University of Southampton

COMP2212 Programming Language Concepts

Walter User Guide

Marco Edoardo Palma & Marcin Wiech

Walter a programming language performing simple computations on potentially unbounded integer sequences.

May 8, 2019

1 Identifiers

You can use letters and decimal digits to name functions and variables, however: every function must start with a uppercase letter, every variable must start with a lowercase letter, and neither can start with a digit. The language is case sensitive.
You cannot use the following keywords: Bool, Int, if, else, Main, stdin, stdout, true, false.
Comments are introduced using //.

2 Data Types

Walter supports Integers (Int) and Booleans (Bool).
Integer Data Type: Signed integer with the range [-2^{29} .. 2^{29}-1].
Boolean Data Type: true and false.
In order to conserve orthogonality, both data types can be arguments of a function's input and output pattern-match.

3 Expressions and Operations

3.1 Assignment & Initialization

- **Initialising a variable**

```
var1 : Int = 0; | var2 : Bool = true;

The pattern is: <variable name>:<variable type>=<constant>
```

- **Assigning a variable**

```
var1 = 23; | var2 = false;
```

3.2 Mathematics Operations

Walter provides operators for standard mathematics operations: addition, subtraction, multiplication, division, negation, modulo, and exponentiation. Examples:

```
var1 = 12 + 3; | var2 = 12/3; | var3 = 12%2; | var5 = 12^4;
```

3.3 Boolean Operations

Walter provides Boolean operations and boolean expressions. Examples:

```
var1 = (true || !false) && true; | var2 = 3 >= 3;
```

3.4 Function's I/O operations with pattern matching

```
[a : Int] -> [a + 3]; //matches one Int to "a" then outputs "a+3"
[a : Bool] -> [!a]; //matches one Bool to "a" then outputs negation of "a"
[a : Int, c : Bool] -> [a + 4, c, !c]; //matches 2 columns and outputs 3 columns
```

Input pattern match (left side of the arrow) is a fixed and strongly typed input segment for a function. It can accept, in any order, an arbitrary number of inputs of both data types.
Output pattern match (right side of the arrow) is a fixed and strongly typed output segment for a function. It can output, in any order, an arbitrary number of boolean and maths operations and variables.

4 If Statements

The pattern for an if statement is:

```
if(<operation that evaluates to Boolean>){<operation 1>}
```

If first operation evaluates to True operation 1 is executed.

The pattern for an if-else statement is:

```
if(<operation that evaluates to Boolean>){<operation 1>}else{<operation 2>}
```

If first operation evaluates to True operation 1 is executed, otherwise operation 2.

5 Functions

5.1 Normal Functions

Here is the general form of a function definition:

```
SumConstant: {k  : Int = 5;}
[a : Int] -> [a + k];
```

This function reads one number at the time and outputs this number plus a constant, k, equal to 5.
The first string represents the function name. Following this is, between curly braces, the variables initialisation area: these variables are the only ones declarable for a function to use, and are strictly bounded the the function and its scope.
Every function body will start with a single input pattern-match declaration. This is then followed by a '->' sign that delimits the start of the routine to be executed for every match until an EOF signal is received. Hence an output match can be introduced at any point of the routine.

5.2 Main Function

Every program requires one function at the bottom of the file, called 'Main' which starts the execution. This feature was introduced to allow piping between functions and I/O.

```
Main:
    stdin >> SumConstant >> stdout
```

Every Main function declaration starts with a stdin statement, have a number of function piping, and end with a stdout statement.
'stdin >>' tells the program to read from standard input into the pattern match of the first function following (SumConstant).
'>>stdout' tells the program to output to standard output the all of the output patterns of the function preceding it (in the example SumConstant).

5.3 Piping

Piping is the feature that allows a function to output to the input pattern match of another function.

```
SumConstant: {k  : Int = 5;}
[a : Int] -> [a + k];

Negate: {}
[a : Int] -> [-a]

Main:
    stdin >> SumConstant >> Negate >> stdout
```

In this example the output patterns of SumConstant are injected into the input pattern of the function Negate, which in turn outputs to standard output. Walton allows multiple functions to be piped, however, one function cannot pipe to itself.

