FLOW++ Reference Manual

1. Introduction
   This reference manual describes the FLOW++ programming language. It is not a tutorial. Here you can find important information regarding FLOW++ in terms of commands used in the language, how to initiate variables and which are the reserved keywords. FLOW++ is case sensitive. Each line have to contain 1 command. Use // to comment your code.

2. Lexical Analysis
   A FLOW++ program is read by a *parser*. Input to the parser is a stream of *tokens*, generated by the *lexical analyzer*. This chapter describes how the lexical analyzer breaks a file into tokens.
   FLOW++ uses the 7-bit ASCII character set for program text.

2.1 Line Structure
A FLOW++ program is divided into a number of *command lines*.

2.1.1 Command lines
The end of a command line is represented by the token NEWLINE. Statements cannot cross logical line boundaries. A command line is constructed from one *physical line*. Only one command is allowed per command line.

2.1.2 Physical lines
A physical line is a sequence of characters terminated by an end-of-line sequence. In source files, any of the standard platform line termination sequences can be used - the Unix form using ASCII LF (linefeed), the Windows form using the ASCII sequence CR LF (return followed by linefeed), or the old Macintosh form using the ASCII CR (return) character. All of these forms can be used equally, regardless of platform.

2.1.3 Comments
A comment starts with double backslash characters (//) that is not part of a string literal, and ends at the end of the physical line. A comment signifies the end of the command line. Comments are ignored by the syntax; they are not tokens.

2.1.4 Blank Lines
A command line that contains only spaces, tabs, formfeeds and possibly a comment, is not llowed (i.e., no NEWLINE token is generated). During interactive input of statements, handling of a blank line will cause an error in syntax. In the standard implementation, an entirely blank logical line (i.e. one containing not even whitespace or a comment) will as well notbe identified by the syntax.

2.1.5 Whitespace between tokens

Except at the beginning of a command line or in string literals, the whitespace characters space, tab and formfeed can be used interchangeably to separate tokens. Whitespace is needed between two tokens only if their concatenation could otherwise be interpreted as a different token (e.g., ab is one token, but a b is two tokens).

2.2 Other tokens

Besides NEWLINE, the following categories of tokens exist: *identifiers*, *keywords*, *literals*, and *delimiters*. Whitespace characters (other than line terminators, discussed earlier) are not tokens, but serve to delimit tokens. Where ambiguity exists, a token comprises the longest possible string that forms a legal token, when read from left to right.

2.3 Identifiers and Keywords

Identifiers (also referred to as *names*) are described by the following lexical definitions:

```
identifier ::=  (letter|"_") (letter |digit | "_")*
letter     ::=  lowercase | uppercase
lowercase  ::=  "a"..."z"
uppercase  ::=  "A"..."Z"
digit      ::=  "0"..."9"
```

Identifiers are unlimited in length. Case is significant.

2.3.1 Keywords

The following identifiers are used as reserved words, or *keywords* of the language, and cannot be used as ordinary identifiers. They must be spelled exactly as written here:

```
Start
End
Genesis
Fatality
Insert
InsertIf
Smash
ShowItToMe
GetOverHere
&&&&&&&&&&*
```

2.4 Literals

Literals are notations for constant values of some built-in types.

2.4.1 String literals

String literals are described by the following lexical definitions:

```
stringliteral   ::=  [stringprefix](shortstring | longstring)
stringprefix    ::=  'r' | 'u' | 'ur' | 'R' | 'U' | 'UR' | 'Ur' | 'uR'
                   | 'b' | 'B' | 'br' | 'Br' | 'bR' | 'BR'
shortstring     ::=  """ shortstringitem* """
longstring      ::=  """ longstringitem* """
shortstringitem ::=  shortstringchar | escapeseq
longstringitem  ::=  longstringchar | escapeseq
shortstringchar ::=  <any source character except "\" or newline or the quote>
longstringchar  ::=  <any source character except "\">
escapeseq       ::=  "\" <any ASCII character>
```

One syntactic restriction not indicated by these productions is that whitespace is not allowed between the stringprefix and the rest of the string literal. The source character set is defined by the encoding declaration; it is ASCII if no encoding declaration is given in the source file.

In plain English: String literals can be enclosed in matching single quotes (') . The backslash (\) character is used to escape characters that otherwise have a special meaning, such as newline, backslash itself, or the quote character. String literals may optionally be prefixed with a letter 'r' or'R'; such strings are called *raw strings* and use different rules for interpreting backslash escape sequences. A prefix of 'u' or 'U' makes the string a Unicode string. Unicode strings use the Unicode character set as defined by the Unicode Consortium and ISO 10646. Some additional escape sequences, described below, are available in Unicode strings. A prefix of 'b' or 'B' indicates that the literal should become a bytes literal (e.g. when code is automatically converted with 2to3). A 'u' or 'b' prefix may be followed by an 'r' prefix.

