

Programming Languages: Instructing the Computer

4A

Specifying syntax
and the role of the compiler



Machine Language



- Algorithms are run on a computer by translating them to a sequence of computer instructions.
- Computer circuitry is designed to execute instructions coded in binary, also called machine language.
 - Machine language varies from processor to processor based on manufacturer, word size, etc.



Programming Languages

- High-level programming languages are designed to allow programmers to write computer instructions in a format more easily understandable by humans.
 - Each language has a set of rules that define what can and cannot be written in that language to define computer instructions.
 - Each language requires some program to translate instructions from this language to machine language.



Example

- Algorithmic Step:
Set z equal to $x + y$.
- In Java:
$$z = x + y;$$
- In binary (byte codes):

0001 1010	(iload_0)
1010 1011	(iload_1)
0110 0000	(iadd)
0011 1101	(istore_2)

Specifying syntax



- The syntax for valid instructions in a programming language can be specified using EBNF (Extended Backus-Naur Form).



John Backus led the team that developed the FORTRAN programming language in the 1950s. Backus developed a formal notation to describe FORTRAN syntax.



Peter Naur used the notation to describe the ALGOL language in the 1960s. ALGOL was the predecessor of famous languages like Pascal, C, Ada and PL/1.

EBNF



- An EBNF description of the syntax of a language is a set of EBNF rules of the form $LHS \leftarrow RHS$
- The LHS contains one word written in lowercase letters.
- The RHS contains some combination of words, literals and controls:
 - | Alternative items (choose one item)
 - [] Optional items (include it or not)
 - { } Repeated items (repeat 0 or more times)



EBNF Example 1

- A string representing an integer can be expressed using the following pair of EBNF rules:

digit \Leftarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

integer \Leftarrow [+|-] digit {digit}

An integer is an optional + or - sign, followed by a digit, followed by zero or more digits.



EBNF Example 1

digit \Leftarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

integer \Leftarrow [+|-] digit {digit}

- Show that -105 is a valid string representing an integer.

integer
[+|-] digit {digit}
- digit {digit}
-1 {digit}
-1 digit digit
-10 digit
-105



EBNF Equivalency

- Are these EBNF expression pairs equivalent?

$0 | 01 \stackrel{?}{\equiv} 0 [1]$

$[0] [1] \stackrel{?}{\equiv} [0 [1]]$

$\{ 0 \} | \{ 1 \} \stackrel{?}{\equiv} \{ 0 | 1 \}$



EBNF Example 2

- A string representing a variable (in some programming language) can be expressed using the following EBNF rules:

letter $\leftarrow a | b | c | d | e | f | g | h | i | j | k | l | m |$
 $n | o | p | q | r | s | t | u | v | w | x | y | z$

digit $\leftarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$

variable $\leftarrow \text{letter} \{ \text{letter} | \text{digit} \}$

A variable is a letter, followed by zero or more letters or digits (in any combination).



EBNF Example 2

```
letter  $\Leftarrow$  a | b | c | ..... | x | y | z  
digit   $\Leftarrow$  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  
variable  $\Leftarrow$  letter { letter | digit }
```

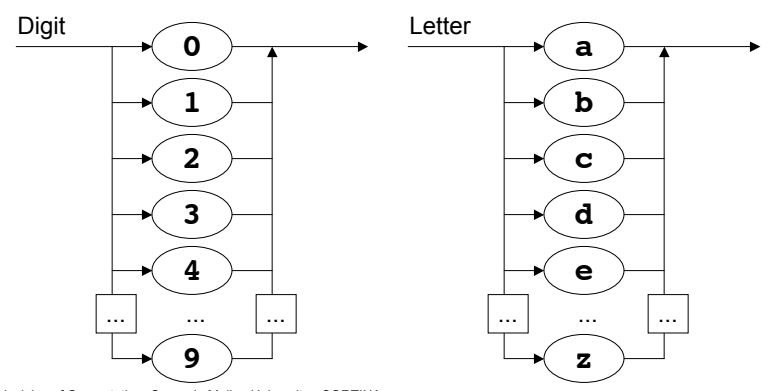
- Show that **r2d2** is a valid string representing a variable.

variable
letter { letter | digit }
r { letter | digit }
r digit letter digit
r2 letter digit
r2d digit
r2d2

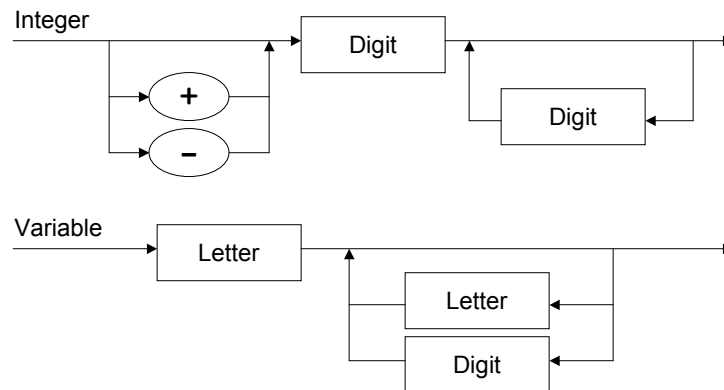


Syntax Diagrams

- Languages can be described using syntax diagrams as well.



