Introduction

CSCI 3136
Principles of Programming Languages

Faculty of Computer Science
Dalhousie University

Winter 2013

Reading: Chapter 1
How to Talk to Your Computer

A programmer’s dream: (Just say what you want . . . in English)

“Computer, please compute the best route from Halifax to San Francisco and display it on your screen.”
A programmer’s dream: (Just say what you want . . . in English)

“Computer, please compute the best route from Halifax to San Francisco and display it on your screen.”

CPU: (Assembly)

...  
addl $1, -4(%rbp)  
movl %eax, %esi  
...
How to Talk to Your Computer

A programmer’s dream: (Just say what you want . . . in English)

“Computer, please compute the best route from Halifax to San Francisco and display it on your screen.”

- Very expressive
- Low “programming” effort
- Sometimes ambiguous

CPU: (Assembly)

...  
addl $1, -4(%rbp)
movl %eax, %esi  
...  

- Very low-level
- Tedious
- Unambiguous
A programmer’s dream: (Just say what you want . . . in English)

“Computer, please compute the best route from Halifax to San Francisco and display it on your screen.”

• Very expressive
• Low “programming” effort
• Sometimes ambiguous

Bridging the gap: (Java, C++, Ruby, Haskell, Scheme, . . . )

class RouteFinder {
    ... 
    public static void main(String[] args) { ... }
    ...
}

CPU: (Assembly)

... 
addl $1, -4(%rbp)
movl %eax, %esi
... 

• Very low-level
• Tedious
• Unambiguous
**How to Talk to Your Computer**

**A programmer’s dream:** (Just say what you want . . . in English)

“Computer, please compute the best route from Halifax to San Francisco and display it on your screen.”

- Very expressive
- Low “programming” effort
- Sometimes ambiguous

**Bridging the gap:** (Java, C++, Ruby, Haskell, Scheme, . . .)

```java
class RouteFinder {
    ...
    public static void main(String[] args) { ... }
    ...
}
```

- Unambiguous
- Fairly expressive
- Fairly efficient to implement in terms of low-level CPU instructions

