Lexical Analysis, Syntax, and Semantics

Learning Objective

This session will introduce students to the concepts of **lexical analysis, syntax,** and **semantics** of programming language. Students will gain an understanding of the problem to be solved, methods for describing syntax, syntax parsing, attribute grammars, and the distinction between static and dynamic semantics.

Since this class is focusing primarily on syntax and semantics, let's begin by defining them.
Webster dictionary defines them as:

Syntax: *In grammar, the arrangement of words as elements in a sentence to show their relationship.*
Semantic: *of meaning, especially meaning in language.*

Statement of the Problem

What's the problem that programming language is attempting to solve? In the first session, we looked at the basic architecture of modern digital computers, particularly the binary representation of instructions and data. Humans simply

cannot effectively work in a language consisting of ones and zeroes. A "higher-order" language, more easily comprehended by the human mind, is needed to program computers. That language is then translated (by a computer program!) into the basic machine language of ones and zeroes. The problem, shown below, is how do you get from "Here" to "There?"

```
Here                 There
int factorial(int n){     01001011100100100111000111010111
int count, fact;          11001011000110000000001110111111
count = n;                00011110101010000001111100000110
fact = 1;                 01110001111100010101010001110010
while(count <> 0){        10001110010101010001000111100111
  fact =  fact * count    01010000111110000111100000001101
  count--;           .
  }                  .
return fact          .
}                    .
```

This example shows on the left a C program to calculate the factorial function. Using a collection of statements constructed according to specific rules, we've crafted a high-order language program. A compiler will then translate these well-defined statements into the machine language of ones and zeroes shown symbolically on the right.

Specifically, the problem is solved by following these steps:

- Step 1: Define the syntax and semantics of a programming language
- Step 2: Write a compiler that can translate the syntax and semantics of that language into machine language
- Step 3: Disseminate a description of the language's syntax and semantics to the users (programmers)
- Step 4: Programmers write code (programs) adhering to the syntactic and semantic rules
- Step 5: The compiler translates that code into machine language.

In the remainder of this course we'll be looking at how programming language helps to solve this problem of getting from "Here" to "There."

Lexical Analysis

Statements (sentences): strings of a language. Example:

```
index = 2*count + 17
```

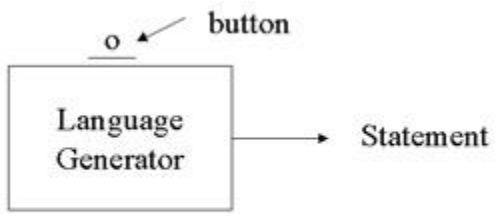
- ***lexemes:*** the lowest level syntactic unit (e.g., identifiers, literals, operators, special words) Examples: + (addition operator); **int** (a C- language special word designating "integer")
- ***token:*** a category of lexeme. Example: the lexeme "index" in the first bullet is an identifier token

Initially, a compiler sees a programming language statement as just a string of characters. **Lexical Analysis** is the process of separating and identifying the component character patterns that represent the **lexemes** and **tokens**. This process must comprise the front end of a syntax analyzer.

Syntax

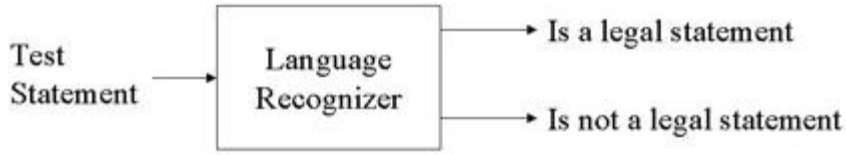
So, working with the lexeme and token components, how do we go about building (or defining) a programming language? There are two ways to do it: through generation or through recognition.

A language generator is a "black box" that produces a syntactically correct statement every time that you push a button. Since the output from the language generator is unpredictable, it doesn't seem like a very practical way to define a programming language. But, as we'll see later, the concept of language generators is very useful.



Language recognizers work in the opposite manner. A recognizer is a "black box" filter that determines if a

given input string is legal.



This approach might not seem to be any more viable than the language generator. One would have to generate candidate generate test statements and run them through the recognizer, and keep track of the legal statements. But again, it turns out that the concept of a language recognize is very useful; a programming language compiler is really a form of language recognizer.

We will now return to the generator concept of defining the syntax of a programming language via the Backus-Naur Form and syntax graphs.

