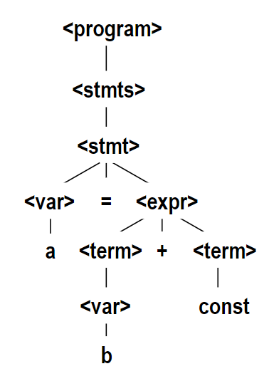
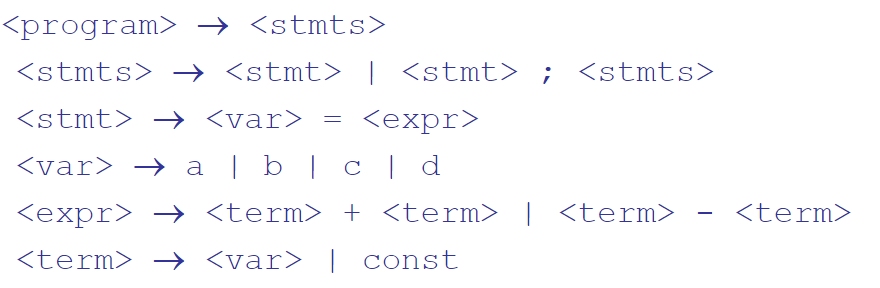
**Programming Language Concepts**

* Syntax and semantics provide a language’s definition
  + Syntax – form or structure of statements
    - Syntax error is missing token e.g. ; ) = etc.
  + Semantics – meaning of the statements
    - Semantic error ex. – type mismatch
* Sentence – string of characters over some alphabet, sentential form with only symbols
* Language – set of sentences
* Lexeme – lowest level syntactic unit of a language (e.g. variable, operator)
  + Lexical error – problem with a lexeme e.g. no end “ for string
* Token – category of lexemes (e.g. identifier)
* Recognizers – reads input strings and determines if is included in the alphabet of the language
* Generators – generates sentences of a language using derivation, if a sentence cannot be generated it is not syntactically correct
  + Derivation – repeated application of grammar rules, starts with start symbol and ends with sentence
    - Sentential form – string of symbols in derivation
    - Leftmost derivation – leftmost nonterminal in each sentential form is expanded
    - Derivation can be leftmost, rightmost, or neither
    - Example Derivation: a = b + const

<program> -> <stmts>

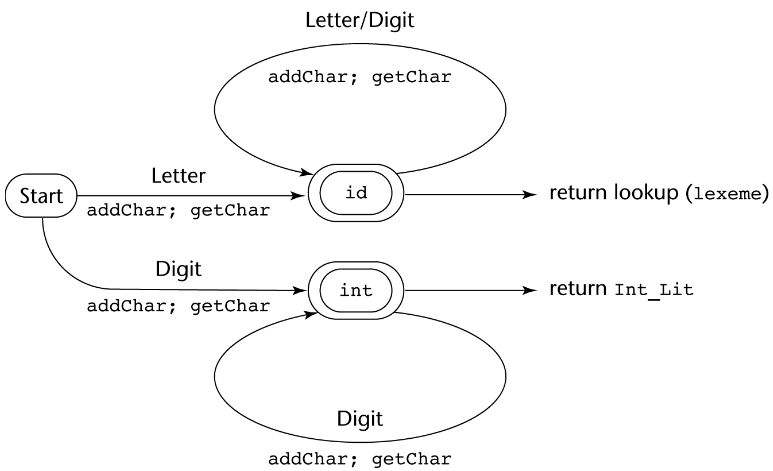
* + - * <stmt>
      * <var> = <expr>
      * <var> = <term> + <term>
      * <var> = <term> + const
      * <var> = <var> + const
      * <var> = b + const
      * A = b + const
    - Parse tree – other hierarchal form of derivation, cannot use tree when order of operations is needed, no ambiguity
    - Ambiguous grammar produces 2 or more equal parse trees for a sentence
* Context Free Grammers (e.g. Backus-Naur Form) – a set of rules for making language sentences
  + Made up of Nonterminals (placeholders), terminals (lexemes and tokens), productions (rules like in blue image), and start symbol
  + Left recursive – when a rule has left side also appearing in the beginning of the right side
  + Right recursive – when the left side is also in the end of the right side, right associativity
  + Optional parts of EBNF are placed in brackets [] e.g. <proc\_call> -> ident [<expr\_list>]
  + Alternative parts are in parenthesis and separated by bars e.g. <term> - > <term> (+|-) const
  + Repetitions (0 or more) are placed inside braces {} e.g. <ident> -> letter {letter|digit}
  + Finite languages have grammers without recursion or the potential to never end
* Perl regex pattern matching is similar
  + () used to group, [] denotes a set in which one element is to be picked
    - {num} gives number of times the [] is repeated
  + \* after group means repeat 0 or more times, + for repeat 1 or more times
  + \ before key char turns it into char
  + | is choice, pick one of 2 options
  + ? is optional, can be used or not

