CSC 432 Principles of Programming Languages II

Lecture Note 3_3

Language Semantics

Semantics is the study of the relationship between words and how we draw meaning from those words.

Semantics involves the deconstruction of words, signals, and sentence structure. It influences our reading comprehension as well as our comprehension of other people's words in everyday conversation. Semantics play a large part in our daily communication, understanding, and language learning without us even realizing it.

Since meaning in language is so complex, there are actually different theories used within semantics, such as formal semantics, lexical semantics, and conceptual semantics etc

- **Formal Semantics** is the study of grammatical meaning in natural languages using formal tools from logic and theoretical computer science.- Formal semantics uses techniques from math, philosophy, and logic to analyze the broader relationship between language and reality, truth and possibility. Has your teacher ever asked you to use an "if… then" question? It breaks apart lines of information to detect the underlying meaning or consequence of events.
- **Lexical Semantics** - Lexical semantics deconstruct words and phrases within a line of text to understand the meaning in terms of context. This can include a study of individual nouns, verbs, adjectives, prefixes, root words, suffixes, or longer phrases or idioms.
- **Conceptual Semantics** - Conceptual semantics deals with the most basic concept and form of a word before our thoughts and feelings added context to it. For example, at its most basic we know a cougar to be a large wild cat. But, the word cougar has also come to indicate an older woman who's dating a younger man. This is where context is important.
- **Denotation** A denotational definition of a language consists of three parts: the abstract syntax of the language, a semantic algebra defining a computational model, and valuation

functions. The valuation functions map the syntactic constructs of the language to the semantic algebra. Recursion and iteration are defined using the notion of a limit. The programming language constructs are in the syntactic domain while the mathematical entity is in the semantic domain and the mapping between the various domains is provided by valuation functions. Denotational semantics relies on defining an object in terms of its constituent parts.

- **Connotation**, on the other hand, refers to the associations that are connected to a certain word or the emotional suggestions related to that word. The connotative meanings of a word exist together with the denotative meanings. The connotations for the word snake could include evil or danger. Connotation is created when you mean something else, something that might be initially hidden.
- **Axiomatic semantics** -The axiomatic semantics of a programming language are the assertions about relationships that remain the same each time the program executes. Axiomatic semantics are defined for each control structure and command. The axiomatic semantics of a programming language define a mathematical theory of programs written in the language.

A mathematical theory has three components:

- *Syntactic rules:* These determine the structure of formulas which are the statements of interest;
- *Axioms:* These describe the basic properties of the system;
- *Inference rules:* These are the mechanisms for deducing new theorems from axioms and other theorems

Operational Semantics-An operational definition of a language consists of two parts: an abstract syntax and an interpreter. An interpreter defines how to perform a computation. When the interpreter evaluates a program, it generates a sequence of machine configurations that define the program's operational semantics. The interpreter is an evaluation relation that is defined by rewriting rules. The interpreter may be an abstract machine or recursive functions.

Expressions, Statements, and Blocks

Variables and operators are basic building blocks of programs. You combine literals, variables, and operators to form expressions — segments of code that perform computations and return values. Certain expressions can be made into statements — complete units of execution. By grouping statements together with braces — { and } — you create blocks of code.

Expressions

Expressions perform the work of a program. Among other things, expressions are used to compute and to assign values to variables and to help control the execution flow of a program. The job of an expression is twofold: to perform the computation indicated by the elements of the expression and to return a value that is the result of the computation.

Definition: An *expression* is a series of variables, operators, and method calls (constructed according to the syntax of the language) that evaluates to a single value.

As discussed in the previous section, operators return a value, so the use of an operator is an expression. This partial listing of the `MaxVariablesDemo` program shows some of the program's expressions in boldface:

```
...
//other primitive types
char aChar = 'S';
boolean aBoolean = true;

//display them all
System.out.println("The largest byte value is "
                   + largestByte);
...

if (Character.isUpperCase(aChar)) {
    ...
}
```

Each expression performs an operation and returns a value, as shown in the following table.

Some Expressions from MaxVariablesDemo

Expression	Action	Value Returned
`aChar = 'S'`	Assign the character 'S' to the character variable `aChar`	The value of `aChar` after the assignment ('S')
`"The largest byte value is " + largestByte`	Concatenate the string `"The largest byte value is "` and the value of `largestByte` converted to a string	The resulting string: `The largest byte value is 127`
`Character.isUpperCase(aChar)`	Call the method `isUpperCase`	The return value of the method: `true`

The data type of the value returned by an expression depends on the elements used in the expression. The expression `aChar = 'S'` returns a character because the assignment operator returns a value of the same data type as its operands and `aChar` and `'S'` are characters. As you see from the other expressions, an expression can return a boolean value, a string, and so on.