6 Examples

Example 1 - Read different data types and output twice

```
//Reads two numbers per row. If the second number is true (1) output first number+1.
IfTrueAddPrintTwice: {}
[a : Int, b : Bool] -> if(b){
                        [a + 1];
                        [a + 2];
                       }

//Read from stdin, pipe to IfTrueAddPrintTwice, ouput to stdout two new values.
Main:
    stdin >> IfTrueAddPrintTwice >> stdout

Input:  Output:
1 1     2
2 0     3
3 1     4
        5
```

Example 2 - Boolean orthogonality when piping and in I/O

```
Function1: {k  : Int = 1;}
[a : Int, b : Int] -> if(b > 3){
                        [a, true];
                      } else{
                        [2*a, false];
                      }

Function2: {}
[a : Int, b : Bool] -> if(b){
                           [a^2];
                       } else{
                           [-a+k];
                       }

Main:
        stdin >> Function1 >> Function2 >> stdout

Input:  Output:
1 4     1
2 1     -3
3 5     9
```

7 Type checking & informative error messages.

7.1 Syntax Errors

Syntax errors are reported including line and column number of where the first error was detected. Syntax errors include both the bad use of identifiers, and invalid sequence of Walter source elements (grammar).

7.2 Main and its location in script

For a text file to be considered valid Walter Source code, there must exist at least a Function and the Main function. The latter one must be located at the end of the source file, and make at least one function call from stdin to stdout.

```
[Error] Multiple declarations of Main function
[Error] Main function missing or not placed at the end
```

7.3 Scope errors

In Walter, variables are strictly bounded to a function. Hence an error will be reported if a variable outside a function's scope is invoked. Functions themselves have a scope, which is limited to the source file: an error is returned if this requirement is not met:

```
[Error] Multiple declarations of Main function
[Error] Main function missing or not placed at the end
[Error] No instance of function <fName> in Main function declaration.
[Error] No instance of variable <varName> in function <functionName>
```

7.4 Type Errors in initialization & reassignment

Walter is strongly typed. Hence at any point of the computation it is able to verify if a variable's type is consistent. These include initialization and reassignment of a variable's value.

```
[Error] var2 : Bool could not be assigned to Int
```

7.5 Type Errors in operations

Includes the prevention of adding Int to Bool and vice versa. Ensures maths operations are performed on Ints and Bool operations on Bools. Eg:

```
[Error] Invalid maths operation on type <Type> and type <Type> in function functionName
[Error] Invalid statement in output of functionName
[Error] Condition does not evaluate to type <Type> in function <fName> condition <expression>
```

7.6 Type Errors in piping

It is crucial for the both the input and output pattern to match in type, number and order of the arguments. The type checker is able to detect such inconsistencies and let the user know with

```
[Error] Invalid pipe statement between <sourceFunction> >> <destinationFunction>
```

8 Appendix

Problem1

```
Problem1:
{ var1 : Int = 0; }
[a : Int] -> [var1];
             var1 = a;

Main:
    stdin >> Problem1 >> stdout
```

Problem2

```
Problem2: {}
[a : Int] -> [a, a];

Main:
    stdin >> Problem2 >> stdout
```

Problem3

```
Problem3: {}
[a : Int, b : Int] -> [a + 3*b];

Main:
    stdin >> Problem3 >> stdout
```

Problem4

```
Problem4: { sum : Int = 0; }
[a : Int] -> sum = sum + a;
             [sum];

Main:
    stdin >> Problem4 >> stdout
```

Problem5

```
Problem5:
{
        var1 : Int = 0;
        var2 : Int = 0;
        x : Int = 0;
}
[a : Int] -> x = var1 + var2 + a;
             [x];
             var1 = var2;
             var2 = x;

Main:
    stdin >> Problem5 >> stdout
```

Problem6

```
Problem6: {var1 : Int = 0;}
[a : Int] -> [a,var1];
             var1 = a;
Main:
    stdin >> Problem6 >> stdout
```

Problem7

```
Problem7: {}
[a : Int, b : Int] -> [a-b, a];

Main:
    stdin >> Problem7 >> stdout
```

Problem8

```
Problem8: {var1 : Int = 0;}
[a : Int] -> [a + var1];
             var1 = a;

Main:
    stdin >> Problem8 >> stdout
```

Problem9

```
Problem9:
{
    previous : Int = 0;
    accumulator : Int = 0;
}

[a : Int] -> previous = a + previous + accumulator;
             accumulator = accumulator + a;
             [previous];

Main:
    stdin >> Problem9 >> stdout
```

Problem10

```
Problem10:
{
    var1 : Int = 0;
    var2 : Int = 0;
    x : Bool = true;
}
[a : Int] -> if(x){
                var1 = a + var1;
                [var1];
             } else{
                var2 = a + var2;
                [var2];
             }
             x = !x;

Main:
    stdin >> Problem10 >> stdout
```