CPSC 411: Introduction to Compiler Construction

Ron Garcia
rxg@cs.ubc.ca
http://www.ugrad.cs.ubc.ca/~cs411/
Land Acknowledgment

• http://aboriginal.ubc.ca/community-youth/musqueam-and-ubc/

• Our course is held on the UBC Point Grey (Vancouver) campus, which sits on the traditional, ancestral, unceded territory of the xʷməθkʷəy̓əm (Musqueam) First Nation.
Who am I?

• Ronald Garcia

• How should you address me:
  – Ron
  – Dr. Garcia
  – Prof. Garcia

• Research Area: Programming Languages!
Basic Course Information

• Prerequisites
  – CPSC 311 and CPSC 213

• If you are on the wait list then something is wrong!:
  – (https://www.cs.ubc.ca/students/undergrad/courses/waitlists)

• Programming assignments will be in Java (JDK 1.8?)
  – We’ll be using IntelliJ as our IDE (CE 2018.3)
  – We’ll be using Maven3 as our Project Management Tool
  – We’ll be using JUnit5 as our Testing Framework
  – We’ll be using github.ugrad.ubc.ca as our source code control system (log into github today, make sure your ugrad account is activated first)
Contacting Me

- Office: 387 ICICS/CS
- Office hours: TO BE DETERMINED
- Do not email me with general questions about assignments or course material.
- Instead:
  - Use Piazza: https://piazza.com/ubc.ca/winterterm12018/cpsc411
  - Ask a TA during office hours
  - Ask me after class
  - Visit me during my office hours
    - Especially FMOOWMP victims (http://bit.ly/1N34dI3)
Use email to:

- Make appointments or bring marking errors and that sort of thing to my attention
- Privacy regulations make it difficult to respond via email to issues concerning requests for grades or personal information
- To make sure the message isn’t missed, the subject line of the email should contain [cpsc411]
- Email response times can vary widely from a few hours to several days. Don’t expect any response on weekends or after hours
More Course Info

- Textbook is required

Your TAs

• TAs:
  – Nico Ritschel
  – Rui Ge
  – Hiroki Takezawa

• TA DLC Office Hours: COMING SOON!
Course Workload

• 2 Individual Assignments – Language Processors in Java
• 2 Group Projects: Compile to x86!
  – Compiler for a simple programming language
  – Compiler for MiniJava, a subset of Java
• Read pretty much the whole book
• Much Code Spelunking (SE Lyfe)
• Examinations (Practice and Theory)
  – 1 Midterm exam (After Reading Week)
  – 1 Final exam (During Final Exam Period)
Course Grading Scheme

- Assignments 5%
- Compiler projects 45%
- Midterm/Final Exam 50%
- Instructor reserves the right to make minor modifications to the marking scheme
Class policies

• Read the policies on the website
Academic Conduct

• You are encouraged to collaborate by:
  – Helping each other understand material and assignments
  – Exploring/discussing solutions to assignments
    • Caveat – no looking at each others code or exchanging anything written (i.e., talk but don’t write)
    • The “Gilligan’s Island” “Better Call Saul” rule (???)
  – Use of existing public approaches to problem (but see next slide!)
  – Discussing with current 411 students existing approaches to solving a problem
  – Discussion of requirements
  – Discussing the merits of a proposed solution with the course instructor or TA
Academic Conduct cont’d

• What’s not allowed
  – Submitting someone else’s work as your own. Examples include:
    • Having in your possession previous solutions to the assignments - either someone else’s or the instructor’s
    • Working in a group and then handing in the work, even a part of it
    • Work you have handed in to another course – all work must be new work
  – Making a solution available as an aid to others, either now, or in the future!
Academic Conduct cont’d

• What to do if you are uncertain?
  – Ask
  – Read (and re-read) the department’s policy on academic misconduct

– Resources
  – http://www.cs.ubc.ca/about/policies/collaboration.shtml
Possible Penalties

1. A failing grade or zero in the course, exam or assignment.
2. Suspension from the University.
3. Reprimand with a letter placed in the student’s file.
4. A notation on the student’s permanent record of the penalty imposed.
If you are tempted:

1. Think of the lost opportunity to learn
2. Think of the disappointment that I will feel
3. Think of the disappointment that your family will feel
4. Think of the extra work that you put on me
5. Just Don’t!
6. Some thought provoking reading on the subject:
Let’s get going!!!

