Introduction to Programming Languages

- Today
  - Syllabus
  - Overview

Office hours

- TR 2-3
  - Occasionally a few minutes late due to meetings between 1 and 2
  - Email me to schedule another time if you cannot make office hours
- See me in person especially for programming questions
  - I will not debug your code through emails
- Start working your homework and projects early

Grading

- Homework, Surveys, Quizzes, Group assignments (25%)
  - Around 5 written assignments (15%)
  - Surveys (2%)
  - Pop Quizzes (3%)
  - Group presentation (5%)
- Projects (40%)
  - Compile SomeLife
    - Four phases
  - Scheme programming
- Exams (35%)

Introduction to Programming Languages (Objectives)

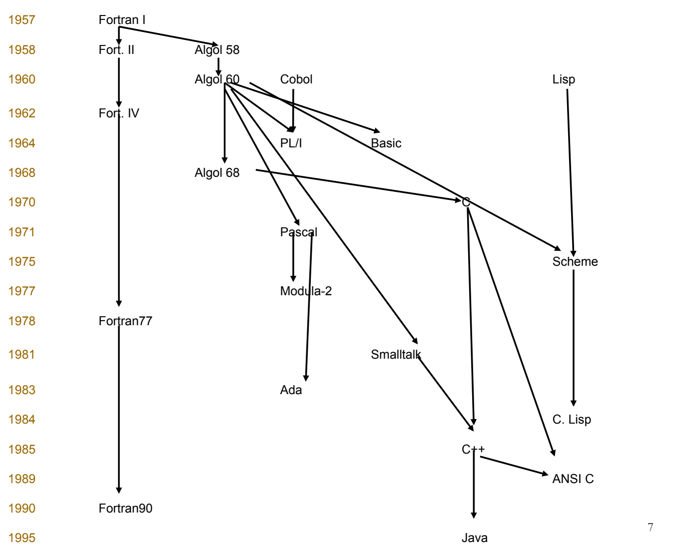
- Given a language the student will be able to evaluate the language using common design criteria
- The student will be able to name and describe the different phases of a compiler.

Reasons for Studying Programming Languages

- Increased capacity to express ideas
  - new paradigms → new problem solving skills
- Improved background for choosing an appropriate language
- Increased ability to learn new languages
  - some languages are similar
  - understand obscure language features
- Better understanding of implementation of languages
  - understand implementation costs
  - figure out how to do things in languages that don't support them explicitly
    - Simulate language features
- Increased ability to design new languages
  - Or make better use of language technology wherever it appears

Programming Domains

- Scientific Applications
  - Computationally intensive
- Business Applications
  - I/O intensive
- Artificial Intelligence
  - Use of symbolics
- Systems software
  - More low-level interactions
- Scripting
  - Lists of commands often doing simple processing
- Special purpose
  - Verilog



Language Evaluation Criteria

- Readability
- Writability
- Reliability

Language Evaluation Criteria

- Readability
  - Simplicity
    - Small number of basic components
    - Examples against simplicity
      - feature multiplicity
      - operator overloading
  - Orthogonality
    - A relatively small set of primitive constructs can be combined in a relatively small number of ways to build control and data structures.
    - Consistent simple rules
      - Functions that can't return any data type
    - Related to simplicity

Language Design Criteria

- Readability (cont.)
  - Control statements
    - Structured control flow (no gotos)
  - Data types and structures
    - Adequate facilities for user-defined types
    - No records in FORTRAN 77
    - No Boolean type in C
  - Syntax Considerations
    - Do not restrict identifier forms (FORTRAN77)
      - Six characters at most
    - Use keywords (none in PL/I)
      - if then = else then else = then else then = else;
    - Appearance indicates their purpose
      - For example, **static** in C

Language Design Criteria

- Writability
  - Simplicity and orthogonality
    - Not a large number of constructs
    - Consistent set of rules
  - Support for abstraction
    - Data abstraction and process abstraction
    - Ability to define and use complicated structures
      - Not Fortran 77
  - Expressivity
    - Ability to conveniently express common functionality
    - Power of a language
      - Dynamic types, first-class functions in Scheme.

