**c1**:The study of programming languages is valuable for a number of reasons:

– Increase our capacity to use different constructs

– Enable us to choose languages more intelligently

– Makes learning new languages easier.

Most important criteria for evaluating programming languages include:

– Readability, writability, reliability, cost

Major influences on language design have been machine architecture and software development methodologies

The major methods of implementing programming languages are: compilation(Translate high-level program   
into machine code), pure interpretation (No translation), and hybrid implementation (A high-level language program is translated to an intermediate language that allows easy interpretation)

**c2**:Development, development environment, and evaluation of a number of important programming languages

• Perspective into current issues in language design.

**c3**:BNF and context-free grammars are equivalent meta-languages – Well-suited for describing the syntax of programming languages

• An attribute grammar is a descriptive formalism that can describe both the syntax and the semantics of a language

• Three primary methods of semantics description– Operation, axiomatic, denotational

Syntax: the form or structure of the expressions, statements, and program units

Semantics: the meaning of the expressions, statements, and program units

• Semantics: the meaning of the expressions, statements, and program units

• Syntax and semantics provide a language’s definition

• A sentence is a string of characters over some alphabet

• A language is a set of sentences

• A lexeme is the lowest level syntactic unit of a language (e.g., \*, sum, begin)

• A token is a category of lexemes (e.g., identifier)

• Recognizers – A recognition device reads input strings over the alphabet of the language and decides whether the input strings belong to the language

• Generators – A device that generates sentences of a language – One can determine if the syntax of a particular sentence is syntactically correct by comparing it to the structure of the generator

Context-Free Grammars – Developed by Noam Chomsky in the mid-1950s – Language generators, meant to describe the syntax of natural languages – Define a class of languages called context-free languages

Backus-Naur Form (1959) – Invented by John Backus to describe the syntax of Algol 58 – BNF is equivalent to context-free grammars

BNF Fundamentals: abstractions are used to represent classes of syntactic structures--they act like syntactic

variables (also called nonterminal symbols, or just terminals). Terminals are lexemes or tokens. A rule has a left-hand side (LHS), which is a nonterminal, and a right-hand side (RHS), which is a string of terminals and/or nonterminals. Nonterminals are often enclosed in angle brackets

Grammar: a finite non-empty set of rules. A start symbol is a special element of the nonterminals of a grammar

Syntactic lists are described using recursion <ident\_list> → ident | ident, <ident\_list>

A derivation is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols).

Every string of symbols in a derivation is a sentential form.

A sentence is a sentential form that has only terminal symbols

A leftmost derivation is one in which the leftmost nonterminal in each sentential form is the one that is expanded

A derivation may be neither leftmost nor rightmost

Parse Tree: A hierarchical representation of a derivation

A grammar is ambiguous if and only if it generates a sentential form that has two or more distinct parse trees

If we use the parse tree to indicate precedence levels of the operators, we cannot have ambiguity

Operator associativity can also be indicated by a grammar

EBNF: Optional parts are placed in brackets [ ] Alternative parts of RHSs are placed inside parentheses and separated via vertical bars. Repetitions (0 or more) are placed inside braces { } Alternative RHSs are put on separate lines. Use of a colon instead of => .Use of opt for optional parts. Use of one of for choices

Static Semantics: Nothing to do with meaning. Context-free grammars (CFGs) cannot describe all of the syntax of programming languages

Categories of constructs that are trouble: - Context-free, but cumbersome (e.g., types of operands in expressions) - Non-context-free (e.g., variables must be declared before they are used)

Attribute grammars (AGs) have additions to CFGs to carry some semantic info on parse tree nodes

Primary value of AGs: – Static semantics specification – Compiler design (static semantics checking)

Definition: An attribute grammar is a context-free grammar G = (S, N, T, P) with the following

additions: – For each grammar symbol x there is a set A(x) of attribute values – Each rule has a set of functions that define certain attributes of the nonterminals in the rule – Each rule has a (possibly empty) set of predicates to check for attribute consistency

Let X0 → X1 ... Xn be a rule

• Functions of the form S(X0) = f(A(X1), ... , A(Xn)) define synthesized attributes

• Functions of the form I(Xj) = f(A(X0), ... , A(Xn)), for i <= j <= n, define inherited attributes

• Initially, there are intrinsic attributes on the leaves

How are attribute values computed? – If all attributes were inherited, the tree could be decorated in top-down order. – If all attributes were synthesized, the tree could be decorated in bottom-up order. – In many cases, both kinds of attributes are used, and it is some combination of top-down and bottom-up that must be used.

Semantics: – Programmers need to know what statements mean – Compiler writers must know exactly what language constructs do – Correctness proofs would be possible – Compiler generators would be possible – Designers could detect ambiguities and inconsistencies

Operational Semantics: Describe the meaning of a program by executing its statements on a machine, either

simulated or actual. The change in the state of the machine (memory, registers, etc.) defines the meaning of the statement (To use operational semantics for a high- level language, a virtual machine is needed)

Hardware pure interpreter- too expensive. Software pure interpreter: actions difficult to understand, machine- dependent. A better alternative: A complete computer simulation: the process: – Build a translator (translates source code to the machine code of an idealized computer) – Build a simulator for the idealized computer. Evaluation of operational semantics: – Good if used informally (language manuals, etc.) – Extremely complex if used formally (e.g., VDL), it was used for describing semantics of PL/I.

Uses of operational semantics: - Language manuals and textbooks - Teaching programming languages

• Two different levels of uses of operational semantics: - Natural operational semantics - Structural operational semantics

• Evaluation:- Good if used informally (language manuals, etc.) - Extremely complex if used formally

Denotational Semantics: • Based on recursive function theory • The most abstract semantics description method.