9-9

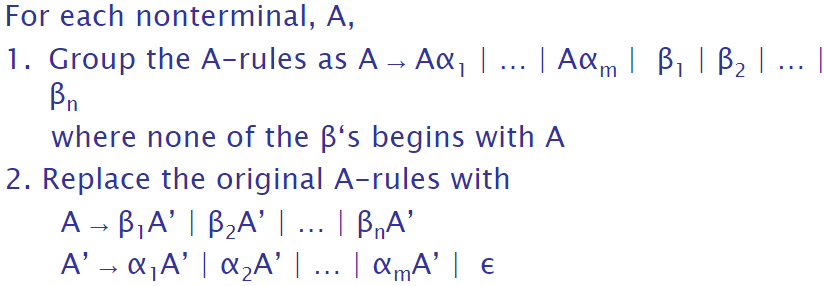
* Static semantics of language – for things that are difficult with BNF
  + ex. Operands in expressions, declaring variables before used
* Attribute grammars – addons to CFGs to carry semantic info on parse tree nodes
  + Used for static semantic specification
  + For each grammar symbol x there is a set A(x) of attribute values
  + Each rule has a set of functions that define attributes of its nonterminals
  + Rules may have predicates to check attribute consistency
  + Synthesized attributes – gets values from attributes attached to children nonterminal
  + Inherited attributes – gets values from attributes attached to parent nonterminal
  + Intrinsic attributes – default, initial attributes
  + Fully attributed – when a parse tree has all attribute values completed
* Operational semantics – describes meaning of a program by executing its statements on a machine
  + For high level language, virtual machine is needed
* Denotational semantics – defines a math object for each language entity, function maps instances of entities onto instances of objects
  + State of a program is a set variables (in pair form)

**CH 4**

* Language implementation systems must analyze source code
* Nearly all syntax analysis is based on formal description of language syntax (BNF)
  + Has 2 parts, low level lexical analyzer and high-level syntax analyzer/parser
    - Lexical analyzer – pattern matcher for character strings, identifies substrings in source code that belong together (lexemes/tokens)
    - Reserved words and identifiers can be identified by a table lookup
    - Characters are interpreted and given classes ex. Digit
    - Lexical analyzer subprograms:
      * getChar – gets next char in input, stores in nextChar, determines its class and stores it in charClass
      * addChar – puts nextChar char into place lexeme is being accumulated
      * lookup – determines whether the lexeme string is a reserved word
* State Diagram – maps and describes tokens in a diagram
  + One approach to making a lexical analyzer



* Parsing tasks: find all syntax errors, for each, produce a diagnostic message, and produce parse tree or at least a trace of it, for the program
  + Parsers that work for any unambiguous grammars are complex and inefficient O(n3)
    - Compilers use parsers that only work for subset of unambiguous grammars O(n)
  + Top-down parser – creates the parse tree starting at the root, leftmost derivation
    - Recursive-Descent parsing – each nonterminal in grammar has a subprogram which parses sentences that the nonterminal generates
      * EBNF makes RD parsing more efficient, minimizes nonterminals
      * Associated parsing subprogram – compares first terminal systems in each RHS of rule with the token stored in nextToken, if match continue, else error
      * If top down parser is used, grammar cannot use left recursion or have pairwise disjointness
        + Left recursion solution:



* + - * + Pairwise Disjointness – for rules with multiple RHS, none can have the same first terminal
  + Bottom-up parser – creates parse tree starting at the leaves, reverse of rightmost derivation
    - Finds substring of right sentential form that is RHS of rule that reduces to previous sentential form
    - LR family – a table driven parser family that stores the states of the parser
    - Parse action table

