CS 230
Programming Languages

09 / 08 / 2022

Instructor: Michael Eckmann

Today's Topics

- Introduction
- Review the syllabus
- Chapter 1 material
  - Programming domains
  - Language design influences and tradeoffs
  - Language categories

Homework

- Read chapter 1 in Sebesta.
- An assignment will be due Wednesday by 11:59 pm. September 14 via email to meckmann@skidmore.edu

Who is your instructor?

- I'm Mike Eckmann and I have taught at Skidmore since 2004. Before that I was at Lehigh University in PA.
- I studied Mathematics and Computer Engineering and Computer Science all at Lehigh University.
- I was employed as a programmer (systems analyst) for eight years.

Syllabus

- Office hours
- Text book
- Assignments
  - programs
  - homeworks
- Collaboration policy
- Grading schema
- Workload
- Student preparation before class

Note: The most up-to-date syllabus will be found on the course web page.

Reason to study Prog. Langs.

- Increased ability to express ideas
  - Hard to conceptualize what you can't describe
  - Limited grasp of lang. -> limited complexity of thought
  - (read paragraphs from text) page 2.
  - May be able to apply what is learned in one language to another. How?
- For choosing appropriate languages
  - The more you know, the better you can fit the job to the language
  - What language will you choose if you only know one language?

Reason to study Prog. Langs.

- Anyone here know multiple natural (spoken/written, non-programming languages)?
  - How much do you agree with this: learning a second language allows you to learn more about your first language. How about: if you know two languages, it is easier to learn a 3rd, etc.
- Understand significance of implementation
  - Use language more intelligently, e.g. Faster execution, limit your use of dangerous features, find bugs easier
- Ability to design new languages
- Overall advancement of computing

Programming Domains

- Scientific applications
- Business applications
- Artificial intelligence
- Systems programming
- Web programming

