

MODULE 2

Syntax and Semantics

What is Syntax?

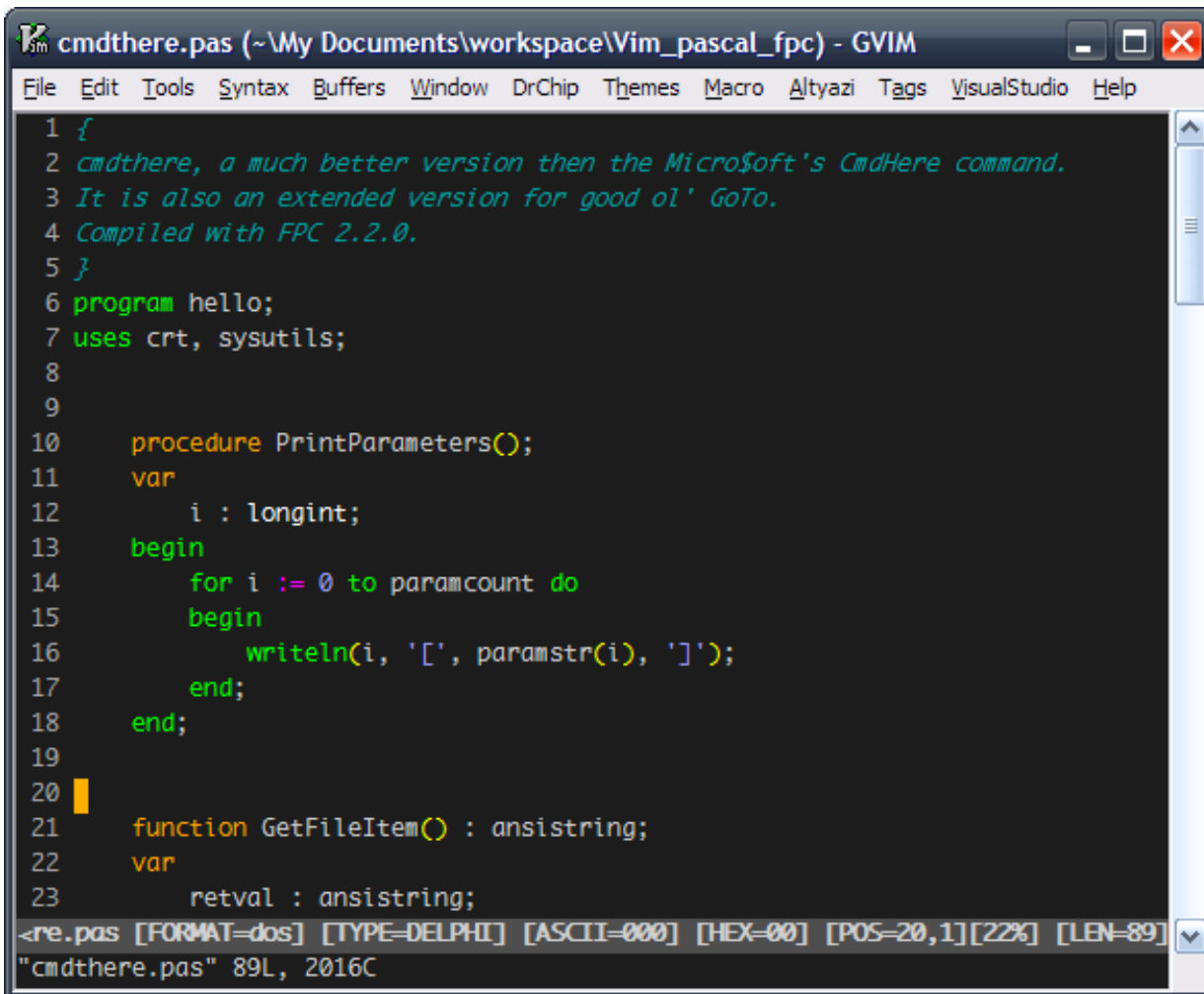
According to the Webster Dictionary

Definition of *syntax*

- 1 **a** : the way in which linguistic elements (such as words) are put together to form constituents (such as phrases or clauses)

 b : the part of grammar dealing with this
- 2 : a connected or orderly system : harmonious arrangement of parts or elements
 // the *syntax* of classical architecture
- 3 : syntaxics especially as dealing with the formal properties of languages or calculi