**CH 5**

* Imperative languages are abstractions of von Neumann architecture
  + Data and programs stored in memory
* In all languages except Fortran, key words are reserved words, can not be user defined names
* Named constant – variable that is bound to the same value through its lifetime
* Variable – abstraction of a memory cell(s), defined by attributes ex. name, memory address, value, type, lifetime, and scope
  + Variable names are not always defined; reserved words, name length, and case sensitivity can be restricting
  + Addresses of variables can change during execution
    - Aliases – variables that share a memory location
    - Abstract memory cell – physical cell(s) associated with a variable
  + Type determines range of values and set of operations
    - Explicit variable declaration is normal type declaration
    - Implicit declaration uses conventions like type inferencing
    - Strong typing – detecting all type errors
  + Scope – range of statements a variable is visible to
    - Local variables - defined with local scope (only for class or function)
    - Nonlocal variable – visible in the code block but not defined there
      * Global variable – type of nonlocal variables, accessed in local scope with keyword global
    - Referencing environment – a collection of names that are visible to the statement
    - Static Scoping - to find the variable a name references, a program searches for a declaration, first locally, then in increasing scope sizes
      * Enclosing scopes are called static ancestors, nearest ancestor is static parent
      * Referencing environment is local var + vars in static ancestors
    - Dynamic Scoping – based on function calls, not block nesting
      * Referencing environment is local var + vars in active subprograms
        + Active Subprogram - execution has begun but hasn’t ended
  + Explicit declaration – manual defining of variable type ex. Java
  + Implicit declaration – automatic defining of variable through type inferencing ex. Python, C#
* Binding – association between entity and attribute
  + Ex. Between variable and type or value, operation and symbol
  + Static binding – first occurs before runtime and remains unchanged through execution
  + Dynamic binding – first occurs during execution or can change during execution
    - Dynamic type binding – variable type can change
      * Pro is flexibility but Con is high cost, hard error detection
  + Binding time is time when binding takes place, 5 different types:
    - Language design time ex. Bind operator symbols to operations
    - Language implementation time ex. Bind floating point type to a representation
    - Compile time – during compilation ex. Bind variable to type in Java or C
    - Load time – when static variable is loaded to memory cell
    - Runtime – bind non-static local variable to memory
  + Storage bindings – allocation and deallocation of cells to and from pool of free memory cells
  + Binding lifetime – time a variable is stored in a cell
    - Static – first occurs before runtime and remains unchanged through execution
    - Stack dynamic – storage bindings are created for variables when their declaration statements are executed, all attributes except address are statically bound
      * Pros: allows recursion, conserves storage; Cons: allocation overhead, inefficient addressing
    - Explicit heap dynamic – anonymous variables allocated and deallocated in heap memory during execution (created by keyword like new)
      * Accessed only through pointer or reference (not static memory)
      * Ex. All objects in java
    - Implicit heap dynamic - anonymous variables allocated and deallocated in heap memory during execution by assignment (‘=’)
      * Pro: variable flexibility, Cons: inefficient (dynamic attributes) and loss of error detection
  + Let construct – in most languages, has a variable binding part and expressions part
* Everything static – run at compile time
* Everything dynamic – run at run time