The Java programming language allows you to construct compound expressions and statements from various smaller expressions as long as the data type required by one part of the expression matches the data type of the other. Here's an example of a compound expression:

```
x * y * z
```

In this particular example, the order in which the expression is evaluated is unimportant because the results of multiplication is independent of order; the outcome is always the same, no matter what order you apply the multiplications. However, this is not true of all expressions. For example, the following expression gives different results, depending on whether you perform the addition or the division operation first:

```
x + y / 100     //ambiguous
```

You can specify exactly how you want an expression to be evaluated, using balanced parentheses — `(` and `)`. For example, to make the previous expression unambiguous, you could write:

```
(x + y)/ 100    //unambiguous, recommended
```

If you don't explicitly indicate the order in which you want the operations in a compound expression to be performed, the order is determined by the *precedence* assigned to the operators in use within the expression. Operators with a higher precedence get evaluated first. For example, the division operator has a higher precedence than does the addition operator. Thus, the two following statements are equivalent:

```
x + y / 100
x + (y / 100) //unambiguous, recommended
```

When writing compound expressions, you should be explicit and indicate with parentheses which operators should be evaluated first. This pratice will make your code easier to read and to maintain.

The following table shows the precedence assigned to the operators in the Java platform. The operators in this table are listed in precedence order: The higher in the table an operator appears, the higher its precedence. Operators with higher precedence are evaluated before operators with a relatively lower precedence. Operators on the same line have equal precedence. When operators of equal precedence appear in the same expression, a rule must govern which is evaluated first. All binary operators except for the assignment operators are evaluated from left to right. Assignment operators are evaluated right to left.

Operator Precedence

postfix operators	`expr++ expr--`
unary operators	`++expr --expr +expr -expr ~ !`
multiplicative	`* / %`
additive	`+ -`
shift	`<< >> >>>`
relational	`< > <= >= instanceof`
equality	`== !=`

bitwise AND	`&`
bitwise exclusive OR	`^`
bitwise inclusive OR	`|`
logical AND	`&&`
logical OR	`||`
conditional	`? :`
assignment	`= += -= *= /= %= &= ^= |= <<= >>= >>>=`

Statements

Statements are roughly equivalent to sentences in natural languages. A *statement* forms a complete unit of execution. The following types of expressions can be made into a statement by terminating the expression with a semicolon (`;`):

- Assignment expressions
- Any use of `++` or `--`
- Method calls
- Object creation expressions

These kinds of statements are called *expression statements*. Here are some examples of expression statements:

```
aValue = 8933.234;                      //assignment statement
aValue++;                               //increment       "
System.out.println(aValue);             //method call     "
Integer integerObject = new Integer(4); //object creation "
```

In addition to these kinds of expression statements, there are two other kinds of statements. A *declaration statement* declares a variable. You've seen many examples of declaration statements.

```
double aValue = 8933.234;             //declaration statement
```

A *control flow statement* regulates the order in which statements get executed. The `for` loop and the `if` statement are both examples of control flow statements. You'll learn about control flow statements in the section Control Flow Statements◆.

Blocks

A *block* is a group of zero or more statements between balanced braces and can be used anywhere a single statement is allowed. The following listing shows two blocks from the `MaxVariablesDemo`◆ program, each containing a single statement:

```
if (Character.isUpperCase(aChar)) {
    System.out.println("The character " + aChar
                       + " is upper case.");
} else {
    System.out.println("The character " + aChar
                       + " is lower case.");
}
```

Summary of Expressions, Statements, and Blocks

An expression is a series of variables, operators, and method calls (constructed according to the syntax of the language) that evaluates to a single value. You can write compound expressions by combining expressions as long as the types required by all of the operators involved in the compound expression are correct. When writing compound expressions, you should be explicit and indicate with parentheses which operators should be evaluated first.

If you choose not to use parentheses, then the Java platform evaluates the compound expression in the order dictated by operator precedence. The table in section Expressions, Statements, and Blocks◆ shows the relative precedence assigned to the operators in the Java platform.

