
CODE READING

Our children learn to read long before they can write successfully. A seven-year old can read far more sophisticated language than he or she can write. When we learn a foreign language this is also true, as it is in music, or any another “language”. It makes sense, therefore to learn how to read a computer language before you can write good programs. We can draw an excellent analogy between learning a natural language and learning a computer language. The table below shows how close this is.

Reading stage	Natural language	Computer language
Pre-reading	the alphabet	the symbol system
Flash cards	nouns	identifiers and names
Action pictures	verbs	operators
Simple stories	sentence sequences	command sequences
Descriptive passages	adjectives, adverbs	expressions
Anaphora	pronouns	parameter passing
Discourse understanding	topic, flow, punctuation	algorithms, control flow, punctuation
Literature	Themes, motives, roles	modular structure and program design

So, just as you cannot be expected to read a whole book (much less to write one) before you can recognize words and their meanings, then you must also learn to recognize the parts of a computer program before you can understand the whole thing. In this course, we will greatly compress the reading aspect of the computer language, since you have already learned at least one natural language. In any case, computer languages are much simpler than any natural language, so it goes by much faster.

SYNTAX

There are two things that can help in understanding the syntax of a language. The first is obvious: lots of examples. The second is not so obvious, and only comes later after a genuine feel for the language has been acquired: the grammar for the language. Studying whole programs, and not just fragments, best satisfies the first. This is because there is not time, in the context of a semester-long course to present hundreds of tiny examples, such as you find a child’s first reading book. By the end of five or six books full of simple sentences like “John loves Mary”, a child gets the message of “subject verb object” pretty well, without actually knowing about the parts of speech. However, as adults we can cope with more complexity earlier, as long as the complexity has only one dimension. So students can

cope with whole programs, as long as it is only the syntactic aspects that are being studied.

Your language course instructor will give you a whole program to study and learn from. Here we give some general principles that cover almost all the common languages. After these general principles, we will look at ways of presenting grammars.

SYNTACTIC ELEMENTS

All of the languages in the CS1x7 group share common elements. Some of these are at the level of *tokens*. Token is the word we use in computer languages to talk about the smallest meaningful unit. It corresponds to the word in ordinary language, although, as we shall see, punctuation is often considered at the token level as well. On the other hand, some elements of a language are structural. This is where punctuation really comes in, because we often use punctuation to help us figure out what structure a program has: where the boundaries of a unit or module are, where one thing ends and the next begins, and so on. Natural languages have phrases, sentences, paragraphs, sections, chapters and books. Some of these are signaled by layout conventions, such as chapters always start on a new page, etc., but computer languages almost always use more explicit syntax. The aim is to show how to recognize these elements in a large program listing.

THE CHARACTER SET

Computers use numbers as the basis for all their computations. Storing characters, for instance in someone's name, or in a description, such as the color of their hair needs the ability to store *characters*. The standard set of characters that are used in computer programming is the ASCII set. The American Standard Code for Information Interchange has been the standard for many years, and is in universal usage. The only other code that was used in this country was the EBCDIC code developed by IBM, but this has largely dropped out of use.

Every key on the standard computer keyboard produces one of these code, which is essentially a number, so that the computer can store these characters in its memory. Below is the ASCII table, which lists every key

	0	1	2	3	4	5	6	7	8	9
0	␣	␣	␣	␣	␣	␣	␣	␣	␣	␣
1	␣	␣	␣	␣	␣	␣	␣	␣	␣	␣
2	␣	␣	␣	␣	␣	␣	␣	␣	␣	␣
3	␣	␣		!	"	#	\$	%	&	'
4	()	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[\]	^	_	`	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{		}	~	␣		

and its corresponding code. The column and row number give the actual numeric code. For instance, the letter A is number 65 and the left bracket, [, is number 91. A symbol □ means the character does not print, or show on the screen. These characters are used for controlling the appearance of characters and the text that are made up of these characters. For instance, code number 10 is used as a line feed (next line) and code 9 is the tab character. These two, together with the space (code 32) and the carriage return (code 13) make up the *whitespace* characters.

Notice also that the last two boxes (codes, 128 and 129) are blank. Actually there are only 128 ASCII characters, with codes from 0 to 127. This number is adequate for our Roman script (52 letters, plus the digits and a bunch of punctuation) with room to spare for characters like \$, @ and #. In these days of increasing worldwide computer usage, people who speak languages with other scripts, such as Arabic, Hindi and Korean, have forced a radical expansion of the ASCII table into the idea of *Unicode*. Now there are a total of 65000 plus boxes in the table, which is adequate for all of the worlds scripts. Each script has a designated part of the table, but the Roman script retains its place as the first block, so that all the millions of lines of code that rely on the ordering of the original ASCII table do not have to be changed.