**CH 6 Data Types**

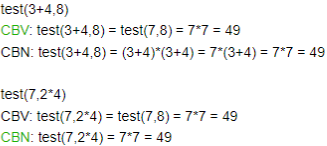
* Data Type – collection of data objects and set of predefined operations on those objects
* Descriptor – collection of attributes of a variable
* Object – represents an instance of a user-defined data type
* Primitive Data Types – most languages have
  + As many as 8 types of integers
    - 2s complement or sign magnitude notation for negative
  + Floats are only approximations of decimals
    - IEEE standard 754 is commonly used
  + Some languages (C#, COBOL) have decimal type
    - Each decimal number converted to byte or half-byte
    - More accurate than float but less range and storage efficient
  + Complex numbers (part real part imaginary) are primitive in some languages
  + Boolean is simplest, often stored in bytes
  + Characters encoded in ASCII (1 byte), some like Java, Javascript and C# use Unicode (2 byte)
  + String design vary by language, in some is char array in others is special array type
    - Comparison, catenation, substring, and pattern matching are common operations
    - In C and C++, no array size checking so functions like strcpy(str1, str2) could overflow into other storage
    - SNOBOL4 is string manipulation language
    - Regular expressions used with pattern matching operations in many languages
    - String length could be static size, limited dynamic (ended by end symbol), or dynamic without size limit
* Enumeration types – all possible values given with definition
  + C# ex. enum days {mon, tue, wed, thu, fri, sat, sun};
  + Types are translated into ints in C, not java or C#
* Arrays are accessed by subscripting (indexing): a mapping from indices to elements (usually uses [])
  + C and C++ don’t range check, Java and C# do
  + Static array – subscript ranges and storage allocation are static (before run time)
  + Fixed stack-dynamic – subscript ranges are static, but allocation is performed in run time
  + Stack-dynamic array – both binding of subscript ranges and storage allocation in run time
    - Once range is bound and storage is allocated, both remain fixed for array lifetime
  + Fixed heap-dynamic array – same as stack-dynamic except storage is allocated from heap
    - Ex. all java arrays
  + Heap-dynamic array – dynamic, binding subscript ranges and storage allocation can change
  + Heterogeneous array – elements do not have to have the same type
  + Rectangular array – multidimensional array where all rows and columns are same size
  + Jagged array – multidimensional array in which rows and columns have different sizes
    - Array of arrays
  + Slice – part of an array with properties of the whole array, could be row, column, range of subscripts or list of subscripts
  + Associative Array is like dictionary in python, key value pairs
    - Accessed with name{key}, deleted with key word delete
* Records are stored as fields in objects, faster access than array data
  + Each record stores record name, type, and offset address
* Tuple is like record except without names, allows multiple returns from methods
* Lists come in various forms in different languages
  + In LISP and schema, lists are in () without commas
    - Everything is in that format, including function calls, ‘ before ( signifies data
    - Lisp operations: CAR returns first list element, CDR returns all but first element, CONS puts first element into second list element to make new list, LIST turns all parameters into a list
  + Lists in ML are just like in Java, with [,] and homogeneous
  + In F# [;] is used and there is class type
  + Python lists don’t need to be homogenous and can be mutable
* Union – type whose variables can store different type values at different times during execution
  + Free Union – unions with no type checking like in C and C++
    - Unsafe, easy to access wrong data, not used in Java and C#
  + Discriminated Unions – unions have a type indicator called discriminant
* Pointer type stores addresses used for indirect addressing and dynamic memory
  + Pointers are extremely flexible but must be used carefully
  + Pointer assignment gives the pointer variable an address
  + Dereferencing gives the value stored at the pointer’s address
    - Explicit with \*varName like in C and C++
    - Implicit without that symbol like in Java
  + Dangling pointers – store addresses that are no longer being used, unintended consequences
    - Tombstone – one solution, makes an extra heap cell that the variable now points to, tombstone points to deallocated variable, tombstone set to nil
      * Costly in time and space
    - Locks-and-Keys – another solution, heap-dynamic variables are represented as var + lock cell
      * When var is allocated, lock value is created to say that the cell is in use
    - Not a problem in language with Garbage Collector like Java
  + Lost heap-dynamic variable – a variable that is no longer accessible to the program
    - Process called memory leakage
  + Pointer arithmetic can be used with array elements; stuff[100], stuff[5] == \*(stuff+5)
  + & symbol in front of a variable or array element returns its address
  + Subtracting 2 pointers gives the distance between them
  + C++ and Java have object references, more limited than pointer
* Heap Management has complex run-time process, 2 approaches to reclaim garbage:
  + Reference counters – every cell has a counter for the number of pointers pointing at the cell
    - Simple incremental design but requires more space and execution time, circularly connected cells are a problem
  + Mark-Sweep – every heap cell has an extra bit for algorithm, all cells initially set to garbage, those that are pointed to are set as not, all others are put in list of available cells
    - Simple but causes execution delays when run
    - Variable-size cells (as oppose to single-size) makes more complicated, initial setting of indicators and maintaining list of available space is difficult
      * Most programming languages require variable-size cells
* Type Checking – activity of ensuring an operator’s operands have compatible types
  + Compatible type – legal for the operator or allowed to be coerced by the language rules
    - Coercion – automatic type conversion during assignment ex. double = int in Java
    - Type error – inappropriate type for an operation
  + If type bindings are static, type checking is too, same with dynamic
  + Strongly Typed – when a programming language always detects type errors
    - C and C++ are weakly typed and ML and F# are strongly typed
    - Java and C# are almost strongly typed because of explicit type casting
    - Coercion rules can weaken strong typing considerably
* Type Equivalence – variables having the same type
  + Structure Type Equivalence – variables have equivalent types if the types have identical structures
  + Derived Type – derived from an existing type, Ada ex. type Celsius is new Float
    - Identical structure but different type, inherits parent properties
  + Ada’s Subtype – similar but different from derived type, usually constrains value range
  + Anonymous variables in Ada are assigned without type, cannot be type equivalent
  + Ada is strongly typed

