Compilers

• 40-414: Compiler Design

  http://sharif.edu/~sani/courses/compiler/

• Computer Engineering Dept., Sharif University

• Instructor: GholamReza GHASSEM SANI
Compilers

- Lectures:
  - Time: Sundays and Tuesdays, 16:30–18:00
  - Location: https://vc.sharif.edu/ch/sani, or https://vclass.ecourse.sharif.edu/ch/sani

- Evaluation:
  
  4 Written Assignments, and 20%
  4 Programming Assignments 40%
  2 Exams 40%
Acknowledgement

- Most Lecture Notes are from a similar course (i.e., CS-143) taught by Professor Alex Aiken in Stanford University
Text

• The Purple Dragon Book

• Aho, Lam, Sethi & Ullman

• Not required
  - But a useful reference
The Course Project

- A big project
- ... in 4 rather easy parts
- Start early!
Academic Honesty

• Don’t use work from uncited sources
  - Including old code

• We use plagiarism detection software
  - many cases in past offerings

PLAGIARISM

Prof. Aiken
How are Languages Implemented?

• Two major strategies:
  - Interpreters (older)
  - Compilers (newer)

• Interpreters run programs “as is”
  - Little or no preprocessing

• Compilers do extensive preprocessing
Compilers

Source Program → Compiler → Target Program

Input → Target Program → Output

Errors
Interpreters

- Translates line by line
- Executes each translated line immediately
- Execution is slower because translation is repeated
A Hybrid Compiler

Source Program → Translator → Intermediate Program → Virtual Machine → Output
Different Types of Compilers

Construction

Single Pass

Multiple Pass
History of Compilers

• 1954 IBM develops the 704
  - Successor to the 701

• Problem
  - Software costs exceeded hardware costs!

• All programming done in assembly
The Solution

• “Speedcoding”
  • an early example of an interpreter
  • developed in 1953 by John Backus
  • much faster way of developing programs
  • programs were 10-20 times slower than hand-written assembly
  • needed 300 bytes = 30% machine memory
FORTRAN I

• FORmula TRANslation Project
• FORTRAN ran from 1954 To 1957
• By 1958, over 50 percent of all of programs were in FORTRAN

John Backus
FORTRAN I

• The first compiler
  - Huge impact on computer science

• Led to an enormous body of theoretical work

• Modern compilers preserve the outlines of FORTRAN I
The Structure of Fortran Compiler

1. Lexical Analysis
2. Parsing
3. Semantic Analysis
4. Optimization
5. Code Generation

The first 3, at least, can be understood by analogy to how humans comprehend English.
Lexical Analysis

• First step: recognize words.
  - Smallest unit above letters

This is a sentence.
More Lexical Analysis

• Lexical analysis is not trivial. Consider:

ist his ase nte nce
And More Lexical Analysis

- Lexical analyzer divides program text into “words” or “tokens”
  
  If x == y then z = 1; else z = 2;

- Units:
  - Keywords { if, then, else }
  - Identifiers { x, y, z }
  - Numbers { 1, 2 }
  - Operators { ==, = }
  - Separators { blanks, ; }
Parsing

• Once words are understood, the next step is to understand sentence structure

• Parsing = Diagramming Sentences
  - The diagram is a tree
Diagramming a Sentence

This line is a longer sentence

article noun verb article adjective noun

subject object

sentence
Parsing Programs

- Parsing program expressions is the same
- Consider:
  
  \[
  \text{If } x == y \text{ then } z = 1; \text{ else } z = 2; 
  \]

- Diagrammed:

```
  x  ==  y
    |    |    |
    |    |    |
    |    |    |
  relation
    |    |    |
    |    |    |
  predicate

  z    1
    |    |    |
    |    |    |
  assign

  z    2
    |    |    |
    |    |    |
  assign

  if-then-else

  stmt
    |    |    |
    |    |    |
  then-stmt

  stmt
    |    |    |
    |    |    |
  else-stmt
```
Semantic Analysis

• Once sentence structure is understood, we can try to understand “meaning”
  - But meaning is too hard for compilers

• Compilers perform limited semantic analysis to catch inconsistencies
Semantic Analysis in English

• Example:
  Jack said Jerry left his assignment at home.
  What does “his” refer to? Jack or Jerry?

• Even worse:
  Jack said Jack left his assignment at home?
  How many Jacks are there? (1, 2, or 3)
  Which one left the assignment?
Semantic Analysis in Programming

• Programming languages define strict rules to avoid such ambiguities.

• This C++ code prints “4”; the inner definition is used:
  ```cpp
  int Jack = 3;
  {
    int Jack = 4;
    cout << Jack;
  }
  ```
More Semantic Analysis

• Compilers perform many semantic checks besides variable bindings

• Example:
  
  Jack left her homework at home.

