# Flow: A Reference Manual

# Table of Contents

1. Introduction 3

2. Source Files Types 3

2.1 Typedef 3

2.2 Graph Definition 3

2.3 Solver 4

2.4 Relationship Between the Source Files 4

2.4.1 Association 4

2.4.2 Dependencies 4

3. Lexical Conventions 5

3.1 Tokens 5

3.2 Comments 5

3.3 Identifiers 5

3.4 Keywords 5

3.5 Constants 5

3.5.1 String 6

3.5.2 Integers 6

3.5.3 Float 6

3.5.4 List 6

4. Data Types 6

4.1 Basic 6

4.1.1 int 7

4.1.2 float 7

4.1.3 string 7

4.2 Basic Type Conversion 8

4.2.1 Float to Integer 8

4.2.2 Number to String 8

4.2.3 String to Numerical Types 8

4.3 Basic Type Truth Values 9

4.4 Aggregate Data Types 9

4.4.1 Node 9

4.4.2 Arc 10

4.4.3 Graph 10

4.4.4 List 11

4.5 Aggregate Type Truth Values 11

5. Objects and Identifiers 12

5.1 Scope 12

5.2 Labels 12

6. Expressions 13

6.1 Primary Expressions 13

6.2 Unary Operators 14

6.2.1 Numerical Negation Operator 14 6.3 Casts 14

6.4 Multiplicative Operators 14

6.5 Additive Operators 15

6.6 Relational Operators 15

6.7 Equality Operators 16

6.8 Logical AND Operator 16

6.9 Logical OR Operator 16

6.10 Assignment Expressions 16

7. Declarations 17

7.1 Node Declaration 17

7.2 Arc Declaration 17

7.3 Variable Declarations 18

7.4 Function Declaration 18

8. Statements 19

8.1 Expression Statements 19

8.2 Compound Statements 19

8.3 Selection Statements 19

8.4 Iteration Statements 20

8.5 Jump Statements 20

8.6 Function Calls 20

9. Graph Definition 21

9.1 Node Definition 21

9.2 Arc Definition 21

10. Quick Reference 21

10.1 Reserved Words 21

10.2 Symbol Table 22

10.3 Precedence Chart 22

10.4 Context-Free Grammar 23

10.5 Runtime Environment 27

# Section 1: Introduction

This is the reference manual for Flow, a language for solving graphs. Flow was written for Columbia University's 2011 Spring Semester Programming Languages and Translators course, taught by Al Aho. This manual describes the way that Flow will be implemented, so it will serve as an extremely reliable resource to anyone trying to learn it. There are currently no other resources available on how to use Flow other than the in-line documentation and the language tutorial, but we will update the manual with any additional resources as they become available.

As mentioned before, since this text describes the exact way the language will be implemented, everything is precisely as it should act experimentally. If you find through your own experiments that our results are impossible to reproduce, contact us immediately—our github url is [https://github.com/MootyWaffles/Flow](https://github.com/MootyWaffles/FLOW), and we would appreciate it if you alerted us to any error, or better yet, forked and fixed it yourself (but still let us know about it!).

# Section 2: Source Files Types

In Flow, for each application there are three different source file types:

1) Typedef

2) Graph definition

3) Solver

Each file serves a different role in creating the desired application, and all are required for the application.

## 2.1 Typedef

The typedef file serves as the interface between the Graph definition and the Solver. It defines the attributes of the Nodes and Arcs used in the application and specifies the labels that must exist within the graph.

Typedef files have the form

typedef:

Node-declarations

Arc-declarations

Typedef files have the extension .flowt.

## 2.2 Graph Definition

The Graph definition file defines a graph using the types defined in the typedef.

Graph definition files have the form

graph-definition:

typedef-association

Node-definition

Arc-definition

function-definition

graph-statements

Graph definition files have the extension .flowg.

## 2.3 Solver

The solver defines a sequentially executed set of operations on the associated Graph.

Solvers have the form

solver:

typedef association

graph association

function-definition

solver-statements

Solver files have the extension .flow.

