8/31/17

* Compilers vs. Interpreters
  + There are compiled languages (C) and interpreted languages (Python, Java)
  + Compiler
    - A compiler reads and translates a program entirely before running
    - Source program 🡪 Lexical analyzer (outputs lexical units/tokens) 🡪 Syntactic analyzer (outputs parse tree) 🡪 Semantic analyzer (outputs intermediate code) 🡪 Code generator => Machine language code
  + Interpreter
    - An interpreter reads and interprets lines of code as they are run
    - Source program 🡪 Lexical analyzer 🡪 Syntactic analyzer 🡪 Semantic analyzer 🡪 Code generator 🡪 (Android, PC, Mac, etc. JVM code, depending on platform)
    - Java creates .class file

9/05/17

* Program
  + Form/Syntax
    - A precise description of *all* grammatically correct programs
    - Analogous to grammar – set of rules
    - Noam Chomsky (1957) hierarchy:
      * Regular
      * Context-free
      * Context-sensitive
      * Free
    - Least to most expressive
    - Context-free grammar (CFG)
      * In programming, = Backus-Naur Form (BNF)
        + Set of terminal symbols, T, alphabet
        + Set of non-terminal symbols, N

e.g., start symbol, S

* + - * + Set of productions

A (non-terminal) 🡪 W (string of terminals and non-terminals)

Sentential form

String of terminals & non-terminals

* + Meaning/Semantics
  + BNF for integers
    - What is T?
      * Digits – 0-9
    - What is N?
      * Digit 🡪 0 |
        + Means digit represents 0 or 1 or 2 or … all
      * Integer 🡪 Digit | Integer Digit
        + Recursive definition: integer can be replaced by digit or integer + digit which is integer digit + digit which is …
    - Verification: Is 256 an integer using these rules? (Can we derive 256 using these rules)
      * Need start symbol (is “Integer”):
        + Apply second production rule (Integer 🡪 Integer Digit)

Integer => Integer Digit

* + - * + Apply second production rule again

=> Integer Digit Digit

* + - * + Apply first rule (Integer 🡪 Digit)

=> Digit Digit Digit

* + - * + One by one, replace digits: leftmost first

=> 2 Digit Digit

=> 2 5 Digit

=> 2 5 6

* + - * + Result is a sentence
      * ^ Leftmost derivation
      * You can shortcut steps, like from Digit Digit Digit to 256 using “=>\*” (star goes above) (meaning “in 0 or more steps)
    - Note: => means “derives”, 🡪 means “can be replaced by”
  + Language = set of all sentences that can be derived from the start symbol
    - Denoted L(G) where G is grammar
* Parse Trees
  + Integer 🡪 Integer Digit

Integer

* + - * Integer Digit
  + Use this to create 256:

Integer

* + - * + Integer Digit
      * Integer Digit 6
    - Digit 5
  + 2
* Expressions
  + e.g., 1 – 2 + 3
  + BNF for this?
    - T = {operators, operands} = {+, -, 0, 1, … 9}
    - Call digit “Term”
    - N:
      * Expr 🡪 Term + Expr | Term – Expr | Term (wrong)
      * Term 🡪 0 | 1 | … | 9
    - Derive 1 – 2 + 3:
      * Expr 🡪 Term – Expr
      * 🡪 Term – Expr + Expr
      * 🡪 Term – Term + Term
      * 🡪 1 – Term + Term
      * 🡪 1 – 2 + Term
      * 🡪 1 - 2 + 3
      * Can draw like expanding tree as you choose production rules
    - This leads to problems with order of operations. This evaluates to 1 – (2 + 3) = -4, should be (1 – 2) + 3 = 2. Production rule needs to be:
      * Expr 🡪 Expr + Term | Expr – Term | Term
      * Term 🡪 0 | 1 | … | 9
      * Derive:
        + Expr 🡪 Expr + Term
        + 🡪 Expr – Term + Term
        + 🡪 Term – Term + Term
        + …
      * Notice that this starts with the other production rule so the order is correct
    - First was “Right recursion”, where the tree expands to the right. Second was “Left recursion”, where tree expands to the left.
  + Rule of thumb:
    - Use left recursion for left association
    - Use right recursion for right association (e.g., exponentiation: )