Language Design Criteria

- Reliability
  - Type checking
    - All type errors should be caught at compile- or run-time.
      - Anything in C
  - Exception handling
    - Ability to intercept run-time errors and take corrective action
  - Aliasing
    - Do not have two or more distinct references to the same memory cell
  - Security
    - Do not allow access to non-user data
      - Stack overflow in C

Influences on Language Design

- Computer Architecture
  - von Neumann
  - Parallel machines
- Programming methodologies
  - Data abstraction
    - Object oriented vs. procedure oriented

13

Paradigms

- A programming paradigm is a way of conceptualization of what it means to perform computation, of structuring and organizing how tasks are carried out in a computer.
- Example
  - A block-structured paradigm is a set of programming languages that support nested block structures including procedures.

14

Example Paradigms

- Imperative
  - Program = steps of computation via state changes
  - traditional model of program modifying memory
  - features include variables, assignments, arrays, ...
  - C, Pascal, Fortran
- Object oriented
  - everything is an object
  - an object has its own memory
  - computation performed by communicated objects
  - objects are an instance of class having both data and methods
  - Java, Eiffel, Part of C++

15

Example Paradigms

- Functional (Applicative)
  - Values are single entities and are not "stored" in memory
  - Computation performed by applying functions
  - Functions are 1st-class values which means they can be used like data (created, returned from function calls, ...)
  - Parts of Scheme, Lisp
- Logic (Declarative)
  - A program is a declaration of facts, rules of inference and queries
  - Computation is done by a (backtracking) inference engine that tries to do a "proof".

16

This Class

- Syntax analysis
  - scanners
  - parsers
- Semantic analysis
  - compilers
- Functional programming
  - study the features of Scheme by writing programs
- Language Security

Why Scanners and Parsers?

- To understand programming languages we need something expressing syntactic structure.

1. English
   - not precise
2. Grammar
   - precise but hard to use
3. Scanner and Parser
   - precise and automatic

Why Compilers?

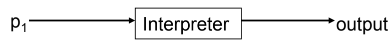
- To understand the meaning of a programming language, we need a tool to express that meaning.
- We interact with compilers all the time
  - Java
  - C, C++
- Develops skills for you to reason about program behavior.
- Specify language semantics
  - English
    - not precise
    - awkward

Why Compilers?

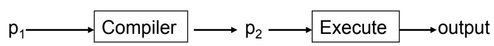
- Specify language semantics
  - Denotational Semantics
    - precise
    - mathematically elegant
    - hard to read
  - Interpreter
    - precise
    - elegant
    - useful beyond theory
    - high-level specification
  - Compiler
    - precise
    - useful beyond theory
    - understanding of implementation increases understanding of concept
    - low-level specification

Compilation vs. Interpretation

- An interpreter is a program that takes another program, p_1, and evaluates p_1 to determine its meaning.



- A compiler is a program that takes second program, p_1, and produces a third program, p_2, which when evaluated gives the meaning of p_1. (note that p_2 does not need to be machine language)

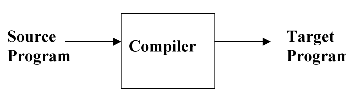


Compilation vs. Interpretation

- Interpretation:
  - Greater flexibility
  - Portability (Java)
- Compilation
  - Better performance
- Most languages are a combination of both
  - Java (compilation to Java byte code, which is interpreted and possibly compiled into machine code)
  - C (mostly compiled, but I/O formats interpreted)

What is a compiler?

- A compiler is just a program that takes other programs and converts them into another language. That language is often assembler so that it can be assembled, linked and run on a computer



Principles of Compiler Design

- The compiler must preserve the meaning of the program being compiled
  - The compiler must faithfully implement the defined semantics of a programming language.
- The compiler must improve the source code in a discernable way
  - A direct translation of a source program results in highly inefficient code.