## 2.4 Relationship Between the Source Files

### 2.4.1 Association

The graph declaration and solver files must each reference a typedef file. This typedef provides an interface for interaction between them. This means that any solver that using a particular typedef can be applied to any graph that uses the same typedef. The ability to mix and match graphs and solvers is a key aspect of Flow - versatility.

The typedef file is referenced using the use keyword, as follows:

use nodeType.flowt;

This use statement must appear as the first meaningful statement (that is, the very first line that is not a comment) in the graph definition and solver file.

### 2.4.2 Dependencies

The Solver must have exactly one typedef and at least one Graph associated with it. The Graph must have exactly one typedef associated with it. The typedef is independent.

# Section 3: Lexical Conventions

Flow is compiled in two parts—first the graph is compiled into a Java file, and then the solver is compiled into Java with the graph as input. At runtime, the solver runs on the graph. Unlike other languages, there is no preprocessor, so the program is checked and tokenized by the compiler, not the preprocessor.

## 3.1 Tokens

Flow consists of six different types of tokens: comments, identifiers, keywords, whitespace, operators, and constants. All consecutive whitespace, i.e. newlines, carriage returns, tabs, horizontal breaks, comments, and spaces, will be ignored after the first sequential whitespace in a line.

The input-stream will be tokenized greedily, first finding the longest possible string to tokenize, and then recursively finding more longest strings to tokenize.

## 3.2 Comments

Comments can be done with the /\* \*/characters, where all of the characters in between/\* and \*/ will not be compiled, and will in fact be considered whitespace characters. Please note—there is no way to “escape” \*/ within a comment, and comments cannot be nested-- /\* ----- /\* \*/ \*/ will be interpreted that the comments go from /\* ---- /\* \*/, but do not extend to the second \*/.

## 3.3 Identifiers

An identifier is a sequence of letters, digits, and \_ (underscore). The first character must be a letter. Identifiers are case sensitive, so lowercase (i.e. a) and uppercase (i.e. A) letters are interpreted differently (i.e. apple is different from Apple). It is a convention that final variables should be declared in the solver in all capital letters, but in the solver, all variables from the graph declaration are also final, so it should not be regarded as a strict convention that variables in all capital letters are finals, and the rest are not.

## 3.4 Keywords

The following words may not be used as identifiers:

int, float, string, List, Node, Arc, Graph, while, if, else, use, print, return, of

## 3.5 Constants

There are several types of constants in Flow. They are string constants, floating point constants, integer constants, and List constants.

### 3.5.1 String Constants

String constants will start and end with either “ or '. One cannot start with one and end with the other, the same one must be the beginning and the terminal character. There are no character constants—if you want to use one, use a string constant of length one. In between the quotes, there can be any kind of character, as long as it is not the terminal character.

### 3.5.2 Integer Constants

There are only non-negative integer constants—the negative sign is used exclusively as a unary operator (which we will discuss later). Hence, x = 0 and x = -0 are equivalent statements, and the only constant part of the integer is the 0. Furthermore, integers are composed only of digits, as defined in the identifiers subsection.

### 3.5.3 Floating Point Constants

Floating point constants have an integer part and a decimal part. They must be represented as the integer followed by a decimal and the decimal part (##.####, like 12.3456) or just a decimal point followed by the decimal part (.###, like .789), and cannot use E/exponential notation (#.## E #, like 3.56 E 2, is NOT allowed). *Note: In the previous sentences, each # represents a digit of an arbitrary number. This format is simply being used to describe the general form of floating point constants - the number of # is NOT representing how many digits are allowed*.

### 3.5.4 List Constants

List constants can be composed of any other type of constant, are contained in square brackets, and are separated by commas. An example of a valid List constant is [3, 2, 4, 9]. An example of an invalid List constant is [“this is a string”, 1, 4]. This is invalid because every member of a List must have the same type.

# Section 4: Data Types

## 4.1 Basic Types

There are three basic data types in Flow: int, float, and string. The int type represents an integer, float represents a floating-point number, and string represents a character string.

### 4.1.1 int

Type int represents an integer. The typical implementation is to represent an int as a 4-byte two’s-complement number, giving it a range of -2147483648 to 2147483647. This size of an int is also the range that Java has, and since Flow is compiled into Java, this is the range of int in Flow.