A statement forms a complete unit of execution and is terminated with a semicolon (`;`). There are three kinds of statements: expression statements, declaration statements, and control flow statements.

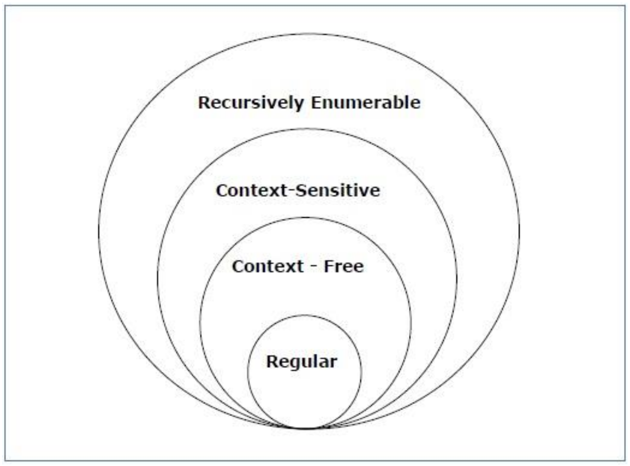
You can group zero or more statements together into a block with braces: `{` and `}`. Even though not required, we recommend using blocks with control flow statements even if only one statement is in the block.

Chomsky Classification of Grammars

According to Noam Chomosky, there are four types of grammars − Type 0, Type 1, Type 2, and Type 3. The following table shows how they differ from each other −

Grammar Type	Grammar Accepted	Language Accepted	Automaton
Type 0	Unrestricted grammar	Recursively enumerable language	Turing Machine
Type 1	Context-sensitive grammar	Context-sensitive language	Linear-bounded automaton
Type 2	Context-free grammar	Context-free language	Pushdown automaton
Type 3	Regular grammar	Regular language	Finite state automaton

Take a look at the following illustration. It shows the scope of each type of grammar −



Type - 3 Grammar

Type-3 grammars generate regular languages. Type-3 grammars must have a single non-terminal on the left-hand side and a right-hand side consisting of a single terminal or single terminal followed by a single non-terminal.

The productions must be in the form **X → a or X → aY**

where **X, Y ∈ N** (Non terminal)

and **a ∈ T** (Terminal)

The rule **S → ε** is allowed if **S** does not appear on the right side of any rule.

Example

$X \rightarrow \varepsilon$
$X \rightarrow a \mid aY$
$Y \rightarrow b$

Type - 2 Grammar

Type-2 grammars generate context-free languages.

The productions must be in the form $\mathbf{A \rightarrow \gamma}$

where $\mathbf{A \in N}$ (Non terminal)

and $\boldsymbol{\gamma \in (T \cup N)^*}$ (String of terminals and non-terminals).

These languages generated by these grammars are be recognized by a non-deterministic pushdown automaton.

Example

$S \rightarrow X\ a$
$X \rightarrow a$
$X \rightarrow aX$
$X \rightarrow abc$
$X \rightarrow \varepsilon$

Type - 1 Grammar

Type-1 grammars generate context-sensitive languages. The productions must be in the form

$\boldsymbol{\alpha\ A\ \beta \rightarrow \alpha\ \gamma\ \beta}$

where $\mathbf{A \in N}$ (Non-terminal)

and $\boldsymbol{\alpha, \beta, \gamma \in (T \cup N)^*}$ (Strings of terminals and non-terminals)

The strings $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ may be empty, but $\boldsymbol{\gamma}$ must be non-empty.

The rule $\mathbf{S \rightarrow \boldsymbol{\varepsilon}}$ is allowed if S does not appear on the right side of any rule. The languages generated by these grammars are recognized by a linear bounded automaton.

Example

$AB \rightarrow AbBc$
$A \rightarrow bcA$
$B \rightarrow b$

Type - 0 Grammar

Type-0 grammars generate recursively enumerable languages. The productions have no restrictions. They are any phase structure grammar including all formal grammars.

They generate the languages that are recognized by a Turing machine.

The productions can be in the form of **α** → **β** where **α** is a string of terminals and nonterminals with at least one non-terminal and **α** cannot be null. **β** is a string of terminals and non-terminals.

