Language Reference Manual

Team No. 10
iWAL: intuitive Web Automation Language

TABLE OF CONTENTS

1. Introduction

iWAL is designed to be intuitive and easy to use and understand and will therefore be similar to a scripting language. We, therefore, do not provide an object oriented structure but rather allow users to define functions to perform web automation tasks. We also provide an array of inbuilt functions, which would make the language very useful to those users that do not have a lot of programming experience. The language reference manual specifies the lexical convention decisions and syntax for iWAL as well as the grammar that is used to represent statements, expressions, operators, functions etc. in iWAL.

2. Lexical conventions

The source program is treated as a character stream. The lexical analysis phase of the compiler would then break up this character stream into a sequence of tokens. The tokens can be classified into identifiers, keywords, operators, separators or constants. Apart from these, we have whitespace, which would include one or more of space, tab or newline and comments. Whitespace and comments are ignored by the compiler because they do not classify as tokens.

Below, we explain each of our token classes:

2.1 Identifiers

An identifier uniquely identifies an entity in the program. iWAL specifies that identifiers should begin either with a letter or an underscore and it can then contain a sequence of letters, digits and underscores. Identifiers are case sensitive.

2.2 Keywords

Keywords are reserved words that are used to convey specific information to the compiler. Since these words are reserved, they cannot be used as identifiers.
The keywords that are reserved in iWAL are:

for	def
if	return
else	break
boolean	repeat
void	until
int	import
char	String
double	true
key	false

2.3. Operators

iWAL supports a variety of operators. These range from arithmetic operators to logical operators, equality operators etc. A complete list of iWAL operators is provided below.

Operator	**Purpose**
+	Addition of integer or double values, or concatenation of strings
-	Subtraction of integer or double values
*	Multiplication of of integer or double values
/	Divide an integer or float value by another value
=	Assignment operator
==	Equality comparison
>	Greater than comparison
<	Less than comparison
>=	Greater than or equal to comparison
<=	Less than or equal to comparison
!	Not operator
!=	Not equal to operator
||	OR operator
&&	AND operator

2.4. Separators

Semi-colon (‘;’) is a separator. It indicates the end of a statement.

2.5. Constants

Constants values can belong to any one of the data type that iWAL supports. For example,

- 5 is an integer constant
- a is a character constant
- true and false are boolean constants

- Enter, Space, Shift, Alt, Ctrl, Backspace, etc are key constants

2.6. Comments

Comments in iWAL have the same format as that of Java.

Single line comments can be given as :

```
// All text on this line is a comment
```

Multi-line comments can be given as :

```
/* This comment can go
 on for multiple lines.. */
```

3. Types

iWAL supports a variety of different primitive and a few derived data types. iWAL is a statically typed language. Therefore, the type compatibility will be checked at compile time and any errors related to the data type mismatch would be reported to the user.

3.1 Primitive types

1. int - This type will be similar to the int data type within Java. so, it is a 32-bit signed two's complement integer, which has a minimum value of -2^{31} and a maximum value of $2^{31}-1$

2. double - Similar to the double data type in Java, the double data type in iWAL is a double-precision 64-bit IEEE 754 floating point.

3. char - The char data type is a single 16-bit Unicode character. It has a minimum value of '\u0000' (or 0) and a maximum value of '\uffff' (or 65,535 inclusive).

4. boolean - The boolean data type has two possible values, true and false. It can be used as a simple flag to check for occurrence of various conditions, especially during the execution of loops.

5. key - The key data type can take value of any key on a standard keyboard. Thus, it is useful to simulate pressing of keys in a web automation language such as iWAL.
For example, we can have something as follows to simulate the pressing of a tab key:

```
key k = Tab;
tap(k);
```

3.2 Derived types

1. String - The String is a derived data type. It can be used to hold any string. It is represented as a sequence of characters with double quotes. For example, we can use the String data type to have variables which can store urls or any other information.

String url = “https://courseworks.columbia.edu/welcome/”;

2. Array- Similar to Java, we allow iWAL to have arrays. An array holds a fixed number of values of a single type. This type can be any one of the data types that iWAL supports. The length of an array is established when the array is created. After creation, its length is fixed.

An array of size 50 can be defined as follows:
int myArray [50];

4. Scope

4.1 Lexical Scope

The lexical scope for a declared variable will start at the end of its declaration and persist until its current scope exits. Variable names in the same scope must be unique.

4.2. Global Scope

Variables that are declared globally, outside any function definition will have a global scope. Global scope of a variable will persist throughout the entire program.

4.3. Function Scope

The variables that are declared within a function definition will be available only in that function, that is, the scope of the variable persists only within the function body and not outside of it.

4.4. Linkage Scope

iWAL supports a variety of inbuilt library functions that can be imported and used within programs by the users. These imported functions or variables would have linkage scope and they would be available to be used within the program file, in which the they are imported.

5. Expressions

An *expression* is "a sequence of operators and operands that specifies a computation. Examples of expressions include 2, 4+5, “Hello” etc. Assignments are expressions in iWAL.