The following operators are valid on ints.

+ - Additive operators

\* / % Multiplicative operators

< > <= >= Relational operators

== != Equality operators

= += -= \*= /= %= Assignment operators

For a full description of these operators and their functions, see section 6.

### 4.1.2 float

Type float represents floating-point numbers. The typical implementation is an IEEE double-precision floating-point number, but this may vary from one system to another. The maximum and minimum possible values of a float in Flow will be analogous to the range of floats in Java.

The following operators are valid on floats. The list is very similar to the list of operators valid on ints except the modulus operation is not supported on floats.

+ - Additive operators

\* / Multiplicative operators

< > <= >= Relational operators

== != Equality operators

= += -= \*= /= Assignment operators

For a full description of these operators and their functions, see section 6.

### 4.1.3 string

A string is a sequence of characters, and the character set allowed in Flow strings is the set of printable characters in ASCII. string literals are denoted by single or double quotes, as in “this is a string” and 'this is also a string'. The quotes must match each other - that is, if you begin with double quotes, you must end with double quotes, and the same goes for single quotes.

Every string has an attribute called length that is accessed using the dot operator. This attribute is an int and represents the length of the string in number of characters. For example, “this is a string”.length will be the int value 16.

The following operators are valid on strings.

+ Concatenation

< > <= >= Relational operators

== != Equality operators

For a full description of these operators and their functions, see section 6.

## 4.2 Basic Type Conversions

There is no automatic promotion of data types in Flow. An operator such as + (plus) may operate on two ints or on two floats, but never one of each. For example, the expression 2.5 \* 3 will result in a type error because 2.5 is a float literal and 3 is an int literal. Because of this, Flow provides a built-in mechanism to convert from one basic data type to another through typecasting. The cast operator consists of parentheses with the name of the desired type inside them, as in (float) 3. The foregoing example converts the int literal 3 into a float. This example is trivial, as one could simply use a float literal, but the cast operation is useful in many other situations.

### 4.2.1 Converting from float to int

There is no danger at all in converting a value of type int to a value of type float as in the example above because no information is lost. The new float will be numerically equal to the int value used to create it. However, the same cannot be said when converting a float value into an int value because ints have no fractional component. Any fractional portion of the float value will be truncated (lost) when converting to an int. For example, (int) 3.4 will result in an int with the value 3.

### 4.2.2 Converting Numbers into strings

The casting operators can also be used to convert numerical types (float and int) into type string. This is necessary if the number is to be concatenated with a string. For example, (string) 3.4 will convert the float value 3.4 into a string with value “3.4”. This will work every time regardless of the value of the numerical type because a number can always be represented as a string.

### 4.2.3 Converting strings into Numerical Types

The same cannot be said when converting strings into numerical types because not every character string makes sense as a number. The rules and restrictions for converting strings to numerical types are the same as the rules for specifying numerical literals, except that no other characters are permitted in the string, even after the numerical value.

To convert a string into an int, the characters in the string must be only the digits 0-9 with an optional sign character (+ or -) at the front. For instance, (int) “-34” will work, but (int) “96 salads” and (int) “twenty-one” will fail.

There are also restrictions on the characters allowed to be in a string converted to a float. In addition to the optional sign character and one or more digits, the string being converted to a float may contain at most one decimal point. For example (float) “34.687” will work, but (float) “3.67.8” will fail.

## 4.3 Basic Type Truth Values

Flow does not have an explicit Boolean type. Instead, every basic type has a “truth value” associated with it that is used in logical operations. An int is defined to be false if it is exactly 0 and is true otherwise. A float is false if it is equal to 0.0 or -0.0, and is true otherwise. Any non-empty string (with length > 0) is true, while the empty string (denoted “” or '') is false.

## 4.4 Aggregate Data Types

Flow provides four built-in aggregate types: Node, Arc, Graph, and List.

The nature of the Node, Arc, and Graph types can be changed in the typedef section of the program to fit the needs of a particular application. Attributes can be assigned to the Node and Arc types, and these can also be renamed. (Fans of the Object-Oriented model may wish to think of this as defining a subclass of the parent Node or Arc type with additional fields.) Additionally, the typedef can require that certain Nodes in the Graph be given labels, which then become attributes of the Graph type. For more on defining custom Node and Arc types, see Section 7.