• A “type mismatch” between her and Jack; we know they are different people
  - Presumably Jack is male
Optimization

- No strong counterpart in English,
  - but *a little bit like* editing
  - but *akin to* editing

- Automatically modify programs so that they
  - Run faster
  - Use less memory

- Your project has no optimization component :D
Optimization Example

\[ X = Y \times 0 \] is the same as \[ X = 0 \]

NOT ALWAYS CORRECT

\[ \text{NaN} \times 0 = \text{NaN} \]
Code Generation

• Produces assembly code (usually)

• A translation into another language
  – Analogous to human translation
Compilers Today

• The overall structure of almost every compiler adheres to our outline

• The proportions have changed since FORTRAN
  - Early: lexing, parsing most complex, expensive
  - Today: optimization dominates all other phases, lexing and parsing are cheap
Compiler Front-end and Back-end

Source Program

1. Lexical analyzer
2. Syntax Analyzer
3. Semantic Analyzer
4. Intermediate Code Generator
5. Code Optimizer
6. Code Generator
7. Peephole Optimization

Analyses

Syntheses

1, 2, 3, 4, 5: Front-End
6, 7: Back-End

Target Program
Front-End

- Front end maps source code into an IR representation
- Back end maps IR onto machine code
- Simplifies retargeting
Front-End (Cont.)

Scanner:

- Maps characters into tokens – the basic unit of syntax
  - $x = x + y$ becomes $<\text{id}, x> \leq, > <\text{id}, x> \text{\#}, > <\text{id}, y>$
- Eliminate white space (tabs, blanks, comments)
Front-End (Cont.)

Source code → Scanner → tokens → Parser → Parse Tree

errors

Parser:

- Recognize context-free syntax
- Guide context-sensitive analysis
- Produce meaningful error messages
Back-End

IR → Instruction selection → Register Allocation → Machine code

Back-End:

- Translate IR into machine code
- Choose instructions for each IR operation
- Decide what to keep in registers at each point
Two Main Components of Back-End

Code Generator:

- Produce compact fast code
- Use available addressing modes
Back-End (Cont.)

Peephole Optimization:

- **Limited resources**
- **Optimal allocation is difficult**
Phase 1. Lexical Analysis

Easiest Analysis - Identify tokens which are the basic building blocks

For Example:

\[
\text{Position} := \text{initial} + \text{rate} \times 60;
\]

All are tokens

Blanks, Line breaks, etc. are scanned out
Phase 2. Syntax Analysis or Parsing

Parse Tree:

assignment statement

:= expression

+ expression

expression

* expression

expression

number

identifier

expression

identifer

initial

rate

Nodes of tree are constructed using a Grammar for the source language
• **Finds Semantic Errors**

- Syntax Tree
  - position
    - initial
      - *
        - rate
          - 60

- Conversion Action
  - position
    - initial
      - *
        - rate
          - inttoreal

• **One of the Most Important** Activities in This Phase:
  - Type Checking - Legality of Operands
Supporting Phases

• Symbol table creation / maintenance
  - Contains info (address, type, scope, args) on certain Tokens, typically identifiers
  - Data structure created/initialized during lexical analysis; and updated during later analysis & synthesis

• Error handling
  - Detection of different errors which correspond to all phases; and deciding what happens when an error is found
An example of the Entire Process

position = initial + rate * 60

**lexical analyzer (Scanner)**

```
<id, 1>  <=  <id, 2>  <=  <id, 3>  <=  <num, 60>
```

**syntax analyzer (Parser)**

```
:=
  +
  *
<id, 1>
<id, 2>
<id, 3>
<num, 60>
```

**semantic analyzer**

```
:=
  +
  *
<id, 1>
<id, 2>
<id, 3>
<inttoreal>
<num, 60>
```

Symbol Table

- 1 position real ...
- 2 initial real ...
- 3 rate real ...

**Error Handler**

**intermediate code generator**
An example of the Entire Process

Symbol Table
1 position real …
2 initial real …
3 rate real …

intermediate code generator

\[ t_1 := \text{inttoreal}(60) \]
\[ t_2 := id_3 \times t_1 \]
\[ t_3 := id_2 + t_2 \]
\[ id_1 := t_3 \]

code optimizer

\[ t_1 := id_3 \times 60.0 \]
\[ id_1 := id_2 + t_1 \]

final code generator

LD R1, id3
MUL R1, R1, #60.0
LD R2, id2
ADD R1, R1, R2
ST id1, R1

Error Handler

position = initial + rate \times 60