2. Ders

Language Evaluation Criteria:

1- Readability :
   - The most important
   - Orthogonality (Features that are unique. There is no intersection. (For example: There are 3 ways to implement loop in C. This is not orthogonality)
   - Overall simplicity (Too many features is bad. Multiplicity of features is bad)

2- Writability :
   - Simplicity and orthogonality
   - Support for abstraction
   - Development enviroment
   - Expressivity (Conflicting with simplicity and orthogonality. There are so many ways to implement a thing)

- Orthogonality improves Read/Writability. Orthogonal languages are easier to learn. Assembly is orthogonal language.

3- Reliability :
   - Program behavior is the same on different platforms. (early versions of Fortran)
   - Type errors are detected (C vs ML)
   - Semantic errors are properly trapped (C vs C++)
   - Memory leaks are prevented (C vs Java)
   - Efficient implementation (If it is web page. Is it responsive?)

-

	CRITERIA		
Characteristic	READABILITY	WRITABILITY	RELIABILITY
Simplicity	•	•	•
Orthogonality	•	•	•
Data types	•	•	•
Syntax design	•	•	•
Support for abstraction		•	•
Expressivity		•	•
Type checking			•
Exception handling			•
Restricted aliasing			•

4- Cost :
   - Programmer training
   - Software creation
   - Compilation / Execution
   - Compiler cost
   - Maintenance

Partial and Total Functions

- Value of an expression may be undefined (3/0) or nonterminate (does not return anything), this type of functions called Partial functions. These functions may not give us an answer.
- A total function is one that will generate a value and do not terminate no matter what.
- Programming languages implements partial functions.

Paradigms

1- Imperative
   a. Procedural (C , Fortran)
   b. Object-Oriented (C++ , Java , C#)
2- Declarative
   a. Functional (Lisp , Haskell , Scheme)
   b. Logic (Prolog)
   c. Mathematical

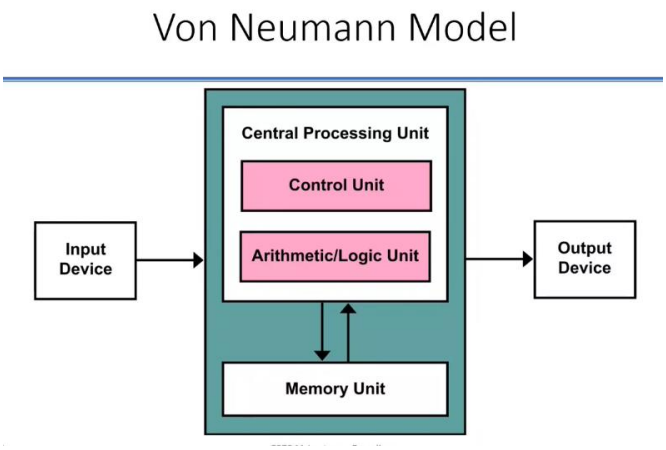
- In reality, very few languages are "pure"

A. Imperative (Procedural)
*Programs consists of actions to effect state change, principally through assignment operations or side effects.

*Easy to track the code. That is why it is mostly used

B. Functional & Logic

*Focuses on function evaluation; avoids updates, assignment , mutable state, side effects



Language Translation

- Native-code compiler (produces machine code)
    - Fortran, C , C++ , SML ...
- Interpreter (translates into internal form and immediately executes)
    - Scheme, Haskell, Python
- Byte-code compiler (produces portable bytecode, which is executed on virtual machine)
    - Java
- Hybrid approaches (Source-to-source translation (early C++ to C compile))

Compilation Process

i. Compilation: source code -> relocatable object code (binaries)

ii. Linking: many relocatable binaries (modules plus libraries) -> one relocatable binary

iii. Loading: relocatable -> absolute binary (with all code and data references bound to the addresses occupied in memory)

iv. Execution: control is transferred to the first instruction of the program

Phases of Compilation

a. Preprocessing (conditional macro text substitution)
b. Lexical Analysis (convert keywords, identifiers, constants into a sequence of tokens)
c. Syntactic Analysis (check that token sequence is syntactically correct)
d. Generate abstract syntax trees (check types)
e. Intermediate code generation ("walk" the ASTs)
f. Final code generation (produce machine code)

Language Interpretation

- Read in an expression, translate into internal form
- Evaluate internal form
- Print the result of evaluation
- Loop back to read the next expression

3. Ders

Von Neumann Programming Languages

*PL's that are high-level abstract isomophic copies of von Neumann architecture

Program variables <-> computer storage cells

Control statements <-> computer test-and-jump instructions

Assignment statements <-> fetching , storing instructions

Expressions <-> memory reference and arithmetic instructions

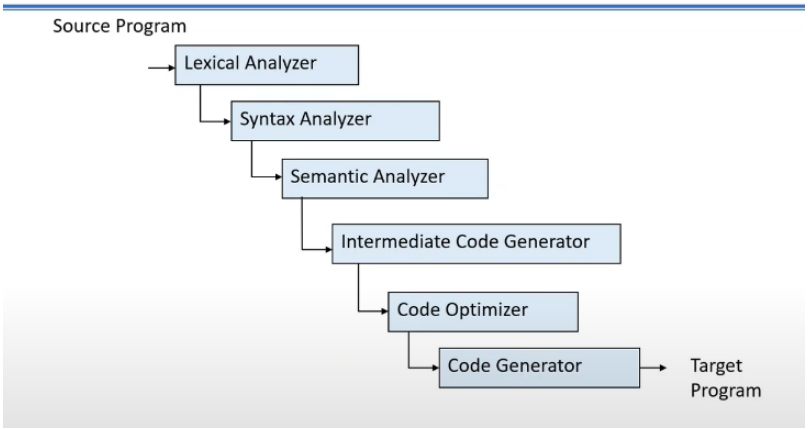
For Ex: C is very fundamental von Neumann language

Syntax and Semantics of Programs

- Syntax = The symbols used to write a program
- Semantics = The actions that ocur when a program is executed

Programming language implementation = Syntax -> Semantics

Typical Compiler



Syntax

- Syntax of a programming language is a precise description of all grammatically correct programs
  - Precise formal syntax was first used in ALGOL 60
- Lexical Syntax = Basic Symbols (names, values, operators, etc.)
- Concrete Syntax = Rules for writing expressions, statements, programs
- Abstract Syntax = Internal representation of expressions and statements, capturing their "meaning" (i.e., semantics)