### 4.4.1 Node

The Node type represents one node (or vertex) of the Graph. In that regard, a Node is just a named point, but Nodes may also contain an arbitrary amount of information in their attributes. A Node may only be created or modified in the graph declaration section of the program, but the information inside it can be accessed at any point. Node attributes have the names assigned to them in the typedef section of the program and are accessed using the dot operator. For example, node1.value denotes the attribute called value associated with the Node called node1.

Every Node comes with six attributes that are created automatically. The attribute inDegree is the number of Arcs that point to the Node. The attribute outDegree is the number of Arcs that point away from the Node. If the application is one in which it is more convenient to think of Graph as being undirected, the attribute degree is provided. It is always equal to the sum of inDegree and outDegree. The attributes arcsIn and arcsOut are Lists of Arcs pointing into and away from the Node, respectively. Again, if it more convenient to think of the Graph as undirected, the attribute arcs is a list of all Arcs regardless of direction.

In addition to the automatic Node attributes, the programmer can define additional attributes in a typedef file. By default, these attributes (as well as all attributes mentioned in the previous paragraph) are immutable, meaning their values cannot be changed in the solver portion of the program. This is reasonable because extracting data from a graph should not change the nature or structure of that graph.

### 4.4.2 Arc

An Arc represents an arc (or edge) of the Graph and as such serves to connect exactly two Nodes. Like a Node, an Arc can contain an arbitrary amount of data in its attributes, as specified in the typedef section of the program. Every Arc type automatically has three attributes, called from and to. For an Arc called arc2, the expression arc2.from denotes the Node that arc2 points away from, and arc2.to is the Node that arc2 points toward. If it makes more sense to consider the Graph undirected, the Arc attribute nodes is provided, which is a List containing both of the Nodes that the Arc connects. Additional attributes may be defined in the typedef.

### 4.4.3 Graph

A Graph is the data type that refers to the entire output of the graph declaration section of the program. The Graph has several built-in attributes that can be accessed with the dot operator. Graph.numNodes and Graph.numArcs are int values that represent the number of Nodes in the Graph and the number of Arcs in the Graph, respectively. Graph.nodes is a List of every Node in the Graph, and Graph.arcs is a List of every Arc in the Graph.

In addition, the Graph has an attribute for every label declared in the typedef section, and labels are also accessed using the dot operator. For example, if the typedef specifies a Graph that looks like a binary tree that has a Node designated as the label root, the expression Graph.root represents that Node. Labels are discussed further in section 5.

### 4.4.4 List

A List is an ordered collection of items. In Flow, any single List must contain items that are all of the same type, but that type can be any basic or aggregate type (including List). Lists may appear in Node or Arc attribute, they may be declared in the graph declaration (although they will not be available in the solver), or they may be declared in the solver. The built-in attributes of the List type make it easy to use as several different well-known list data structures.

The attribute length is an int representing the number of items in the List, and those items can be accessed by index (as in an array) with square brackets. Indices begin at 0, so nodeList[nodeList.length - 1] would give the last item in a List called nodeList.

The push, peek, and pop functions allow the List to act like stack, a first-in-last-out list. Example: intList.push(1) adds the int value 1 to the end of the List intList, intList.pop() removes the last int from intList, and intList.peek() is semantically equivalent to intList[intList.length - 1].

If an application calls for a first-in-first-out list, the List type provides functions enqueue (which adds an item to the end of the List) and dequeue (which removes the first item in the List).

More generally, List provides the functions append (which adds an item to the end of List), prepend (which adds an item to the beginning of the List), and remove (which takes as an argument an int value and removes from the List the item at index indicated by the argument).

It is interesting to note that append, push, and enqueue all to exactly the same thing: add an item to the end of the list and increase the value of length by 1. All three are available for the convenience and preference of the programmer.

Only one binary operator can be used with Lists. The + operator joins two Lists into single List. The two operand Lists must contain items of the same type.

## 4.5 Aggregate Type Truth Values

Truth values for aggregate data types work a little differently than truth values for basic data types.