• The process of building a denotational specification for a language: - Define a mathematical object for each language entity – Define a function that maps instances of the language entities onto instances of the corresponding mathematical objects • The meaning of language constructs are defined by only the values of the program's variables.

program state: The state of a program is the values of all its current variables. s = {<i1, v1>, <i2, v2>, ..., <in, vn>} Let VARMAP be a function that, when given a variable name and a state, returns the current value of the variable VARMAP(ij, s) = vj

<dec\_num> → '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | <dec\_num> ('0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9')  
Mdec('0') = 0, Mdec ('1') = 1, ..., Mdec ('9') = 9  
Mdec (<dec\_num> '0') = 10 \* Mdec (<dec\_num>)  
Mdec (<dec\_num> '1’) = 10 \* Mdec (<dec\_num>) + 1  
...  
Mdec (<dec\_num> '9') = 10 \* Mdec (<dec\_num>) + 9

Expressions: Map expressions onto Z ∪ {error} We assume expressions are decimal numbers, variables, or binary expressions having one arithmetic operator and two operands, each of which can be an expression

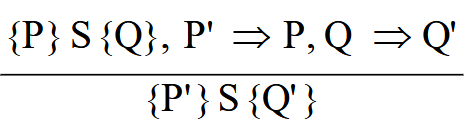
Loop: The meaning of the loop is the value of the program variables after the statements in the loop have been executed the prescribed number of times, assuming there have been no errors • In essence, the loop has been converted from iteration to recursion, where the recursive control is mathematically defined by other recursive state mapping functions

Denotational Semantics: • Can be used to prove the correctness of programs • Provides a rigorous way to think about programs • Can be an aid to language design • Has been used in compiler generation systems • Because of its complexity, it are of little use to language users

Axiomatic Semantics: • Based on formal logic (predicate calculus) • Original purpose: formal program verification • Axioms or inference rules are defined for each statement type in the language (to allow transformations of logic expressions into more formal logic expressions) • The logic expressions are called assertions. An assertion before a statement (a precondition) states the relationships and constraints among variables that are true at that point in execution • An assertion following a statement is a postcondition • A weakest precondition is the least restrictive precondition that will guarantee the postcondition

Program Proof Process: • The postcondition for the entire program is the desired result – Work back through the program to the first statement. If the precondition on the first statement is the same as the program specification, the program is correct.

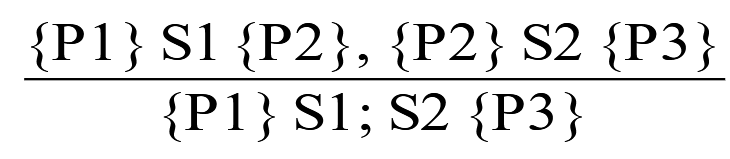
Axiomatic Semantics: Assignment: An axiom for assignment statements (x = E): {Qx->E} x = E {Q}

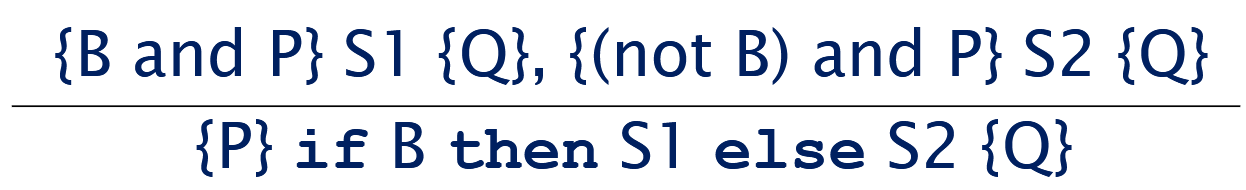
The Rule of Consequence: 

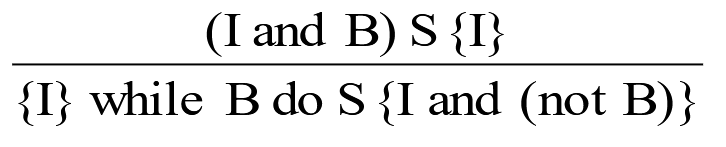
Axiomatic Semantics: Sequences: An inference rule for sequences of the form S1; S2

{P1} S1 {P2}

{P2} S2 {P3}



Axiomatic Semantics: Selection: An inference rules for selection - if B then S1 else S2

Axiomatic Semantics: Loops: An inference rule for logical pretest loops: {P} while B do S end {Q}(I is the loop invariant)

Axiomatic Semantics: Axioms: Characteristics of the loop invariant: I must meet the following conditions:

– P => I -- the loop invariant must be true initially

– {I} B {I} -- evaluation of the Boolean must not change the validity of I

– {I and B} S {I} -- I is not changed by executing the body of the loop

– (I and (not B)) => Q -- if I is true and B is false, Q is implied

– The loop terminates -- can be difficult to prove

Loop Invariant: The loop invariant I is a weakened version of the loop postcondition, and it is also a precondition. I must be weak enough to be satisfied prior to the beginning of the loop, but when combined with the loop exit condition, it must be strong enough to force the truth of the postcondition

Evaluation of Axiomatic Semantics: • Developing axioms or inference rules for all of the statements in a language is difficult • It is a good tool for correctness proofs, and an excellent framework for reasoning about programs, but it is not as useful for language users and compiler writers • Its usefulness in describing the meaning of a programming language is limited for language users or compiler writers

Denotation Semantics vs Operational Semantics: In operational semantics, the state changes are defined by coded algorithms • In denotational semantics, the state changes are defined by rigorous mathematical functions

**c4:** Syntax Analysis portion of a language processor nearly always consists of two parts: – A low-level part called a lexical analyzer – A high-level part called a syntax analyzer, or parser.