Syntax Diagrams



15-105 Principles of Computation, Carnegie Mellon University - CORTINA

13

EBNF Example 3



- A string representing a simple assignment statement can be expressed using the following EBNF rules:

$\text{arith_op} \leftarrow + \mid - \mid * \mid /$

$\text{expression} \leftarrow \text{variable} \{ \text{arith_op} \text{variable} \}$

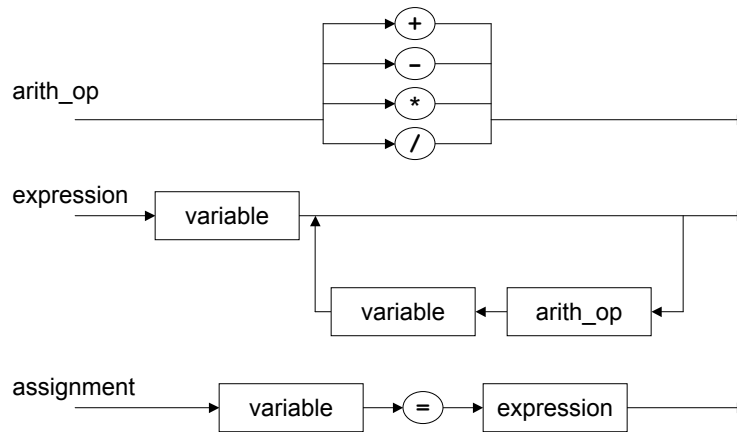
$\text{assignment} \leftarrow \text{variable} = \text{expression}$

An assignment statement is a variable followed by an equals sign followed by an expression.

15-105 Principles of Computation, Carnegie Mellon University - CORTINA

14

More Syntax Diagrams



15-105 Principles of Computation, Carnegie Mellon University - CORTINA

15

EBNF Example 3



Validate: $w = x + y * z$

assignment

variable = expression

letter { letter | digit } = expression

w { letter | digit } = expression

w = expression

w = variable { arith_op variable }

w = letter { letter | digit } { arith_op variable }

$w = x$ { letter | digit } { arith_op variable }

15-105 Principles of Computation, Carnegie Mellon University - CORTINA

16

EBNF Example 3 (cont'd)



```
w = x { arith_op variable }
w = x arith_op variable arith_op variable
w = x + variable arith_op variable
w = x + letter { letter | digit } arith_op variable
w = x + y { letter | digit } arith_op variable
w = x + y arith_op variable
w = x + y * variable
w = x + y * letter { letter | digit}
w = x + y * z { letter | digit}
w = x + y * z
```

Syntax vs. Semantics



- **Syntax** refers to the structure or grammar of the instruction that is to be executed.
- **Semantics** refers to the meaning of the instruction that is to be executed.
- Examples:
 - **while (x < z) x += y ;**
What if y were a string instead of a number?
 - *The purple scream ate the smelly thunderstorm.*
This is a valid sentence grammatically, but...

Programming Process



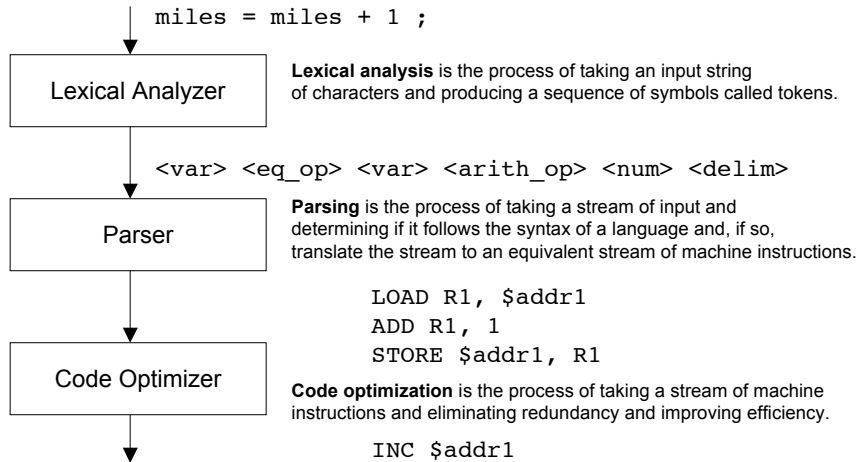
- An algorithm is designed to solve a computational problem.
- A programmer translates this algorithm into instructions written in a high-level programming language.
- The program is translated to machine instructions by a special program called a **compiler**.
- The machine language program is executed on the computer to solve the problem.

Programming Process



- If the programmer does not follow the syntax rules of the programming language, the compiler will report **syntax errors**.
- If the algorithm is incorrect or the programmer does not translate the algorithmic steps correctly, but does follow syntax rules of the programming language, the compiler will translate the computer code.
 - When the machine language program is executed, the results will likely show **logical errors**.

How a compiler works



Compiling vs. Interpreting



- A **compiler** translates an entire program from high-level language to machine instructions before execution begins.
- An **interpreter** translates each high-level instructions one at a time, executing each instruction immediately.
 - Compilers tend to be more complex to build.
 - Interpreters tend to run programs more slowly.

A compiler is only a program



A program accepts input, performs some computations and generates some output:



A compiler is a program that accepts a computer program as its input and generates the machine code for that program:



But since a compiler is just a program...



We can generate a new compiler by writing it in a high-level language and then using the current compiler (for that language) to translate it to machine code.



This is how compilers are created. *But how was the first compiler created if we need a compiler to create it?*