Example

S → ACaB
Bc → acB
CB → DB
aD → Db

Regular Expressions

A Regular Expression can be recursively defined as follows −

- **ε** is a Regular Expression indicates the language containing an empty string. **(L (ε) = {ε})**
- **φ** is a Regular Expression denoting an empty language. **(L (φ) = { })**
- **x** is a Regular Expression where **L = {x}**
- If **X** is a Regular Expression denoting the language **L(X)** and **Y** is a Regular Expression denoting the language **L(Y)**, then
    - **X + Y** is a Regular Expression corresponding to the language **L(X) ∪ L(Y)** where **L(X+Y) = L(X) ∪ L(Y)**.
    - **X . Y** is a Regular Expression corresponding to the language **L(X) . L(Y)** where **L(X.Y) = L(X) . L(Y)**
    - **R*** is a Regular Expression corresponding to the language **L(R*)**where **L(R*) = (L(R))***
- If we apply any of the rules several times from 1 to 5, they are Regular Expressions.

Some RE Examples

Regular Expressions	Regular Set
(0 + 10*)	L = { 0, 1, 10, 100, 1000, 10000, … }
(0*10*)	L = {1, 01, 10, 010, 0010, …}
(0 + ε)(1 + ε)	L = {ε, 0, 1, 01}
(a+b)*	Set of strings of a's and b's of any length including the null string. So L = { ε, a, b, aa , ab , bb , ba, aaa…….}

(a+b)*abb	Set of strings of a's and b's ending with the string abb. So L = {abb, aabb, babb, aaabb, ababb, ………….}
(11)*	Set consisting of even number of 1's including empty string, So L= {ε, 11, 1111, 111111, ……….}
(aa)*(bb)*b	Set of strings consisting of even number of a's followed by odd number of b's , so L = {b, aab, aabbb, aabbbbb, aaaab, aaaabbb, ………….}
(aa + ab + ba + bb)*	String of a's and b's of even length can be obtained by concatenating any combination of the strings aa, ab, ba and bb including null, so L = {aa, ab, ba, bb, aaab, aaba, ………….}

Regular Sets

Any set that represents the value of the Regular Expression is called a Regular Set.

Properties of Regular Sets

Property 1. *The union of two regular set is regular.*

Proof –

Let us take two regular expressions

$RE_1 = a(aa)^*$ and $RE_2 = (aa)^*$

So, L_1 = {a, aaa, aaaaa,.....} (Strings of odd length excluding Null)

and L_2 ={ ε, aa, aaaa, aaaaaa,.......} (Strings of even length including Null)

$L_1 \cup L_2$ = { ε, a, aa, aaa, aaaa, aaaaa, aaaaaa,.......}

(Strings of all possible lengths including Null)

RE $(L_1 \cup L_2) = a^*$ (which is a regular expression itself)

Hence, proved.

Property 2. *The intersection of two regular set is regular.*

Proof –

Let us take two regular expressions

$RE_1 = a(a^*)$ and $RE_2 = (aa)^*$

So, L_1 = { a,aa, aaa, aaaa, ...} (Strings of all possible lengths excluding Null)

L_2 = { ε, aa, aaaa, aaaaaa,.......} (Strings of even length including Null)

$L_1 \cap L_2$ = { aa, aaaa, aaaaaa,.......} (Strings of even length excluding Null)

RE ($L_1 \cap L_2$) = aa(aa)* which is a regular expression itself.

Hence, proved.

Property 3. *The complement of a regular set is regular.*

Proof −

Let us take a regular expression −

RE = (aa)*

So, L = {ε, aa, aaaa, aaaaaa, ...} (Strings of even length including Null)

Complement of **L** is all the strings that is not in **L**.

So, L’ = {a, aaa, aaaaa, ...} (Strings of odd length excluding Null)

RE (L’) = a(aa)* which is a regular expression itself.

Hence, proved.

Property 4. *The difference of two regular set is regular.*

Proof −

Let us take two regular expressions −

RE_1 = a (a*) and RE_2 = (aa)*

So, L_1 = {a, aa, aaa, aaaa, ...} (Strings of all possible lengths excluding Null)

L_2 = { ε, aa, aaaa, aaaaaa,.......} (Strings of even length including Null)

$L_1 – L_2$ = {a, aaa, aaaaa, aaaaaaa, ...}

(Strings of all odd lengths excluding Null)

RE ($L_1 – L_2$) = a (aa)* which is a regular expression.

Hence, proved.

Property 5. *The reversal of a regular set is regular.*

Proof −

We have to prove $\mathbf{L^R}$ is also regular if **L** is a regular set.

