bsum\b+x*10; \n}\n```

• How does the compiler recognize the keywords, identifiers, the structure, etc.?
First Step: Lexical Analysis (Scanning)

- The compiler scans the input file and produces a stream of tokens

  WHILE, LPAR, <ID,sum>, LT, <ID,total>, RPAR, LBRACE, 
  <ID,sum>, EQ, <ID,sum>, PLUS, <ID,x>, TIMES, <NUM,10>, 
  SEMICOL, RBRACE

- Each token has a corresponding lexeme, the character string that corresponds to the token
  - For example, “while” is the lexeme for token WHILE
  - “sum”, “x”, “total” are lexemes for token ID
Lexical Analysis (Scanning)

- Compiler uses a set of patterns to specify valid tokens
  - tokens: LPAREN, ID, NUM, WHILE, etc.

- Each pattern is specified as a regular expression
  - LPAREN should match: ( 
  - WHILE should match: while 
  - ID should match: [a-zA-Z][0-9a-zA-Z]*

- It uses finite automata to recognize these patterns
  ![ID automaton diagram]
Lexical Analysis (Scanning)

• During the scan the lexical analyzer gets rid of the **white space** (\b, \t, \n, etc.) and **comments**

• Important additional task: Error messages!
  - `Var%1` → Error! Not a token!
  - `while` → Error? It matches the identifier token.

• Natural language analogy: Tokens correspond to words and punctuation symbols in a natural language
Next Step: Syntax Analysis (Parsing)

• How does the compiler recognize the structure of the program?
  – Loops, blocks, procedures, nesting?

• Parse the stream of tokens -> parse tree
  – program will be on the leaves of the tree
Syntax Analysis (Parsing)

```
WHILE LPAREN RelExpr RPAREN LBRACE Stmt RBRACE

Stmt
DeclStmt
AssignStmt
Block

RelExpr
LT <ID,sum> <ID,total>

Stmt

AssignStmt
Expr EQ Expr SEMICOL

Expr
PLUS ArithExpr

ArithExpr
TIMES Expr

Expr

<ID,sum> <ID,sum> <ID,x> <NUM,10>
```
Syntax Analysis (Parsing)

• The syntax of a programming language is defined by a set of recursive rules. These sets of rules are called context free grammars.

\[
\begin{align*}
\text{Stmt} & \rightarrow \text{WhileStmt} \mid \text{Block} \mid \ldots \\
\text{WhileStmt} & \rightarrow \text{WHILE LPAREN Expr RPAREN Stmt} \\
\text{Expr} & \rightarrow \text{RelExpr} \mid \text{ArithExpr} \mid \ldots \\
\text{RelExpr} & \rightarrow \ldots
\end{align*}
\]

• Compilers apply these rules to produce the parse tree
• Again, important additional task: Error messages!
  – Missing semicolon, missing parenthesis, etc.
• Natural language analogy: It is similar to parsing English text. Paragraphs, sentences, noun-phrases, verb-phrases, verbs, prepositions, articles, nouns, etc.
Intermediate Representations

• The parse tree representation has too many details
  - LPAREN, LBRACE, SEMICOL, etc.

• Once the compiler understands the structure of the input program, it does not need these details (they prevent ambiguities during parsing)

• Compilers generate a more abstract representation after constructing the parse tree, which does not include the details of the derivation

• Abstract syntax trees (AST): Nodes represent operators, children represent operands
Intermediate Representations

while < <id,sum> <id,total> 
  
assign + <id,sum> 
  
  * <id,sum> 
    
    * <id,x> <num,10>
Semantic (Context-Sensitive) Analysis

- Not everything that we care about is related to the *structure* of the program, in some cases we have to check the meaning (or semantics)

- Are variables declared before they are used?
  - We can find out if “while” is declared by looking at the symbol table

- Do variable types match?
  \[ \text{sum} = \text{sum} + x*10; \]
Semantic (Context-Sensitive) Analysis

Symbol Table

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td>float</td>
</tr>
<tr>
<td>x</td>
<td>int</td>
</tr>
</tbody>
</table>

+<id,sum> *<id,x> <num,10>

sum can be a floating point number,
x can be an integer

may become

+<id,sum>
int2float

*<id,x> <num,10>
Runtime Environment

• Efficient implementation of programming language abstractions
  – Symbolic names
  – Name spaces
  – Procedures
  – Parameters
  – Control Flow

• Bridge the gap between useful idea and practical application
• Abstract syntax trees are a high-level intermediate representation used in earlier phases of the compilation

• There are lower-level (i.e., closer to the machine code) intermediate representations
  – Three-address code: Every instruction has at most three operands. Very close to (MIPS, x86) assembly
  – Stack based code: Assembly language for JVM (Java Virtual Machine), an abstract stack machine.