Grammar

- A meta-language is a language used to define other languages
- A grammar is a meta-language used to define the syntax of a language. It consists of (Called BNF which is Backus-Naur Form):
  - Finite set of terminal symbols
  - Finite set of non-terminal symbols
  - Finite set of production rules
  - Start symbol
  - Language = (possibly infinite) set of all sequences of symbols that can be derived by applying production rules starting from the start symbol

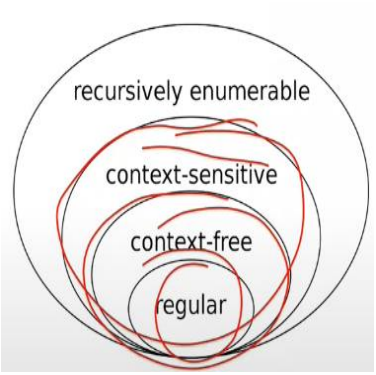
- Grammar for unsigned decimal integers
  - Terminal symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
  - Non-terminal symbols: Digit, Integer
  - Production rules:
    - Integer → Digit | Integer Digit
    - Digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
  - Start symbol: Integer
- Can derive any unsigned integer using this grammar

BNF

Derivation of 352 as an Integer

Integer	→ Integer Digit
	→ Integer 2
	→ Integer Digit 2
	→ Integer 5 2
	→ Digit 5 2
	→ 3 5 2

Chomsky Hierarchy



Regular Grammars:

- Regular expressions, finite-state automata
- Used to define lexical structure of the language

Context-free Grammars:

- Non-deterministic pushdown automata
- Used to define concrete syntax of the language

Context-sensitive Grammars:

- Unrestricted grammars
- Recursively enumerable languages, Turing machines

1. *Regular Grammars*

Left Regular grammar

- All production rules have the form
  A -> w or A-> Bw or A -> e (A: Non-terminal, w: Terminal, e: Empty)

Right regular grammar

- A -> w or A -> wB or A -> e
- Non-terminal symbol at the right of the terminal symbol (A -> wB)

Lexical Analysis

- Source code = long string of ASCII characters
- Lexical analyzer splits it into **tokens**
  - Token = sequence of characters (symbolic name) representing a single terminal symbol
- Identifiers: myVariable ...
- Literals: 123 5.67 true ...
- Keywords: char sizeof ...
- Operators: + - * / ...
- Punctuation: ; , } { ...
- Discards whitespace and comments

Regular Expressions -> A way to describe regular grammar

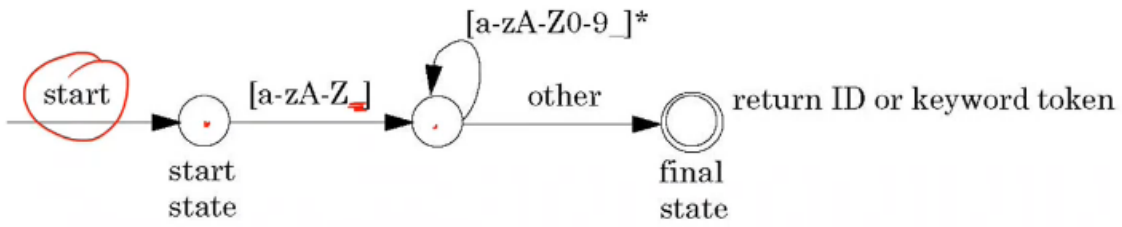
* We cannot use Regular Grammar for everything. We need to use Context-Free Grammar but it makes the algorithm complex.

Automatic Scanner Generation

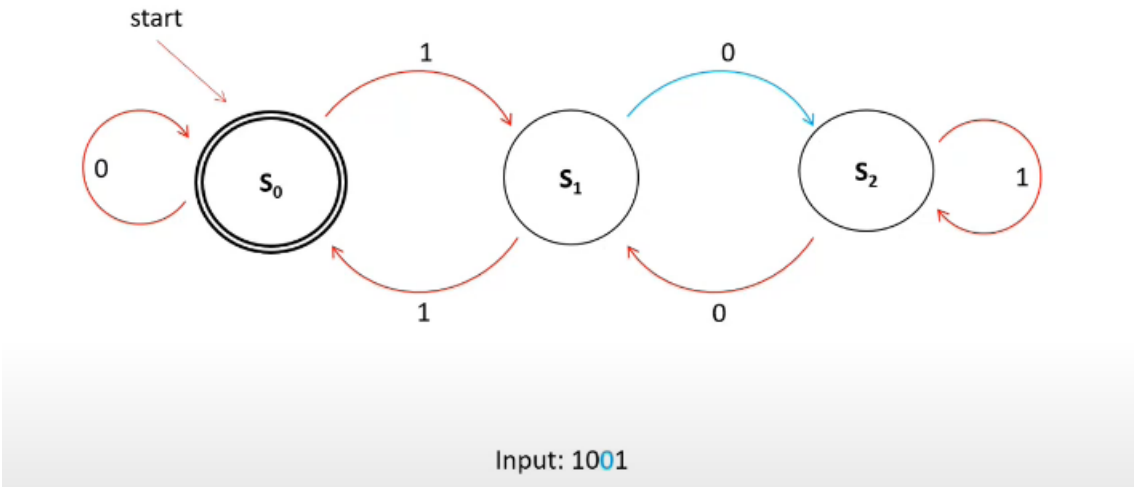
- Lexer or scanner recognizes and separates lexical tokens
    - Parser usually calls lexer when it's ready to process the next symbol (lexer remembers where it left off)
- Scanner code usually generated automatically
    - Input: Lexical definition (e.g., regular expressions)
    - Output: Code implementing the scanner
    - Typically, this is a deterministic finite automaton (DFA)
    - Examples: Lex, Flex (C and C++) , JLex (Java)

Finite State Automata (DFA)

- Set of states
    - Usually represented as graph nodes
- Input alphabet + unique "end of program" symbol
- State transition function
    - Usually represented as directed graph edges (arcs)
    - Automaton is deterministic if, for each state and each input symbol, there is at most one outgoing arc from the state labeled with the input symbol -> uniqueness of computation



- By default, any unmapped input => Error!



Traversing DFA: Start with leftmost (from given input), follow the states. We need to go to final state (So in this example). If we can't go then there is a problem.