**CPU:** (Assembly)

```assembly
addl $1, -4(%rbp)
movl %eax, %esi
...```

- Very low-level
- Tedious
- Unambiguous
Part I: Program Semantics and Translation

Ways to specify programming languages formally

- What is a syntactically correct program in this language?
- What does a syntactically correct program mean? What is the computation it specifies?

Basics of compilation and interpretation

- How do we translate a syntactically correct program into (machine) code that carries out exactly the computation specified by the program?
Programming paradigms

- Different high-level ways of thinking about computation.
- Each paradigm is suited best for certain classes of applications.
- Each programming language designed to express a certain subset of these paradigms more easily.
Why Is This Useful?

- Makes learning new languages and understanding obscure features easier.
- Helps choose the most appropriate language for the job, evaluate trade-offs.
- Learn to simulate useful features in languages that lack them.
- Learn to use features of a language most effectively.
History of Programming Languages

- Machine language, assembly language
- FORTRAN (1957) → versions I–IV, … 1995
- COBOL (1959), APL (1960), SNOBOL (1962)
- BASIC (1964) → Visual Basic (1990)
- Prolog (1973)
Why So Many Programming Languages?

- Evolution
- Special purpose
- Personal preference

What makes a programming language successful?

- Expressive power
- Easy to learn
- Easy to implement
- Open-source
- Good compilers (e.g., FORTRAN)
- Economics, patronage, inertia
Programming Paradigms

Imperative programming

- Closest to von Neumann machine (assemblers, FORTRAN, BASIC, COBOL)
- Structured programming (ALGOL, Pascal, C)

Object-oriented programming

- Smalltalk, C++, Java

Functional programming

- Lisp, Scheme, ML, Haskell

Logic programming

- Prolog, VisiCalc
Course Outline

Introduction

Program translation

- Lexical analysis, regular languages, and finite automata
- Parsing, context-free languages, and push-down automata
- Semantic analysis and attribute grammars

Flow of control and information

- Names, scopes, and binding
- Control flow

Building abstractions

- Subroutines and control abstractions
- Data types
- Object-oriented programming
- Functional programming
• **Required textbook:** Michael L. Scott. *Programming Language Pragmatics*, 3rd edition.

• Slides  
  (available from [http://www.cs.dal.ca/~nzeh/Teaching/3136](http://www.cs.dal.ca/~nzeh/Teaching/3136)).

• Hopcroft et al. *Introduction to Automata Theory*.

• Tucker and Noonan. *Programming Languages*.

• More relevant books listed on the course website. Some are available online.

• Other links can be found on the course website.
Evaluation and Plagiarism Policy

Evaluation

- 40% assignments (weekly)
- 20% midterm exam
- 40% final exam

Assignment groups

- You are allowed to form groups of up to 3 students working together on an assignment.
- Each group hands in one joint assignment. Thus, all members of the same group receive the same mark.
- The constitution of groups may change between assignments.

Plagiarism policy

- Collaboration between groups is not allowed.
- Any use of reference material (book, web, ... ) must be acknowledged.
- According to Faculty policy, any suspected case of plagiarism is referred to the Academic Integrity Officer and may be forwarded to the Senate Discipline Committee.
But I Know Only Java

This courses uses Java, C, C++, Python, Ruby, Perl, Scheme, and Haskell to illustrate important programming concepts.

Other languages may be mentioned in passing where appropriate.

Search the web for tutorials on how to use these languages, tinker with them.

Learning new languages is fun!
Overview of Program Translation

CSCI 3136
Principles of Programming Languages

Faculty of Computer Science
Dalhousie University

Winter 2013

Reading: Chapter 1
Compilation vs Interpretation

Compilation

Interpretation
Compilation vs Interpretation

Compilation

Source program → Compiler → Target program

Input → Target program → Output

Interpretation
Compilation vs Interpretation

Compilation

Source program → Compiler → Target program

Input → Target program → Output

Interpretation

Source program → Interpreter → Output

Input → Interpreter → Output
Trade-Off Between Compilation and Interpretation
Trade-Off Between Compilation and Interpretation

Advantages of compilation

- Standalone code
- Faster code
- Smaller code
Trade-Off BetweenCompilation and Interpretation

Advantages of compilation

- Standalone code
- Faster code
- Smaller code

Advantages of interpretation

- Faster development
- More flexibility and possibly more expressive power
- Late binding
- Dynamic features
... is fuzzy:

- Just in time (JIT) compilation
- The Perl interpreter compiles program into intermediate code.
- The Java compiler compiles the program into “byte code” that is run by an interpreter.
Compilation-Interpretation Boundary

... is fuzzy:

- Just in time (JIT) compilation
- The Perl interpreter compiles program into intermediate code.
- The Java compiler compiles the program into “byte code” that is run by an interpreter.

Generally, we consider a translation program that translates source code into machine code or some intermediate code to be a compiler if it needs to perform a semantic analysis of the source code to produce the intermediate code:

- Java is compiled: the byte code is close to machine code and requires semantic analysis to produce.
- Perl is interpreted: the intermediate code is produced only when the program is run.
- BASIC is interpreted: early interpreters stored program in tokenized form. No semantic analysis is required to produce this.
Phases of Compilation

Front end
- Source program (character stream)
  → Scanner (lexical analysis)
  → Token stream
  → Parse tree
  → Parser (syntactic analysis)
  → Semantic analysis and code generation
  → Abstract syntax tree or other intermediate form

Back end
- Modified intermediate form
- Machine-independent code improvement
- Target code generation
- Target language (e.g., assembly)
- Modified target language
- Machine-specific code improvement
Lexical Analysis and Parsing

Lexical analysis

- Group input characters into tokens (e.g., identifiers, keywords, numbers, …)
- Remove extraneous characters (e.g., spaces, tabs, newline characters) and comments

Parsing

- Organize tokens into a parse tree according to a grammar describing the language
- Ensure the sequence of tokens conforms to the syntax defined by the language grammar
Semantic Analysis

Generates symbol table and intermediate representation of program (e.g., syntax tree) from the parse tree.

- The symbol table maps each identifier to information known about it.
- The abstract syntax tree includes only important nodes of the parse tree but also stores annotations of these nodes (e.g., pointers from identifiers to symbol table entries).

Enforces rules not captured by the context-free grammar (e.g., use identifier only after it has been declared).

Generates code for run-time checks (e.g., array bounds).
A Sample Pascal Program

{ calculate greatest common divisor }
program gcd( input, output );
var i, j : integer;
begin
  read( i, j );
  while i <> j do
    if i > j then i := i - j
    else j := j - i;
  writeln( i );
end.
{ calculate greatest common divisor }

program gcd ( input , output ) ;
var i , j : integer ;
begin
    read ( i , j ) ;
    while i <> j do
        if i > j then i := i - j
        else j := j - i ;
    writeln ( i ) ;
end .
A Context-Free Grammar for the Language in the Example

Rules

\[ \text{Program} \rightarrow \text{program \ identifier ( identifier More\_identifiers ) ; Block .} \]
\[ \text{Block} \rightarrow \text{Labels Constants Types Variables Subroutines} \]
\[ \text{begin Statement More\_statements end} \]

\[ \text{More\_identifiers} \rightarrow \varepsilon \]
\[ \text{More\_identifiers} \rightarrow , \ \text{identifier More\_identifiers} \]

Terminals

- program, var, integer, begin, end, while, do, if, then, else
- (,),,, ;, :, :, :=, <>, >, -
- identifier

Non-terminals

- Program, More\_identifiers, Block, Labels, Constants, Types, …
The Parse Tree of the Program

```
program identifier ( identifier (input) More_identifiers )

, identifier More_identifiers

identifier (output)
```

```
begin end Subs

stmts...
```

```
Variables

Types

Consts

Lbls
```

```
end Morestmts
```

Introduction
# The Abstract Syntax Tree and the Symbol Table

## Index

<table>
<thead>
<tr>
<th>Index</th>
<th>Symbol</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>integer</td>
<td>type</td>
</tr>
<tr>
<td>2</td>
<td>textfile</td>
<td>type</td>
</tr>
<tr>
<td>3</td>
<td>input</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>output</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>gcd</td>
<td>program</td>
</tr>
<tr>
<td>6</td>
<td>i</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>j</td>
<td>1</td>
</tr>
</tbody>
</table>

## Program

```
Program
    read
    read
    while
    if
    <>
    :=
    :=
    write
    writeln
```

### Example Code

```
Program
    read(5) := read(3)
    > read(6)
    if <> := := write(4) writeln(6)
```

### Defines

- **Predefined**
  - `integer`: type
  - `textfile`: type
  - `input`: 2
  - `output`: 2
- **User-defined**
  - `gcd`: program
  - `i`: 1
  - `j`: 1