Advantages of Using BNF to Describe Syntax: Provides a clear and concise syntax description • The parser can be based directly on the BNF • Parsers based on BNF are easy to maintain

Reasons to Separate Lexical and Syntax Analysis: • Simplicity - less complex approaches can be used for lexical analysis; separating them simplifies the parser • Efficiency - separation allows optimization of the lexical analyzer • Portability - parts of the lexical analyzer may not be portable, but the parser always is portable

Lexical Analysis: A lexical analyzer is a pattern matcher for character strings • A lexical analyzer is a “front end” for the parser • Identifies substrings of the source program that belong together – lexemes The lexical analyzer is usually a function that is called by the parser when it needs the next token • Three approaches to building a lexical analyzer: – Write a formal description of the tokens and use a software tool that constructs a table-driven lexical analyzer from such a description – Design a state diagram that describes the tokens and  
write a program that implements the state diagram – Design a state diagram that describes the tokens and  
hand-construct a table-driven implementation of the state diagram In many cases, transitions can be combined

to simplify the state diagram – When recognizing an identifier, all uppercase and lowercase letters are equivalent – When recognizing an integer literal, all digits are equivalent - use a digit class Reserved words and identifiers can be recognized together

Convenient utility subprograms: – getChar - gets the next character of input, puts it in nextChar, determines its class and puts the class in charClass – addChar - puts the character from nextChar into the place the lexeme is being accumulated, lexeme – lookup - determines whether the string in lexeme is a reserved word (returns a code)

The Parsing Problem: Goals of the parser, given an input program: – Find all syntax errors; for each, produce an appropriate diagnostic message and recover quickly – Produce the parse tree, or at least a trace of the parse tree, for the program

Two categories of parsers–Top down - produce the parse tree, beginning at the root • Order is that of a leftmost derivation • Traces or builds the parse tree in preorder

Bottom up - produce the parse tree, beginning at the leaves • Order is that of the reverse of a rightmost derivation

• Useful parsers look only one token ahead in the input

Top-down Parsers – Given a sentential form, xAα , the parser must choose the correct A-rule to get the next sentential form in the leftmost derivation, using only the first token produced by A • The most common top-down parsing algorithms: – Recursive descent - a coded implementation – LL parsers - table driven implementation

Bottom-up parsers – Given a right sentential form, α, determine what substring of α is the right-hand side of the rule in the grammar that must be reduced to produce the previous sentential form in the right derivation – The most common bottom-up parsing algorithms are in the LR family

The Complexity of Parsing – Parsers that work for any unambiguous grammar are complex and inefficient ( O(n3), where n is the length of the input ) – Compilers use parsers that only work for a subset of all unambiguous grammars, but do it in linear time ( O(n), where n is the length of the input )

Recursive-Descent Parsing: • There is a subprogram for each nonterminal in the grammar, which can parse sentences that can be generated by that nonterminal

• EBNF is ideally suited for being the basis for a recursive-descent parser, because EBNF minimizes the number of nonterminals <expr> → <term> {(+ | -) <term>}

Assume we have a lexical analyzer named lex, which puts the next token code in nextToken

• The coding process when there is only one RHS: – For each terminal symbol in the RHS, compare it with the next input token; if they match, continue, else there is an error – For each nonterminal symbol in the RHS, call its associated parsing subprogram• This particular routine does not detect errors • Convention: Every parsing routine leaves the next token in nextToken• A nonterminal that has more than one RHS requires an initial process to determine which RHS it is to parse – The correct RHS is chosen on the basis of the next token of input (the lookahead) – The next token is compared with the first token that can be generated by each RHS until a match is found – If no match is found, it is a syntax error

The LL Grammar Class – The Left Recursion Problem • If a grammar has left recursion, either direct or indirect, it cannot be the basis for a top-down parser

The other characteristic of grammars that disallows top-down parsing is the lack of pairwise disjointness – The inability to determine the correct RHS on the basis of one token of lookahead

• Pairwise Disjointness Test: – For each nonterminal, A, in the grammar that has more than one RHS, for each pair of rules, A → αi and A → αj, it must be true that FIRST(αi) ⋂ FIRST(αj) = φ  
Bottom-up Parsing: The parsing problem is finding the correct RHS in a right-sentential form to reduce to get the previous right-sentential form in the derivation

Intuition about handles: – The handle of a right sentential form is its leftmost simple phrase – Given a parse tree, it is now easy to find the

handle – Parsing can be thought of as handle pruning

Shift-Reduce Algorithms – Reduce is the action of replacing the handle on the top of the parse stack with its corresponding LHS – Shift is the action of moving the next token to the top of the parse stack

Advantages of LR parsers: – They will work for nearly all grammars that

describe programming languages. – They work on a larger class of grammars than other bottom-up algorithms, but are as efficient

as any other bottom-up parser. – They can detect syntax errors as soon as it is possible. – The LR class of grammars is a superset of the class parsable by LL parsers.