Let, L = {01, 10, 11, 10}

RE (L) = 01 + 10 + 11 + 10

L^R = {10, 01, 11, 01}

RE (L^R) = 01 + 10 + 11 + 10 which is regular

Hence, proved.

Property 6. *The closure of a regular set is regular.*

Proof −

If L = {a, aaa, aaaaa, ...} (Strings of odd length excluding Null)

i.e., RE (L) = a (aa)*

L* = {a, aa, aaa, aaaa ,aaaaa,……………} (Strings of all lengths excluding Null)

RE (L*) = a (a)*

Hence, proved.

Property 7. *The concatenation of two regular sets is regular.*

Proof −

Let RE_1 = (0+1)*0 and RE_2 = 01(0+1)*

Here, L_1 = {0, 00, 10, 000, 010, ...} (Set of strings ending in 0)

and L_2 = {01, 010,011,.....} (Set of strings beginning with 01)

Then, L_1 L_2 = {001,0010,0011,0001,00010,00011,1001,10010,.............}

Set of strings containing 001 as a substring which can be represented by an RE − (0 + 1)*001(0 + 1)*

Hence, proved.

Identities Related to Regular Expressions

Given R, P, L, Q as regular expressions, the following identities hold −

- ∅* = ε
- ε* = ε
- RR* = R*R
- R*R* = R*
- (R*)* = R*
- RR* = R*R
- (PQ)*P =P(QP)*
- (a+b)* = (a*b*)* = (a*+b*)* = (a+b*)* = a*(ba*)*

- $R + \emptyset = \emptyset + R = R$ (The identity for union)
- $R \varepsilon = \varepsilon R = R$ (The identity for concatenation)
- $\emptyset L = L \emptyset = \emptyset$ (The annihilator for concatenation)
- $R + R = R$ (Idempotent law)
- $L (M + N) = LM + LN$ (Left distributive law)
- $(M + N) L = ML + NL$ (Right distributive law)
- $\varepsilon + RR^* = \varepsilon + R^*R = R^*$

Declarations

In programming, a declaration is a statement describing an identifier, such as the name of a variable or a function. Declarations are important because they inform the compiler or interpreter what the identifying word means, and how the identified thing should be used.

A declaration may be optional or required, depending on the programming language. For example, in the C programming language, all variables must be declared with a specific data type before they can be assigned a value.

Declarations provide information about the name and type of data objects needed during program execution.

Two types of declaration:

- implicit declaration
- explicit declaration

Implicit declaration or default declaration:

They are those declarations which are done by compiler when no explicit declaration or user defined declaration is mentioned.

Example

$abc='astring';

$abc=7;

In 'perl' compiler implicitly understand that

$abc ='astring' is a string variable and

$abc=7; is an integer variable.

Explicit declaration of data object:

Float A,B;

It is an example of Float A,B, of c language. In explicit we or user explicitly defined the variable type. In this example it specifies that it is of float type of variable which has name A & B.

A "Declaration" basically serves to indicate the desired lifetime of data objects.

Declarations of operations:

- Compiler need the signature of a prototype of a subprogram or function so it can determine the type of argument is being used and what will be the result type.

*** Before the calling of subprogram, Translator needs to know all these information. ***

Purpose of Declarations:

1) Choice of storage representation: AS Translator determines the best storage representation of data types that why it needs to know primarily the information of data type and attribute of a data object.

2) Storage Management: It make to us to use best storage management for data object by providing its information and these information as tells the lifetime of a data object.

For Example:-

In C language we have many options for declaration for elementary data type.

1) Simple Declaration: Like float A,B;

It tells lifetime is only at the end of execution **as lifetime of every data objects can be maximum to end of execution time.**

But simple declaration tells the single block of memory will be allocated.

2) Runtime Declaration: C language and many more language provide us the feature of dynamic memory allocation by keywords "Malloc and Calloc."

So in this special block of memory is allocated in memory and their lifetime is also different.

3) Polymorphic operations: In most language, some special symbol like + to designate any one of the several different operation which depends on type of data or argument is provided.
In this operation has some name like as we discussed + in this case operation symbol is said to be overloaded because it does not designate one specific operation.

Ada: allows programmer to overload subprograms.

ML: Expands this concept with full polymorphism where function has one name but variety of implementation depending on the types of arguments.

4) Type checking:- Declaration is basically for static type checking rather than dynamic.