Introduction

COMP3220 – Principle of Programming Languages

Zhitao Gong

2016 Spring

Outline

Course Information

Introduction to Programming Language

Contacts

- Instructor Zhitao Gong

Tigermail	zzg0009
Office	Shelby 2307
Office Hour	Friday 1000 – 1100

- TA Wenlu Wang

Tigermail	wzw0022
Office	Shelby 2319
Office Hour	Wednsday 1500 – 1600

- Course Website
  http://auburn.edu/~zzg0009/comp3220/spring2016/

Course Materials

- Textbook is *Concepts of Programming Languages*, tenth edition by Robert W. Sebesta.
- Other Reference materials is provided on course website.

Grading Policy

All assignments are submitted electronically on Canvas. *NO LATE SUBMISSION IS ACCEPTED*.

Table: Grade weight distribution

Description	Points
Assignment	40
Midterm Exam	20
Final Exam	20
Term Project	20
Total	100

Topics

Tutorial Three PL tutorials are included, imperative PL `C++`, functional PL `Scheme` and logic PL `Prolog`.

Compiler Formal languages, regular expression (RE), BNF, context-free grammars, parsing.

PL basics Names, types, binding, memory management, procedure and data abstraction, parameter passing.

Course Goals

- Learn basic structures in most PL.
  E.g., variables, data types, control structures, procedure abstraction and etc. So that it is easier to learn a new one.
- Study different language paradigms.
  Imperative, functional and logic PL so that you have more choice.

Outline

Course Information

Introduction to Programming Language

What is Programming Language (PL)?

A language intended for use by a person to express a process by which a computer can solve a problem
– Hope and Jipping

A set of conventions for communicating an algorithm
– E. Horowitz

The art of programming is the art of organizing complexity
– Dijkstra

It is just a string of characters.
– Compiler LOL

PL Design Consideration

Readable Comments, names, syntax. E.g., `C++`, `Python` vs `BrainFuck`, `Whitespace`.

Learning Curve Small number of core concepts combine regularly and systematically. E.g., `C`, `Lua` vs `Haskell`.

Portable Language standardization and platform support. E.g., `Java`, `C++`.

Abstraction Control and data structures that hide details. E.g., floating numbers.

Efficient E.g., `C++` vs `Python`.

Purpose General programming, scientific computing, string manipulating, etc. E.g., `Python`, `R` and `Matlab`, `awk`, etc.

Why Learn More than One PL?

- Choose the right language for a given problem. E.g., `Ruby` and `Python` vs `C++` for web development, `Matlab` and `R` vs `C` for scientific computing.

  If all you have is a hammer,
  every problem looks like a nail.

- Learn new languages more easily later. PL's are ever changing. Different jobs may require different PL's.
- Learn thinking about problems in different ways, or *paradigms*.

What is Paradigm?

Paradigm is the way of thinking about problems.
PL's can be *loosely* classified into two categories.

Imperative Program consists of commands describing *how* to get solution. E.g., `C`, `Pascal`, `Assembler`.

Declarative Program consists of expressions describing *what* is the solution. Key concept is *referential transparency*.

- Functional PL like `Lisp`, `Haskell` and `Erlang`.
- Logic PL like `Prolog`.

There are languages that are *multi-paradigm*, e.g., `C++`, `Javascript`, `Python`, etc.

Imperative Paradigm

Imperative PL follow the model of computation described in the *Turing Machine* – they maintain the fundamental notion of a *state*.

- State of program are values stored in the memory and register.
- Program issues to the machine orders to change the state of the machine.
- Fits closely Von Neumann architecture.

Key commands

Assignment Changes the state of the machine.

Branch Changes the state of the program.

Sequence Used to chain commands together.