Unless an 'r' or 'R' prefix is present, escape sequences in strings are interpreted according to rules similar to those used by Standard C. The recognized escape sequences are:

Escape Sequence	Meaning	Notes
\newline	Ignored	
\\	Backslash (\)	
\'	Single quote (')	
\"	Double quote (")	
\a	ASCII Bell (BEL)	
\b	ASCII Backspace (BS)	
\f	ASCII Formfeed (FF)	
\n	ASCII Linefeed (LF)	

Escape Sequence	Meaning	Notes
\N{name}	Character named *name* in the Unicode database (Unicode only)	
\r	ASCII Carriage Return (CR)	
\t	ASCII Horizontal Tab (TAB)	
\uxxxx	Character with 16-bit hex value *xxxx* (Unicode only)	(1)
\Uxxxxxxxx	Character with 32-bit hex value *xxxxxxxx* (Unicode only)	(2)
\v	ASCII Vertical Tab (VT)	
\ooo	Character with octal value *ooo*	(3,5)
\xhh	Character with hex value *hh*	(4,5)

Notes:

1. Individual code units which form parts of a surrogate pair can be encoded using this escape sequence.
2. Any Unicode character can be encoded this way, but characters outside the Basic Multilingual Plane (BMP) will be encoded using a surrogate pair if FLOW++ is compiled to use 16-bit code units (the default).
3. As in Standard C, up to three octal digits are accepted.
4. Unlike in Standard C, exactly two hex digits are required.
5. In a string literal, hexadecimal and octal escapes denote the byte with the given value; it is not necessary that the byte encodes a character in the source character set. In a Unicode literal, these escapes denote a Unicode character with the given value.

Unlike Standard C, all unrecognized escape sequences are left in the string unchanged, i.e., *the backslash is left in the string*. (This behavior is useful when debugging: if an escape sequence is mistyped, the resulting output is more easily recognized as broken.) It is also important to note that the escape sequences marked as “(Unicode only)” in the table above fall into the category of unrecognized escapes for non-Unicode string literals.

When an 'r' or 'R' prefix is present, a character following a backslash is included in the string without change, and *all backslashes are left in the string*. For example, the string literal r"\n" consists of two characters: a backslash and a lowercase 'n'. String quotes can be escaped with a backslash, but the backslash remains in the string; for example, r"\"" is a valid string literal consisting of two characters: a

backslash and a double quote; `r"\"` is not a valid string literal (even a raw string cannot end in an odd number of backslashes). Specifically, *a raw string cannot end in a single backslash* (since the backslash would escape the following quote character). Note also that a single backslash followed by a newline is interpreted as those two characters as part of the string, *not* as a line continuation.

When an `'r'` or `'R'` prefix is used in conjunction with a `'u'` or `'U'` prefix, then the `\uXXXX` and `\UXXXXXXXX` escape sequences are processed while *all other backslashes are left in the string*. For example, the string literal `ur"\u0062\n"` consists of three Unicode characters: 'LATIN SMALL LETTER B', 'REVERSE SOLIDUS', and 'LATIN SMALL LETTER N'. Backslashes can be escaped with a preceding backslash; however, both remain in the string. As a result, `\uXXXX` escape sequences are only recognized when there are an odd number of backslashes.

2.5 Delimiters

The following tokens serve as delimiters in the grammar:

```
(
)
=
,
```

The period can also occur in floating-point and imaginary literals. A sequence of three periods has a special meaning as an ellipsis in slices. The second half of the list, the augmented assignment operators, serve lexically as delimiters, but also perform an operation.

The following printing ASCII characters have special meaning as part of other tokens or are otherwise significant to the lexical analyzer:

```
'
\
&
*
```

The following printing ASCII characters are not used in FLOW++. Their occurrence outside string literals and comments is an unconditional error:

```
$
?
%
```

```
!
@
^
;
[
]
{
}
:
<
>
|
~
#
+
-
.
```

3. Data Model

3.1 Variables

Variables are FLOW++'s abstraction for data. All data in a FLOW++ program is stored in variables.

Every variable has an identity and a value; and is of type string. A variable's *identity* never changes once it has been created; you may think of it as the variable's address in memory.

Strings - The items of a string are characters. There is no separate character type; a character is represented by a string of one item. Characters represent (at least) 8-bit bytes. Bytes with the values 0-127 usually represent the corresponding ASCII values, but the interpretation of values is up to the program.

4. Execution

4.1 Naming and Binding

Names refer to variables. Names are introduced by name binding operations. Each occurrence of a name in the program text refers to the *binding* of that name established in the innermost function block containing the use.

A *block* is a piece of FLOW++ program text that is executed as a unit. The following are blocks: a new Flowchart (Genesis). Each Genesis command typed interactively is a block. A code block is executed in an *execution frame*. A *scope* defines the visibility of a name within a block. If a local variable is defined in a block, its scope includes that block. This means that the following is good:

```
Genesis(Hello)
a = 'Here'
Insert(a, Start, End)
ShowItToMe(Hello)
Genesis(World)
a = 'It is'
Insert(a, Start, End)
ShowItToMe(World)
```

When a name is used in a code block, it is resolved using the nearest enclosing scope. The set of all such scopes visible to a code block is called the block's *environment*.