• Intermediate code generation for these lower level representations and machine code code generation are similar
Compilers can improve the quality of code by static analysis
- Data flow analysis, dependence analysis, code transformations, dead code elimination, etc.

```java
while (sum < total) {
    sum = sum + x*10;
}
```

We do not need to recompute \(x \times 10\) in each iteration of the loop

```java
temp = x*10;
while (sum < total) {
    sum = sum + temp;
}
```
Code Generation: Instruction Selection

- **Source code**
  
  \[
  \begin{align*}
  a &= b + c; \\
  d &= a + e;
  \end{align*}
  \]

- **Target code**

  If we generate code for each statement separately we will not generate efficient code

  ```
  \begin{align*}
  &\text{code for the first statement} \\
  &\quad \begin{cases}
  \text{MOV } b, R0 \\
  \text{ADD } c, R0 \\
  \text{MOV } R0, a
  \end{cases} \\
  &\text{code for the second statement} \\
  &\quad \begin{cases}
  \text{MOV } a, R0 \\
  \text{ADD } e, R0 \\
  \text{MOV } R0, d
  \end{cases}
  \end{align*}
  ```

  This instruction is redundant
There are a limited number of registers available on real machines.

Registers are valuable resources (keeping the values in registers prevents memory access), the compiler has to use them efficiently.

<table>
<thead>
<tr>
<th>source code</th>
<th>three-address code</th>
<th>assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td>( d = (a-b)+(a-c)+(a-c); )</td>
<td>( t = a - b; )</td>
<td>MOV a,R0</td>
</tr>
<tr>
<td></td>
<td>( u = a - c; )</td>
<td>SUB b,R0</td>
</tr>
<tr>
<td></td>
<td>( v = t + u; )</td>
<td>MOV a,R1</td>
</tr>
<tr>
<td></td>
<td>( d = v + u; )</td>
<td>SUB c,R1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADD R1,R0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADD R1,R0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MOV R0,d</td>
</tr>
</tbody>
</table>
Compilers

A) Why do we need a compiler?
B) What steps do we need to take to realize a compiler?
C) How is a compiler put together?
History of Compiler Development

Source program

Compiler

Target program

+ Error Diagnosis
+ Code Optimization

Monolithic Compiler | Two-pass Compiler | Modular Optimizing Compiler | Multicore/GPU Compiler | Autotuning Compiler | ML/Quantum Compiler


Our observation: From Monolithic to Modular to Architecture(input)-aware to Domain-Specific.
High-level View of a Compiler

- Must recognize legal (and illegal) programs
- Must generate correct code
- Must manage storage of all variables (and code)
- Must agree with OS and linker on format for object code
A Higher Level View: How Does the Compiler Fit In?

- collects the source program that is divided into separate files
- macro expansion

Preprocessor → Compiler

Protects source program

Assembler → Loader/Linker

- links the library routines and other object modules
- generates absolute addresses

generates machine code from the assembly code

library routines, relocatable object files

executable machine code
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 and multiple passes
  - Typically, front end is $O(n)$ or $O(n \log n)$, back end is NP-complete
- Different phases of compiler also interact through the symbol table
Responsibilities

- Recognize legal programs
- Report errors for the illegal programs in a useful way
- Produce IR and construct the symbol table
- Much of front end construction can be automated
Scanner

- Maps character stream into words—the basic unit of syntax
- Produces tokens and stores lexemes when it is necessary
  - $x = x + y ;$ becomes
    - $<id,x> \text{ EQ} <id,x> \text{ PLUS} <id,y> \text{ SEMICOLON}$
  - Typical tokens include `number, identifier, +, -, while, if`
- Scanner eliminates white space and comments
The Front End

Parser
- Uses scanner as a subroutine
- Recognizes context-free syntax and reports errors
- Guides context-sensitive analysis (type checking)
- Builds IR for source program
- Scanning and parsing can be grouped into one pass
The Front End

Context Sensitive Analysis

- Check if all the variables are declared before they are used
- Type checking
  - Check type errors such as adding a procedure and an array
- Add the necessary type conversions
  - int-to-float, float-to-double, etc.
The Back End

Responsibilities
- Translate IR into target machine code
- Choose instructions to implement each IR operation
- Decide which values to keep in registers
- Schedule the instructions for instruction pipeline

Automation has been much less successful in the back end
Instruction Selection

- Produce fast, compact code
- Take advantage of target language features
  - E.g., addressing modes
- Usually viewed as a pattern matching problem
  - *Ad hoc* methods, pattern matching, dynamic programming
- Especially problematic when instruction sets are complex
  - RISC architectures simplified this problem
Instruction Scheduling

- Avoid hardware stalls (keep pipeline moving)
- Use all functional units productively
- Optimal scheduling is NP-Complete
Register Allocation

- Have each value in a register when it is used
- Manage a limited set of registers
- Can change instruction choices and insert LOADs and STOREs
- Optimal allocation is NP-Complete

Compilers approximate solutions to NP-Complete problems
Traditional Three-pass (Optimizing) Compiler

Code Optimization

- Analyzes IR and transforms IR
- Primary goal is to reduce running time of the compiled code
  - May also improve space, power consumption (mobile computing)
- Must preserve “meaning” of the code
The Optimizer (or Middle End)

Modern optimizers are structured as a series of passes

Typical Transformations

• Discover and propagate constant values (constant propagation)
• Move a computation to a less frequently executed place
• Discover a redundant computation and remove it
• Remove unreachable code