Describing Syntax

One method of generating syntactically correct language statements is to use a notation invented by John Backus (IBM) and Peter Naur (of ALGOL fame) in the late 1950s. Their notation is known as the Backus-Naur Form, or BNF. Amazingly, BNF is similar to a technique used 2,500 years ago by an Indian linguist named Panini to describe Sanskrit grammar.

Languages have rules that describe their syntax. These rules are called the language's ***grammar***. According to Webster, grammar is *that part of the study of languages which deals with the forms and structure of words and sentences (syntax).*

BNF notation can describe a *context-free* grammar. Context free simply means that the meaning of any statement is not dependent on other statements; therefore, the terms "grammar" and "BNF" can be used interchangeably.

BNF is a *meta-language*, i.e., a language used to describe another language. BNF uses *abstractions* to describe syntactic structures, e.g., a C assignment statement can be represented by the abstraction <assign>. Brakets (< >) are used to delimit (enclose) abstractions.

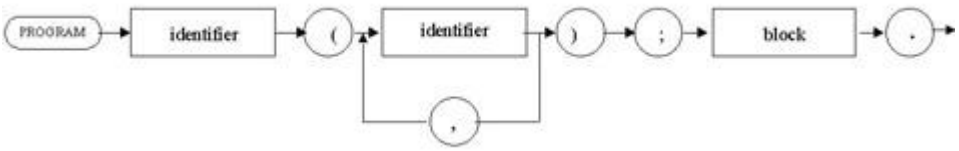
Some of the important points include:

- BNF is simple, yet capable of describing most of the syntax of programming languages.
- Although a grammar may be very small, it may be capable of generating an infinite set of different statements if it (the grammar) contains *recursive* rules.

The grammar (rule set) may be flawed in that it produces ambiguous statements. More about this later.

BNF has been expanded somewhat to improve its writability and readability; the newer notation is called Extended BNF, or EBNF. The extensions include the use of square brackets [ ] to indicate optional symbols, the use of braces { } to indicate optional or repetitious parts (e.g., lists), and the use of parentheses ( ) and the "OR" operator | to indicate multiple choices.

Syntax graphs are another powerful tool for producing context-free grammars. The notation is quite simple: ellipses (rectangles with rounded ends) are used to represent terminal symbols, and rectangles are used to represent non-terminals. The notation is attractive for its compactness. The extent of the compactness is shown below:



This is a top-level syntax graph for the entire Pascal language (from Grogono). The graph shows that every Pascal program begins with the reserved word "PROGRAM," followed by an identifier (the program's name), followed by an identifier. The identifier is a list of zero or more arguments, separated by commas. The list of arguments is enclosed by parentheses, followed by a semicolon. The rectangle labeled "block" is the body of the program. The program is terminated by a period.

Grogono further decomposes the non-terminals (the rectangles in the graph) using 20 additional syntax graphs, similar in complexity to the one shown above.

Grammar

As mentioned, a set of rules that define a language is called its **grammar**. The **BNF** is a meta-language that is capable of specifying the rules of a syntactically correct grammar. The rules of a grammar can be applied iteratively, starting with a **start symbol**, to generate a sentence of the language. The process of doing this is called a **derivation**.

Please study this derivation carefully; it contains many important concepts of programming language. A few notes on this derivation follow:

- The symbol => means "derives"
- The first line starts with the <program> rule.
- Each subsequent line is generated by replacing the left-most non-terminal in the previous line with one of that non-terminal's definitions in the grammar
- Replacing non-terminals in this manner is called a *leftmost derivation*
- Derivation order (leftmost, rightmost, or a combination) has no effect on the language that is generated.
- A language as simple as the one described by this grammar is *infinite*, i.e., it is capable of generating an infinite number of different statements. Do you understand why?

Now that we've seen how to generate the sentences of a programming language, let's look at the technique used to recognize those sentences.

Parsing

Another approach for formulating the derivation of a sentence from a grammar is through the use of a parse tree. While a parse tree provides the same result as a derivation, it shows the hierarchical structure or pattern of the of the sentences much better.

It is possible that a grammar can produce sentences that can be parsed in more than one way; such grammars are said to be ***ambiguous***. This is an important defect in a grammar; if a language is based on an ambiguous grammar, the compiler will not be able to uniquely parse its statements.