9/07/17

* Expr 🡪 Expr + Prod | Expr – Prod | Prod, where “Prod” = Production symbol
* Prod 🡪 Prod \* Term | Term
* Term 🡪 0 | 1 | … | 9
* Conditional Statements
  + If – then – else
  + T = { ;, =, x, y, i, f, e, l, s, e, (, ) }
  + Stmt (statement) 🡪 Assignment | IfStmt
  + Asnmt 🡪 Id = Expr ; , where Id = Identifier for a variable
  + Id 🡪 x | y
  + Expr 🡪 …
  + IfStmt 🡪 if (BoolExpr) Stmt | if (BoolExpr) Stmt else Stmt
  + BoolExpr 🡪 … (later)
  + Can lead to “Dangling Else Problem” – hard to know which IfStmt to choose, given nested if’s with else inside (see slides)

9/12/17

* Extended BNF (EBNF)
  + Iteration {} – 0 or more iterations
  + Choice ( | ) – must take one
  + Option [] – either take or leave
  + “Kleen Star” – 0 or more times
  + “Kleen Plus” – 1 or more times
* BNF
  + Expr 🡪 Expr + Term | Expr – Term | Term
  + IfStmt 🡪 if(BoolExpr) Stmt | if(BoolExpr) Stmt else Stmt
* EBNF
  + Expr 🡪 Term{(+|-)Term}
  + IfStmt 🡪 if(BoolExpr) Stmt [else Stmt]
* Replaced recursion with loop
  + {} are like for loop – can choose + or – each time
* BNF is as powerful as EBNF!
* EBNF
  + A 🡪 x {y} z
* BNF
  + A 🡪 x A’ z
  + A’ 🡪 A’ y |
  + (empty string – no symbol)
* EBNF
  + A 🡪 x (y | z) w
* BNF
  + A 🡪 x y w | x z w
* EBNF
  + A 🡪 x [y] z
* BNF
  + A 🡪 x y z | x z
* You cannot replace right recursion using loops in EBNF.

9/14/17

* Lexical Syntax
  + Identifiers
    - Names – Stack, x, i push
  + Literals
    - 123, ‘x’, 3.25, true
  + Keywords
    - bool char else false float if int main true while
  + Operators
  + Punctuation
* Lexemes 🡪 Tokens
  + Identifier 🡪 Letter { Letter | Digit }
  + Letter 🡪 a | b | … | z | A | B | … | Z
  + Digit 🡪 0 | 1 | … | 9
  + Literal 🡪 Integer | Boolean | Float | Char
  + Integer 🡪 Digit { Digit }
  + Boolean 🡪 true | false
  + Float 🡪 Integer . Integer
  + Char 🡪 ‘ ASCII Char ‘

9/21/17

* Finite State Automata (FSA)
  + Behind the scenes of regular expressions
* Deterministic Finite State Automata (DFA)
  + At most one outgoing arc from any state for any particular input symbol
  + Easy to parse, not as expressive
* Non-deterministic F.A (NFA)
  + Allows multiple outgoing arcs from a state for the same input symbol
  + Expressive, but difficult to parse
* See notebook for Regex 🡪 NFA 🡪 DFA

9/26/17

10/03/17

* Can’t do recursive descent parsing with left recursion (hence the algorithm to remove left recursion)