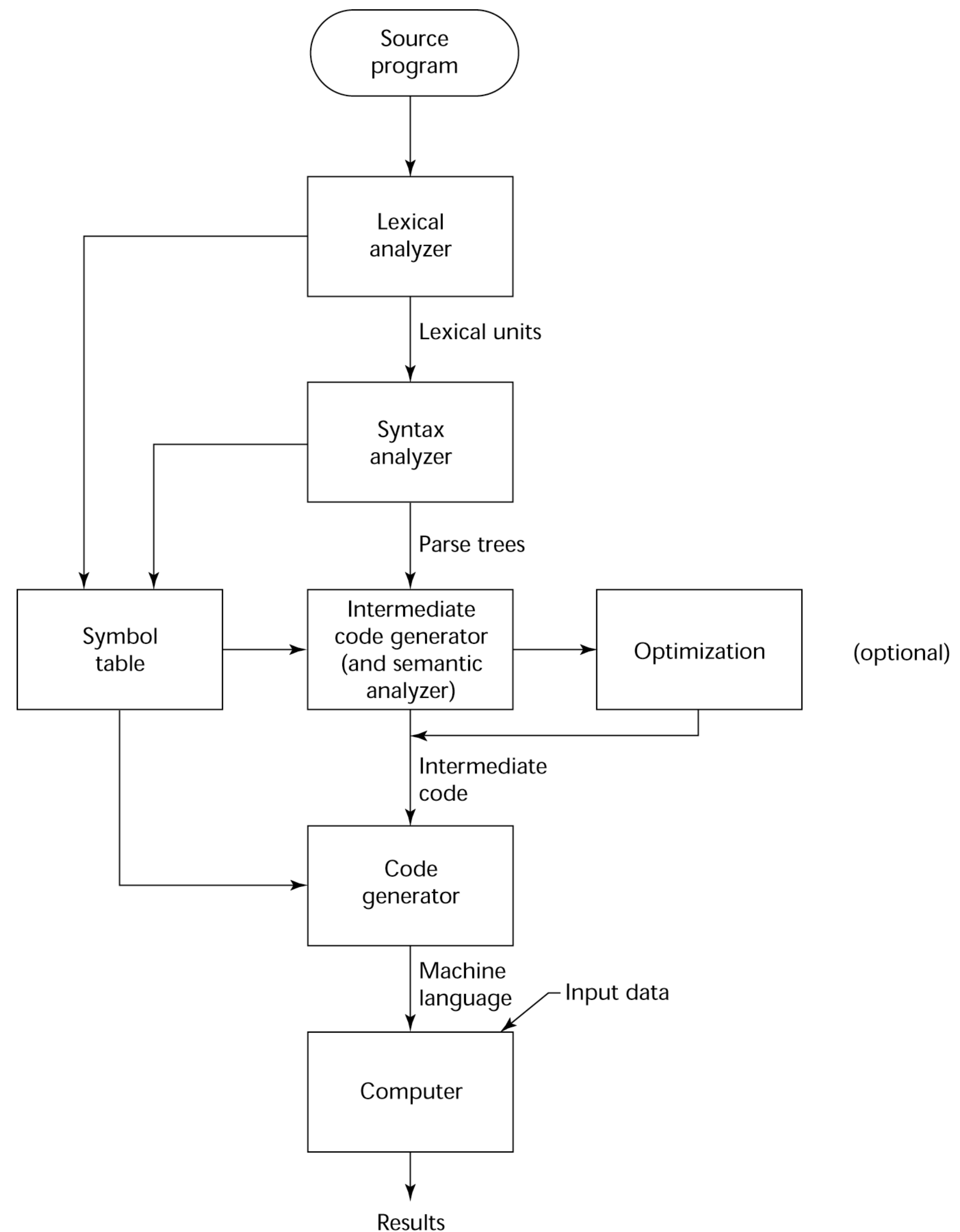
The major Language Categories

- Imperative
- Functional
- Logic
  - Rule based, no order of execution
- Object-Oriented (grew out of Imperative)
  - Abstract data types
  - Inheritance
  - Dynamic binding of method calls to methods

Compilation, Interpretation, Hybrids

- What is a compiler?
- What is an interpreter?
- What is a hybrid of these?
  - Java uses hybrid model
  - Compiler generates intermediate code that is then interpreted
  - Any advantages/disadvantages?

Compilation Process



Compilation Process

- Lexical analyzer – groups the code into variable names, reserved words, punctuation, etc.
- Syntax analyzer – constructs parse trees to determine if the lexical units in order are syntactically correct
- Both create items in the **symbol table** to refer to these units
- Intermediate code generator (and semantic analyzer) generates assembly-like code from the symbol table and parse trees.

Compilation Process

- Optimization may occur at this stage
  - What might the compiler try to optimize?
- Code generator takes the intermediate code and the symbol table to create the machine code for the particular architecture on which the code is compiled.
- Before execution, the machine code must be linked to any necessary programmed libraries and/or systems programs.

Language Evaluation

- When we talk about Language evaluation we're talking about evaluating how effective the **features** of a language are at allowing programmers to create programs that are readable, writeable, reliable, costly etc.
- In that way we can then compare languages to each other in terms of how readable, writeable, reliable, costly, etc. the programs in each of those languages tend to be.
- This is different from comparing, for instance, how readable 2 programs written in the same language are.

Language Evaluation

- Readability
  - Why care about readability?
  - Software lifecycle (time/$ spent on maintenance vs. on initial coding)
  - Simplicity (1 way vs. mult., op. overload) refer to sec. 1.3.1.1
  - Orthogonality (set of primitives combined in few ways in all combinations to build data structures) --- fewer exceptions to the rules – refer to parag. on p. 9
  - Data types (e.g. Using an int where a boolean is best)
  - Structures (e.g. Records vs. arrays of individuals)
  - Syntax (identifier rules (size, valid chars), special words, ending all blocks with same keyword or symbol decr. readab.)

Language Evaluation

- Writability
  - Orthogonality
    - if small set and all combinations make sense +
    - if large set and all combinations exist but some don't make sense to ever use or would be rarely used – (could cause undetected errors)
  - Abstraction (functional (methods/functions) & data (classes))
    - don't need to know/be reminded of/replicate the implementation details, just need to use what you wrote + allows details to be ignored
  - Expressivity (e.g. count++, for vs. while ...)
- Reliability
  - (+)Type checking (at run-time vs. compile-time vs. neither)
    - when is better?
  - (+)Exception Handling
  - (-)Aliasing (why?)

Language Evaluation

- Let's expand on the idea of Orthogonality
  - set of primitives combined in few ways in **all** combinations to build the control and data structures
  - all possible combinations are legal and meaningful
- Increased orthogonality implies decreased exceptions.
- Examples of non-orthogonality in C
  - structs can be returned from functions but arrays cannot
  - structs cannot have itself as a member (i.e. No recursive records).
  - arrays cannot have functions as elements
  - parameters are passed by value except arrays are essentially passed by reference

Language Evaluation

- Anyone want to give examples of non-orthogonality in Java?

Language Evaluation

- Anyone want to give examples of non-orthogonality in Java?
  - <, >, <=, >= allowed on all primitive types except boolean
  - All parameters are pass-by-value but
    - parameters of primitive types get their values copied, parameters of class types (which are references/addresses) get the address copied and hence the data at which the reference refers can change inside a method (and the change is reflected afterwards) --- not true for primitive types
    - So there's no way to make a primitive type variable change its value when passing in as a parameter
    - Similar situation as in C with arrays passed as a parameter
  - + is overloaded in String but no other operator overloading
  - chars are sometimes treated as ints and sometimes as the char itself because they are stored as an int but are displayed as a character
  - In early versions of Java, cases of a switch only allow an integer type

Language Evaluation

- Cost (time == $)
  - training programmers in the language
  - writing --- IDEs
  - compiling, executing --- compiler optimization tradeoff
  - Implementation system (e.g. The compiler and/or interpreter available for free?)
  - Poor reliability
  - Maintenance (up to 4 times cost of developing)
- Can you think of others?

Let's think about other evaluation criteria

- Any other language evaluation criteria you can think of?
- What are the pros and cons of case sensitivity in user defined names?