Examples of issues that can cause ambiguous grammar problems include:

- Operator precedence
- Associativity of operators, e.g., (A + B) + C = A + (B + C) (The order
- Embedded **if-then-else** statements

Operator precedence issues concern the order in which different operators in the same expression should be applied. Operator associativity concerns precedence issues for multiple instances of the same operator in an expression. Embedded **if-then-else** statement issues are created when a single **else** construct follows two **if** constructs. It could refer to either one, but the meaning is different in each case.

Methods of avoiding ambiguity in a programming language include the use of additional non-terminals (e.g., "factor" and "term"), and the use of additional grammar rules, as necessary.

Since parsers are language recognizers, they are a "natural" for programming-language compilers. The compiler's parser reads each source code statement and attempts to find the unique parse tree that fits it. Finding this pattern provides the key to interpreting the meaning of the lexemes.in the statement. Parsers can be built to work *top-down* (i.e., start at the root and work towards the leaves), or *bottom-up* (i.e., work from the leaves back to the root) when searching the grammar's rules to find the tree that fits a statement. We'll focus on top-down parsers.

An elegant way to implement a top-down parser is through the use of a ***recursive descent*** algorithm. We described recursion in the previous session; it's simply the ability of a function (or computer subroutine) to call itself. A recursive-descent parser is built by writing a subprogram (function) that is associated with each grammar non-terminal. The parser traces out each sub-tree that is rooted at the non-terminal, according to the grammar's rules. The parser may be recursive, if the grammar is recursive. Note the impossibility of using recursive descent on left-recursion

grammars, e.g. <A> -> <A> + <B>. The parser would fall into an infinite loop and never get a result to compare with the input string! (Look at Section 4.4.2: The LL Grammar Class.)

Semantics (in brief)

At this point, we have seen how to construct a context-free grammar to describe the syntax of a programming language. Next, we need to explore ways of describing the semantics (meaning) of the programming language statements, expressions, etc.

There are two categories of semantics:

- ***Static semantics***: semantic rules that can be checked during compile time (i.e., prior to runtime). As an example, variables in a program must be "declared" before they are referenced.
- ***Dynamic semantics:*** semantic rules that apply during the execution of a program.

Part of the problem of describing the semantics (meaning) of a programming language can be solved through the use of attribute grammars, i.e., grammars that have been enhanced through the addition of *attributes* (variables that can have values assigned to them), *attribute computation functions*(specify how attribute values are computed), *and predicate functions* (state the syntax and semantic rules).

Since attribute grammars contain some basic semantic information, they describe more of a programming language's structure than is possible with a context-free grammar. There are three types of attributes:

- ***Synthesized attributes:*** used to pass semantic information up a parse tree
- ***Inheritance attributes:*** used to pass semantic information down a parse tree
- ***Intrinsic attributes:*** synthesized attributes determined outside of the parse tree.

Attribute grammars still have the shortcoming of not being able to describe the dynamic semantics of a programming language. Dynamic semantics is still in its infancy - there is no universally accepted notation to describe it. Consequently, (imprecise) English text is often used to describe dynamic semantics. Three approaches are used:

- ***Operational semantics:*** describe program operation in terms of a lower-level language; first used to describe PL/1
- ***Axiomatic semantics:*** use axioms and inference rules for each kind of statement in the language
- ***Denotational semantics:*** map a language construct into a mathematical object.

Operational semantics is the easiest concept to describe. By way of example, consider the following use of a lower-level language to describe the semantics of a FORTRAN IV "DO" statement:

DO label variable = initial, terminal, [stepsize]

Clearly, operational semantics is the least formal of the three methods of describing dynamic semantics, yet it is effective. In some cases, circular definitions can result, where concepts end up being defined in terms of themselves.
Axiomatic semantics uses axioms and inference rules for each kind of statement in the language being described. ***Assertions*** (also known as ***predicates***) either ***pre-condition*** or ***post-condition*** a language statement.

The ***weakest pre-condition*** is the least restrictive precondition that will satisfy the post-condition. In the above example, the weakest pre-condition is {X > 0}.

A significant problem with axiomatic semantics is that an axiom or inference rule must be defined for every statement type, and this can be very difficult. Consequently, axiomatic semantics is really more of a research tool than a practical way of describing the meaning of language constructs.

