CS 230
Programming Languages

09 / 12 / 2022

Instructor: Michael Eckmann

Today's Topics

- Questions? / Comments? --- from the reading?
- Language evaluation continued …
- von Neumann architecture
- Imperative vs. Functional languages
- Describing the Syntax of a programming language

Homework

- Read chapter 1 in Sebesta.
- Assignment 1 is due Wednesday by 11:59 pm. September 14 via email to meckmann@skidmore.edu

Language Evaluation

- Recall our discussion of 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, not too costly etc.
- Readability, Writeability, Reliability, Cost are common criteria with which to evaluate language features
- We covered readability, writeability and reliability criteria last time

Language Evaluation

- 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?
  - <, >, <=, >= 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

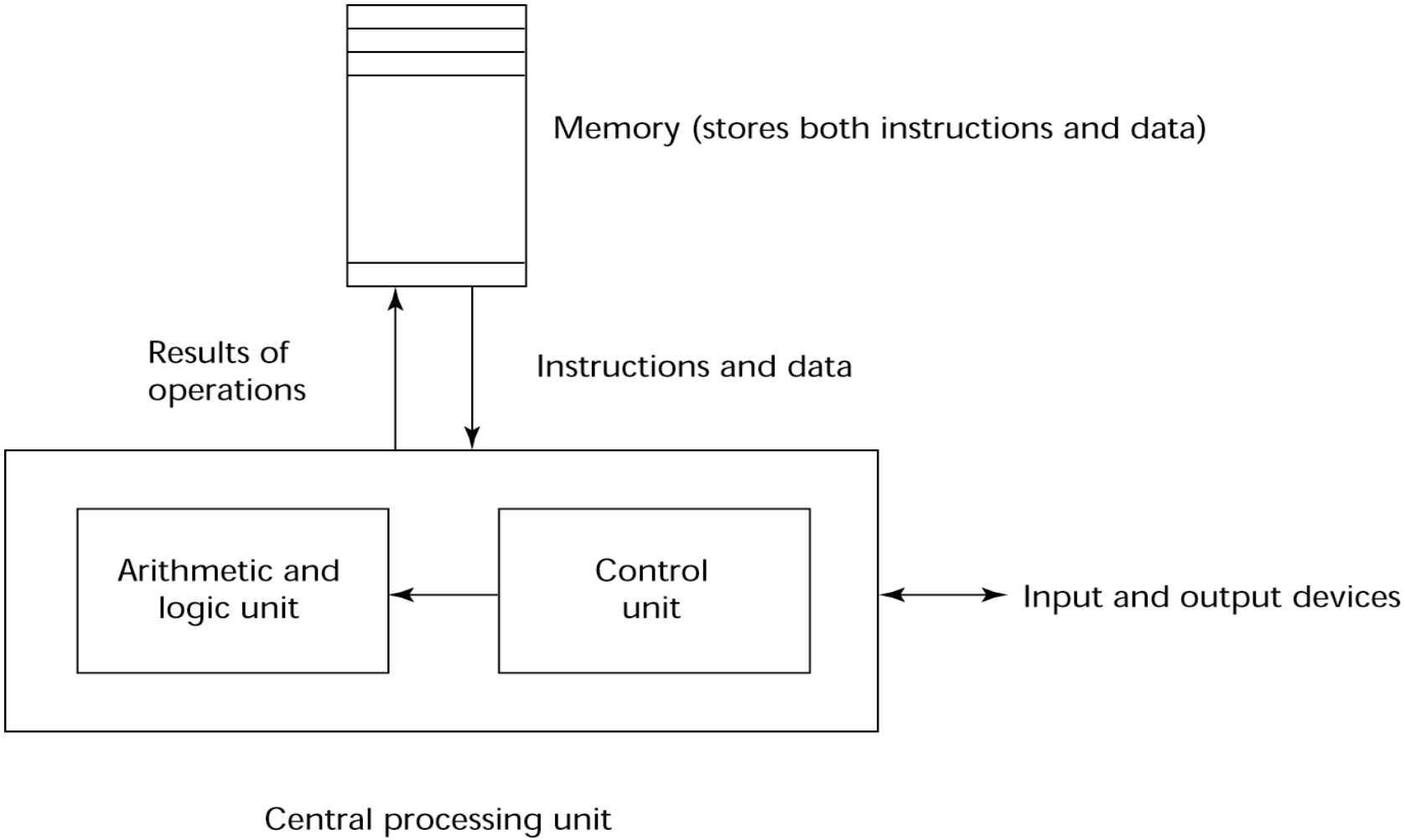
Language Evaluation

- 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?

von Neumann Architecture



Execution Process

- Fetch-execute cycle on a von-Neumann machine
  - Uses a program counter to know which instruction is next to be fetched.
  - While (true) {Fetch, increment, decode, execute}

von Neumann bottleneck

- transmission/piping between memory and CPU takes longer than executing instructions within the CPU.
- How is that a bottleneck?