The question of how these character codes are actually stored in programs is dependent on the language. Your instructor will tell you this when you need to know.

TOKENS

Tokens can be of six different kinds:

- 1) Names
- 2) Constants
- 3) Keywords (or reserved words)
- 4) Operators
- 5) Punctuation
- 6) Special, language-dependent

Names are easy to spot because they look like words, albeit words that do not appear in the dictionary. Every language allows the programmer to invent new names for things in the program. Below are names from the CS1x7 languages. You can see that letters, perhaps digits, and maybe one or two other characters make up these names. Most languages insist on starting a name with a letter, although there are differences. Your instructor will tell you the rules for your language.

```
aName          Inventory      pqrst_37      a7012
AVERYLONGNAMEWITHMANYCHARACTERS
```

As you can see, names can be long, short, meaningful or not, but they all start with a letter, and cannot

contain a space. In fact, the space is the most important of the punctuation characters, since it can separate a name from other names.

Constants can be either numeric or character strings. Some languages allow you to name a constant, which then gives you the benefit of having a name for a commonly-used quantity. Numbers are either integers (whole numbers) or floating-point numbers which allow decimal fractions. Examples of numbers are:

12345 -56 12.345 1.2345e3

This last one is a special form called scientific notation, where a power of 10 (the exponent) is given after the letter e.

Examples of character strings in C, C++ and Java are:

"a very long string (with embedded punctuation!)" 'X' '\n'

These last two are actually single characters, so single quote marks are used. Pascal uses single quote marks for all strings. Fortran uses special "Hollerith" constants for strings.

Keywords are names that have special meaning in the language and may not be used for any other than the intended purpose. Examples are the type names, and words that are involved with declarations. Every language has a short list of such words.

Operators are also pretty easy to spot since they almost always contain nonalphabetic characters. They can be grouped according to their use in the following way:

Arithmetic: the usual operations of addition, subtraction, multiplication and division are signified by the operators +, -, *, and /, respectively. Many languages have a modulus or remainder operator, %. Pascal uses the keywords **mod** and **div** for the remainder operation and integer division. C, C++, and Java have unary operators for incrementing and decrementing integers. They are ++ and --.

Relational: when we want to compare quantities in a program, we use a relational operator. The most important of these is equality, signified by =, or by == in C, C++ and Java. Fortran 77 uses special forms, such as .EQ.

Assignment: values are stored inside variables by assignment. The = sign is most often used, although Pascal uses :=. C, C++ and Java also have operators that combine arithmetic operators with =, such as += which means add, then assign.

Logical: the logical operators can join logical expressions in larger expressions. All the languages have and, or and not, the basic operations. In C, C++ and Java, these are &&, || and !, respectively. Again Fortran uses special forms such as .AND.

Special, language-dependent operations are also available. C and C++ have the bit-wise operators, and C, C++ and Pascal have operators that manipulate pointers. C, C++, Java and Pascal have the field access operator.

Punctuation serves two purposes in a computer language. One is to set off parts of syntax from each

other, and the second is to improve readability.

All the languages use the blank or space character to separate names from keywords or other names. Mostly, other *whitespace* characters, such as the newline character, and the tab character are ignored and are only used for readability. This means that the nicely indented layout you typically see in C, C++, Java and Pascal programs is purely for human use; the compiler ignores it all. Fortran is different in that the newline character is used to separate commands, which thus end up one per line, making for easy readability.

Perhaps the next most important punctuation character is the semicolon, which is used by C, C++, Java and Pascal to separate statements. Technically C, C++ and Java use it as a statement terminator, but the distinction is minor.

C, C++ and Java use the braces, { and }, to show where a block of statements begins and ends. Pascal uses the words begin and end for the same purpose.

Apart from these, there is little commonality among the languages, and much use of other characters for punctuation. In general, if a character or group of characters is not an operator, then it must be punctuation, since it cannot be part of a name.

Special-purpose syntax is a part of every language. Once the general rules have been learned, the few oddities and quirks just have to be approached one at a time. Mostly, however, these special-purpose pieces of syntax are related to a concept that is only relevant to that particular language and is thus studied separately.