Denotational semantics is yet another way of describing the dynamic semantics of a programming language. The approach here is to use mathematical objects to represent language constructs; if necessary, functions (possibly recursive) are used to convert these constructs to the mathematical objects. Denotational semantics has the potential to provide rigor in describing semantics, in that language constructs are mapped to well-known mathematical objects. However, in practice this approach is too complex for general-purpose use.

Wrap Up

This session, we've looked at techniques for describing the syntax and semantics of programming language. With respect to syntax, we've examined different methods of describing syntax, syntax parsing, and attribute grammars. With respect to semantics, we've looked at the distinction between static and dynamic semantics, and we've looked at several methods of describing them.

Next session, we'll "drill down" a bit further, and look at the *semantics* of variables, in particular, how they are named and bound to memory, and we'll explore the concepts of scope, lifetime, and referencing environment.

Recommended Readings:

The Formal Semantics of Programming Languages: An Introduction
by Glynn Winskel
Mit Press; 1993;
ISBN: 0262231697

Action Semantics (Cambridge Tracts in Theoretical Computer Science, No 26)
by Peter D. Mosses
Cambridge Univ Press;
ISBN: 0521403472

Programming in PASCAL, Revised Edition
by Peter Grogono; 363 pages,
Published by Addison-Wesley Publishing Company, Inc.; 1980;
ISBN 9-201-02775-5

Names, Bindings, Type Checking, and Scope

Learning Objective

This session will introduce students to the concepts surrounding programming language variables: their attributes, how they are bound, how they are type-checked, their scope and visibility, and their lifetime.

Overview

In the last session, we developed the concepts of lexical analysis, syntax, and semantics as applied to programming language. We saw that these programming-language concepts are important to both language developers and language users. We learned that a lexical analysis to identify the tokens (e.g., identifiers, literals, operators, and special words) is the initial step taken by a compiler when reading a line of code.

This session, we'll be looking at concepts associated with these tokens. We'll primarily be focusing on the ***semantics*** of ***variables.*** As we've seen in an earlier session, variables are associated with imperative programming languages which grew out of the von Neumann (fetch-execute) computer architecture. In this model, variables correspond to memory cells, either directly or indirectly.

Some of the specific topics that we'll investigate are variable names, the concepts of variable binding, scope, and lifetime, referencing environments, named constants, and variable initiation.

Names

A ***name*** iss "*a string of characters used to identify some entity in a program."* Names are also sometimes referred to as ***identifiers.*** Program entities that may have names include:

- ***labels***: an example is the Ada label "SUMMATION," in the code snippet shown below:

```
goto SUMMATION
...
<<SUMMATION>>SUM := SUM + NEXT
```

The bracketing notation << >> used in Ada makes the label easier to spot, and hence more readable.

- ***subprograms*:** Subprograms are units of code called repeatedly from different places in a program. An example of a subprogram call in FORTRAN is:

```
CALL AVERAGE(X, NUM)
```

where AVERAGE is the name of a subprogram.

- ***parameters:*** Parameters are used to pass data to and/or from subprograms and functions. X and NUM are named parameters in the preceding FORTRAN example.

Some of the issues associated with names include the maximum name length, the use of connector characters, case sensitivity, and special words. Looking at each of these considerations:

- ***Maximum name length:*** Early languages often used short names, as few as one or two characters. FORTRAN I allowed up to six characters; more modern languages, such as C and FORTRAN 90, allow names to use up to 31 characters, while Ada and C++ have no restriction on name length. There are trade-offs associated with name lengths: longer names are more readable (assuming the programmer picks meaningful names), but shorter names compile more efficiently.
- ***Connector characters:*** Programming languages generally don't allow names to include spaces, yet multiple-word names are often more readable. As an example, a variable that indicates the length of a list might naturally be called LIST LENGTH, if spaces were allowed. Many languages (e.g., C, C++, Java, etc.) allow the use of the underscore (_) character to enhance readability, leading to the name LIST_LENGTH in our example.
- ***Case sensitivity:*** Names in some languages (e.g., Java, C, and C++) are *case-dependent*, i.e., upper- and lower-case characters are treated as different characters. In a case-sensitive language, the names Average and average are different. Some early languages, such as FORTRAN 77, allowed only upper case names. Case sensitivity affects both readability and writability. As an example, the predefined Java name "getAudioClip" is more readable with the upper case A and C, but is less writable since the programmer must get the correct case for every letter in the name.
- ***Special words:*** special words include *keywords, reserved words, and predefined names.* A *keyword* is a special, context-sensitive word. Looking at the Sebesta FORTRAN example on p.192,