**Lambda Calculus**

* System of mathematical logic for expressing computation based on function abstraction and application using variable binding and substitution
* Functions can be written in lambda notation, f(x) = x as λx.x
* Equality of functions – functions are equal if they give the same output given the same input
* A variable after ‘.’ Is called lambda term (lambda abstraction)
* Operation on lambda variables is lambda application
* λ is binder of variables
* Bound variable – variable after λ
* Free variable – variable in application not bound
* α Equivalence – functions with the same structure but different variable names are the same
  + λx.x is same as λy.y
* β Reduction is fundamental to evaluating lambda functions
  + Set bound variable equal to part of expression right after the binding lambda’s application
  + Can only reduce one variable at a time

**Scala**

Programming Paradigms:

* Imperative Programming – variable mutating with assignments, control structures (if statements, loops) etc.
* Functional Programming – all features of imperative programming but with functions added, based around functions
* Logic Programming – programs are a set of logical rules and are run based on logic deductions
* Object-Oriented Programing – based around objects/classes
* Substitution model – expression and function evaluation, reduces an expression to a value or smaller expression
  + Formalized by lambda calculus
* Call By Name (CBN) vs Call By Value (CNV)
  + Both strategies result in same value if they both terminate and reduced expression consists of pure functions
  + CNB is not simplified/reduced until it is necessary
  + Different result example:

def first(x: Int, y: Int) = x

def loop: Int = loop

* + - first(1,loop) terminates with CBN but loops forever with CBV
* Scala key words/symbols:
  + val is fixed value ex. val = 1 + 2
  + var is modifiable variable ex. var = 4 + 1
  + def defines a function and uses CBN semantics ex. def first(x: Int, y: Int) = x
    - Can create blocks with {} after = for functions, can define other functions in blocks
      * A block is considered an expression with the value of its last expresion
  + If-else statements use true, false, !, &&, ||, and comparison operations
    - Ex. def abs(x: Int) = if (x >= 0) x else –x
  + Semicolons are optional unless used to separate expressions on the same line
  + Single expressions can use multiple lines if in () or operator is on first line
* Higher-Order Functions – functions that take other functions as a parameter
  + Ex. def sum(f: Int=> Int, a: Int, b: Int): Int=if (a > b) 0 else f(a) + sum(f, a+1, b)
* Anonymous functions have no names, are just function literals
  + Ex. def sumCubes(a: Int, b: Int) = sum(x => x \* x \* x, a,)
* Can create Objects which make a block which can have functions and expressions
* Currying – calling a function with multiple ()s of parameters essentially calls the function recursively with each parameter
  + Ex. Def f = (args1 => (args2 => ... (argsn=> E)...)
* Lists are immutable, homogeneous, and type inferencing
  + Empty list contains Nil
  + List constructor is recursive ex. fruit = “apple” :: (“orange” :: (“mangoes” :: Nil)
  + 3 basic functions: isEmpty, head (first element) and tail (list of all after tail)

**CH 7 Expressions & Assignments**

* Expressions specify computations in a programming language
  + Order of operations and operand evaluation is essential
  + Arithmetic Expressions were motivation for first languages
    - Unary, binary, and ternary operators have 1, 2, and 3 operands
    - Operator precedence rules for evaluation determine which operations are used first
      * Typically parenthesis, unary operators, exponents, \*/, +-
      * With APL, all operators have equal precedence and evaluate right to left
    - In Ruby and Lisp, arithmetic operations call methods, which can be overridden
    - C, C++ and Java have a 1 line conditional formal
      * Ex. average = (count == 0)? 0 : sum / count