EXPRESSIONS

Expressions are used in all languages to support computation, which is usually, but not exclusively, a numeric kind of operation. There are two ways that an expression can occur in the context of a statement. The first is as the whole right-hand side of an assignment statement. All languages will allow the following piece of arithmetic:

$$\text{COST} = \text{QUAN} * \text{PRICE}$$

The expression is everything after the = sign. The second context is as an argument to a procedure or function call. Again, every language can say:

$$\text{CALCULATE}(\text{QUAN} * \text{PRICE})$$

Here, the expression is the only argument to the `CALCULATE` procedure.

Simple expressions, like these are of the form **name operator name**, but *literal constants* can also be used instead of the names. The biggest problem with recognizing expressions is that they are often nested. One expression can be contained inside another, as in:

$$\text{COST} = \text{QUAN} * (\text{PRICE} + \text{TAX})$$

Here, the expression to the right of the = sign is one expression: `PRICE + TAX`, nested inside the `QUAN` * ... expression. Parentheses are often used to show *sub-expressions*, although some languages also rely on

the order of precedence of operators where parentheses are not used. Sub-expressions may be nested to any depth, or at least to a depth that most programmers would agree is too much!

Special syntax is used when a *function call* forms part of an expression. All languages use the same syntax: `function-name(argument, ... argument)` where the ... means any number of *arguments* is possible. For example:

```
COST = QUAN * CALCTAX(PRICE)
```

Here, the function `CALCTAX` is being called with the argument `PRICE` to produce a value to be multiplied by `QUAN`.

STATEMENTS OR COMMANDS

All of the languages in CS1x7 use *statements* to achieve their results. A statement is either an assignment, or a procedure call. Assignment statements typically look like **name assignment-operator expression**. Procedure calls look like **procedure-name(argument, ... argument)** where ... means any number of arguments is possible. Statements are separated by semicolons (or a newline in Fortran), so a good way to identify statements is to look for semicolons first.

MODULES OR PROGRAM LAYOUT

Although some languages have a flexible policy on how a program should be structured and laid out as a text file, programmers have developed a set of useful conventions that apply to whatever language they are using. It is very wise to follow these conventions, since the vast majority of programs written today are going to be read and worked on by more than one person. It makes sense, in this case, because just saying Java or Fortran, and following their syntax is not enough. We must also observe the conventions of the language-using community in order to be understood better.

Every language uses the idea of a *module* to split up the program text into more readable chunks. The highest level of modularity is the *procedure* or *function*, in C, Pascal and Fortran, and the *class* in C++ and Java. These modules should be separated by at least one blank line, and preferably commented as well. The comments allow the reader to spot the beginnings and ends of modules more easily, and of course to convey information to the reader as to their purpose.

Other high-level structures are *global declarations*, which are “outside” of any module, and *compiler directives*, which allow the compiler to carry out special operations.

C, C++, Java and Fortran allow the *text file* itself to be a structuring mechanism, and make multi-file programs. Pascal, however, has a nested module structure where procedures are contained within other procedures, and the top-level procedures are contained in the program itself.

The set of conventions for program layout are different for each language and your instructor will tell you what they are. However, some basic principles are:

1. Separate modules by at least one blank line.

2. Separate sections of code relating to a particular sub-task by a blank line.
3. Where the code has a block structure (C, C++, Java use braces for this, Pascal uses **begin**, **end** pairs) indent nested blocks and line up the start and end of a block horizontally. e.g.

```

{ ...
  { ...
    ...
  }
  ...
}

```

4. Put multiple-line comments before the code they talk about.
5. Put single-line comments either before the code, or after the end of the statement, on the same line.
6. Separate operators from operands by a single space. e.g.

```
QUAN * (PRICE + TAX)
```

NOT

```
QUAN*(PRICE+TAX)
```

Not everyone agrees with the last convention, but almost all text books follow it.

GRAMMARS AND LANGUAGE DEFINITIONS

A *grammar* is a formal definition of the syntax of a language. English and other natural languages have enormously complicated grammars, but computer languages have simple ones, by comparison. Often they are only useful for people writing compilers and other software designed to manipulate text files written in the language, but they can also be useful for programmers as a reference to what is allowable in the language. A grammar will not tell you what to write, but it might tell you how to write it.

BNF

Most grammars are presented in a form call Backus-Naur Form, or *BNF*. There are two main concepts in a BNF definition. The first is that of a *terminal symbol*, which is something that can occur in a text file written in the language in question. Terminal symbols are things like operators, punctuation symbols, and the characters that make up words and literal constants. The second concept is that of a non-terminal symbol, which is a category to which a particular syntactic construct can belong. Non-terminal symbols have names, but these are not names we put in programs, they are names that only have meaning in the context of the grammar. One such name might be statement. Another might be expression. Others may be less understandable, like storage-qualifier, or access-modifier. People who write grammars often make up pretty unreadable names.