- The program definition must be correct programs, otherwise, we will have a syntax error.
- The syntax of the program must be correct before it can be executed.
- Unlike in the English language, a sentence with syntax errors can still be understood.
- This is because computers are not as intelligent as humans.



```
1 {  
2 cmdthere, a much better version then the Micro$oft's CmdHere command.  
3 It is also an extended version for good ol' GoTo.  
4 Compiled with FPC 2.2.0.  
5 }  
6 program hello;  
7 uses crt, sysutils;  
8  
9  
10 procedure PrintParameters();  
11 var  
12     i : longint;  
13 begin  
14     for i := 0 to paramcount do  
15     begin  
16         writeln(i, '[', paramstr(i), ']');  
17     end;  
18 end;  
19  
20  
21 function GetFileItem() : ansistring;  
22 var  
23     retval : ansistring;
```

<re.pas [FORMAT=dos] [TYPE=DELPHI] [ASCII=000] [HEX=00] [POS=20,1][22%] [LEN=89]
"cmdthere.pas" 89L, 2016C

Describing Syntax

- A language is a set of strings of characters from some alphabet.
- The strings of a language are called **sentences** or statements.
- The lowest-level syntactic units are called **lexemes**.
 - It may include some literals, operators, and special words.

Consider the following Java statement:

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

The lexemes and tokens of this statement are

<i>Lexemes</i>	<i>Tokens</i>
index	identifier
=	equal_sign
2	int_literal

*	mult_op
count	identifier
+	plus_op
17	int_literal
;	semicolon

Backus-Naur Form and Context-Free Grammars

In the middle to late 1950s, two men, Noam Chomsky and John Backus, in unrelated research efforts, developed the same syntax description formalism, which subsequently became the most widely used method for programming language syntax.

This revised method of syntax description became known as Backus-Naur Form, or simply BNF. BNF is a natural notation for describing syntax. In fact, something similar to BNF was used by Panini to describe the syntax of Sanskrit several hundred years before Christ (Ingerman, 1967).

Although the use of BNF in the ALGOL 60 report was not immediately accepted by computer users, it soon became and is still the most popular method of concisely describing programming language syntax.

It is remarkable that BNF is nearly identical to Chomsky's generative devices for context-free languages, called context-free grammars. In the remainder of the chapter, we refer to context-free grammars simply as grammars. Furthermore, the terms BNF and grammar are used interchangeably.

A metalanguage is a language that is used to describe another language. BNF is a metalanguage for programming languages.

`<assign> → <var> = <expression>`

The text on the left side of the arrow, which is aptly called the left-hand side (LHS), is the abstraction being defined. The text to the right of the arrow is the definition of the LHS. It is called the right-hand side (RHS) and consists of some mixture of tokens, lexemes, and references to other abstractions.

`total = subtotal1 + subtotal2`

$\langle \text{if_stmt} \rangle \rightarrow \text{if (} \langle \text{logic_expr} \rangle \text{) } \langle \text{stmt} \rangle$

$\langle \text{if_stmt} \rangle \rightarrow \text{if (} \langle \text{logic_expr} \rangle \text{) } \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle$

or with the rule

$\langle \text{if_stmt} \rangle \rightarrow \text{if (} \langle \text{logic_expr} \rangle \text{) } \langle \text{stmt} \rangle$

| $\text{if (} \langle \text{logic_expr} \rangle \text{) } \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle$

Although BNF is simple, it is sufficiently powerful to describe nearly all of the syntax of programming languages.

Describing Lists

$\langle \text{ident_list} \rangle \rightarrow \text{identifier}$
 $\quad \quad \quad | \text{identifier}, \langle \text{ident_list} \rangle$

Grammars and Derivations

A grammar is a generative device for defining languages. 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. In a grammar for a complete programming language, the start symbol represents a complete program and is often named `<program>`.