Knuth’s insight: A bottom-up parser could use the entire history of the parse, up to the current point, to make parsing decisions

• Syntax analysis is a common part of language implementation

• A lexical analyzer is a pattern matcher that isolates small-scale parts of a program – Detects syntax errors – Produces a parse tree

• A recursive-descent parser is an LL parser – EBNF

• Parsing problem for bottom-up parsers: find the substring of current sentential form

• The LR family of shift-reduce parsers is the most common bottom-up parsing approach

**c5** Length: C99: no limit but only the first 63 are significant; also, external names are limited to a maximum of 31. C# and Java: no limit, and all are significant. C++: no limit, but implementers often impose one

Special characters: – PHP: all variable names must begin with dollar signs – Perl: all variable names begin with special characters, which specify the variable’s type – Ruby: variable names that begin with @ are instance variables; those that begin with @@ are class variables

Case sensitivity – Disadvantage: readability (names that look alike are different) • Names in the C-based languages are case sensitive • Names in others are not • Worse in C++, Java, and C# because predefined names are mixed case (e.g. IndexOutOfBoundsException)

Special word: – An aid to readability; used to delimit or separate statement clauses – A keyword is a word that is special only in certain contexts – A reserved word is a special word that cannot be used as a user-defined name – Potential problem with reserved words: If there are too many, many collisions occur (e.g., COBOL has 300 reserved words!)

Variables: – Name – Address – Value – Type – Lifetime – Scope

Variables Attributes: Name. Address - the memory address with which it is associated .Type - determines the range of values of variables and the set of operations that are defined for values of that type; in the case of floating point, type also determines the precision • Value - the contents of the location with which the variable is associated. Abstract memory cell - the physical cell or collection of cells associated with a variable

A binding is an association between an entity and an attribute, such as between a variable and its type or value, or between an operation and a symbol • Binding time is the time at which a binding takes place. • Language design time -- bind operator symbols to operations • Language implementation time—bind floating point type to a representation • Compile time -- bind a variable to a type in C or Java • Load time -- bind a C or C++ static variable to a memory cell) • Runtime -- bind a nonstatic local variable to a memory cell

• A binding is static if it first occurs before run time and remains unchanged throughout program execution. • A binding is dynamic if it first occurs during execution or can change during execution of the program

An explicit declaration is a program statement used for declaring the types of variables • An implicit declaration is a default mechanism for specifying types of variables through default conventions, rather than declaration statements • Basic, Perl, Ruby, JavaScript, and PHP provide implicit declarations – Advantage: writability (a minor convenience) – Disadvantage: reliability (less trouble with Perl). Some languages use type inferencing to determine types of variables (context) – C# - a variable can be declared with var and an initial value. The initial value sets the type – Visual Basic 9.0+, ML, Haskell, and F# use type inferencing. The context of the appearance of a variable determines its type

Dynamic Type Binding (JavaScript, Python, Ruby, PHP, and C# (limited)): Specified through an assignment statement– Advantage: flexibility (generic program units) – Disadvantages: • High cost (dynamic type checking and interpretation) • Type error detection by the compiler is difficult

Storage Bindings & Lifetime – Allocation - getting a cell from some pool of available cells – Deallocation - putting a cell back into the pool  
• The lifetime of a variable is the time during which it is bound to a particular memory cell

Static--bound to memory cells before execution begins and remains bound to the same memory cell throughout execution, e.g., C and C++ static variables in functions – Advantages: efficiency (direct addressing), history-sensitive subprogram support – Disadvantage: lack of flexibility (no recursion)

Stack-dynamic--Storage bindings are created for variables when their declaration statements are elaborated. (A declaration is elaborated when the executable code associated with it is executed) • If scalar, all attributes except address are statically bound – local variables in C subprograms (not declared static) and Java methods

• Advantage: allows recursion; conserves storage • Disadvantages: – Overhead of allocation and deallocation – Subprograms cannot be history sensitive – Inefficient references (indirect addressing)

Explicit heap-dynamic -- Allocated and deallocated by explicit directives, specified by the programmer, which take effect during execution • Referenced only through pointers or references, e.g. dynamic objects in C++ (via new and delete), all objects in Java • Advantage: provides for dynamic storage management • Disadvantage: inefficient and unreliable

Implicit heap-dynamic--Allocation and deallocation caused by assignment statements – all variables in APL; all strings and arrays in Perl, JavaScript, and PHP • Advantage: flexibility (generic code) • Disadvantages: – Inefficient, because all attributes are dynamic – Loss of error detection

The scope of a variable is the range of statements over which it is visible • The local variables of a program unit are those that are declared in that unit • The nonlocal variables of a program unit are those that are visible in the unit but not declared there • Global variables are a special category of nonlocal variables • The scope rules of a language determine how references to names are associated with variables

Static Scope: Based on program text • To connect a name reference to a variable, you (or

the compiler) must find the declaration • Search process: search declarations, first locally, then in increasingly larger enclosing scopes, until one is found for the given name • Enclosing static scopes (to a specific scope) are called its static ancestors; the nearest static ancestor is called a static parent • Some languages allow nested subprogram definitions, which create nested static scopes (e.g., Ada, JavaScript, Common Lisp, Scheme, Fortran 2003+, F#, and Python)

Variables can be hidden from a unit by having a "closer" variable with the same name

Blocks: A method of creating static scopes inside program units--from ALGOL 60

“LET” Construct: A let construct has two parts – The first part binds names to values – The second part uses the names defined in the first part

Declaration Order: C99, C++, Java, and C# allow variable declarations to appear anywhere a statement can appear In C++, Java, and C#, variables can be declared in for statements

Global Scope: • C, C++, PHP, and Python support a program structure that consists of a sequence of function definitions in a file. C and C++have both declarations (just attributes) and definitions (attributes and storage). PHP – Programs are embedded in HTML markup documents, in any number of fragments, some statements and some function definitions – The scope of a variable (implicitly) declared in a function is local to the function – The scope of a variable implicitly declared outside functions is from the declaration to the end of the program, but skips over any intervening functions. Python – A global variable can be referenced in functions, but can be assigned in a function only if it has been declared to be global in the function