Is same as

If (count == 0)

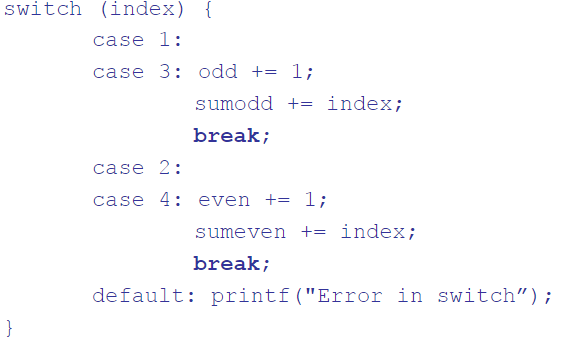
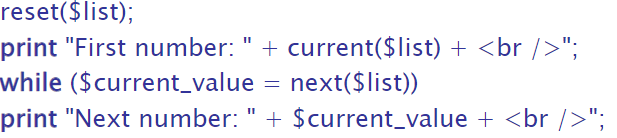
Average = 0

Else

Average = sum / count

* + - Side Affect – when a function or operation performs other actions in addition to returning a value, ex ++ or –
  + Operator Overloading – using an operator for more than 1 purpose (like \* in C)
    - In C++, C# and F#, user-defined overloaded operators are allowed
  + Short Circuit Evaluation – when an expression result is determined without evaluating all operations, ex. (a \* 2) \* (b++) when a = 0
* Different types of type conversion
  + Widening conversion – to type with extra space and no loss, ex short to long
  + Narrowing conversion – to type with less space and data loss, ex double to int
  + Mixed mode expressions have operands of different types, involve coercion
    - Casting – explicit type conversion, ex (int)sum
    - In ML and F#, no coercion allowed and no mixed mode
* Assignment Statements
  + Perl has conditional target assignment statements that assigns 1 of the 2 variables based on a condition, ex. ($flag ? $total : $subtotal) = 0
  + Compound Assignment Operator ex a += b
  + In Perl, Javascript, and C based languages, an assignment produces a result and can be used as an operand
  + Perl, Ruby, and Lua allow multiple-source, multiple-target assignments

**CH 8 Statement-Level Control Structures**

* Selection Statement – chooses between 2 or more paths of execution
  + 2 way selectors – standard if else
  + Multiple way selectors – more than 2 options, switch case structure
    - C based languages use this structure:
      * Control expression can only be integer type
      * Default clause for unpresented values
      * If no break, executes next case statements
    - C# requires a break or goto for every case and allows strings as control expression
  + Counter controlled loops in Python or Java 5
    - For x in list:
  + In loops for C, C++, and Python, continue skips the rest of the iteration but does not break the loop
  + Displays all elements of an array $list in PHP:

**CH 9 Subprograms**

* Subprogram declaration - gives the protocol but not the body of the subprogram
  + Formal parameter is listed in subprogram declaration
    - In some languages, formal parameters can have default values if no actual
  + Actual parameter is used in a subprogram call
  + Formal and actual parameters can correspond via position in list or by keyword (name =)
  + Different types of parameter passing
    - Pass-by-value (In Mode) – value of actual parameter is copied to initialize corresponding formal parameter
      * Used in C based languages and most language
    - Pass-by-result (Out Mode) – formal parameter acts as local variable, its value is copied to actual parameter
    - Pass-by-value-result (Inout Mode) – combination of both modes
    - Pass-by-reference (Inout Mode) – passes an access path
    - Pass-by-name (Inout Mode) – by textual substitution
* 2 subprogram categories:
  + Procedure – collection of statements that define parameterized computations
  + Functions structurally resemble procedures but are semantically modeled on math functions
* In most modern languages, local variables are stack dynamic
* Subprograms can be passed as parameters
  + Shallow binding – when environment of statement that calls passed subprogram is used
  + Deep binding – when environment in which passed subprogram is defined
  + Ad hoc binding – environment of call
* Generic or polymorphic subprogram takes parameters of different types
  + Ad hoc polymorphism - subprogram overloading
  + Subtype polymorphism – variable of type T can access any type T object or derived object
  + Parametric polymorphism – a subprogram having generic formal parameters
* Static scoping – wide
* Dynamic scoping – narrow
* delegates