A Grammar for a Small Language

$\langle \text{program} \rangle \rightarrow \mathbf{begin} \langle \text{stmt_list} \rangle \mathbf{end}$

$\langle \text{stmt_list} \rangle \rightarrow \langle \text{stmt} \rangle$

$\quad \quad \quad | \langle \text{stmt} \rangle ; \langle \text{stmt_list} \rangle$

$\langle \text{stmt} \rangle \rightarrow \langle \text{var} \rangle = \langle \text{expression} \rangle$

$\langle \text{var} \rangle \rightarrow A \mid B \mid C$

$\langle \text{expression} \rangle \rightarrow \langle \text{var} \rangle + \langle \text{var} \rangle$

$\quad \quad \quad | \langle \text{var} \rangle - \langle \text{var} \rangle$

$\quad \quad \quad | \langle \text{var} \rangle$

A derivation of a program in this language follows:

```
<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 + C ; <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
```


$\langle \text{program} \rangle \rightarrow \text{begin } \langle \text{stmt_list} \rangle \text{ end}$

$\langle \text{stmt_list} \rangle \rightarrow \langle \text{stmt} \rangle$

$\quad \mid \langle \text{stmt} \rangle ; \langle \text{stmt_list} \rangle$

$\langle \text{stmt} \rangle \rightarrow \langle \text{var} \rangle = \langle \text{expression} \rangle$

$\langle \text{var} \rangle \rightarrow A \mid B \mid C$

$\langle \text{expression} \rangle \rightarrow \langle \text{var} \rangle + \langle \text{var} \rangle$

$\quad \mid \langle \text{var} \rangle - \langle \text{var} \rangle$

$\quad \mid \langle \text{var} \rangle$

$\langle \text{program} \rangle \Rightarrow \text{begin } \langle \text{stmt_list} \rangle \text{ end}$

$\Rightarrow \text{begin } \langle \text{stmt} \rangle ; \langle \text{stmt_list} \rangle \text{ end}$

$\Rightarrow \text{begin } \langle \text{var} \rangle = \langle \text{expression} \rangle ; \langle \text{stmt_list} \rangle \text{ end}$

$\Rightarrow \text{begin } A = \langle \text{expression} \rangle ; \langle \text{stmt_list} \rangle \text{ end}$

$\Rightarrow \text{begin } A = \langle \text{var} \rangle + \langle \text{var} \rangle ; \langle \text{stmt_list} \rangle \text{ end}$

$\Rightarrow \text{begin } A = B + \langle \text{var} \rangle ; \langle \text{stmt_list} \rangle \text{ end}$

$\Rightarrow \text{begin } A = B + C ; \langle \text{stmt_list} \rangle \text{ end}$

$\Rightarrow \text{begin } A = B + C ; \langle \text{stmt} \rangle \text{ end}$

$\Rightarrow \text{begin } A = B + C ; \langle \text{var} \rangle = \langle \text{expression} \rangle \text{ end}$

$\Rightarrow \text{begin } A = B + C ; B = \langle \text{expression} \rangle \text{ end}$

$\Rightarrow \text{begin } A = B + C ; B = \langle \text{var} \rangle \text{ end}$

$\Rightarrow \text{begin } A = B + C ; B = C \text{ end}$

A Grammar for Simple Assignment Statements

$\langle \text{assign} \rangle \rightarrow \langle \text{id} \rangle = \langle \text{expr} \rangle$

$\langle \text{id} \rangle \rightarrow A \mid B \mid C$

$\langle \text{expr} \rangle \rightarrow \langle \text{id} \rangle + \langle \text{expr} \rangle$

$\mid \langle \text{id} \rangle * \langle \text{expr} \rangle$

$\mid (\langle \text{expr} \rangle)$

$\mid \langle \text{id} \rangle$

$$A = B * (A + C)$$

is generated by the leftmost derivation:

$$\langle \text{assign} \rangle \Rightarrow \langle \text{id} \rangle = \langle \text{expr} \rangle$$