Any Node, Arc, or Graph that has been initialized (the identifier refers to an actual memory location) is true, while any aggregate data type identifier that has not been initialized (and hence does not refer to an actual memory location) is defined to be false.

For Lists, the empty List is defined to be false, but any List with one or more items is defined to be true, regardless of the type or value of those items.

# Section 5: Objects and Identifiers

An object is a location in memory, and an identifier represents a name given to a location in memory. For example, the statement int x = 5; carves out a piece of memory large enough to store an int object, then fills it with the value 5. (More on declarations in Section 7.) That object may then be referenced from elsewhere in the program by its name, the identifier x. Identifiers are used to reference values of any basic or aggregate type as well as function names.

As with most programming languages, Flow uses a recursive definition for identifiers. That is, an identifier is either a letter, or it is an identifier followed by a letter, a number, or an underscore with no intervening whitespace. For example, node1 and get\_arc\_weight would be legal Flow identifiers, but 1node would not. Identifiers must not conflict with any of Flow’s reserved words (see Section 10.1).

## 5.1 Identifier Scope

A file is divided into a hierarchy of blocks by the nesting of the braces { and }. Any code that does not appear within the braces is said to be at the root block. Identifiers declared within a block are accessible to all code blocks within it that appear below the declaration. The parameters of a function should be considered declared within the block that the function is associated with.

Identifiers declared in the graph declaration section are not available in the solver section. Likewise identifiers declared in the solver are not available to the graph declaration. This means that identifiers in the graph declaration can be reused in the solver with no consequences. This also means that Lists and identifiers of basic types used in the graph declaration are not available for use in the solver (unless they are attributes of Nodes or Arcs).

Identifiers declared in the typedef are available to both the graph declaration and the solver. Thus, the typedef provides a useful interface over which the graph and solver interact. Labels (explained in the next section) and Node or Arc attributes are available in both the graph declaration and solver sections, but that is because they are all declared in the typedef section.

## 5.2 Labels

Labels are special identifiers (or names) mandated by the typedef section of a program that apply to Nodes. Identifiers used as labels follow the same rules that other identifiers do: they must begin with a letter and consist only of letters, numbers, and the underscore character, and they must not conflict with any of Flow’s reserved words. A label is assigned to a Node using the colon operator as follows:

/\* label root mandated in the typedef \*/

Node myNode(int value) root;

/\* label assigned to a Node in the graph declaration \*/

root: @rootNode;

/\* Node referred to by label root accessed in solver \*/

print Graph.root.value;

In the above example, the identifier rootNode used in the graph declaration does not persist into solver, but the label root does. The colon operator is used to apply the label to the Node declared in the graph declaration section. The label appears in the solver as an attribute of the Graph type.

# Section 6: Expressions

The order of operations in Flow is defined by the order of the sections as they appear in this section of the reference manual, from other of highest to lowest precedence. If any individual operator has a certain associativity, it will be specified in its section.

A full grammar can be found in Section 10.4. For convenience, a precedence chart and symbol table are also included following the grammar in Sections 10.3 and 10.2.

## 6.1 Primary Expressions

In Flow, identifiers, constants, strings, and expressions contained inside parentheses are considered to be primary expressions.

primary-expression:

identifier

constant

string

(expression)

An identifier is a primary expression, given that it follows the lexical conventions as

specified in Section 3.3. Each identifier has a type, which is determined by its declaration.

A constant is a primary expression, given that it is one of the supported types and follows that type's lexical conventions as specified in Section 3.5.

A string literal is a primary expression. It has the type string - note that it is a lowercase s, denoting that it is considered a "primitive" type in Flow. It is immutable. Lexical conventions for strings are discussed in detail in Section 3.5.1. Note that a nonempty string will return true (in the form of the integer 1) if evaluated logically.

An expression within parentheses is a primary expression. It has the same type and value as the same expression without parentheses. The parentheses merely serve to alter or enforce the order of operations.

## 6.2 Unary Operators

### 6.2.1 Numerical Negation Operator

The numerical negation operator can only be used on operands of type integer or float. Following arithmetic logic, the result of applying the minus operator is the mathematical negative of the operand's value. Note that the negative of 0 is 0.