REAL APPLE {REAL is a keyword here (precedes a name)}
REAL = 3.4 {REAL is not a keyword, just a name here}

Reserved words are special words that cannot be used as a name. ANSI C has 32 keywords:

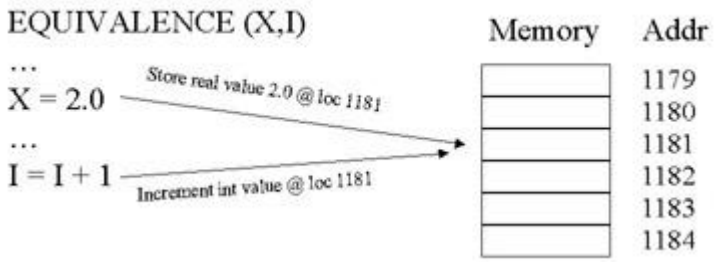
auto	**double**	**int**	**struct**
break	**else**	**long**	**switch**
case	**enum**	**register**	**typedef**
char	**extern**	**return**	**union**
const	**float**	**short**	**unsigned**
continue	**for**	**signed**	**void**
default	**goto**	**sizeof**	**volatile**
do	**if**	**static**	**while**

Reserved words such as the above list are usually shown in **bold.**

Variables

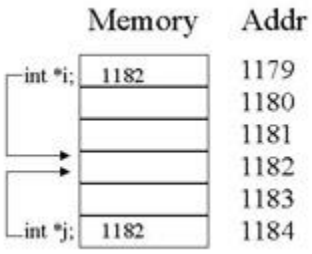
Variables are abstractions of memory cells, e.g., they associate a *name* with a *memory location.* The process of forming such associations is referred to as **binding**. Variables are characterized by the six attributes of name, address, type, value, lifetime, and scope. We've looked at the name attribute in the preceding section; now, we'll look at the other five attributes.

- **address:** the address is the memory location of a variable. A variable's address is referred to as its **l-value** (l as in left). It's entirely possible, and not at all unusual, for a program to use a single name to refer to the same or different memory addresses, in different parts of the program. It's also possible for different identifiers (names) to refer to the same memory addresses: these different names are call ***aliases***. In the FORTRAN language, aliases can be created with the EQUIVALENCE statement.



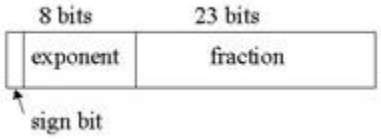
In this example, the float variable X and the integer variable I are aliases that both refer to the same memory location, cell 1181.

Variables are ***pointers*** if they point to another memory location, i.e., their type is a memory address and their value is the address of the variable they are said to point to. Two variables that point to the same memory location are also a type of alias, as shown in this figure.



Here, *i is a pointer in memory location 1179 that points at memory cell 1182, and int *j is a pointer in memory cell 1184 that that also points at cell 1182.

- **type:** the type attribute of a variable determines the range of values the variable can have, and the set of operations that can be performed on that variable (addition, concatenation, etc.). A ***float*** type, for example, would have the following 32-bit structure in memory and would support floating-point operations.



- **value:** the value attribute corresponds to the *contents* of a variable (i.e., memory cell(s)). Note that as exemplified by a float (which required four memory cells), a variable may require more than one memory cell to contain its value. A variable's value is called its ***r-value*** because it is on the right side of an assign statement (e.g., X = 2.0). Note that the l-value is needed to find the r-value.
- **lifetime:** The lifetime of a variable is the time during which a variable is **bound** to (associated with) a specific memory location.
- **scope:** The scope of a variable is the range of program statements over which a variable is visible, i.e., its value can be obtained.

We will discuss the concepts of binding, lifetime, and scope in more detail below.

Binding

Binding means an association, such as your name is bound to you as a person. For a programming language, binding takes place at the binding time, which might be at the time the language was designed, at compilation time, or at execution (run) time. Binding can be **static** if it occurs before run time, and remains unchanged throughout program execution, or it can be **dynamic** if it occurs during run time or can change during execution.