2. *Context-Free Grammars*

- Used to describe concrete syntax
    - Typically using BNF Notation
- Production rules have the form A -> w
    - A is a non-terminal symbol, w is a string of terminal and non-terminal symbols
- Parse Tree = graphical representation of derivation
    - Each internal node = LHS (Left Hand Side) of a production rule
        # Internal node must be a non-terminal symbol
    - Children nodes = RHS of this production rule
    - Each leaf node = terminal symbol (token) or "empty"

5. Ders

Syntactic Correctness

- Lexical analyzer produces a stream of tokens
- Parser (syntactic analyzer) verifies that this token stream is syntactically correct by constructing a valid parse tree for the entire program
    - Unique parse tree for each language construct
    - Program = Collection of parse trees rooted at the top by a special start symbol
- Parser can be built automatically from the BNF description of the language's CFG

CFG For Floating Point Numbers

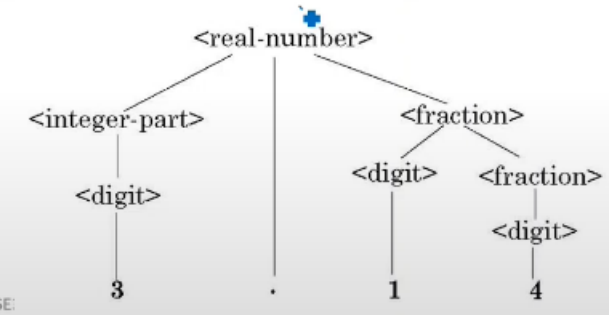
```
<real-number> ::= <integer-part> '.' <fraction-part>
<integer-part> ::= <digit> | <integer-part> <digit>
<fraction> ::= <digit> | <digit> <fraction>
<digit> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

::= stands for production rule

<...> are non-terminals

| represents alternatives for the right-hand side of a production rule

Sample parse tree:



Parse Trees

Grammars define hierarchical syntactic structure → parse trees

```
<assign> ::= <id> = <expr>
<id> ::= A | B | C
<expr> ::= <id> + <expr> | <id> * <expr> | ( <expr> ) | <id>
```

A = B * (A + C) can be generated by leftmost derivation...

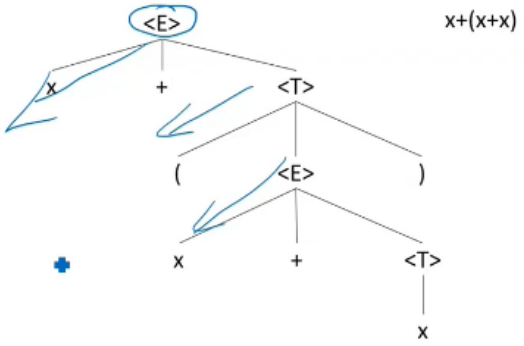
```
<assign> ⇒ <id> = <expr>
         ⇒ A = <expr>
         ⇒ A = <id> * <expr>
         ⇒ A = B * <expr>
         ⇒ A = B * ( <expr> )
         ⇒ A = B * ( <id> + <expr>)
         ⇒ A = B * ( A + <expr>)
         ⇒ A = B * ( A + <id>)
         ⇒ A = B * ( A + C)
```

Parsers

- Determine if the input program is syntactically correct
- Produce a parse tree for the correct input program
- Classify parsers by the order in which they build the parse tree
    - Top-down
    - Bottom-up

Recursive Descent Parsing

- Perform a depth-first search of the derivation tree for the input being parsed – top-down



LL Algorithms

- Recursive descent – coded directly from BNF
- Parsing table – do not implement BNF rules
- Both are versions of LL algorithms
    - Works on same subset of all CFGs
    - First L: Left to right scan of input
    - Second L: Leftmost derivation is generated (Top-down and leftmost comes first)

LR Parsing

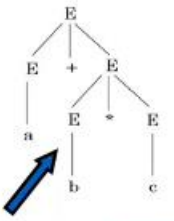
- Bottom-up parsing
- LR Algorithms
    - L: Left to right scan of input
    - R: Rightmost derivation is generated

Syntactic Ambiguity

<expr> ::= <expr> + <expr> | <expr> * <expr> | a | b | c

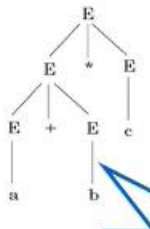
How to parse a+b*c using this grammar?

Parse Tree from a rightmost derivation starting from <expr> + <expr>



Both parse trees are syntactically valid

Parse Tree from a leftmost derivation starting with <expr> * <expr>



Only this tree is semantically correct (operator precedence and associativity are semantic, not syntactic rules)

Problem: this tree is syntactically correct, but semantically incorrect

Sentential Form

- For a grammar G, with start symbol S, any derivation $S \Rightarrow \alpha$ is called a sentential form:
  - If contains only terminal symbols, is a sentence in L(G)
  - If contains one or more non-terminals, it is just a sentential form (not a sentence in L(G))
- A left-sentential form is a sentential form that occurs in the leftmost derivation of some sentence
- A right-sentential form is a sentential form that occurs in the rightmost derivation of some sentence
- E.g.:
  - <P> $\Rightarrow$ begin <stmt_list> end

Derivation Order

- Leftmost derivation: the replaced non-terminal is always the left-most non-terminal in the previous sentential form
  - <assign> ⇒ <id> = <expr>
  - <assign> ⇒ A = <expr>
  - <assign> ⇒ A = <id> * <expr>
  - <assign> ⇒ A = B * <expr>
  - <assign> ⇒ A = B * <id>
  - <assign> ⇒ A = B * C
- Rightmost derivation: other way around…
  - <assign> ⇒ <id> = <expr>
  - <assign> ⇒ <id> = <id> * <expr>

<assign> := <id> = <expr>
<id> := A | B | C
<expr> := <id> + <expr> | <id> * <expr>
| (<expr>) | <id>

- Derivation order has no effect on the language generated by grammar.

Handle

- A handle of a right-sentential form γ is a pair $<A \Rightarrow \beta, k>$ where $A \Rightarrow \beta$ is a production rule and k is the position in γ of β's rightmost symbol.
- If $<A \Rightarrow \beta, k>$ is a handle, then replacing β in γ at position k with A produces the previous right-sentential form from which γ is derived in a rightmost derivation
  - $S \Rightarrow \alpha\ A\ \omega \Rightarrow \alpha\ \beta\ \omega$
- Because γ is a right-sentential form, the substring to the right of handle contains only terminal symbols ➔ the parser does not need to scan past the handle (only lookahead)
- If the grammar is **unambiguous**, then every right sentential form of the grammar has exactly one handle

- Handle denilen şey şu: Bottom-up parsingte en uzun eşleşebilecek deger alınır. Buradaki en uzun eşlenebilecek deger (string, cümle) bulunduktan sonra buna "handle" denir. Daha sonra reducing yapılarak bu "handle" daha abstract bir şeye dönüştürülür. "Handle" oluşurken "shifting" yapılarak kelime öbekleri sırasıyla en uzun cümleyi bulur.

Shift-Reduce Parsing

E ::= E + T | T
T ::= T * F | F
F := (E) | id

id+id*id

Step	Parse Stack	Right-most Sentential	LookAhead	Unscanned	Parser Action
0	*empty*		*id*	id+id*id	Shift
1	id	id+id*id	+	+id*id	Reduce
2	F	F+id*id	+	+id*id	Reduce
3	T	T+id*id	+	+id*id	Reduce
4	E	E+id*id	+	+id*id	Shift
5	E+	E+id*id	id	id*id	Shift
6	E+id	E+id*id	*	*id	Reduce
7	E+F	E+F*id	*	*id	Reduce
8	E+T	E+T*id	*	*id	Shift
9	E+T*	E+T*id	id	id	Shift
10	E+T*id	E+T*id	empty	empty	Reduce
11	E+T*F	E+T*F	empty	empty	Reduce
12	E+T	E+T	empty	empty	Reduce
13	E	E	empty	empty	Accept

Why not E*id?

Conflicts

- Generic shift-reduce strategy:
  - If there is a handle on top of the stack, reduce
  - Otherwise, shift
- But what if there is a choice?
  - If it is legal to shift or reduce, there is a
    - shift-reduce conflict
  - If it is legal to reduce by two different
    - productions, there is a reduce-reduce conflict
- Source of conflicts:
  - Ambiguous grammars always cause conflicts
  - So do many non-ambiguous grammars

Removing Ambiguity

- Define a distinct non-terminal symbol for each operator precedence level
- Define RHS of production rule to enforce proper associativity
- Extra non-terminal for smallest subexpressions