Evaluation of Static Scoping: Works well in many situations • Problems: – In most cases, too much access is possible – As a program evolves, the initial structure is destroyed and local variables often become global; subprograms also gravitate toward become global, rather than nested

Dynamic Scope: Based on calling sequences of program units, not their textual layout (temporal versus spatial) • References to variables are connected to declarations by searching back through the chain of subprogram calls that forced execution to this point. Advantage: convenience – Disadvantages: 1. While a subprogram is executing, its variables are visible to all subprograms it calls 2. Impossible to statically type check 3. Poor readability- it is not possible to statically determine the type of a variable

Scope and lifetime are sometimes closely related, but are different concepts • Consider a static variable in a C or C++ function

The referencing environment of a statement is the collection of all names that are visible in the statement • In a static-scoped language, it is the local variables plus all of the visible variables in all of the enclosing scopes • A subprogram is active if its execution has begun but has not yet terminated • In a dynamic-scoped language, the referencing environment is the local variables plus all visible variables in all active subprograms

A named constant is a variable that is bound to a value only when it is bound to storage • Advantages: readability and modifiability • Used to parameterize programs • The binding of values to named constants can be either static (called manifest constants) or dynamic • Languages: – C++ and Java: expressions of any kind, dynamically bound – C# has two kinds, readonly and const - the values of const named constants are bound at compile time - The values of readonly named constants are dynamically bound

Case sensitivity and the relationship of names to special words represent design issues of names • Variables are characterized by the sextuples: name, address, value, type, lifetime, scope • Binding is the association of attributes with program entities • Scalar variables are categorized as: static, stack dynamic, explicit heap dynamic, implicit heap dynamic • Strong typing means detecting all type errors

**c6** Primitive Data Types: Those not defined in terms of other data types

Integer: Java’s signed integer sizes: byte, short, int, long

Floating Point: Model real numbers, but only as approximations (EEE Floating-Point Standard 754)

Complex: Each value consists of two floats, the real part and the imaginary part

Decimal: • Store a fixed number of decimal digits, in coded form (BCD) • Advantage: accuracy • Disadvantages: limited range, wastes memory

Boolean: Could be implemented as bits, but often as bytes

Character: Stored as numeric codings. Most commonly used coding: ASCII

String: Values are sequences of characters. Typical operations: – Assignment and copying – Comparison (=, >, etc.) – Catenation – Substring reference – Pattern matching• C and C++ – Not primitive – Use char arrays and a library of functions that provide operations • SNOBOL4 (a string manipulation language) – Primitive – Many operations, including elaborate pattern matching • Fortran and Python – Primitive type with assignment and several operations • Java (and C#, Ruby, and Swift) – Primitive via the String class • Perl, JavaScript, Ruby, and PHP - Provide built-in pattern matching, using regular expressions

String Length: Static: COBOL, Java’s String class. Limited Dynamic Length: C and C++. Dynamic (no maximum): SNOBOL4, Perl, JavaScript

String Implementation: • Static length: compile-time descriptor • Limited dynamic length: may need a run- time descriptor for length (but not in C and C++) • Dynamic length: need run-time descriptor; allocation/deallocation is the biggest implementation problem

User-Defined Ordinal Types: the range of possible values can be easily associated with the set of positive integers(Java – integer – char– Boolean)

Enumeration Types: All possible values, which are named constants, are provided in the definition(C# enum days {mon, tue, wed, thu, fri, sat, sun};)

array: homogeneous aggregate of data elements in which an individual element is identified by its position in the aggregate, relative to the first element.

Array Indexing: Fortran and Ada use parentheses Most other languages use brackets (• FORTRAN, C: integer only • Java: integer types only • Index range checking - C, C++, Perl, and Fortran do not specify range checking - Java, ML, C# specify range checking)

Subscript Binding and Array Categories: Static: subscript ranges are statically bound and storage allocation is static. Fixed stack-dynamic: subscript ranges are statically bound, but the allocation is done at declaration time. Fixed heap-dynamic: similar to fixed stack- dynamic: storage binding is dynamic but fixed after allocation. Heap-dynamic: binding of subscript ranges and storage allocation is dynamic and can change any number of times – Advantage: flexibility (arrays can grow or shrink during program execution) (• C and C++ arrays that include static modifier are static • C and C++ arrays without static modifier are fixed stack-dynamic • C and C++ provide fixed heap-dynamic arrays • C# includes a second array class ArrayList that provides fixed heap-dynamic • Perl, JavaScript, Python, and Ruby support heap-dynamic arrays)

Heterogeneous Arrays: the elements need not be of the same type • Supported by Perl, Python, JavaScript, and Ruby

Arrays Operations: • APL provides the most powerful array processing operations for vectors and matrixes as well as unary operators (for example, to reverse column elements) • Python’s array assignments, but they are only reference changes. Python also supports array catenation and element membership operations • Ruby also provides array catenation

Rectangular and Jagged Arrays: A rectangular array is a multi-dimensioned array in which all of the rows have the same number of elements and all columns have

the same number of elements • A jagged matrix has rows with varying number of elements. C, C++, and Java support jagged arrays • F# and C# support rectangular arrays and jagged arrays

Slices: A slice is some substructure of an array; nothing more than a referencing mechanism • Slices are only useful in languages that have array operations

Accessing Multi-dimensioned Arrays: – Row major order (by rows) – used in most languages – Column major order (by columns) – used in Fortran