• What is this course about?
Course Plan

1. Introduction (today)
2. Writing Language Processors in Java (311 Immigration)
3. A whirlwind trip through a simple compiler
4. A slightly slower trip through a slightly more complex compiler
   - Syntax
   - Semantics
   - Translation
5. A more careful trip through a complex compiler for a “real” language (Mini-Java)
   - Instruction selection
   - Optimization
   - Register allocation
Chapter 1: Introduction

GOAL this lecture:
What is this course about... a high-level perspective.

OVERVIEW
– Levels of Programming Languages (A Caricature)
– Language processors
– Specification of a programming language
# Levels of Programming Languages

| High level program | class Triangle {  
|                   |   ...  
|                   |   float surface()  
|                   |     return b*h/2;  
|                   | }  |
| Low level program | LOAD r1,b  
|                   | LOAD r2,h  
|                   | MUL r1,r2  
|                   | DIV r1,#2  
|                   | RET  |
| Executable Machine code | 00010010010000101  
|                        | 0010010011101100  
|                        | 10101101001...  |
Levels of Programming Languages

Some high level languages:
- Pascal, Java, C, C++, Ada, ...

Some low level languages:
- x86_64 assembly language, PowerPC assembly language, Java Virtual Machine assembly language...
Levels of Programming Languages

What makes a high level language different from a low level language?

Things found in HL languages but typically not in LL languages
- expressions
- control structures/abstractions: while, repeat-until, if-then-else procedures
- data types
  - distinguish several different types of data
  - composite data types
  - user defined data types
- encapsulation
  modules, procedures, objects
A **high level language** is more abstract than a low level language.

More abstract? What does that mean?

**Abstraction:** Separate the ‘what’ from the ‘how’.
Or ‘what is implemented’ from ‘how is it implemented’.
e.g., procedural abstraction = separate ‘what does it do’ from ‘how does it do it’

HL languages abstract away from the underlying machine
=> much more portable
Q: How do the following make a HL language more abstract?

- Expressions
- control structures: while, repeat-until, if-then-else,
  procedures
- data types
- encapsulation
  modules, procedures, objects
A programming language processor is any system (software or hardware) that manipulates programs.

Examples:
- Editors
- Translators (e.g., compiler, assembler, disassembler)
- Interpreters
Language Processors: Why do we need them?

How to bridge the “semantic gap”? 

Programmer

Compute surface area of a triangle?

Programmer

Concepts and Ideas

Java Program

JVM Assembly code

JVM Binary code

JVM Interpreter

X86_64 Processor

Hardware

0101001001...

Hardware
Language Processors: Another alternative

Programmer

Compute surface area of a triangle?

How to bridge the “semantic gap”?

Hardware

0101001001...

Programmer

Concepts and Ideas

Java Program

X86_64 Assembly code

X86_64 Binary (.o)

X86_64 Binary (.exe)

X86_64 Processor

Hardware
Assignments and Projects

• Assignment 1
  – Due Monday, January 14

• Project
  – You will be working in groups of 2 or 3 (mostly 3)
  – You will be working with the same partner(s) throughout the term.
  – Start thinking about who you would like to partner with!
Programming Language Specification

• Why?
  – A communication device between people who need to have a common understanding of the PL:
    • language designer, language implementor, programmer

• What to specify?
  – Specify what is a ‘well formed’ program
    • syntax
    • contextual constraints (also called static semantics):
      – scoping rules
      – type rules
  – Specify what is the meaning of (well formed) programs
    • semantics (also called dynamic/runtime semantics)
Why?

What to specify?

How to specify?

- Formal specification: use some kind of precisely defined formalism
- Informal specification: description in English.

- Usually a mix of both (e.g., Java specification)
  - Syntax => formal specification using CFG
  - Contextual constraints and semantics => informal
Syntax Specification

Syntax is specified using “Context Free Grammars”:

– A finite set of terminal symbols
– A finite set of non-terminal symbols
– A start symbol
– A finite set of production rules

Usually CFG are written in “Bachus Naur Form” or BNF notation.

A production rule in BNF notation is written as:

\[ N ::= \alpha \]  
where \( N \) is a non terminal 
and \( \alpha \) a sequence of terminals and non-terminals

\[ N ::= \alpha | \beta | ... \] is an abbreviation for several rules with \( N \) as left-hand side.
A CFG defines a set of strings. This is called the language of the CFG.