```
<expr> ::= <expr> + <expr> | <expr> * <expr> | a | b | c
```

6. Ders

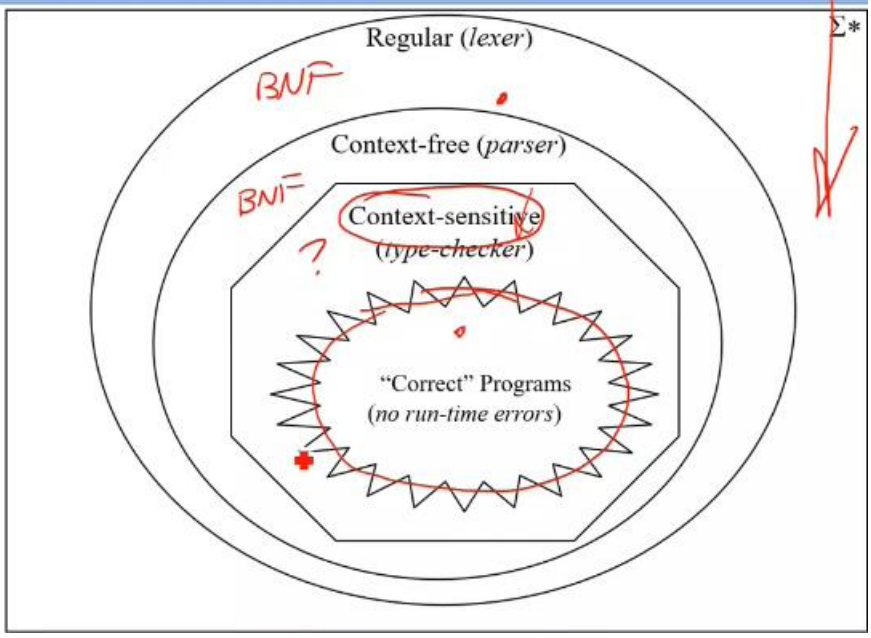
Shift-Reduce Parsing

- Major actions: Shift or reduce or give an error or give a success

Solving the Dangling Else Ambiguity

- Algol 60, C, C++: associate each else with closest if; use { … } or begin … end to override
- Algol 68, Modula, Ada: use an explicit delimiter to end every conditional (e.g., if … endif)
- Java: rewrite the grammar and restrict what can appear inside a nested if statement
  - IfThenStmt → if (Expr) Stmt
  - IfThenElseStmt → if (Expr) StmtNoShortIf else Stmt
    - The category StmtNoShortIf includes all except IfThenStmt

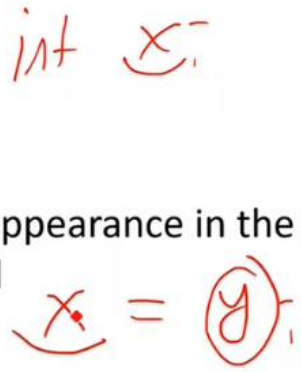
Grammars



Semantic Analysis

- Beyond context free grammar
  - Is x declared before it is used?
  - Is x declared but never used?
  - Is an expression type consistent?
  - Is an array reference in bounds?
  - ...
- Choices
  - Use context sensitive grammars
    - Hard to define and costly to use
  - Use attribute grammar
    - Can help to some extend
  - Use ad hoc methods
    - Mostly required

- Explicit type declaration
  - as program statement
- Implicit type declaration
  - default mechanism - at the first appearance in the program
  - e.g., in FORTRAN, BASIC, and Perl



Dynamic Binding

- JavaScript, PHP, Common Lisp
- Specified through an assignment statement e.g., JavaScript
  - list = [2, 4.33, 6, 8];
  - list = 17.3;
- Advantage: flexibility (generic program units)

Static Variables

- Static variables
  - bound to memory cells before program execution begins
  - remain bound to same memory cells until program execution terminates
- Example
  - all FORTRAN 77 variables
  - C static variables
- Advantages:
  - efficiency (direct addressing)
  - history-sensitive subprogram support
- Disadvantage:
  - less flexible (no recursion)

Stack Dynamic Variables

- Storage bindings are created for variables when their declaration statements are elaborated (storage allocation and binding)
  - Allocated from the run-time stack
  - De-allocate after execution (garbage collection)
  - Can happen at the beginning of block or anywhere
- Advantage:
  - allows recursion
  - conserves storage – allocate when and where needed
- Disadvantages (compared to static vars):
  - Overhead of allocation and de-allocation
  - Subprograms cannot be history sensitive
  - Indirect addressing (longer time)

Explicit Heap-Dynamic Variables

- Nameless abstract memory cells (heap)
  - Allocated and de-allocated by explicit directives
  - Specified by the programmer
  - Takes effect during execution
  - Referenced only through pointers or references
    - e.g. dynamic objects in C++ (via new and delete)