Associative Arrays: an unordered collection of data elements that are indexed by an equal number of values called keys. Built-in type in Perl, Python, Ruby, and Swift

record is a possibly heterogeneous aggregate of data elements in which the individual elements are identified by names (COBOL)

• Records are used when collection of data values is heterogeneous • Access to array elements is much slower than access to record fields, because subscripts are dynamic (field names are static) • Dynamic subscripts could be used with record field access, but it would disallow type checking and it would be much slower

A tuple is a data type that is similar to a record, except that the elements are not named (Python, ML, and F#)

List: Lists in Lisp and Scheme are delimited by parentheses and use no commas. quote it with an apostrophe

List Operations in Scheme – CAR returns the first element of its list parameter (CAR ′(A B C)) returns A – CDR returns the remainder of its list parameter after the first element has been removed (CDR ′(A B C)) returns (B C) - CONS puts its first parameter into its second parameter, a list, to make a new list (CONS ′A (B C)) returns (A B C) - LIST returns a new list of its parameters (LIST ′A ′B ′(C D)) returns (A B (C D))

List Operations in ML – Lists are written in brackets and the elements are separated by commas – List elements must be of the same type – The Scheme CONS function is a binary operator in ML, :: 3 :: [5, 7, 9] evaluates to [3, 5, 7, 9] – The Scheme CAR and CDR functions are named hd and tl, respectively

F# Lists – Like those of ML, except elements are separated by semicolons and hd and tl are methods of the List class

Python Lists – The list data type also serves as Python’s arrays – Unlike Scheme, Common Lisp, ML, and F#, Python’s lists are mutable – Elements can be of any type

Unions: variables are allowed to store different type values at different times during execution C and C++ provide union constructs in which there is no language support for type checking; the union in these languages is called free union. Type checking of unions require that each union include a type indicator called a discriminant(ML, Haskell, and F#) Free unions are unsafe. Java and C# do not support unions

pointer type variable has a range of values that consists of memory addresses and a special value, nil • Provide the power of indirect addressing

Assignment is used to set a pointer variable’s value to some useful address • Dereferencing yields the value stored at the location represented by the pointer’s value

C++ includes a special kind of pointer type called a reference type that is used primarily for formal parameters. Java extends C++’s reference variables and allows them to replace pointers entirely. C# includes both the references of Java and the pointers of C++

Representations of Pointers: Large computers use single values • Intel microprocessors use segment and offset

Tombstone: extra heap cell that is a pointer to the heap-dynamic variable Locks-and-keys: Pointer values are represented as (key, address) pairs

Reference counters: maintain a counter in every cell that store the number of pointers currently pointing at the cell

Mark-Sweep: The run-time system allocates storage cells as requested and disconnects pointers from cells as necessary; mark-sweep then begins

Variable-Size Cells: All the difficulties of single-size cells plus more

Optional Types: useful when there is a need for a variable to indicate that it currently has no value (C#, F#, and Swift)

Type checking is the activity of ensuring that the operands of an operator are of compatible types

Strong Typing: – C and C++ are not: parameter type checking can be avoided; unions are not type checked – Java and C# are, almost (because of explicit type casting) - ML and F# are

Name Type Equivalence: the two variables have equivalent types if they are in either the same declaration or in declarations that use the same type name

Structure type equivalence means that two variables have equivalent types if their types have identical structures

Type theory is a broad area of study in mathematics, logic, computer science, and philosophy

The data types of a language are a large part of what determines that language’s style and usefulness • The primitive data types of most imperative languages include numeric, character, and Boolean types • The user-defined enumeration and subrange types are convenient and add to the readability and reliability of programs • Arrays and records are included in most languages • Pointers are used for addressing flexibility and to control dynamic storage management

**c7** unary: 1 operand, binrary: 2, ternary 3

precedence levels – parentheses – unary operators – \*\* (if the language supports it) – \*, / – +, -

Arithmetic Expressions: operator associativity rules for expression evaluation define the order in which adjacent operators with the same precedence level are evaluated

Ruby – All arithmetic, relational, and assignment operators, as well as array indexing, shifts, and bit-wise logic operators, are implemented as  
methods - One result of this is that these operators can all be overriden by application programs  
• Scheme (and Common Lisp) - All arithmetic and logic operations are by explicitly called subprograms - a + b \* c is coded as (+ a (\* b c))

Conditional Expressions/

Operand Evaluation Order: 1. Variables: fetch the value from memory 2. Constants: sometimes a fetch from memory; sometimes the constant is in the machine language instruction 3. Parenthesized expressions: evaluate all operands and operators first 4. The most interesting case is when an operand is a function call

Functional side effects: when a function changes a two-way parameter or a non-local variable

referential transparency: if any two expressions in the program that have the same value can be substituted for one another anywhere in the program, without affecting the action of the program

Overloaded Operators: Use of an operator for more than one purpose

A narrowing conversion is one that converts an object to a type that cannot include all of the values of the original type

A widening conversion is one in which an object is converted to a type that can include at least approximations to all of the values of the original type

mixed-mode expression is one that has operands of different types

Short Circuit Evaluation: An expression in which the result is determined without evaluating all of the operands and/or operators

**c8**. A control structure is a control statement and the statements whose execution it controls

Selection Statements/Clause Form/Nesting Selectors/Selector Expressions/Multiple-Way Selection

Approaches: – Multiple conditional branches – Store case values in a table and use a linear search of the table – When there are more than ten cases, a hash table of case values can be used – If the number of cases is small and more than half of the whole range of case values are represented, an array whose indices are the case values and whose values are the case labels can be used

iterative Statements: repeated execution of a statement or compound statement is accomplished either by iteration or recursion