Example:
Start ::= Letter
  | Start Letter
  | Start Digit
Letter ::= a | b | c | d | ... | z
Digit ::= 0 | 1 | 2 | ... | 9

Q: What is the “language” defined by this grammar?
Example: Syntax of “Expression”

Expression is a very simple calculator-like language.

An example program:

```plaintext
// This is a comment.
zero = 0;
one = 1;
two = one + zero;
three = two + one;
four = three + two;
five = four + three;
print five
```

- Expression
- Statement
Example: Syntax of “Expression”

Program ::= Statements print Expression
Statement
  ::= Identifier = Expression ;
Statements ::= Statement
  | Statements Statement
  | ...
Example: Syntax of “Expression” (continued)

Expression ::= primary-Expression
  | Expression Operator primary-Expression
  | Expression ? Expression : Expression
primary-Expression ::= Literal
  | Identifier
  | ! primary-Expression
  | ( Expression )
Identifier ::= Letter
  | Identifier Letter
  | Identifier Digit
Literal ::= Digit
  | Literal Digit
Operator ::= + | - | * | / | <
Example: Syntax of “Expression” (continued)

Comment ::= // CommentLine eol
CommentLine ::= Graphic | CommentLine Graphic
Graphic ::= any printable character or space
Contextual Constraints

Syntax rules alone are not enough to specify the format of well-formed programs.

**Example 1:**
\[
m = 2; \\
\text{print } m + x \\
\text{Undefined!}
\]

**Example 2:**
\[
m = 2; \\
n = m < 4; \\
\text{print } n + 1 \\
\text{Type error!}
\]
Scope Rules

Scope rules regulate visibility of identifiers. They relate every applied occurrence of an identifier to a binding occurrence.

Example 1

```plaintext
m = 2;
r = 10 * m;
print r
```

Example 2:

```plaintext
m = 2;
print m + x
```

Terminology:

*Static (or lexical) scoping* vs. *dynamic scoping*
Type Rules

Type rules regulate the expected types of arguments and types of returned values for the operations of a language.

Type rule of $<$ :

$E_1 < E_2$ is type correct and of type $\text{Boolean}$ if $E_1$ and $E_2$ are type correct and of type $\text{Integer}$

Type rule of $E \ ? \ V_1 : V_2$

$E \ ? \ V_1 : V_2$ is type correct and of the same type as $V_1$ if $E$ is of type $\text{Boolean}$ and $V_1$ and $V_2$ are type correct and have the same type

Terminology:

$Static\ typing\ vs.\ dynamic\ typing$
Semantics

Specification of semantics is concerned with specifying the “meaning” of well-formed programs.

Terminology:

Expressions are evaluated and yield values (and may or may not perform side effects)

Statements are executed and perform side effects.

Declarations are elaborated to produce bindings

Side effects:

- change the values of variables
- perform input/output
Semantics

Example: The (informally specified) semantics of statements in Expression.

*Statements are executed to define constants and perform output.*

The statement $C = E;$ is executed as follows:

1. first the expression $E$ is evaluated to yield a value $v$
2. then $v$ is assigned to the constant named $C$

The sequential statement $S_1 \ S_2$ is executed as follows:

1. first the statement $S_1$ is executed
2. then the statement $S_2$ is executed

etc.
Semantics

The statement \texttt{print } \( E \) is executed as follows:

first the expression \( E \) is evaluated to yield a value \( v \)

then \( v \) is printed on the standard output
Semantics

Example: The semantics of expressions.

An expression is evaluated to yield a value.

A (literal expression) \( L \) yields the integer value of \( L \)

The (constant name) expression \( C \) yields the value of the constant named \( C \)

The (binary operation) expression \( E_1 \ op \ E_2 \) yields the value obtained by applying the binary operation \( op \) to the values yielded by (the evaluation of) expressions \( E_1 \) and \( E_2 \)

e tc.
Semantics

• The ultimate semantics processing in a compiler is translation
• Generate code in the target language that is equivalent in every respect to the program in the source language
• Instruction selection
• Register allocation
• Code improvement (optimization)
Compilers

• This course is about Compilers.
• Compilers are language processors
  – translating one language into another
  – (usually high-level language into low-level language)
  – The challenge: bridging “semantic gap” between languages
• Language specification
  – needed for communication between language designers, implementers and users
  – Three “parts”
    • Syntax of the language: usually formal: EBNF
    • Static Semantics: usually informal: type rules and scope rules (written in English
    • Dynamic Semantics: usually informal: English
A whirlwind trip through a compiler