```
int  * i;
i = new int;
...
delete i;
```

    - all objects in Java
- Advantage:
  - provides for dynamic storage management (flexibility)
- Disadvantage:
  - inefficient by comparison (cost of reference)
  - reliability must be shown

Implicit Heap-Dynamic Variables

- Bound to heap storage automatically
  - Only when they are assigned values
  - All attributes are bound every time they are assigned
- Allocation and de-allocation caused by assignments
  - e.g. all variables in APL
  - all strings and arrays in Perl an JavaScript
- Example: Java statement "highs = [74, 84, 490, 44, 45];"
  - "highs" is now an array …
- Advantage:
  - Flexibility
  - Writability
- Disadvantages:
  - Inefficient, because all attributes are dynamic
  - More work to do error detection

Variable Bindings

- Static
  - E.g., C static variables
- Stack dynamic
  - E.g., C method variables
- Heap dynamic variables
  - Explicit Heap-Dynamic Variables
    - E.g., dynamic objects in C++ (new and delete)
  - Implicit Heap-Dynamic Variables
    - E.g., arrays in JavaScript

Type Checking

- Type checking ensures that the operands and the operator are of compatible types
- Generalized to include subprograms and assignments

- Compatible type is either
  - legal for the operator, or
  - language rules allow it to be converted to a legal type
- Coercion
  - Automatic (implicit) conversion
- Type error
  - Application of an operator to an operand of incorrect type

- Nearly all type checking can be static for static type bindings
- Type checking must be dynamic for dynamic type bindings

Strong Typed Languages: Java, C#, Haskell, Python, Ruby

Weak Typed Languages: C, C++, JavaScript, PHP

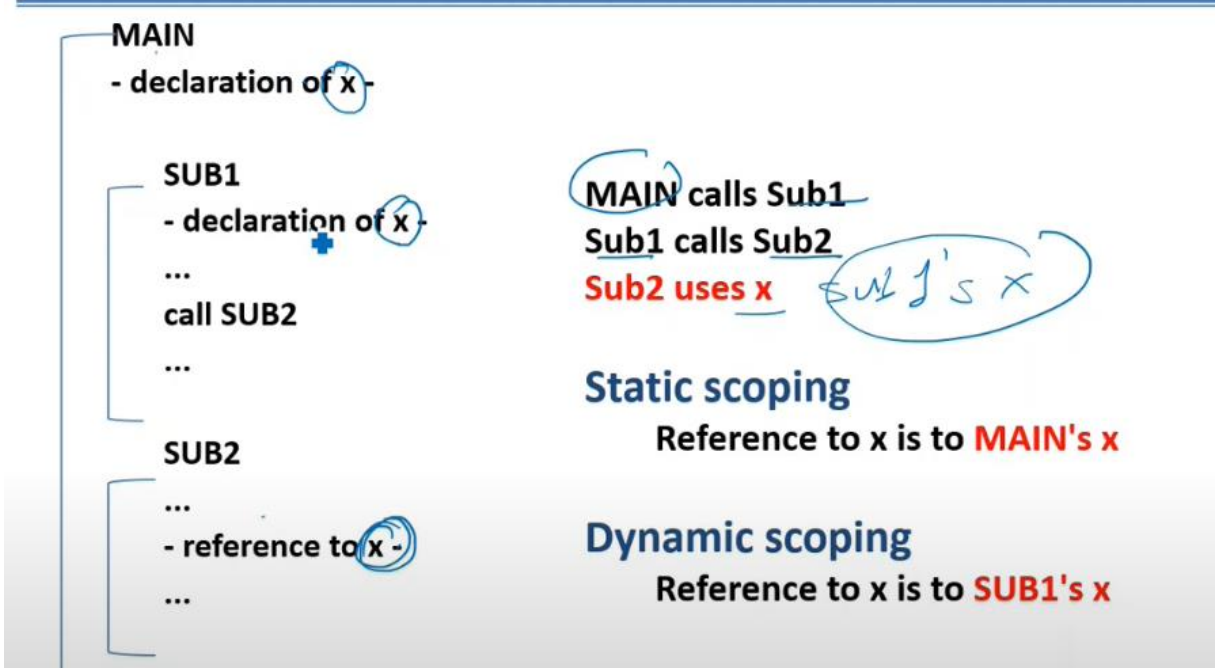
Name Type Compatibility

- Variables have compatible types if they are either
  - in the same declaration
  - or in declarations that use the same type name
- Easy to implement but highly restrictive

Structure Type Compatibility

- Variables have compatible types if their types have identical structures
  - More flexible
  - But harder to implement
    - Because of structured types (comparison of whole structures instead of names)
    - Others are much simpler
- Questions:
  - Structurally the same record types with different field names?
  - Array types with the same base type but different subscripts? E.g. [1..10] vs. [0..9]
  - Enumeration types whose components are spelled differently?
- With structural type compatibility, you can not differentiate between types of the same structure

Scope Example



Scope Evaluation

- Advantage of dynamic scoping
  - convenience (no need to pass variables)
  - flexibility
- Disadvantage
  - poor readability (same name different non-local vars)
  - reliability (accessibility from multiple subprograms)
  - longer access times for non-local variables

Type

- A type is a collection of computable values that share some structural property
- A type is also a set of operations on the values
- Examples:
  - Integers, Strings, int → bool, (int → int) → bool, …
- Not type examples:
  - {3, true}, , Even integers
- Distinction between sets that are types and sets that are not types is language-dependent

Type constructors: Defining New Types

- **Type constructors as set operations:**
  - Cartesian product
  - Union
  - Subset
  - Functions (Arrays)
- **Some type constructors do not correspond to set operations** (e.g., pointers)
- **Some set operators do not have corresponding type constructors (e.g., intersection)**

Type Equivalence

- How to decide if two types are the same?
- Structural Equivalence
  - Types are sets of values
  - Two types are equivalent if they contain the same values
- Name Equivalence
  - Same type names

- Which one is more flexible?
  - Structural equivalence

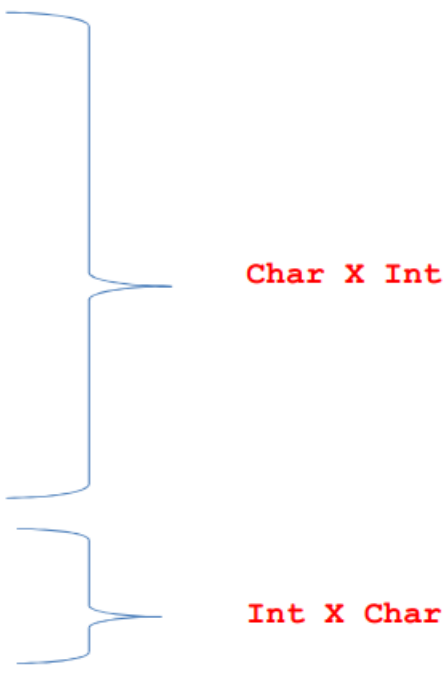
Structural Equivalence of Type

- Two type expressions are structural equivalent if they are
  - same basic type, or
  - formed by applying the same constructor to two structurally equivalent types, or
  - after type declaration type n=T the type name n is structurally equivalent to T

  → two type expressions are structural equivalent if and only if they are identical
- Example:
  - *integer* is equivalent to *integer*
  - *pointer(char)* is equivalent to *pointer(char)*
- Modification is needed for structural equivalence: When array are passed as parameter, we may not wish to include the array bounds as part of the type

Structural Equivalence