Counter-Controlled Loops: A counting iterative statement has a loop variable, and a means of specifying theinitial and terminal, and stepsize values

Logically-Controlled Loops: Repetition control is based on a Boolean expression

Unconditional Branching: Transfers execution control to a specified place in the program

Guarded Commands: : if the order of evaluation is not important, the program should not specify one (Selection /Loop Guarded Command)

Variety of statement-level structures • Choice of control statements beyond selection and logical pretest loops is a trade-off between language size and writability • Functional and logic programming languages use quite different control structures

**c9** A subprogram definition describes the interface to and the actions of the subprogram abstraction – In Python, function definitions are executable; in all other languages, they are non-executable – In Ruby, function definitions can appear either in or outside of class definitions. If outside, they are methods of Object. They can be called without an object, like a function – In Lua, all functions are anonymous • A subprogram call is an explicit request that the subprogram be executed • A subprogram header is the first part of the definition, including the name, the kind of subprogram, and the formal parameters • The parameter profile (aka signature) of a subprogram is the number, order, and types of its parameters • The protocol is a subprogram’s parameter profile and, if it is a function, its return type. Function declarations in C and C++ are often called prototypes • A subprogram declaration provides the protocol, but not the body, of the subprogram • A formal parameter is a dummy variable listed in the subprogram header and used in the subprogram • An actual parameter represents a value or address used in the subprogram call statement

Actual/Formal Parameter Correspondence: • Positional – The binding of actual parameters to formal parameters is by position: the first actual parameter is bound to the first formal parameter and so forth – Safe and effective

• Keyword – The name of the formal parameter to which an actual parameter is to be bound is specified with the actual parameter

categories of subprograms– Procedures are collection of statements that define parameterized computations

– Functions structurally resemble procedures but are semantically modeled on mathematical functions

Pass-by-Value (In Mode): The value of the actual parameter is used to initialize the corresponding formal parameter

Pass-by-Result (Out Mode): When a parameter is passed by result, no value is transmitted to the subprogram; the corresponding formal parameter acts as a local variable; its value is transmitted to caller’s actual parameter when control is returned to the caller, by physical move

Pass-by-Value-Result (inout Mode): A combination of pass-by-value and pass-by-result

Pass-by-Reference (Inout Mode): Pass an access path

Pass-by-Name (Inout Mode): By textual substitution

Parameter Passing Methods: C – Pass-by-value – Pass-by-reference is achieved by using pointers as parameters • C++ – A special pointer type called reference type for pass-by- reference • Java – All non-object parameters are passed are passed by value So, no method can change any of these parameters – Object parameters are passed by reference. Fortran 95+ - Parameters can be declared to be in, out, or inout mode • C# - Default method: pass-by-value – Pass-by-reference is specified by preceding both a formal parameter and its actual parameter with ref • PHP: very similar to C#, except that either the actual or the formal parameter can specify ref • Swift: default passing method is by value, but pass-by- reference can be specified by preceding the formal with inout • Perl: all actual parameters are implicitly placed in a predefined array named @\_ • Python and Ruby use pass-by-assignment (all data values are objects); the actual is assigned to the formal

Overloaded Subprograms: has the same name as another subprogram in the same referencing environment (C++, Java, C#, and Ada include predefined overloaded subprograms)