Imperative Language Features

- Imperative languages are built **FOR** the von Neumann architecture
  - Data and programs stored in same memory
  - Memory is separate from CPU
  - Instructions and data are transmitted from memory to CPU, data transmitted back from CPU to memory
- Variables model memory cells
- Assignment statements model transmission
- Implications of these features
  - Iteration is efficient (b/c instructions are in adjacent memory cells, simple branch)
  - Recursion is inefficient (why?)

Functional vs. Imperative Languages

- Functional (e.g. Lisp, Scheme, et al.)
  - Apply functions to parameters
  - different use of variables
  - No assignment statements
  - No iteration, only recursion

  - Some programmers feel that there are benefits to computing using functional languages. So, why aren't they used more?

Functional vs. Imperative Languages

- Because of von Neumann, that's why!

- What might be a drawback to using a functional language on a von Neumann machine?

Imperative Languages

- von Neumann => imperative languages
- Many programmers may not realize this and think that imperative languages are the **natural** way to program.

Now Chapter 3 ...

- Describing a language
  - How to give it a clear and precise definition so that implementers (compiler writers) will get it right
  - How to describe it to users (programmers) of the language

Syntax and Semantics

- Describing a language involves both its syntax and semantics
- Syntax is the form, semantics is the meaning
  - e.g. English language example:
    - Time flies like an arrow.
      - Syntactically correct but has 3 different meanings (semantics)
- Easier to describe syntax formally than it is to describe semantics formally

Syntax

- A **language** is a set of strings (or sentences or statements) of characters from an alphabet.
  - e.g. the set of all the valid English *sentences* make up the English *language*

Syntax

- Lexemes vs. tokens
  - tokens (more general) are categories of lexemes (more specific)
  - e.g. Some tokens might be: identifier, integerLiteral, additionOperator
  - e.g. Some lexemes might be: idx, 42, +

Syntax

idx = 42 + count;

Lexemes	*Tokens*
idx	identifier
=	assignmentOperator
42	integerLiteral
+	additionOperator
count	identifier
;	statementTerminator

Syntax

- Recognizers and generators are used to define languages.
- Generators generate valid programs in a language.
- Recognizers determine whether or not a program is in the language (that is, valid syntactically.)

- Generators are studied in Chapter 3 (stuff coming up next in this lecture) and recognizers (parsers) in Chapter 4.

- How many valid programs are there for some particular language, say Java?

Syntax

- Context Free Grammars (CFGs) developed by Noam Chomsky are essentially equivalent to Backus-Naur Form (BNF) by Backus and Naur.
- They are used to describe syntax.
- These are termed metalanguages (languages used to describe languages.)

Syntax

- A Context Free Grammar is a four-tuple

(T, N, S, P) where

  - T is the set of terminal symbols
  - N is the set of non-terminal symbols
  - S is the start symbol (which is one of the non-terminals)
  - P is the set of productions of the form:
    - A -> X_1 . . . X_m where
      - A element of N
      - X_i element of N U T, 1 <= i <= m, m>=0

Syntax

- How are CFGs used to describe the syntax of a programming language?
- The nonterminals are abstractions (they represent larger structures)
- The terminals are tokens and lexemes
- The productions are used to describe programs, individual statements, expressions etc. They describe how a nonterminal (abstraction) can be expanded.

Syntax

- Example production:

<while_stmt> → `while` (<logic_expr>) <stmt>

- Everything to the left of the arrow is considered the left-hand side, LHS, and to the right the RHS.
- The only thing that can appear on the LHS is one nonterminal.
- Multiple RHS's for a LHS are separated by the | or symbol, e.g.

<compound_stmt> → <single_stmt> ;

| { <stmt_list> }

Syntax

- Recursion is allowed in productions, e.g.

<ident_list> → `ident`
| `ident`, <ident_list>

Syntax

- **An example grammar:**

<program> → <stmts>

<stmts> → <stmt>

| <stmt> ; <stmts>

<stmt> → <var> = <expr>

<var> → `a` | `b` | `c` | `d`

<expr> → <term> + <term>

| <term> - <term>

<term> → <var> | `const`

Syntax

- **Derivations are repeated applications of production rules.**
- **An example derivation:**

<program> => <stmts>

=> <stmt>

=> <var> = <expr>

=> a = <expr>

=> a = <term> + <term>

=> a = <var> + <term>

=> a = b + <term>

=> a = b + const