Type binding means binding (associating) a variable with a data type. Important considerations are how the type is specified, and when the binding happens. *Static type binding* can be via an ***explicit declaration*** or an ***implicit declaration.*** An explicit declaration uses a type declaration statement, e.g., in the C language: **int** x is an explicit type binding of the variable x to the type **int.** Implicit declarations depend on language conventions. In FORTRAN, variables starting with the letters I, J, K, L, M, and N are implicitly assumed to be integer types.

Dynamic type binding does not use a type declaration; instead, when the assignment statement is executed, the type is determined by the right-hand side (RHS) of the statement. As in the Sebesta example (p.198), the JavaScript language statement list = [10.2, 3.5] results in the variable list being dynamically typed as a single-dimensioned array of length 2, due to the array [10.2, 3.5] on the RHS. Dynamic type binding is very flexible in that "generic" programs can be written to deal with arbitrary data types, but it is also prone to error (e.g., due to typos) that can't be caught at compile time. Dynamic type binding is also inefficient since type checking and memory allocation must be done "on the fly" during program execution.

The ***lifetime*** of a variable is the time the variable is bound to a specific memory location, i.e., it begins when the variable is bound to a memory location and it ends when it is unbound from that location.

Before we go on, we need to consider how memory is allocated in a computer. The issues are two-fold: when memory is allocated, and from where. Memory is statically allocated to variables at compile time, or dynamically allocated at run (execution) time. Depending on the circumstances, memory can be allocated from either the ***stack*** or the ***heap.*** These terms are defined as:

> **Stack:** *In programming, a special type of data structure in which items are removed in the reverse order from which they are added, so the most recently added item is the first one removed.*
>
> **Heap:** *In programming, an area of memory reserved for data that is created at runtime -- that is, when the program actually executes. In contrast, the stack is an area of memory used for data whose size can be determined when the program is compiled.*

Variables may be organized into four categories, depending on how their associated memory is allocated:

- **Static variables:** these are bound to memory before execution (i.e., during compilation) and remain bound throughout execution. Static variables have the advantages of being accessible throughout program execution, and they are very efficient in terms of low run-time overhead. They do not, however, support recursion, and provision must be made for sharing memory between variables (e.g., through the use of aliases) if memory is limited. Early languages (e.g., pre-90 FORTRAN) allowed only static variables; newer languages such as C, C++, and Java allow variables to be optionally specified as static.
- **Stack-dynamic variables:** these variables are statically bound to a type at compilation time, but they are not bound to a memory location until execution of the code reaches the declaration. Unbinding occurs when the procedure containing the declaration ends.
- **Explicit heap-dynamic variables:** these variables are allocated and deallocated via explicit run-time, programmer-specified instructions. The heap, not the stack, is used to provide the required memory cells. As described in Sebesta (p.204), the C++ language **new** operator is used to allocate memory for an explicit heap-dynamic variable. **Implicit heap-dynamic variables:** All the attributes for these variables, including memory cells, are bound when they are assigned a value. The advantage here is total flexibility, e.g.,

totally generic programs can be written. Such programs will be extremely inefficient (run slowly), however, due to the overhead associated with allocating the memory "on the fly."

Type Checking

Type checking is the process of verifying that the operations (e.g., addition, concatenation, logical operations such as **or**, etc.) performed on a variable are compatible with the variable's type. It may be perfectly correct to perform an **or** operation on two Boolean variables, but that operation would not make sense if the two variables were float types.

Type checking applies to both subprograms and assignment operators. Consider the following FORTRAN call to the subroutine SORT:

Call to subroutine:

```
INTEGER LIST, NUM
CALL SORT(LIST,NUM)
...
```

Beginning of subroutine:

```
SUBROUTINE SORT(L,N)
INTEGER L,N
...
```

Note that the arguments LIST and NUM to the subroutine SORT are integers, which is what the subroutine is expecting.

Now, consider a simple assign statement in C:

```
float x, y;
x = y;
```

Again, the variable types appearing on both sides of the = operator are consistent with the meaning (semantics) of that operator.

When are types checked for correctness? Static variable type checking can be accomplished at compile time, but dynamic variables must me checked at run time, at a high overhead cost. A ***type error*** will result if the types used in an assignment or are being passed as subprogram arguments are not compatible. ***Strong typing*** means that type errors are always detected. FORTRAN is not strongly typed, due to language features such as the EQUIVALENCE statement that allows aliasing. Pascal and Ada, however, are nearly strongly typed, having only a few loopholes.