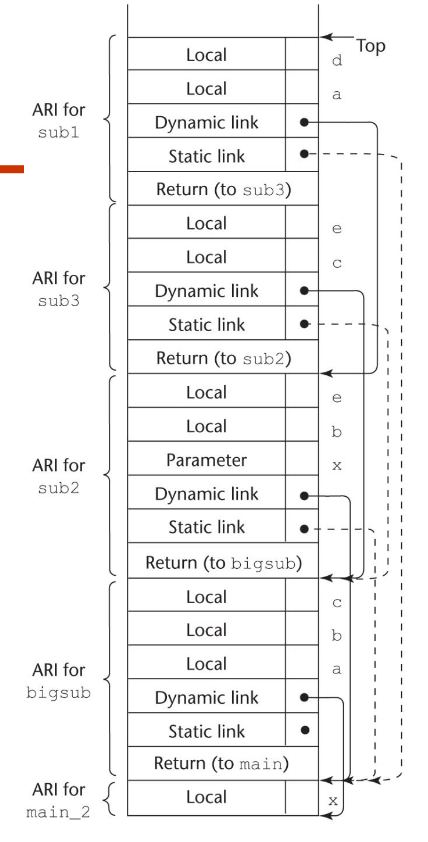
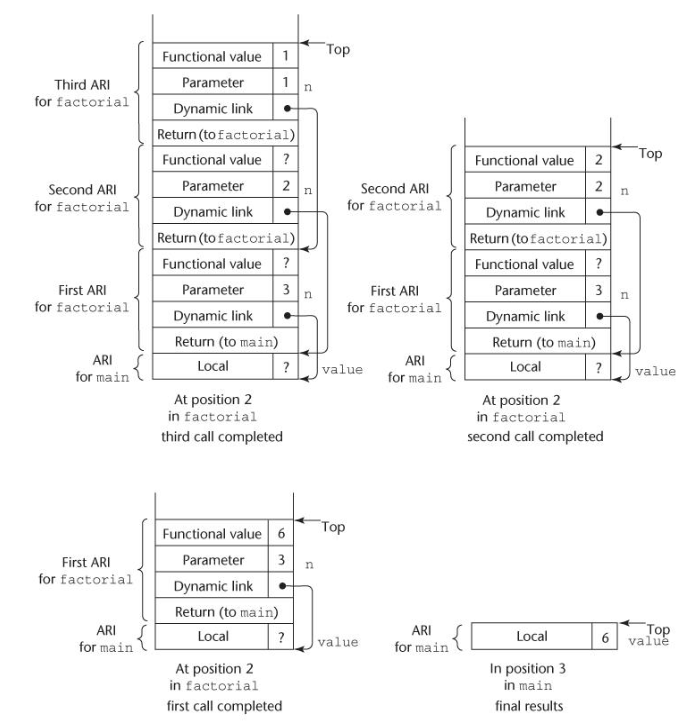
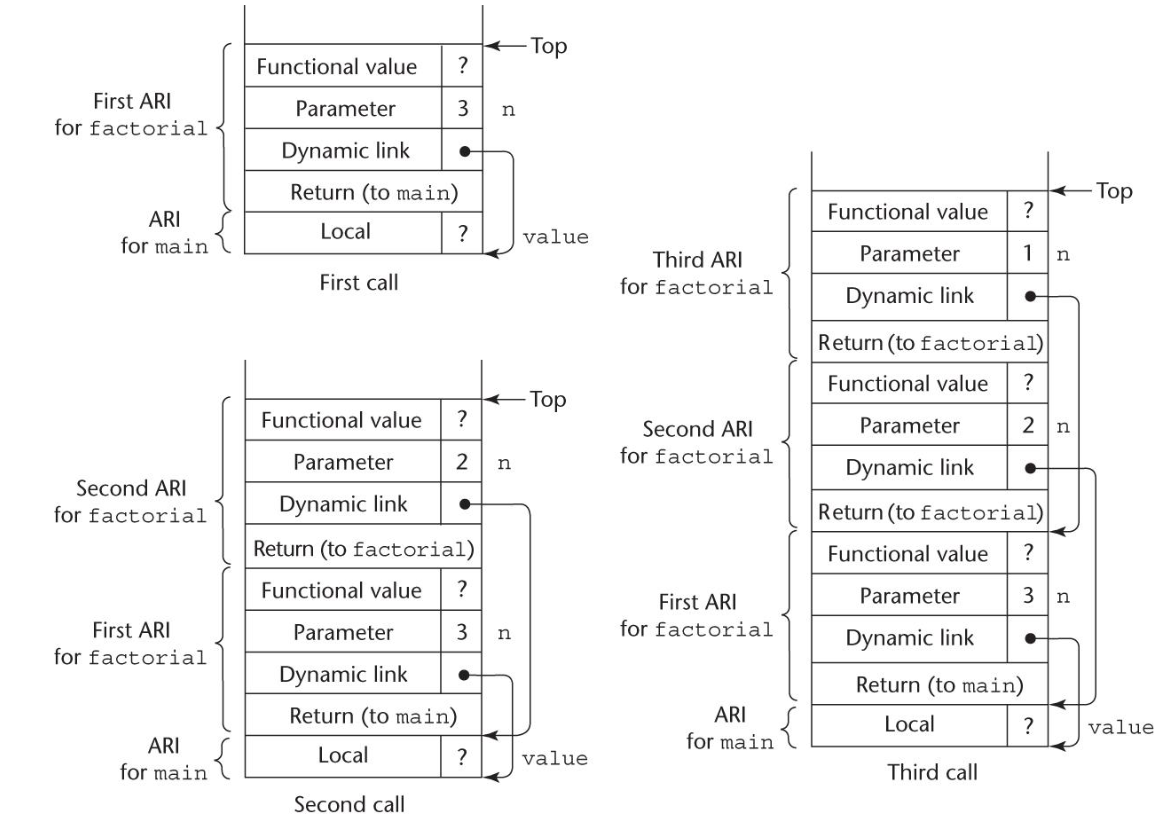
Generic Subprograms: takes parameters of different types on different activations

A subprogram that takes a generic parameter that is used in a type expression that describes the type of the parameters of the subprogram provides parametric polymorphism

Closures: a subprogram and the referencing environment where it was defined

Coroutines: subprogram that has multiple entries and controls them itself – supported directly in Lua

A subprogram definition describes the actions represented by the subprogram • Subprograms can be either functions or procedures • Local variables in subprograms can be stack- dynamic or static • Three models of parameter passing: in mode, out mode, and inout mode • Some languages allow operator overloading • Subprograms can be generic • A closure is a subprogram and its ref. environment • A coroutine is a special subprogram with multiple entries

**c10**. 

Subprogram linkage semantics requires many action by the implementation • Simple subprograms have relatively basic actions • Stack-dynamic languages are more complex • Subprograms with stack-dynamic local variables and nested subprograms have two components – actual code – activation record

**c11**  The concept of ADTs and their use in program design was a milestone in the development of languages • Two primary features of ADTs are the packaging of data with their associated operations and information hiding • Ada provides packages that simulate ADTs • C++ data abstraction is provided by classes • Java’s data abstraction is similar to C++ • C++, Java 5.0, and C# 2005 support parameterized ADTs • C++, C#, Java, and Ruby provide naming encapsulations

**c12**• OO programming involves three fundamental concepts:

ADTs, inheritance, dynamic binding

• Major design issues: exclusivity of objects, subclasses and

subtypes, type checking and polymorphism, single and

multiple inheritance, dynamic binding, explicit and implicit

de-allocation of objects, and nested classes

• Smalltalk is a pure OOL

• C++ has two distinct type systems (hybrid)

• Java is not a hybrid language like C++; it supports only OOP

• C# is based on C++ and Java

• Ruby is a relatively recent pure OOP language; provides some

new ideas in support for OOP

• Implementing OOP involves some new data structures

• Reflection is part of Java and C#, as well as most dynamically

types languages