$$\Rightarrow A = \langle \text{expr} \rangle$$

$$\Rightarrow A = \langle \mathbf{id} \rangle * \langle \text{expr} \rangle$$

$$\Rightarrow A = B * \langle \text{expr} \rangle$$

$$\Rightarrow A = B * (\langle \text{expr} \rangle)$$

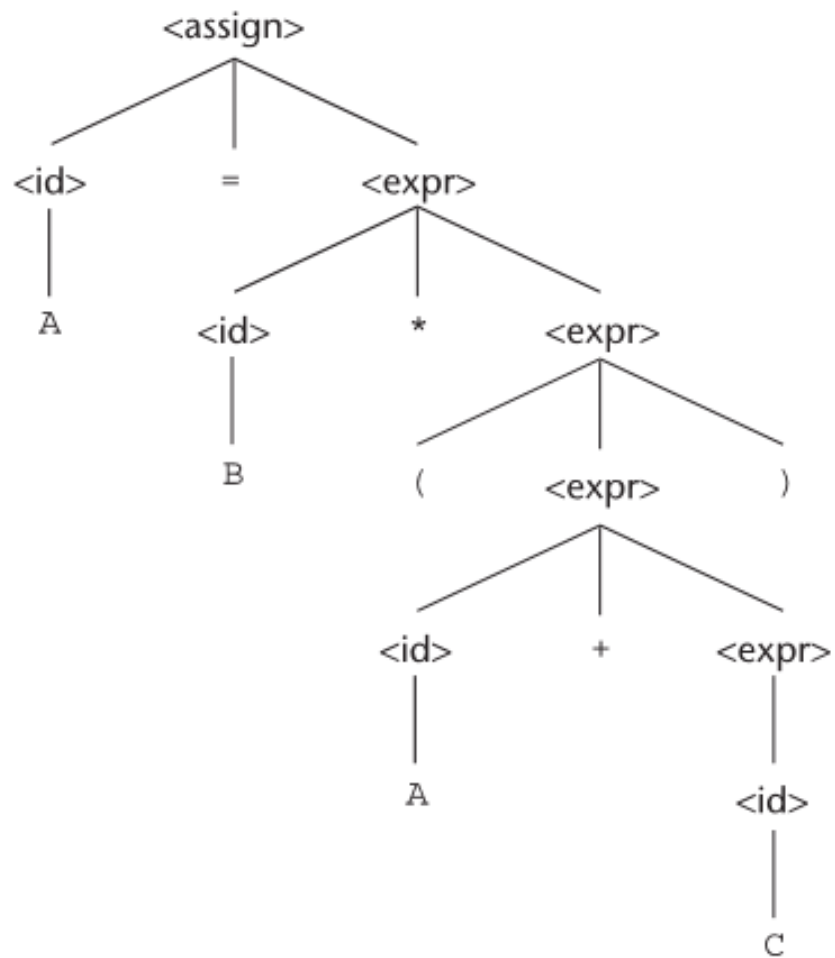
$$\Rightarrow A = B * (\langle \text{id} \rangle + \langle \text{expr} \rangle)$$

$$\Rightarrow A = B * (A + \langle \text{expr} \rangle)$$

$$\Rightarrow A = B * (A + \langle \text{id} \rangle)$$

$$\Rightarrow A = B * (A + C)$$

A parse tree for the
simple statement
A = B * (A + C)



General Syntactic Criteria

Readability - For ordinary user, for example who only knows the language English, COBOL is the most readable among the languages invented so far. However, for a logician, Prolog is the most readable from among the languages.

Writability - Is usually in conflict with the readability features. The more readable a program is the more difficult it is to write the same program. Writability requires more concise and regular structures while readability requires more verbose constructs. “Nakakatamad sumulat ng programs sa COBOL.” - in Filipino.

Prolog on the other hand is the easiest to write simply because it is composed of only one basic statement format.

Ease of translation - Another conflicting objectives of this criteria. To illustrate, take the case of Lisp which is very hard to read and maybe hard to write, but it is very easy to produce a compiler for this language.

- The lesser syntactic construct the better.
- Usually measured by the size.
 - However, it tends to be hard to read and write.
- The ease of translation also affects the popularity of the language. The reason why Algol-68, for example, never became popular is because of the complexity of its language, which makes translation very difficult to do.

Lack of ambiguity - It means that every program in the language must have only one interpretation.

In FORTRAN:

$A(I,J)$

Maybe an access to two-dimensional array or a procedure call.

In Algol-60:

if C1 then if C2 then S1 else S2

Syntactic Elements

Character set - The set of symbol used in the programming language is called its character set or alphabet.

Algol-60 has 52 alphabet, 1- numeric characters and 52 other special characters, a total of 114 characters.

FORTRAN has 47 characters.