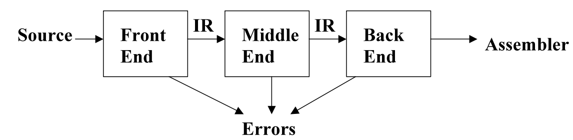
Some Possible Constraints

- Code speed
  - Very fast code might be the highest need of an application
    - e.g., weather
- Code size
  - How much space the object code requires
    - e.g., embedded systems
- Feedback
  - How much feedback is given to the user when an error is encountered
- Compile time
  - programs need to be compiled as fast as possible
- Debugging support
  - code improvements may make debugging difficult

25

Overview of Compiler Phases

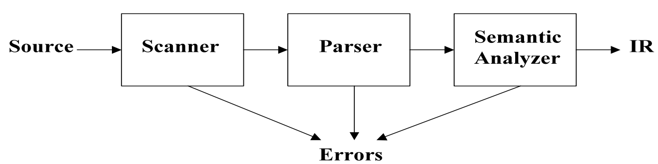
- Basic phases



26

Front End

- Syntax Analysis
  - determine if programs made up of valid sentences
    - scanner - valid words (reserved words, variables, etc.)
    - parser - valid sentence structure (*if* statements, etc.)
    - semantic analyzer - determine if sentences have meaning (type checking)

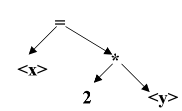


27

Front End

- Lexical Analysis
  - convert words in a program into tokens
    - <var>
    - +, -, =
    - IF, THEN, FOR
- Parsing
  - convert sentences into their structure

x = 2*y



28

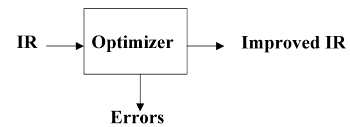
Front End

- Semantic Analysis (context-sensitive analysis)
  - after the program is parsed it is checked to make sure its meaning can be determined
    - type checking
    - number of parameters
    - variables declared
    - functions have prototypes
    - recursion supported or not
- If the program passes semantic analysis, it is converted into an intermediate representation (IR) that is used by the middle end and back end

29

Middle End

- Basic structure
  - often the IR is like assembler



- Optimizer is machine-independent

30

Middle End

- Optimization
  - remove redundancy
  - remove useless code
  - move code out of loops
  - use constants where possible
  - use less expensive operations
  - more … (cs4130/cs5130 in Spring 2013)

31

Back End

- Basic phases

- Order of allocation and scheduling may be different

32

Example

- Consider the following

  w = w * 2 * x * y * w*2

- Compiler first must recognize
  - variable names
  - =, *
- Next it must determine that the statement is in the source language
- Then, it must make sure the types are correct

Example

- Next it must allocate space for the variables
  - stack?
  - data segment?
  - registers?
- Then, the front end might generate

loadAI	r_{sp},@w → r_w
loadI	2 → r_2
mult	r_w, r_2 → r_{t1}
loadAI	r_{sp}, @x → r_x
mult	r_{t1}, r_x → r_{t2}
loadAI	r_{sp}, @y → r_y
mult	r_{t2}, r_y → r_{t3}
loadAI	r_{sp}, @w → r_w
mult	r_{t3}, r_w → r_{t4}
loadI	2 → r_2
mult	r_{t4}, r_2 → r_{t5}
storeAI	r_{t5} → r_{sp},@w

Example

- Optimization is next
  - eliminate extra loads of w and 2 and extra multiplication of w*2

loadAI	r_{sp},@w → r_w
loadI	2 → r_2
mult	r_w, r_2 → r_{t1}
loadAI	r_{sp}, @x → r_x
mult	r_{t1}, r_x → r_{t2}
loadAI	r_{sp}, @y → r_y
mult	r_{t2}, r_y → r_{t3}
mult	r_{t3}, r_{t1} → r_{t4}
storeAI	r_{t4} → r_{sp},@w

  - Code generation converts the intermediate into the target machines assembler.

Why Scheme?

- In this class we will introduce Scheme. Why?
- Developing skills in a functional language improves overall programming skills.
  - Gives more tools to solve problems
  - Recursion is important and powerful
- Scheme is small and simple
- Functional languages are used in AI, natural language recognition, vision systems, expert systems, rapid prototyping, studies of languages, editors (emacs), ...
- Exposes students to a different way to think about programming.

Scheme's Distinguishing Features

- Mostly functional
- Expression oriented
- Recursion
- Automatic storage allocation and collection
- PROGRAMS = DATA
- dynamic type checking