Concepts of Programming Languages

Introduction

Lecturer: Gabriele Keller
Tutor: Liam O’Connor
University of New South Wales
School of Computer Sciences & Engineering
Sydney, Australia

COMP 3161/9161 Week 1
Admin

• Course web page
  - [http://www.cse.unsw.edu.au/~cs3161](http://www.cse.unsw.edu.au/~cs3161)
  - check the course web page for lecture notes, exercises and additional resources
  - course forum if you have any questions about the course content, tutorial exercises, assignments
  - contact me via email for admin questions:
    ‣ keller@cse.unsw.edu.au

• Course material:
  - lecture slides (will be available after the lecture)
  - additional notes will be available online
  - tutorial questions
Assessment

- **Class mark:**
  - Tutorial participation: (3161: 10%, 9161: 0%)
  - Midsession exam: (3161: 10%, 9161: 20%)
  - Assignments (30%)
  - Exam (50%)
    - supplementary exam will be oral exam
  - Overall mark:
    - if at least 40% in each component are achieved
      - (exam + class)/2
    - otherwise
      - min (44, (exam + class)/2)
Assignments

• Released in Week 5 & 9 (approx), four weeks to complete assignment

• Late penalty
  – 4% of assignment worth per day assignment is submitted after the deadline
  – submission accepted until five days after the deadline
Assumed knowledge

• We expect you to know about basic theoretical concepts like boolean logic, predicate logic, induction over natural numbers

• Please check out the tute (not really) exercises for Week 1
Programming Language Courses at CSE

- COMP3131 Programming Languages & Compilers
- COMP3151 Foundations of Concurrency
- COMP3161 Concepts of Programming Languages
- COMP3153 Algorithmic Verification
- COMP4181 Language-based Software Safety
- COMP3141 Software Systems and Implementation
What’s going to happen this week?

Brief intro to *Concepts of Programming Languages*

*What?*

*Why?*

Introduction to programming in Haskell

Next week we will introduce the formal framework we’ll be using throughout the course
Course Content

As programmers and software engineers, programming languages are among our most important tools.

They affect

- the design of a system
- development costs, productivity
- safety and security
- performance
Understanding the principles of programming languages...

To be able to understand new languages/new language features

To become a better programmer and software engineer

- more elegant, more concise, more efficient code
- choose the right tool for the job
- easier to
  - find bugs
  - understand other people’s code
  - learn new languages
  - design your own languages
  - implement compilers and other tools
Languages we use

• **To describe and prove properties of programming languages**
  - Predicate logic, inductive definitions and inference rules

• **Implementation language for assignments**
  - Haskell
    ▸ is a functional programming language
    ▸ high-level language
    ▸ compiler statically checks many program properties
    ▸ short introduction to programming in Haskell this week
How do PL concepts relate to the implementation of PLs?

- Programmer
  - Source Language

How can we bridge this gap?

- Compiler
- Interpreter
- Hybrid (combines both)
Compiler translates source language into machine code.
Interpreter/Abstract machines/Virtual machines

- Programmer
  - Source Language
- Interpreter
  - abstract (aka virtual) machine sits on top of concrete machine
  - executes source language operations
- Computer
  - Machine Code
Hybrid Approach

- **Programmer**
  - **Source Language**
  - **Compiler**
  - **Interpreted Code**
  - **Interpreter**
  - **Computer**
    - **Machine Code**

- abstract (aka virtual) machine sits on top of concrete machine
- executes source language operations
Java Virtual Machine (JVM)

Java

Compiler

Byte Code

JVM (Linux, x86)  JVM (Linux, ARM)  JVM (Windows, x86)
What is LLVM?

- Stood for ‘low level virtual machine’ but should now be seen as name by itself, not acronym

- Is not actually a virtual machine, but a compiler toolkit/collection of compiler components

- many front ends:
  - C, C++, Objective-C, Swift, Java, D, Ada, Fortran, Haskell, Dylan, Gambas, Python, Ruby

- many back ends, for example:
  - ARM, Hexagon, MIPS, Nvidia PTX, PowerPC, R600, SPARC, x86/ x86-64
LLVM (simplified)

- **Language Dependent Frontends**
  - C
  - C++
  - Java Script
  - Haskell

- **Architecture independent LLVM instructions**
  - ARM
  - x86
  - PowerPC

- **Architecture dependent LLVM instructions**

- **Different LLVM Backends**
  - debugger, monitor;
  - compile time
  - link time
  - and run time optimisations
Compilers in detail

Programmer
Source Language

Compiler

Computer
Machine Code
Stages of a Compiler: Lexer

- **Lexer:**
  - decompose program string into a sequence of tokens (i.e., programming language objects)
  - discard white spaces, newlines, comments
  - checks if any illegal ‘words’ are detected

```c
int foo () {
    int i;
    i = 11;
    if ( i > 5) {
        i = i - 1;
    } else {
        i = i + 1;
    }
    return i;
}
```

Ident “int” Ident “foo” LParen RParen LBrace ......
Stages of a Compiler: Parser

• Parser
  
  - the *lexer* produces a flat sequence of tokens
  
  - the *parser* analyses the syntactic structure and builds a hierarchical parse tree
  
  - the structure of a program is determined by the language grammar

• Informal grammar description for C (simplified):
  
  - C function definitions consist of
    
    ▶ an identifier (return type), followed by
    
    ▶ an identifier (function name), followed by
    
    ▶ a possibly empty lists of arguments enclosed in parenthesis, f.b.
    
    ▶ a statement (function body)

  ➔ too verbose, too imprecise
Stages of a Compiler: Parser

• more precise specification (extended Backus-Naur Form - EBNF)

```
funDef ::= Ident₁ Ident₂ (arguments) stmt

stmt  ::= expr; | if expr then stmt₁ else stmt₂;
          | return expr; | {locDec stmts}
          | while (expr) stmt

stmts ::= o | stmt stmts

expr  ::= Num | Ident | expr₁ + expr₂ |
          | Ident = expr | Ident (expr)

locDec ::= Ident₁ Ident₂;

stm    ::= ....

arguments ::= o | ...
```
Stages of a Compiler: Parser

- the parser checks if the token sequence adheres to the grammar, and stores the program as parse tree:
Stages of a Compiler: Semantic Analysis

- checks (static) semantic properties
  - identifier in scope?
  - is the program type correct?
Stages of a Compiler: Optimisation

- Many different optimisations
  - loop unrolling
  - inlining
  - ......
Stages of a Compiler: Code Generation

- allocate register and other resources
- select appropriate machine code instructions
In this course

• What are we looking at in 3161?

- Specify the grammar of programming languages
- Specify and analyse static and dynamic properties of different languages and language features
- Discuss and prove the correctness of different optimisations
- Specify and analyse properties of various abstract machines

Program String

Lexer

Sequence of Tokens

Parser

Parse Tree

Semantic Analysis

Annotated Parse Tree

Optimiser

Intermediate Code

Code Generation

Machine Code