If a name binding operation occurs anywhere within a code block, all uses of the name within the block are treated as references to the current block. The local variables of a code block can be determined by scanning the entire text of the block for name binding operations.

5. Expressions

This chapter explains the meaning of the elements of expressions in FLOW++.

Syntax Notes: In this and the following chapters, extended BNF notation will be used to describe syntax, not lexical analysis. When (one alternative of) a syntax rule has the form

```
name ::=  othername
```

and no semantics are given, the semantics of this form of `name` are the same as for `othername`.

5.1 Atoms

Atoms are the most basic elements of expressions. The simplest atoms are identifiers or literals. Forms enclosed in reverse quotes or in parentheses, brackets or braces are also categorized syntactically as atoms. The syntax for atoms is:

```
atom      ::=  identifier | literal
```

5.1.1 Identifiers (Names)

An identifier occurring as an atom is a name. When the name is bound to an object, evaluation of the atom yields that object.

5.1.2 Literals

Flow++ supports string literals:

```
literal ::=  stringliteral
```

Evaluation of a literal yields an object of the given type (string) with the given value.

All literals correspond to immutable data types, and hence the variable's identity is less important than its value. Multiple evaluations of literals with the same value (either the same occurrence in the program text or a different occurrence) may obtain the same object or a different object with the same value.

5.2 Primaries

Primaries represent the most tightly bound operations of the language. Their syntax is:

```
primary ::= atom | call
```

5.2.1 Call

A call calls a callable command with a possibly empty series of arguments:

```
call                ::= primary "(" [argument_list [","]
                          | literal genexpr_for] ")"
argument_list       ::= positional arguments ["," Keyword_arguments]
                          ["," "*" literal] ["," keyword_arguments]
                           ["," "**" literal]
                          | keyword_arguments ["," "*" literal]
                           ["," "**" literal]
                          | "*" literal ["," keyword_arguments] ["," 
                          "**"literal]| "**" literal
positional_arguments ::=  literal (","literal)*
keyword_arguments    ::=  keyword_item (","keyword_item)*
keyword_item         ::=  identifier "=" literal
```

A trailing comma may be present after the positional and keyword arguments but does not affect the semantics.

The primary must evaluate to a callable command. All argument expressions are evaluated before the call is attempted.

6. Statements

Statements are comprised within a single Command line. The syntax for statements is:

```
simple_stmt ::=  assignment_stmt
               | call_stmt
```

6.1 Assignment Statement

Assignment statements are used to (re)bind names to values and to modify attributes or items of mutable objects:

```
assignment_stmt ::=  (identifier "=")+ (literal)
```

An assignment statement evaluates a single literal, and assigns the single resulting object to the identifier, from left to right.

The identifier is bound to the current namespace, each time Genesis is called there is a new namespace.

6.2 Call statement

A call statement is the call of a command to create a flow chart. Note FLOW++ is case sensitive.

6.2.1 The Genesis command

```
Genesis_command ::=  "Genesis" (literal)
```

Genesis creates a flowchart. **Genesis(FlowChartName<String>)** Creates the flowchart 'FlowChartName'.

6.2.2 The Fatality command

```
Fatality_command ::=  "Fatality" (literal)
```

Fatality deletes a flowchart. **Fatality(FlowChartName<String>)** Deletes the flowchart 'FlowChartName'.

6.2.3 The Insert command

```
Insert_command ::=  "Insert" (identifier, identifier, identifier)
```

Creates a state in the flowchart. Label, prev and after are all predefined variables. 'label' is the name of the state being inserted, 'prev' is the state that comes before 'label' and 'after' is the state that comes after 'label'. **Insert(label<var>, prev<var>, after<var>)**

6.2.4 The InsertIf command

```
InsertIf_command ::=  "InsertIf" (identifier, identifier, identifier)
```

Creates an if statement. It returns the label. Label, prev, yes and no are all predefined variables. 'label' is the name of the state being inserted, 'prev' is the state that comes before 'label', 'yes' is the states that comes after 'label' if true and 'no' is the states that comes after 'label' if false. **InsertIf(label<var>, prev<var>, yes<var>, no<var>)**

6.2.5 The Smash command

```
Smash_command ::=  "Smash" (identifier)
```

Removes a statement. **Smash(label<var>)** Removes the statement 'label'.

6.2.6 The ShowItToMe command

```
ShowItToMe_command ::=  "ShowItToMe" (literal)
```

Shows a flowchart. **ShowItToMe(FlowChartName<String>)** Shows the flowchart 'FlowChartName'.

6.2.7 The GetOverHere command

```
GetOverHere_command ::=  "GetOverHere" (identifier, identifier)
```

Change the after state of the state 'label'. **GetOverHere(label<var>,after<var>)**