- Compiler for the Expression Language
- Parsing
- Typechecking
- Translation to Intermediate Representation (IR code)
  - raw
  - linearized
  - trace-scheduled
- X86_64 code generation
  - flow graph
  - interference graph
  - register allocation
Disclaimer

• This is just one of many ways to build a compiler
• Other ways include:
  – stream-based
    • Many small compilers, each takes as input a representation of the program and produces as output a modified representation
    • The output of the final compiler is the translated program
  – parse-tree based
    • Many passes over a shared data structure
    • The last pass produces the translated program
• Our version is a bit of a hybrid
Expression language

Expression is a very simple calculator-like language.

An example program:

```plaintext
// This is a comment.
zero = 0;
one = 1;
two = one + zero;
three = two + one;
four = three + two;
five = four + three;
print five
```
Example: Syntax of “Expression”

Program ::= Statements print Expression
Statement ::= Identifier = Expression ;
Statements ::= Statement |
             Statements Statement
...
Example: Syntax of “Expression” (continued)

Expression ::= primary-Expression
  | Expression Operator primary-Expression
  | Expression ? Expression : Expression
primary-Expression ::= Literal
  | Identifier
  | ! primary-Expression
  | ( Expression )
Identifier ::= Letter
  | Identifier Letter
  | Identifier Digit
Literal ::= Digit
  | Literal Digit
Operator ::= + | - | * | <
**Example: Syntax of “Expression” (continued)**

<table>
<thead>
<tr>
<th>Comment</th>
<th>:=</th>
<th>//</th>
<th>CommentLine</th>
<th>eol</th>
</tr>
</thead>
<tbody>
<tr>
<td>CommentLine</td>
<td>:=</td>
<td>Graphic</td>
<td></td>
<td>CommentLine Graphic</td>
</tr>
<tr>
<td>Graphic</td>
<td>:=</td>
<td><em>any printable character or space</em></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example programs

sample/fib.exp
sample/less.exp
sample/cond.exp
sample/exp.exp
Expression Language *Parser*

- Specified using JavaCC (Google is your friend)
  - Look at ExpressionsParser.jj in parser.jcc
Example: Concrete Syntax of Expressions (recap)

Expression
::= primary-Expression
   | Expression Operator primary-Expression
   | Expression ? Expression : Expression

primary-Expression
::= Literal
   | Identifier
   | ! primary-Expression
   | ( Expression )
Example: Abstract Syntax of Expressions

Expression
::= Literal IntegerExp
  | Identifier IDExp
  | ! Expression UnaryExp
  | Expression Op Expression BinaryExp
  | Expression ? Expression : Expression ConditionalExp
Abstract Syntax Trees

Abstract Syntax Tree for: \( d = (c + 10) \times n \)
• Do:
  – run parse sample/fib.exp
  – run parse sample/exp.exp
Syntax rules alone are not enough to specify the format of well-formed programs.

**Example 1:**
\[
m = 2; \\
\text{print } m + x
\]
Undefined!

**Example 2:**
\[
m = 2; \\
n = m < 4; \\
\text{print } n+1
\]
Type error!
Scope Rules

Scope rules regulate visibility of identifiers. They relate every applied occurrence of an identifier to a binding occurrence.

Example 1
m = 2;
r = 10 * m;
print r

Example 2:
m = 2;
print m + x

Terminology:

Static binding vs. dynamic binding
Checking scope rules

• Do:
  – run type sample/badscope.exp
Type Rules

Type rules regulate the expected types of arguments and types of returned values for the operations of a language.

Type rule of \(<\) :

\[ E_1 < E_2 \] is type correct and of type \textbf{Boolean} 
if \( E_1 \) and \( E_2 \) are type correct and of type \textbf{Integer}

Type rule of \( E \ ? V_1 : V_2 \):

\[ E \ ? V_1 : V_2 \] is type correct and of the same type as \( V_1 \)
if \( E \) is of type \textbf{Boolean} and \( V_1 \) and \( V_2 \) are type correct and have the same type

Terminology:

\textit{Static typing vs. dynamic typing}
Checking type rules

• Do:
  – run type sample/badtype.exp
Checking Expression programs

• Look at the implementation:
  – In package typechecker.implementation
    • BuildSymbolTableVisitor
    • TypeCheckVisitor
Semantics

Specification of semantics is concerned with specifying the “meaning” of well-formed programs.

