beginner

COLLABORATORS

	TITLE : beginner	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		February 24, 2025	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

beginner

1.1 Constants

```
Constants
*********
```

A constant is a value that does not change. A (literal) number like 121 is a good example of a constant--its value is always 121. We've already met another kind of constant: string constants (see Strings). As you can doubtless tell, constants are pretty important things.

```
  Numeric Constants
  String Constants Special Character Sequences
  Named Constants
  Enumerations
  Sets
```

1.2 Numeric Constants

```
Numeric Constants
=================
```

We've met a lot of numbers in the previous examples. Technically speaking, these were numeric constants (constant because they don't change value like a variable might). They were all decimal numbers, but you can use hexadecimal and binary numbers as well. There's also a way of specifying a number using characters. To specify a hexadecimal number you use a $ before the digits (and after the optional minus sign - to represent a negative value). To specify a binary number you use a % instead.

Specifying numbers using characters is more complicated, because the base of this system is 256 (the base of decimal is ten, that of hexadecimal is 16 and that of binary is two). The digits are enclosed in double-quotes (the " character), and there can be at most four digits. Each digit is a character representing its ASCII value. Therefore, the

character A represents 65 and the character 0 (zero) represents 48. This upshot of this is that character A has ASCII value "A" in E, and "0z" represents ("0" * 256) + "z" = (48 * 256) + 122 = 12,410. However, you probably don't need to worry about anything other than the single character case, which gives you the ASCII value of the character.

The following table shows the decimal value of several numeric constants. Notice that you can use upper- or lower-case letters for the hexadecimal constants. Obviously the case of characters is significant for character numbers.

Number	Decimal value
21	21
-143	-143
$1a	26
-$B1	-177
%1110	14
-%1010	-10
"z"	122
"Je"	19,045
-"A"	-65

1.3 String Constants Special Character Sequences

String Constants: Special Character Sequences
==========

We have seen that in a string the character sequence \n means a linefeed (see Strings). There are several other similar such special character sequences which represent useful characters that can't be typed in a string. The following table shows all these sequences. Note that there are some other similar sequences which are used to control formatting with built-in procedures like WriteF. These are listed where WriteF and similar procedures are described (see Input and output functions).

Sequence	Meaning
\0	A null (ASCII zero)
\a	An apostrophe '
\b	A carriage return (ASCII 13)
\e	An escape (ASCII 27)
\n	A linefeed (ASCII 10)
\q	A double quote (ASCII 34)
\t	A tab (ASCII 9)
\\	A backslash \

An apostrophe can also be produced by typing two apostrophes in a row in a string. It's best to use this only in the middle of a string, where it's nice and obvious:

```
WriteF('Here\as an apostrophe.\n')       /* Using \a */
```

```
       WriteF('Here''s another apostrophe.\n')  /* Using '' */
```

1.4 Named Constants

```
Named Constants
===============
```

It is often nice to be able to give names to certain constants. For instance, as we saw earlier, the truth value TRUE actually represents the value -1, and FALSE represents zero (see Logic and comparison). These are our first examples of named constants. To define your own you use the CONST keyword as follows:

```
     CONST ONE=1, LINEFEED=10, BIG_NUM=999999
```

This has defined the constant ONE to represent one, LINEFEED ten and BIG_NUM 999,999. Named constants must begin with two uppercase letters, as mentioned before (see Identifiers).

You can use previously defined constants to give the value of a new constant, but in this case the definitions must occur on different CONST lines.

```
     CONST ZERO=0
     CONST ONE=ZERO+1
     CONST TWO=ONE+1
```

The expression used to define the value of a constant can use only simple operators (no function calls) and constants.

1.5 Enumerations

```
Enumerations
============
```

Often you want to define a whole lot of constants and you just want them all to have a different value so you can tell them apart easily. For instance, if you wanted to define some constants to represent some famous cities and you only needed to know how to distinguish one from another then you could use an enumeration like this:

```
     ENUM LONDON, MOSCOW, NEW_YORK, PARIS, ROME, TOKYO
```

The ENUM keyword begins the definitions (like the CONST keyword does for an ordinary constant definition). The actual values of the constants start at zero and stretch up to five. In fact, this is exactly the same as writing:

```
     CONST LONDON=0, MOSCOW=1, NEW_YORK=2, PARIS=3, ROME=4, TOKYO=5
```

The enumeration does not have to start at zero, though. You can change the starting value at any point by specifying a value for an enumerated constant. For example, the following constant definitions are equivalent:

```
ENUM APPLE, ORANGE, CAT=55, DOG, GOLDFISH, FRED=-2,
     BARNEY, WILMA, BETTY

CONST APPLE=0, ORANGE=1, CAT=55, DOG=56, GOLDFISH=57,
      FRED=-2, BARNEY=-1, WILMA=0, BETTY=1
```

1.6 Sets

Sets
====

Yet another kind of constant definition is the set definition. This useful for defining flag sets, i.e., a number of options each of which can be on or off. The definition is like a simple enumeration, but using the SET keyword and this time the values start at one and increase as powers of two (so the next value is two, the next is four, the next eight, and so on). Therefore, the following definitions are equivalent:

```
SET ENGLISH, FRENCH, GERMAN, JAPANESE, RUSSIAN

CONST ENGLISH=1, FRENCH=2, GERMAN=4, JAPANESE=8, RUSSIAN=16
```

However, the significance of the values it is best shown by using binary constants:

```
CONST ENGLISH=%00001, FRENCH=%00010, GERMAN=%00100,
      JAPANESE=%01000, RUSSIAN=%10000
```

If a person speaks just English then we can use the constant ENGLISH. If they also spoke Japanese then to represent this with a single value we'd normally need a new constant (something like ENG_JAP). In fact, we'd probably need a constant for each combination of languages a person might know. However, with the set definition we can OR the ENGLISH and JAPANESE values together to get a new value, %01001, and this represents a set containing both ENGLISH and JAPANESE. On the other hand, to find out if someone speaks French we would AND the value for the languages they know with %00010 (or the constant FRENCH). (As you might have guessed, AND and OR are really bit-wise operators, not simply logical operators. See Bitwise AND and OR.)

Consider this program fragment:

```
speak:=GERMAN OR ENGLISH OR RUSSIAN  /* Speak any of these */
IF speak AND JAPANESE
  WriteF('Can speak in Japanese\n')
ELSE
  WriteF('Unable to speak in Japanese\n')
ENDIF
IF speak AND (GERMAN OR FRENCH)
  WriteF('Can speak in German or French\n')
```

```
    ELSE
      WriteF('Unable to speak in German or French\n')
    ENDIF
```

The assignment sets speak to show that the person can speak in German, English or Russian. The first IF block tests whether the person can speak in Japanese, and the second tests whether they can speak in German or French.

When using sets be careful you don't get tempted to add values instead of OR-ing them. Adding two different constants from the same set is the same as OR-ing them, but adding the same set constant to itself isn't. This is not the only time addition doesn't give the same answer, but it's the most obvious. If you to stick to using OR you won't have a problem.