10/12/17

* Scoping
  + ex: i = 10
    - Is there such an i?
    - Are there multiple?
      * Which one are we referring to?
* Static Scoping
  + Each function has its own symbol table
    - Stack of dictionaries
    - Each variable has some ‘binding’ that defines it (binding is line number in slides, but never actually used)
  + When you come across left curly, {, you push a new dictionary onto the stack
  + When you get a right curly, }, you pop
  + Don’t need to run the program to determine which scope a variable is in
* Dynamic scoping
  + One symbol table for the whole program
  + Can lead to strange results
* Modern languages use static scoping
* Other issues
  + Visibility (re-declaring a variable within new braces)
  + Function overloading
    - Same function name
    - How to distinguish?
      * Different parameters – number, type, name
      * Different return value
      * void, int, bool, etc.
  + Lifetime
    - Variable can be out-of-scope but still ‘living’ (one function calls another, the variables of calling function are out of scope but still living)
* Types
  + Statically typed
    - Types are fixed when they are declared at compile time
      * e.g., C
      * ‘int i’, then i can never be any other type
  + Dynamically typed
    - Type of a variable can vary at run time depending on the value assigned
      * e.g., Python
      * i = 10
      * i = ‘hello’
  + Strongly typed
    - Allows all type errors in a program to be detected either at compile time or at run time
    - Independent of static or dynamic
    - “Weakly Typed” leads to confusion – not the opposite of strongly typed, no consensus

10/17/17

* Type conversion
  + Widening
  + Ex: float y = 20.5, int x = 5, y = x results in conversion to float: y = 5.0, but x = y means x = 20 (narrowing)
* Non-basic types
  + pointer, string, array, list, etc. + programmer defined types
  + Dope vector
    - Information about array: element size, element type, # of elements
    - Followed by actual elements
  + 2-D Array?
    - Dope vector becomes | ele. size | ele. type | # rows | # cols | a[0][0] | a[0][1] …. = row major ordering
* C vs. Java Arrays
  + Java
    - Arrays are declared dynamically
    - Contiguous memory allocation not guaranteed
  + Both
    - Size and type cannot be changed after declaration

10/19/17

* Language is complete once type system, semantics, and syntax are well developed

10/24/17

* Exception handling
* Hardware vs. Programming Language
  + Hardware
    - Interrupt handler – div by 0/illegal memory address
    - Interrupt handler routine in hardware 🡪 back to next line of your program
    - ‘Resumption model’
  + Programming language
    - Your program: index out of bound, NULL pointer, etc. 🡪 corresponding exception handler 🡪 back to a specific part of your program
    - Can have multiple exception handlers for the same exception
    - Termination model vs. resumption model
      * In try catch, termination model does not continue the remainder of try block after exception
      * Resumption model goes to the rest of try block after catch
* Memory architecture
  + Static area 🡪 stack 🡪 heap
  + Static area
    - Fixed size, fixed content
    - Global variables – added using compile time
  + Stack
    - Varying size, content
    - Exclusively used for functions
  + Heap
    - Fixed size, varying content
    - Dynamical allocation (new Array)

10/26/17

* Garbage collection
  + Java reclaims unused memory (dynamically allocated)
  + Similar to delete/free from C++, but done automatically
* Reference counting Algorithm:
  + Activated by new, delete, or assigning one pointer to another
  + Data structure: free\_list
    - Linked list of free blocks
    - Each block (of what you allocate) has an RC (reference count)
  + Event 1: creation of a new incoming edge
    - Increase the RC of the block by 1
  + Even 2: deletion of an incoming edge
    - Decrease the RC of the block by 1
    - If the RC hits 0, add the block to the free\_list and decrease the RC of its direct descendent by 1. Recursively apply (b) if the descendent RC becomes 0.
* Mark-sweep Algorihtm
  + Activated by heap overflow
  + Data structure: mark bit (MB)
    - Initially set to 0
  + Two passes
    - Pass 1: Mark all nodes that are (directly or indirectly) accessible from the stack by setting their MB=1
    - Pass 2: Sweep through the entire heap and return all unmarked (MB=0) nodes to the free list
  + Pros
    - Reclaims *all* free blocks
    - Only called into action when heap overflows
  + Cons
    - When it’s called, everything will stand still
    - Needs two passes
* Class participation
  + Mask and sweep won’t come in until third line
* Copy Collection Algorithm