```
struct RecA {
     char x;
     int  y;
}
struct RecB {
     char x;
     int  y;
}
struct RecC {
     char u;
     int  v;
}
struct RecD {
     int  y;
     char x;
}
```



But are they equivalent in these languages?

In C:

```
struct RecA {
    char x;   int  y;
};
struct RecB {
    char x;   int  y;
};
struct RecA a;
struct RecB b;

b = a;
```

(Error: incompatible types in assignment)

Use structural equivalence for everything except unions and structs, which use name equivalence

Type System

- ***Type Constructors***:
  - Build new data types upon simple data types

- ***Type Checking***: The translator checks if data types are used correctly.
  - ***Type Inference***: Infer the type of an expression, whose data type is not given explicitly.
    e.g., x/y
  - ***Type Equivalence***: Compare two types, decide if they are the same.
    e.g., x/y and z
  - ***Type Compatibility:*** Can we use a value of type A in a place that expects type B?

Type Checking vs. Type Inference

- Standard type checking

```
int f(int x) { return x+1; };
int g(int y) { return f(y+1)*2; };
```

  - Look at the body of each function and use declared types of identifiers to check agreement

- Type inference

```
int f(int x) { return x+1; };
int g(int y) { return f(y+1)*2; };
```

  - Look at the code without type information and figure out what types could have been declared

Summary

- Types are important in modern languages
  - Organize and document the program, prevent errors, provide important information to compiler
- Type inference
  - Determine best type for an expression, based on known information about symbols in the expression
- Polymorphism
  - Single algorithm (function) can have many types
- Overloading
  - Symbol with multiple meanings, resolved when program is compiled

Expression vs. Statement

- In pure (mathematical) form:
  - Expression:
    - no side effect
    - return a value
  - Statement:
    - side effect
    - no return value
- Functional languages aim at achieving this pure form
- No clear-cut in most languages

Expression

- Constructed recursively:
  - Basic expression (literal, identifiers)
  - Operators, functions, special symbols
- Number of operands:
  - unary, binary, ternary operators
- Operator, function: equivalent concepts

```
(3+4)*5                    (infix notation)
mul(add(3,4),5)
"*"("+"(3,4),5)            (Ada, prefix notation)
(* (+ 3 4) 5)              (LISP, prefix notation)
```

Expression and Side Effects

- Side Effects:
  - Changes to memory, input/output
  - Side effects can be undesirable
  - But a program without side effects does nothing!
- Expression:
  - No side effect: Order of evaluating sub-expressions does not matter (mathematical forms)
  - Side effect: Order matters

Short-Circuit Evaluation

- `if (false and x) … if (true or x)…`
  - No need to evaluate x, no matter x is true or false
- What is it good for?
  - `if (i <=lastindex and a[i]>=x)…`
  - `if (p != NULL and p->next==q)…`
- Ada: allow both short-circuit and non short-circuit
  - `if (x /= 0) and then (y/x > 2) then ...`
  - `if (x /= 0) and (y/x > 2) then ...` ?
  - `if (ptr = null) or else (ptr.x = 0) then ...`
  - `if (ptr = null) or (ptr.x = 0) then ...` ?

Normal order evaluation (lazy evaluation)

- When there is no side-effect:
  Normal order evaluation (expressions evaluated in mathematical form)
  - Operation evaluated ***before*** the operands are evaluated;
  - Operands ***evaluated only when necessary***.