There are nuances associated with the concept of type compatibility. Types may be named differently but in fact have identical data-storage structures. ***Name type compatibility*** in a

programming language means that two variables must have exactly the same type names to be compatible, even if the underlying data structures are identical. Other languages allow ***structure type compatibility*** if the underlying data structures are identical.

Name type compatibility enable differences other than just structure to differentiate variables.

Scope, Lifetime, and Referencing Environment

Depending on the design of a programming language, a variable may not be "visible" from all parts of a program.

There are three broad categories of variables with respect to scope: ***local variables -*** a variable is local in a program unit if it is defined there, e.g., a variable that is declared within a subroutine; ***non-local variables -*** variables that are visible that were not defined in the program unit; and ***global variables -*** variables that are visible in all program and subprogram units.

Programmers must understand the scope of variables in order to design their code. They must know, for example, when it is necessary to pass a parameter to a subprogram through a formal argument, and when it's appropriate to use a global variable.

We've seen the concept of static (something that happens at compile time) versus dynamic (something that happens at run time) several times now, and we'll see that the concept again applies to scope. ***Static scope*** refers to the binding of names to non-local variables at compile time. When the compiler encounters a variable, the variable's attributes (address, value, type, etc.) are found by finding the statement where the variable was declared. The search for the declaration begins in the sub-program and then continues into the parent (ancestor) program unit(s).

Static scoping has the advantages of run-time efficiency (since the scope of a variable is determined at compile time), and type checking is easily accomplished. The downside of static scoping is that complicated program structures may result to accomplish the desired visibility of variables, with a consequent tendency for the programmer to make too many variables global. Readability can be problematic if variable declarations occur far from where the variables are used.

Dynamic scoping overcomes some of these problems, but of course, introduces some new complications. In dynamic scoping, the scope of a variable will depend on the sequence in which subprograms are called.

Dynamic scoping has the advantage that parameters can be easily passed from parent to child subprograms through variables that are declared in the parent. On the downside, with dynamic scoping all variables of a parent are visible to all child subprograms, like it or not. Also, run-time efficiency is impacted since the compiler cannot do static type checking. Readability can be a problem, since it may be impossible to check all possible calling sequences.

At some point in a program, a variable may not be needed anymore. This concept is referred to as the ***lifetime*** of a variable, i.e., the time during which the variable is bound to a specific memory location. Please note the distinction between the lifetime of a variable and its scope, as exemplified in the C++ code shown in Sebesta (p.219). In this example, note how the lifetime of the variable "sum" extends over the execution time of "printheader," even though "sum" is not visible in printheader.

The ***referencing environment*** of a statement is the collection of all variables visible to that statement. In other words, a software engineer can look at a particular line of code in a program, and depending on the rules (syntax and semantics) of the language, conduct an analysis to determine all of the variables visible from that statement. Again, the referencing environment will depend on whether the variables are statically or dynamically scoped.

Named Constants and Variable Initiation

A ***named constant*** is a variable that is bound to a value only at the time it is bound to storage; its value cannot be changed later in a program. Why would a programmer elect to use a named constant? Typical reasons include increased readability, and improved program maintenance.

As an example, suppose that a programmer was writing code that makes many references to the mathematical constant "Pi" (3.1415...). Rather than use the numeric value 3.1415... throughout the program, it is much better practice to define the constant once at the beginning. In C, such a declaration would look like this:

```
const float  Pi = 3.14159265
```

Statements such as

```
Area = Pi*Radius
```

are highly readable, and more writable since the long string of digits doesn't have to be re-written for every instance. Adding to the reliability is the fact that the constant Pi cannot be changed to another number once it's been declared.

Variables must be **initialized** (bound to a value) prior to being used in a program. Initialization can be either static (done at compile time) or dynamic (done at run time). Initialization can be accomplished in a number of ways: through a specific language feature, such as the FORTRAN DATA statement, through a simple assignment statement (e.g., Pi = 3.1415), or in some languages, at the time the variable's type is declared.

Wrap Up

We've covered a lot of ground this session, having looked at how variables are named and bound in a programming language, how their types are specified and checked, as well as their scope, lifetime, and referencing environment.

Next session will focus on a specific variable attribute: its **type.** We'll look at the various data types, and examine how various data types are stored in a computer's memory.