CMSC 430
Theory of Language Translation
Chau-Wen Tseng

These slides are based on slides copyrighted by Keith Cooper, Linda Torczon & Ken Kennedy at Rice University, with modifications by Uli Kremer at Rutgers University

CMSC 430 — a.k.a. Compilers

• Catalog Description
  → Introduction to compiler construction (emphasis on compiler front ends). Course contents include the following: Formal translation of programming languages, program syntax and semantics, finite state recognizers and regular grammars, context-free parsing techniques such as LL(k) and LR(k). Code generation, improvement, syntax-directed translation schema.

• Course Objectives
  → This course focuses on compilation techniques needed to translate programs written in a standard programming language into executable code on microprocessor architectures. Program analysis and optimization techniques are presented in class lectures. Programming projects provide experience with implementation issues and allow students to develop programming and software engineering skills.

Basis for Grading

• Tests
  → 2 Midterms 15%
  → Final 20%

• Projects
  → Scanner 10%
  → Parser 10%
  → Type Checker 10%
  → Code Generator 10%
  → Optimizer 10%

Notice: This grading scheme is tentative and subject to change.

Basis for Grading

• Exams
  → Midterms
  → Final
  = Closed-notes, closed-book
  = Final is cumulative

• Practice problems
  = Reinforce concepts, provide practice

• Projects
  = Cumulative
  = Don’t fall behind!

Syllabus

• Scanning
• Parsing
• Context Sensitive Analysis
• Runtime Environment
• Intermediate Representations
• Code Generation
• Optimizations
• Dataflow Analysis
• Register Allocation
• Instruction Scheduling

Recommended Textbook

• Engineering A Compiler
  → Keith Cooper & Linda Torczon

ENGINEERING A COMPILER
Keith D. Cooper & Linda Torczon

Rice University
Class-taking technique for CMSC 430

- I will use slides extensively
  → I will moderate my speed, you sometimes need to say "STOP"
- You should read books for details
  → Not all material will be covered in class
  → Book complements the lecture
- CMSC 430 is not a programming course
  → Projects are graded on functionality, documentation, and project reports more than style. However, things should be reasonable
- Use the resources provided to you
  → See me or the TA in office hours if you have questions
  → Post questions regarding projects on Forum

Why Study Compilation?

- Compilers are important system software components
  → They are intimately interconnected with architecture, systems, programming methodology, and language design
- Compilers include many applications of theory to practice
  → Scanning, parsing, static analysis, instruction selection
- Many practical applications have embedded languages
  → Commands, macros...
- Many applications have input formats that look like languages.
  → Mathlab, Mathematica
- Writing a compiler exposes practical algorithmic & engineering issues
  → Approximating hard problems: efficiency & scalability

Compliers

- What is a compiler?
  → A program that translates an executable program in one language into an executable program in another language
  → A good compiler should improve the program, in some way
- What is an interpreter?
  → A program that reads an executable program and produces the results of executing that program
- C is typically compiled, Scheme is typically interpreted
- Java is compiled to bytecodes (code for the Java VM)
  → Which are then interpreted
  → Or a hybrid strategy is used
    - Just-in-time compilation
    - Dynamic optimization (hot paths)

Intrinsic interest

- Compiler construction involves ideas from many different parts of computer science

<table>
<thead>
<tr>
<th>Artificial intelligence</th>
<th>Algorithm</th>
<th>Theory</th>
<th>Systems</th>
<th>Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greedy algorithms</td>
<td>Graph algorithms, union-find</td>
<td>DFA's &amp; PDA's, pattern matching</td>
<td>Allocation &amp; naming, Synchronization, locality</td>
<td>Pipeline &amp; hierarchy management, Instruction set use</td>
</tr>
</tbody>
</table>

Intrinsic merit

- Compiler construction poses challenging and interesting problems:
  → Compilers must do a lot but also run fast
  → Compilers have primary responsibility for run-time performance
  → Compilers are responsible for making it acceptable to use the full power of the programming language
  → Computer architects perpetually create new challenges for the compiler by building more complex machines
  → Compilers must hide that complexity from the programmer
  → Success requires mastery of complex interactions

Making Languages Usable

It was our belief that if FORTRAN, during its first months, were to translate any reasonable "scientific" source program into an object program only half as fast as its hand-coded counterpart, then acceptance of our system would be in serious danger... I believe that had we failed to produce efficient programs, the widespread use of languages like FORTRAN would have been seriously delayed.

— John Backus
High-level View of a Compiler

- Source code
- Compiler
- Machine code
- Errors

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

- Source code
- Front End
- IR
- Back End
- Machine code
- Errors

Implications:
- Use an intermediate representation (IR)
- Front end maps legal source code into IR
- Back end maps IR into target machine code
- Extension: multiple front ends & multiple passes
  *(better code)*

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

A Common Fallacy

- Can we build $n \times m$ compilers with $n \times m$ 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*

The Front End

- 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
- Tokens
- Parser
- IR
- Errors

Scanner
- Maps character stream into words—the basic unit of syntax
- Produces pairs—word & its part of speech
  $x y$ becomes $id id: \text{fn}$
  $x y$ becomes $id id: \text{op}$
- In casusal speech, we call the pair a token
- Typical tokens include number, identifier, +, -, new, while, if
- Scanner eliminates white space
- Speed is important

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:

\[ \text{SheepNoise} \rightarrow \text{SheepNoise sheep} \]

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

---

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>expr</td>
</tr>
<tr>
<td>expr</td>
<td>expr ( \text{op} ) term</td>
</tr>
<tr>
<td>expr ( \text{op} ) y</td>
<td>expr ( \text{op} ) y</td>
</tr>
<tr>
<td>expr ( \text{op} ) term (-y)</td>
<td>expr ( \text{op} ) term (-y)</td>
</tr>
<tr>
<td>expr ( \text{op} ) (2-y)</td>
<td>expr ( \text{op} ) (2-y)</td>
</tr>
<tr>
<td>term (+2-y)</td>
<td>term (+2-y)</td>
</tr>
<tr>
<td>(x+2-y)</td>
<td>(x+2-y)</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)

This contains a lot of unneeded information.

---

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-II
- 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 registers)

Typically, 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

The Optimizer (or Middle End)

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) + 1

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

Modern Restructuring Compiler

Typical Restructuring (source-to-source) 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 (exception handling)
- Interface to the operating system
  - Input and output
- Support of parallelism
  - Parallel thread initiation
  - Communication and synchronization

Classic Compilers

1980: IBM's PL/I 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

1980: IBM's PL/I 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
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

2000: The SGI Pro64 Compiler (now Open64 from Intel)

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

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