## 6.3 Casts

In Flow, there is no automatic type conversion, so it is up to the user to cast types. Thus, Flow is strongly typed. Conversion functions are provided in the form of casting, and are constructed by placing the desired type in parentheses before the expression to be converted.

cast-expression:

unary-expression

(type) cast-expression

Note that the cast operates only on the unary expression immediately following it. Therefore, the use of parentheses on the expression to be cast may be necessary to ensure consistent types and calculation.

## 6.4 Multiplicative Operators

In Flow, the multiplicative operators used are \*, /, and %. Multiplicative operators are evaluated from left to right.

multiplicative-expression:

cast-expression

multiplicative-expression \* cast-expression

multiplicative-expression / cast-expression

multiplicative-expression % cast-expression

Both the \* and / operator can take integers and floats as operands. The % operator can only take integers as operands. All three are binary operators.

The \* operator denotes multiplication. The result is the expected arithmetic calculation. The / operator denotes division. For integers, the result is only the quotient. For floats, the result is the expected result, up to a certain precision. The % operator denotes the modulus function and results in the remainder of the division of the first operand by the second.

## 6.5 Additive Operators

In Flow, the additive operators used are + and -. Additive operators are evaluated from left to right.

additive-expression:

multiplicative-expression

additive-expression + multiplicative-expression

additive-expression - multiplicative-expression

Both the + and - operators can take integers and floats as operands, but not both at the same time. Since they are binary operators, both operands must be only integers or only floats.

The + operator also denotes string concatenation or List join, determined by the type of the operands. If both operands are arithmetic, then it performs addition. If both operands are strings, it performs concatenation. If both operands are Lists, it performs a List join. Both operands must always be of the same type.

The - operator denotes subtraction. The result is the expected arithmetic calculation.

## 6.6 Relational Operators

In Flow, the relational operators used are <, <=, >, and >=. Relational expressions evaluate to true or false, which are expressed as 1 and 0, respectively.

relational-expression:

additive-expression

relational-expression < additive-expression

relational-expression <= additive-expression

relational-expression >= additive-expression

relational-expression > additive-expression

The operator < represents less than, <= less than or equal to, > greater than, and >= greater than or equal to. If the relational expression is evaluated to true, the result will be the int 1, while a false will result in the int 0. Relational expressions apply to ints, floats, and strings.

## 6.7 Equality Operators

In Flow, the equality operators used are == and !=. Equality expressions evaluate to true or false, which are expressed as 1 and 0, respectively.

equality-expression:

relational-expression

equality-expression == relational-expression

equality-expression != relational-expression

The operator == represents equal to, and the != operator not equal to. If the equality expression is evaluated to true, the result will be the int 1, while a false will result in the int 0. Equality expressions apply to ints, floats and strings.

## 6.8 Assignment Expressions

In Flow, the assignment operator used is =.

expression:

equality-expression

unary-expression assignment-operator expression

The type of the left operand must be the end type of the operand on the right.

For the = operator, the value that the expression on the right evaluates to replaces the value of the operand on the left.

# Section 7: Declarations

## Section 7.1 Node Declarations

Nodes are declared within the typedef with a list of attributes of various data types. Also, all labels of this node type are declared with the Node declaration.

Node declarations have the form

node-declaration:

Node node-type-name(attribute-list);

Node node-type-name(attribute-list) label-list;

attribute-list

ε

attribute

attribute, attribute-list

attribute:

datatype-keyword name

label-list:

label-name

label-name, label-list

## Section 7.2 Arc Declarations

Arcs are also defined within the typedef with a list of attributes of various data types. Arcs connect 2 nodes, so the declaration includes both the starting and ending node, as well as the symbol ->.

Arc declarations have the form

arc-declaration:

Arc *id (attr-list);*

attribute-list:

ε

attribute

attribute, attribute-list

attribute:

datatype-keyword name

## Section 7.3 Variable Declarations

Variables are declared as needed within both the Graph definition and the Solver.