Terminology:

Expressions are evaluated and yield values (and may or may not perform side effects)

Statements are executed and perform side effects.

Declarations are elaborated to produce bindings

Side effects:
  • change the values of variables
  • perform input/output
Example: The (informally specified) semantics of statements in Expression.

Statements are executed to define constants and perform output.

The statement \( C = E; \) is executed as follows:

first the expression \( E \) is evaluated to yield a value \( v \)

then \( v \) is assigned to the constant named \( C \)

The sequential statement \( S_1 \ S_2 \) is executed as follows:

first the statement \( S_1 \) is executed

then the statement \( S_2 \) is executed

etc.
Semantics

**Example:** The semantics of expressions.

*An expression is evaluated to yield a value.*

A (literal expression) $L$ yields the integer value of $L$.

The (constant name) expression $C$ yields the value of the constant named $C$.

The (binary operation) expression $E_1 \ O \ E_2$ yields the value obtained by applying the binary operation $O$ to the values yielded by (the evaluation of) expressions $E_1$ and $E_2$.

etc.
Semantics

• The ultimate semantics processing in a compiler is translation
• Generate code in the target language that is equivalent in every respect to the program in the source language
• Translation to intermediate code
• Instruction selection
• Register allocation
• Code improvement (optimization)
Expression translation to IR

- Look at the implementation:
  - in package translate.implementation
    - TranslateVisitor

- Do:
  - run ir sample/cond.exp
  - run ir sample/exp.exp

- IR has four forms:
  - raw
  - linear (canonical)
  - basic blocks
  - trace scheduled
Instruction selection

- Maximal munch (greedy tree matching)
- Express patterns corresponding to machine instructions
- Match the patterns against the (linear) IR
- Works from the leaves up
- Look at the code: (yuck)
  - in backend package codegen.x86_64
    - X64_64Muncher
    - start with CONST and PLUS rules
- Do:
  - run code sample/less.exp
  - run code sample/exp.exp
Optimization (Or more precisely: anti-pessimization)

• A bunch of analysis needs to be done first
  – flow graph
  – liveness
• Then we can perform some correctness-preserving transformations that will improve the code
• I’m not going to show you the code ☹
• Do:
  – run iropt sample/exp.exp
Register allocation

• A bunch of analysis needs to be done first
  – flow graph
  – liveness
  – interference
• Then we can select registers for every temp
• I’m not going to show you the code 😞
• Do:
  – run flow sample/exp.exp & show the graph (show flow*.dot)
  – run live sample/exp.exp & show the graph (show live*.dot)
  – run interf sample/exp.exp & show the graph (first)
  – run reg sample/exp.exp & show the graph (first)
And then we’re all done

• Well, almost
  – We need to assemble the resulting assembly code
    • use gcc to compile exp.s
  – We need a bit of runtime support
    • getting the program running (main?)
    • implementing println
    • for a bigger language, other “builtin” functionality
      – allocating/freeing objects
      – supporting builtin types
And then we’re all done ...

- Do:
  - run final sample/exp.exp
  - gcc –c sample/exp.s
  - gcc –o exp.exe exp.o runtime/runtime.o
  - ./exp.exe

- Ta Da!!
Where are we going?

• The next 4 weeks or so are:
  – Parsing (Chapters 2, 3, and 4)
  – Scopes (Chapter 5)
  – Type checking (Also Chapter 5)
  – Translation to IR (Chapter 7 with a bit of Chapter 6)

• In the context of a slightly bigger language:
  – Expressions + functions
Course Overview

- Introduction (Chapter 1)
- Compiler Frontend:
  - Lexical Analysis & Parsing (Chapters 2, 3, 4)
  - Semantic Analysis (Chapter 5)
  - Activation Records (Chapter 6)
  - Translation to Intermediate Code (Chapter 7)
  - Basic Blocks and Traces (Chapter 8)
- Compiler Backend:
  - Instruction Selection (Chapter 9)
  - Liveness Analysis (Chapter 10)
  - Register Allocation (Chapter 11)
  - Code Emission (Chapter 12)
Course Overview (continued)

• Optimization
  – Dataflow analysis and optimization (Chapter 17)

• Other stuff
  – Garbage collection (Chapter 13)
  – Object oriented languages (Chapter 14)