One non-terminal symbol is designated as the topmost symbol. It usually represents the category of complete programs. You cannot get higher than that. Every non-terminal symbol then needs a definition in terms of

a *rule*. Each rule says how a particular category can be broken-down into a sequence of other categories and literals. A rule looks like this:

```
Non-terminal-symbol ::= sequence of non-terminal-symbols and terminal-
symbols
```

Note the production symbol ::= (rules are also called *productions*). Various forms of BNF will use different production symbols, but ::= is the original one.

As a simple example, let us look at the syntax of an expression involving only the four operations add, subtract, multiply and divide. Our top-level category we will call expression. Now where there are alternatives in a piece of syntax, they are separated by the vertical bar, |. So:

```
expression ::= operand operator operand
```

This shows that an expression is an operand, followed by an operator and then another operand. We need to define operand:

```
operand ::= name | constant | ( expression )
```

Thus an operand can be either a name, or a constant, or an expression in parentheses. Note the terminal symbols for the parentheses. Finally the category operator is defined as:

```
operator ::= + | - | * | /
```

The category name is an interesting one because names, in general can be of any number of characters. The way we express this in BNF is to use recursion. We will also say that the first character must be a letter, something that all languages enforce for names.

```
name ::= letter character-sequence
```

```
character-sequence ::= character | character character-sequence
```

```
character ::= letter | digit
```

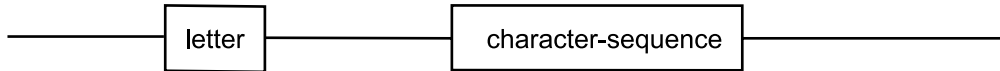
```
letter ::= a | b | ... z | A | B ... Z
```

```
digit ::= 0 | 1 ... 9
```

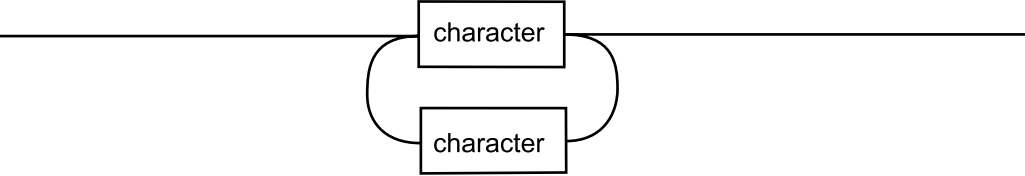
SYNTAX DIAGRAMS

The developers of Pascal introduced a new way of showing a grammar. In a *syntax diagram*, the same concepts of terminal and non-terminal symbols are retained, but the rules are shown as node and line diagrams with branches indicating alternatives, and loops substituting for recursion. Our little expression grammar looks like this in syntax diagram form:

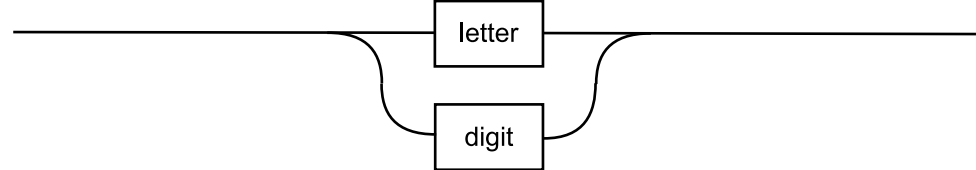
name



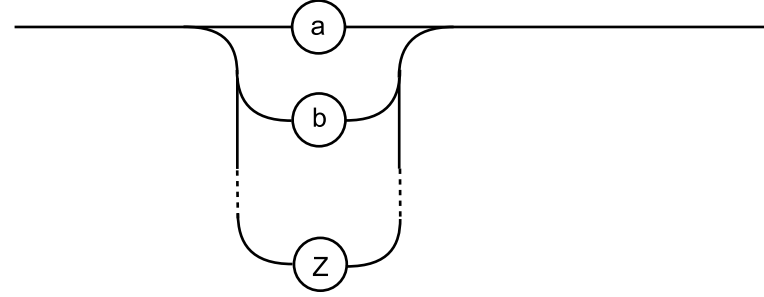
character-sequence



character



letter



digit