Variable declarations have the form

variable-declaration:

list-declaration;

single-variable;

init-declaration;

single-variable:

datatype-keyword variable-name

list-declaration:

datatype-keyword name-list

name-list:

variable-name

variable-name, name-list

init-declaration:

single-variable = variable-name

single-variable = constant

datatype-keyword: one of: int float string List Node Arc Graph

## Section 7.4 Function Declaration

Functions are declared and defined within both Graph definitions and the Solver.

There is no concept of declaring functions without definition in Flow, therefore all function declarations have the form

function-declaration:

datatype-keyword function-name(parameter-list){statement-block}

parameter-list:

ε

single-variable

single-variable, parameter-list

statement-block:

ε

statement

statement; statement-block

# Section 8: Statements

## 8.1 Expression Statements

Expression statements have the form

expression-statement:

expression;

;

An expression-statement is usually an assignment or function call. All side effects are completed before another statement is executed. (For more on expressions, see Section 6.)

## 8.2 Compound Statements

Compound statements have the form

compound-statement:

{ declaration-and-statement-list }

declaration-and-statement-list:

declaration-and-statement-list declaration

declaration-and-statement-list statement

ε

A compound-statement allows multiple statements to be grouped together as one statement. This is useful for specifying an iteration-statement or a selection-statement. (For more on declarations, see Section 7.)

## 8.3 Selection Statements

Selection statements control program Flow.

selection-statement:

if (expression) compound-statement

In the selection statement, the expression in parentheses will evaluate to either true or false. (For more on the truth values of data types, see Section 4.1.) If expression evaluates to true, the compound-statement following the if will be executed.

## 8.4 Iteration Statements

Iteration statements construct loops.

iteration-statement:

while (expression) compound-statement

In the while iteration-statement, the compound-statement is executed as long as the expression inside the parentheses evaluates to true.

## 8.5 Jump Statements

jump-statement:

return expression-statement

return causes the current function to end execution and return the evaluated expression-statement to the calling function.

## 8.6 Function Calls

A function call is a statement that calls a defined function with a specific parameter list with zero or more parameters. It has the form

*function call:*

*function-name(parameter-list)*

The parameter list can contain either primitive constants or variables. The order of the parameters must match the parameter list declared in the function header. The return value of the function is given as an evaluation of the function call and will be treated as the result of an expression.

Example:

foo(4, num) //Calls function foo with parameters 4 and the value of num

# Section 9: Graph Definitions

Once the Node and Arc types have been declared in the typedef, the Graph definition source code employs the types to define components of the graph.

## Section 9.1 Node Definition

Nodes are defined only by their attributes in accordance with the associated type defined in the typedef. The attribute list of the definition must match the typedef in the same order and with the same data types in order to be defined as that Node type.

The Node definitions have the form

node-definition:

@node-type-name

@node-type-name attribute-list

This creates a Node in the Graph.

And to assign a specific node to a label defined in the typedef:

label-node:

label-name : node-definition

## Section 9.2 Arc Definition

Arc definition requires two additional pieces of information in addition to the attributes: the source and the destination of the directed arc. Therefore, Arc definitions have the form

arc-definition:

id1 -> id2

id1 -> id2 attr-list

Arcs cannot be named. They are all anonymous.

# Section 10: Quick Reference

## 10.1 Reserved Words

int, float, string, List, Node, Arc, Graph, in, while, if, else, use, print, return, of

## 10.2 Symbol Table

The symbols in this table have syntactically significant meaning.

|  |  |
| --- | --- |
| Symbol | Description |
| (expression)  (type) | Parentheses to force order of operations, casting |
| \* | Multiplication |
| / | Division |
| % | Modulus |
| + | Addition, string concatenation, list join |
| - | Subtraction, unary negative |
| < <= > >= | Relational operators |
| == != | Equality operators |
| = | Assignment |
| -> | Arc definition |
| : | Label application |
| . | Attribute access |
| , | Function argument separation, list member separation |
| ; | End of statement |
| [ ] | List member access, List literal declaration |
| { } | These surround a code block |
| /\* \*/ | Comment |

## 10.3 Precedence Chart

Precedence in Flow is as follows, from highest to lowest. Associativity (where applicable) is noted on the left.