Imperative Example

The program is like a recipe in which essential states and necessary steps to manipulate them are defined and ordered.

```
int fact(int n) {
  int f = 1;
  while (n > 1) {
    f = f * n;
    n = n - 1;
  }
  return f;
}
```

Functional Paradigm

Functional PL is based on λ-*calculus*, informally, composition of operations on data.
Key commands

λ-abstract $\lambda x.t$ is a definition of an anonymous function that is capable of taking a single input x and substituting it into the expression t. E.g., $\lambda x.x^2$ is an lambda abstraction for function $f(x) = x^2$.

Application ts represents the application of a function t to an input s. E.g., calling the function t and produce $t(s)$.

Functional Example

Functions have no state, and data are immutable.

```
(defun fact (n)
  (if (= n 1)
      1
    (* n (fact (- n 1)))))  ; referential transparent

(fact 4)
```

Logic Paradigm

Logic PL is based on *Horn Clauses*, describing problems as axioms and derivation rules.

Key commands

Unification
: The algorithmic process of solving equations between symbolic expressions.

Non-deterministic Search
: Based on first-order predicate logic.

Logic Example 1

Prolog is most well known logic PL.

```
fact(0,1).
fact(N,F) :-
    N > 0, N1 is N - 1,
    fact(N1,F1), F is N * F1.
```

Logic Example 2

We will talk about about *syllogism* and `Prolog` in later chapters. First we state some facts and predicates.

```
%% Prolog                 Syllogism
man(socrates).          % socrates is a man.
mortal(X) :- man(X).    % All men are mortal.
```

Now we are asking questions

```
%% Is socrates mortal?
mortal(socrates).
```

PL Translation

PL is translated in two ways.

Compilation PL is compiled *in whole* by a compiler into a platform-dependent executable. E.g., `C/C++`.

Interpretation PL is interpreted *one statement at a time* by a virtual machine called interpreter. E.g., `Javascript`.

Some languages are the mixture of both, e.g., `Python`, `Java`.

Translation Overview

Compilers are programs, and generally very large programs. They almost always have a structure based on the *analysis-synthesis* model of translation.

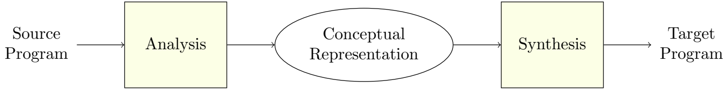
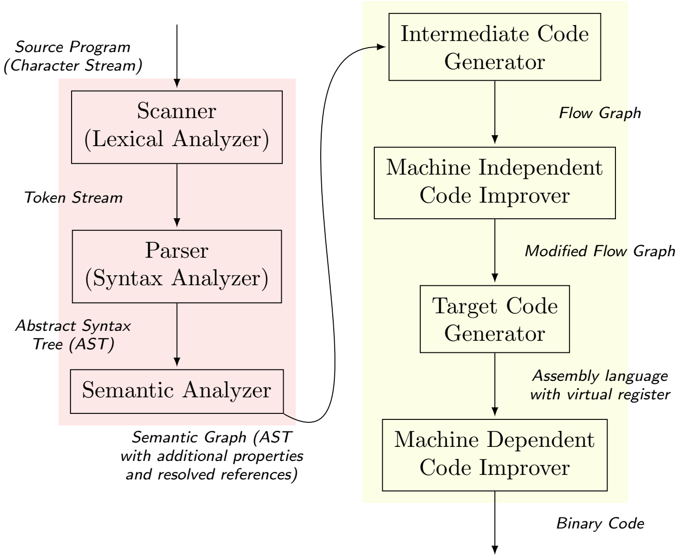


Figure: Every non-trivial translation requires analysis and synthesis.

Compiler Components



Compiler Components 2

Scanner It converts the stream of characters into a stream of tokens, removing whitespace, removing comments and expanding macros along the way.

Parser The parser turns the token sequence into an abstract syntax tree (AST).

Semantic Analysis Checks legality rules and tie up the pieces of the syntax tree (by resolving identifier references, inserting cast operations for implicit coercion, etc.) to form a semantic graph.