COBOL has 52 characters.

PL/I has 60 characters.

Identifiers - Strings used to name data objects, procedures, and keywords.

Example:

BASIC identifiers were restricted to single capital letters, capital followed by a digit, or capital followed by \$.

C identifiers can be of any length but should start with a letter and case sensitive.

Self-Test Exercise

Name your favorite language and mention some keywords or reserved words.

Another is to research some keywords or reserved words for the languages below:

Pascal

Fortran

Operator symbols - These are symbols used to represent the primitive operations in the language.

Example:

PASCAL used symbols +, -, /, *.

LISP used car and cdr (head/tail operator)

Fortran used “.EQ” for equal operator

COBOL used ADD for addition operator

Keywords and reserved words - A keyword is an identifier used as a fixed part of the syntax, for example, the keyword begin and end of Pascal and the switch of C.

A reserved word is a keyword that may not be used as programmer-chosen identifier.

If we do not define them from the language, translation will be difficult.

On the other hand, if we have too many reserved words like in COBOL, it may be very difficult for the programmer to remember all of them.

Comments and noise words - Comments are words ignored during translation.

Example:

Older version of Fortran is placed in column 6 with letter C on it.

PASCAL used { } and (* *)

C uses /* */

C++ have // as single line comment

Noise words, on the other hand, are optional words included in the statements to enhance readability.

Example:

COBOL GO TO <label>

The word TO is noise word since it can be deleted without affecting the program.

GO <label> is the same as GO TO <label>

Delimiters and brackets. A delimiter is a syntactic element used to mark the beginning or end of some syntactic constructs. Some examples of these are the “;” for Pascal and “:” for BASIC.

Brackets on the other hand are paired delimiters used to enclose a group of statements.

In Pascal, we have the “begin” and “end”, while in C the “{” and “}” are used.

Free-field format and fixed-field format.

Example: Pascal and C are free-field format since you can write the ff:

```
for j := 1 to n do
```

```
or
```

```
for
```

```
    j := 1
```

```
to n
```

```
do.
```


Fixed field format is when the positioning of program code in the line are used to convey information.

Older version of Fortran where columns 1 to 5 are for labels.

Expressions. These are basic syntactic elements that are used in most statements. E.g conditions or evaluate values that are assigned to variables.

It could be in the form of prefix, postfix or infix.

Statements. There are several ways by which statements are formulated.

Example, in Pascal where a different syntax for the case statement, if statement, for statement, etc is adopted.

But in Prolog, it adopts a single basic statement format.

```
append ([], L, L) .
```

```
append ([X|L1,L2,[X|L3]] :- append (L1, L2, L3).
```

Overall program-subprogram structure. One is the use of a separate subprogram definition as in FORTRAN.

SUBROUTINE A

...

SUBROUTINE B

....

...

Then, there is the nested subprogram of Pascal.

```
procedure A;
```

```
    procedure B;
```

```
    begin
```

```
end;
```

```
begin
```

```
...
```

```
end;
```

Finally, we have the unseparated subprogram definition (lack of organization), just like in SNOBOL.

MAIN

....

 DEFINE A

 ...

END MAIN <- *end of MAIN*

 ...

END A <- *end of A*

Operator precedence determines the grouping of terms in an expression and decides how an expression is evaluated. Certain **operators** have higher **precedence** than others; for example, the multiplication **operator** has a higher **precedence** than the addition **operator**.

Semantic errors

Semantic errors indicate an improper use of Java statements.

Let us see some examples of semantic errors.

Example 1: Use of a non-initialized variable:

```
int i;
```

```
i++; // the variable i is not initialized
```

Example 2: Type incompatibility:

```
int a = "hello"; // the types String and int are not compatible
```

Example 3: Errors in expressions:

```
String s = "...";
```

```
int a = 5 - s; // the - operator does not support arguments of type String
```


Example 4: Unknown references:

```
Strin x;           // Strin is not defined
```

```
system.out.println("hello"); // system is not defined
```

```
String s;
```

```
s.println();      // println is not a method of the class String
```

Example 5: Array index out of range (dynamic semantic error)

```
int[] v = new int[10];
```

```
v[10] = 100;      // 10 is not a legal index for an array of 10 elements
```

Thank You