|  |  |
| --- | --- |
| Operator | Associativity |
| ( ) |  |
| . | left |
| [] | left |
| - (unary) | right |
| (type) | right |
| \* / | left |
| % | left |
| + - (binary) | left |
| = | right |
| < > <= >= |  |
| == != |  |
| AND | left |
| OR | left |
| , | left |
| : |  |
| ; | left |

## 10.4 Context-Free Grammar

The Flow grammar is given modularly throughout this document, but is reprinted in its entirety here for convenience. Because there are three file types, there are three “start symbols”, typedef, graph-declaration, and solver.

typedef:

Node-declarations

Arc-declarations

graph-declaration:

typedef-association

Node-definition

Arc-definition

function-definition

graph-statements

solver:

typedef-association

function-definition

solver-statements

typedef-association:

use typedef-file-name;

expression:

logical-OR-expression

unary-expression assignment-operator expression

assignment-operator: one of: \*= /= %= += -=

logical-OR-expression:

logical-AND-expression

(logical-OR-expression OR logical-AND-expression)

logical-AND-expression:

equality-expression

(logical-AND-expression AND equality-expression)

equality-expression:

relational-expression

equality-expression == relational-expression

equality-expression != relational-expression

relational-expression:

additive-expression

relational-expression < additive-expression

relational-expression <= additive-expression

relational-expression >= additive-expression

relational-expression > additive-expression

additive-expression:

multiplicative-expression

additive-expression + multiplicative-expression

additive-expression - multiplicative-expression

multiplicative-expression:

cast-expression

multiplicative-expression \* cast-expression

multiplicative-expression / cast-expression

multiplicative-expression % cast-expression

cast-expression:

unary-expression

(type) cast-expression

type: one of: int float string

unary-expression:

primary-expression

!integer

!float

!expression

-integer

-float

primary-expression:

identifier

constant

string

(expression)

node-declaration:

Node node-type-name(attribute-list);

Node node-type-name(attribute-list) label-list;

attribute-list

ε

attribute

attribute, attribute-list

attribute:

datatype-keyword name

label-list:

label-name

label-name, label-list

arc-declaration:

Arc arc-type-name(attribute-list);

attribute-list:

ε

attribute

attribute, attribute-list

attribute:

datatype-keyword name

variable-declaration:

list-declaration;

single-variable;

init-declaration;

single-variable:

datatype-keyword variable-name

list-declaration:

datatype-keyword name-list

name-list:

variable-name

variable-name, name-list

init-declaration:

single-variable = variable-name

single-variable = constant

datatype-keyword: one of: int float string List Node Arc Graph

function-declaration:

datatype-keyword function-name(parameter-list){statement-block}

parameter-list:

ε

single-variable

single-variable, parameter-list

statement-block:

ε

statement

statement; statement-block

statement:

expression-statement

compound-statement

selection-statement

iteration-statement

jump-statement

expression-statement:

expression;

;

compound-statement:

{ declaration-and-statement-list }

declaration-and-statement-list:

declaration-and-statement-list declaration

declaration-and-statement-list statement

ε

selection-statement:

if (expression) compound-statement

if (expression) compound-statement else compound-statement

iteration-statement:

while (expression) compound-statement

for (expression-statement expression-statement expression-statement) compound-statement

for (identifier in identifier) compound-statement

jump-statement:

break;

continue;

return expression-statement

node-definition:

node-type-name(attribute-list)

node-init:

node-name = node-definition

label-node:

label-name : node-definition

label-name : node-init

arc-init:

arc-name = arc-definition

## 10.5 Runtime Environment

Flow programs are compiled into Java, so the runtime environment is Java.

The compilation process is as follows:

1) Compile the graph.

*The result is a graph file that will be used as input for the solver.*

2) Compile the solver with the freshly outputted graph file.

*The result is the solver in a java file that can be run.*

3) Run the solver.

Example:

Flow ExGraph.flowg --> Graph.java

Flow ExSolver.flow ExGraph.java --> Solver.java

java Solver.java

Here is a block diagram to help you visualize this process.

Complier

Graph Source Code  
(.flowg)

Graph File  
(.java file)

Complier

Solver Source Code  
(.flow)

Flow Program  
(Java Program)

Typedef Source Code  
(.flowt)

Output