```
int double (int x) { return x+x; }
int square (int x) { return x*x; }
```

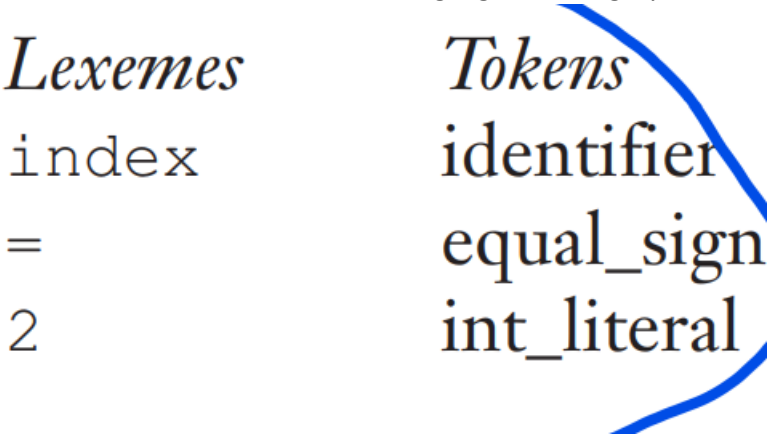
Applicative order evaluation : square(double(2)) = …
Normal order evaluation : square(double(2)) = …

Exception Handling

- We distinguish between two such classes of events:
  - Those that are detected by hardware: e.g., disk read errors, end-of-file
  - Those that are software-detectable: e.g., subscript range errors
- ***Definition:*** An ***exception*** is an unusual event that is detectable by either hardware or software and that may require special processing.
- ***Terminology:*** The special processing that may be required when an exception is detected is called ***exception handling***. The processing is done by a code unit or segment called an ***exception handler***. An exception is ***raised*** when its associated event occurs.

- **WHAT IS SIDE-EFFECT?**
-

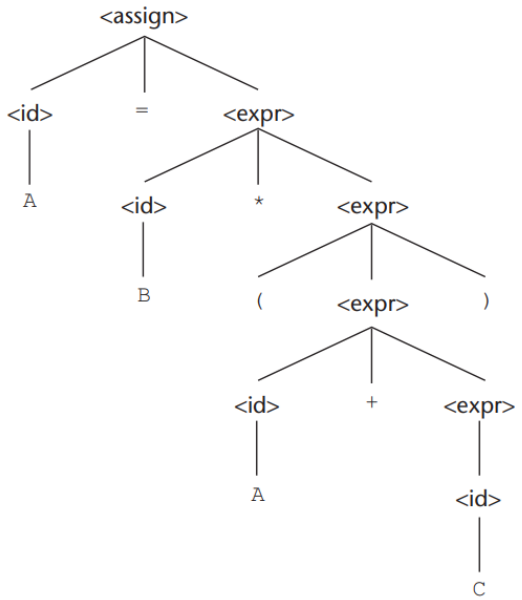
- The syntax of a programming language is the form of its expressions, statements, and program units. Its semantics is the meaning of those expressions, statements, and program units.
- Formal descriptions of the syntax of programming languages, for simplicity's sake, often do not include descriptions of the lowest- level syntactic units. These small units are called lexemes. The description of lexemes can be given by a lexical specification, which is usually separate from the syntactic description of the language. The lexemes of a programming language include its numeric literals, operators, and special words, among others. One can think of programs as strings of lexemes rather than of characters. Lexemes are partitioned into groups— for example, the names of variables, methods, classes, and so forth in a programming language form a group called identifiers. Each lexeme group is represented by a name, or token. So, a token of a language is a category of its lexemes.



- In general, languages can be formally defined in two distinct ways: by recognition and by generation
- The syntax analysis part of a compiler is a recognizer for the language the compiler translates. In this role, the recognizer need not test all possible strings of characters from some set to determine whether each is in the language. Rather, it need only determine whether given programs are in the language. In effect then, the syntax analyzer determines whether the given programs are syntactically correct.
- The forms of the tokens of programming languages can be described by regular grammars. The syntax of whole programming languages, with minor exceptions, can be described by context- free grammars.
- A metalanguage is a language that is used to describe another language. BNF is a metalanguage for programming languages.
- The abstractions in a BNF description, or grammar, are often called nonterminal symbols, or simply nonterminals, and the lexemes and tokens of the rules are called terminal symbols, or simply terminals. A BNF description, or grammar, is a collection of rules.
- The sentences of the language are generated through a sequence of applications of the rules, beginning with a special nonterminal of the grammar called the start symbol. This sequence of rule applications is called a derivation.
- Each of the strings in the derivation, including, is called a sentential form.

<program> => **begin** <stmt_list> **end**
=> **begin** <stmt> ; <stmt_list> **end**
=> **begin** <var> = <expression> ; <stmt_list> **end**
=> **begin** A = <expression> ; <stmt_list> **end**
=> **begin** A = <var> + <var> ; <stmt_list> **end**
=> **begin** A = B + <var> ; <stmt_list> **end**
=> **begin** A = B + C ; <stmt_list> **end**
=> **begin** A = B + ; <stmt> **end**
=> **begin** A = B + C ; <var> = <expression> **end**
=> **begin** A = B + C ; B = <expression> **end**
=> **begin** A = B + C ; B = <var> **end**
=> **begin** A = B + C ; B = C **end**

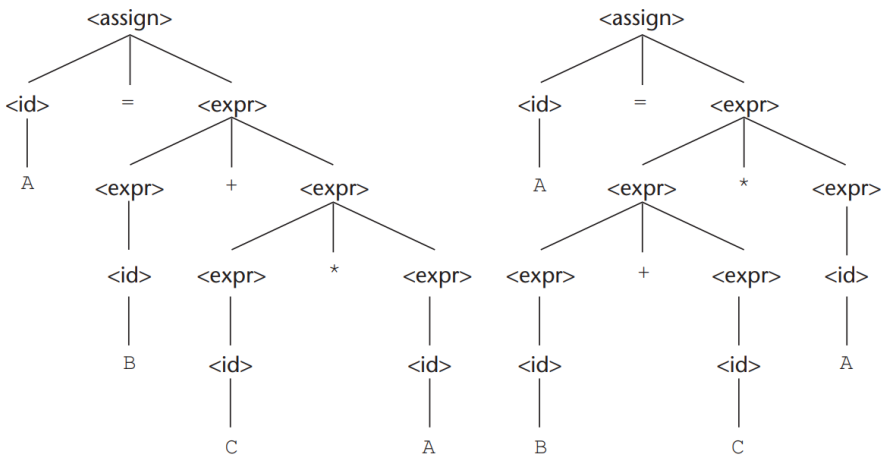
- In this derivation, the replaced nonterminal is always the leftmost nonterminal in the previous sentential form. Derivations that use this order of replacement are called leftmost derivations. The derivation continues until the sentential form contains no nonterminals.
- Derivation order has no effect on the language generated by a grammar.
- One of the most attractive features of grammars is that they naturally describe the hierarchical syntactic structure of the sentences of the languages they define. These hierarchical structures are called parse trees.



- Every internal node of a parse tree is labeled with a nonterminal symbol; every leaf is labeled with a terminal symbol. Every subtree of a parse tree describes one instance of an abstraction in the sentence.
- A grammar that generates a sentential form for which there are two or more distinct parse trees is said to be **ambiguous**.

Figure 3.2

Two distinct parse trees for the same sentence, A = B + C * A



-
- Syntactic ambiguity of language structures is a problem because compilers often base the semantics of those structures on their syntactic form. Specifically, the compiler chooses the code to be generated for a statement by examining its parse tree. If a language structure has more than one parse tree, then the meaning of the structure cannot be determined uniquely
- There are several other characteristics of a grammar that are sometimes useful in determining whether a grammar is ambiguous.1 They include the following: (1) if the grammar generates a sentence with more than one leftmost derivation and (2) if the grammar generates a sentence with more than one rightmost derivation. Some parsing algorithms can be based on ambiguous grammars. When such a parser encounters an ambiguous construct, it uses nongrammatical information provided by the designer to construct the correct parse tree. In many cases, an ambiguous grammar can be rewritten to be unambiguous but still generate the desired language.
- When an expression includes two operators that have the same precedence (as * and / usually have)—for example, A / B * C—a semantic rule is required to specify which should have precedence. This rule is named associativity.
- An attribute grammar is a device used to describe more of the structure of a programming language than can be described with a context-free grammar. An attribute grammar is an extension to a context-free grammar. The extension allows certain language rules to be conveniently described, such as type compatibility.
- These problems exemplify the categories of language rules called static semantics rules. The **static semantics** of a language is only indirectly related to the meaning of programs during execution; rather, it has to do with the legal forms of programs (syntax rather than semantics). Many static semantic rules of a language state its type constraints. Static semantics is so named because the analysis required to check these specifications can be done at compile time
- Synthesized attributes are used to pass semantic information up a parse tree, while inherited attributes pass semantic information down and across a tree. (Synthesized -> value, Inherited-> type (Examples))
- A lexical analyzer serves as the front end of a syntax analyzer. Technically, lexical analysis is a part of syntax analysis.
- State diagrams of the form used for lexical analyzers are representations of a class of mathematical machines called finite automata. Finite automata can be designed to recognize members of a class of languages called regular languages. Regular grammars are generative devices for regular languages. The tokens of a programming language are a regular language, and a lexical analyzer is a finite automaton.

- The part of the process of analyzing syntax that is referred to as syntax analysis is often called parsing.
- Parsers for programming languages construct parse trees for given programs.
- Parsers are categorized according to the direction in which they build parse trees. The two broad classes of parsers are top-down, in which the tree is built from the root downward to the leaves, and bottom-up, in which the parse tree is built from the leaves upward to the root.
- A top-down parser traces or builds a parse tree in preorder. A preorder traversal of a parse tree begins with the root. Each node is visited before its branches are followed. Branches from a particular node are followed in left-to-right order. This corresponds to a leftmost derivation.
- Imperative programming languages are, to varying degrees, abstractions of the underlying von Neumann computer architecture. The architecture's two primary components are its memory, which stores both instructions and data, and its processor, which provides operations for modifying the contents of the memory. The abstractions in a language for the memory cells of the machine are variables. In some cases, the characteristics of the abstractions are very close to the characteristics of the cells; an example of this is an integer variable, which is usually represented directly in one or more bytes of memory. In other cases, the abstractions are far removed from the organization of the hardware memory, as with a three-dimensional array, which requires a software mapping function to support the abstraction.
- A variable can be characterized as a sextuple of attributes: (name, address, value, type, lifetime, and scope).
- The address of a variable is sometimes called its l- value, because the address is what is required when the name of a variable appears in the left side of an assignment.
- It is possible to have multiple variables that have the same address. When more than one variable name can be used to access the same memory location, the variables are called **aliases**.
- The value of a variable is the contents of the memory cell or cells associated with the variable. It is convenient to think of computer memory in terms of abstract cells, rather than physical cells.
- A variable's value is sometimes called its r- value because it is what is required when the name of the variable appears in the right side of an assignment statement.
- A binding is an association between an attribute and an entity, such as between a variable and its type or value, or between an operation and a symbol. The time at which a binding takes place is called binding time.
- A binding is **static** if it first occurs before run time begins and remains unchanged throughout program execution. If the binding first occurs during run time or can change in the course of program execution, it is called **dynamic**.
- An **explicit declaration** is a statement in a program that lists variable names and specifies that they are a particular type. An **implicit declaration** is a means of associating variables with types through default conventions, rather than declaration statements. In this case, the first appearance of a variable name in a program constitutes its implicit declaration. Both explicit and implicit declarations create static bindings to types.
- Implicit variable type binding is done by the language processor, either a compiler or an interpreter.

- Another kind of implicit type declarations uses context. This is sometimes called type inference. In the simpler case, the context is the type of the value assigned to the variable in a declaration statement. For example, in C# a var declaration of a variable must include an initial value, whose type is taken as the type of the variable.
- With dynamic type binding, the type of a variable is not specified by a declaration statement, nor can it be determined by the spelling of its name. Instead, the variable is bound to a type when it is assigned a value in an assignment statement. When the assignment statement is executed, the variable being assigned is bound to the type of the value of the expression on the right side of the assignment. Such an assignment may also bind the variable to an address and a memory cell, because different type values may require different amounts of storage.
- The greatest disadvantage of dynamic type binding is cost.
- The **lifetime** of a variable is the time during which the variable is bound to a specific memory location. So, the lifetime of a variable begins when it is bound to a specific cell and ends when it is unbound from that cell.
    - **Static variables** are those that are bound to memory cells before program execution begins and remain bound to those same memory cells until program execution terminates. One advantage of static variables is efficiency.
    - A **stack-dynamic variable** is one that is bound to an address on the stack, which is dynamically (during run-time) allocated for that purpose. It may also be unbound during run-time, and its memory cell deallocated by being popped off the stack. In most current programming languages, the formal parameters and local variables of subroutines (functions, methods) are stack-dynamic variables. Memory cells are allocated for them when the subroutine begins execution, and are deallocated when the subroutine ends execution.
    - **Explicit heap-dynamic variables** are those nameless variables allocated on the heap for dynamic data structures or for instances of objects in OO programming languages. In Java or C++, they are allocated by the new operator.
    - An **implicit heap-dynamic variable** is like an explicit heap-dynamic variable, but is created without an explicit allocation operator. *Sebesta* gives as an example the JavaScript statement, list = [10.2, 3.5], where the variable storing the two-element array (not the variable list) is an implicit heap-dynamic variable.

- Automatic conversion is called a **coercion**. For example, if an int variable and a float variable are added in Java, the value of the int variable is coerced to float and a floating-point add is done.
- C and C++ are not strongly typed languages because both include union types, which are not type checked.
- The scope of a variable is the range of statements in which the variable is visible
- The method of binding names to nonlocal variables called **static scoping**
- **Dynamic scoping** is based on the calling sequence of subprograms, not on their spatial relationship to each other. Thus, the scope can be determined only at run time.
- The **referencing environment** of a statement is the collection of all variables that are visible in the statement. The referencing environment of a statement in a static scoped language is the variables declared in its local scope plus the collection of all variables of its ancestor scopes that are visible.

- A subprogram is **active** if its execution has begun but has not yet terminated. The referencing environment of a statement in a dynamically scoped language is the locally declared variables, plus the variables of all other subprograms that are currently active.
- A **named constant** is a variable that is bound to a value only once. Named constants are useful as aids to readability and program reliability. Readability can be improved, for example, by using the name pi instead of the constant 3.14159265. C++ ve Java'da dynamic binding ile constant yapılabiliyor.
    - Örnek: C++ -> const int PI = 3.14; ~ Java -> final int PI = 3.14;
-