5.1 Primary Expressions

A primary expression is the most elementary type of expression. It can be an identifier, a string literal, a constant or a parenthesized expression. Primary expressions can be used to build more complex expressions.

primary-expression:
ID
STRING
NUMBER
‘(‘ expression ‘)’

5.2 Function Calls

The syntax of function call is as follows:

function-expression:
ID ‘(‘ parameter-list ‘)’

parameter-list:
    parameter-declaration-optional
    parameter-list ‘,’ parameter-declaration

A function call consists of function name followed by parentheses containing an empty or a comma-separated list of expressions that are the arguments to the function.
Parameters are passed in by value and evaluated in the order they are specified in the parameter list.

5.3 Operators

5.3.1. Arithmetic

1.Multiplicative Operators

The multiplicative operators are * and /, and they group left-to-right. The following is the syntax for the multiplicative operators:

multiplicative-expression:
    primary-expression
    multiplicative-expression ‘*’ primary-expression
    multiplicative-expression ‘/’ primary-expression

Implicit type-casting is not supported.

Operands of * and / must have arithmetic type (int or double).

The binary * operator indicates multiplication, and its result is the product of the operands.

The binary / operator indicates division of the first operator (dividend) by the second (divisor) and yields the quotient. If the second operand is 0, an error is thrown. Division with both int and/or long operands results in discarding the fractional portion of the result.

2. Additive Operators

The additive operators + and - associate from left to right. The usual arithmetic conversions are performed. The syntax for the additive operators is as follows:

additive-expression:
    multiplicative-expression
    additive-expression '+' multiplicative-expression
    additive-expression '-' multiplicative expression

The result of the + operator is the sum of the operands. The result of the - operator is the difference of the operands.

If the operands are strings, then + represents concatenation.

5.3.2 Relational Operators

The relational operators are left associate. The syntax is as follows:

relational-expression:
    additive-expression
    relational-expression '<' additive-expression
    relational-expression '>' additive-expression
    relational-expression '<=' additive-expression
    relational-expression '>=' additive-expression

If the operands are arithmetic, the < (less), > (greater), <= (less or equal), >= (greater or equal) operators return a boolean value true if the specified relation is true and false if it is false.

5.3.3 Equality Operators

The equality operators == and != function as the relational operators do, but they have lower precedence. They are used as follows:

equality-expression:
relational-expression
equality-expression ‘==’ relational-expression
equality-expression ‘!=’ relational-expression

The operands are arithmetic (int or double), the == (equal to) and != (not equal to) operators return a boolean
value true if the specified relation is true and false if it is false.

5.3.4 Logical Operators

The logical operators && and || are used when evaluating two expressions to obtain a single relational result. The operator && corresponds to the boolean logical operation AND and operator || corresponds to boolean logical operation OR

1. Logical AND Operator (&&)

The && operator groups left-to-right.

logical-AND-expression:
equality-expression
logical-AND-expression ‘&&’ equality-expression

If both operands evaluate to true or a boolean equivalent, the logical AND expression yields a boolean with value true; otherwise, it yields false.

2. Logical OR Operator (||)

The || operator groups left-to-right.

logical-OR-expression:
logical-AND-expression
logical-OR-expression ‘||’ logical-AND-expression

If either of the two operands evaluate to true or a boolean equivalent, the logical OR expression yields a boolean with value true; otherwise, it yields false.

Note: iWAL supports *short-circuiting*. It means that when using the conditional AND and OR operators (&& and ||), iWAL does not evaluate the second operand unless it is necessary to resolve the result. To be more explicit, the right operand (operand2) is not evaluated in the cases:

false && operand2
(The final result is false)
true || operand2
(The final result is true)

3. Logical NOT Operator (!)

The operator ! is the iWAL operator for the boolean operation NOT. It has only one operand, to its right, and inverts it, producing false if its operand is true, and true if its operand is false.
The operand must be of type boolean.

5.3.4 Assignment Operator

An assignment operator stores the value of the right expression into the left expression.

assignment-expression:
ID '=' assignment-expression
logical-OR-expression

The left operand must be an identifier. iWAL is dynamically typed, so the type of the left operand is same as that of the right operand.

5.3.5 Operator Precedence Table

Precedence	Operator	Function	Associativity
1	()	*Function call*	Left-to-right
2	!	*Logical NOT*	Right-to-left
3	* /	*Multiplication, Division*	Left-to-right
4	+ −	*Addition, Subtraction*	Left-to-right
5	< <= > >=	*For relational operators < and ≤ respectively* *For relational operators > and ≥ respectively*	Left-to-right
6	== !=	*For relational = and ≠ respectively*	Left-to-right
7	&&	*Logical AND*	Left-to-right
8	||	*Logical OR*	Left-to-right

5.4 Declarations

Declarations tell us the structure of identifiers and how they should be understood. Function definitions specify the structure of how functions are defined in a program.

external-declaration:
    function-definition
    statement

function-definition:
    type ‘def’ ID ‘(‘ parameter-list ‘)’ ‘{‘ statement-list ‘}’
    type ‘def’ ID ‘(‘ parameter-list ‘)’ ‘{‘ ‘}’

5.5 Statements

Statements perform a certain task-either a declaration, definition, compounded blocks or control flow statements.

statement:
    declaration-statement
    expression-statement
    compound-statement
    selection-statement
    iteration-statement
    return-statement
    break-statement

5.5.1 Declaration Statement

Used for declaring a function, an identifier or assigning a value to some variable.

parameter-declaration:
    type ID
    type ID ‘=’ assignment-expression

5.5.2 Expression Statement

A statement which comprises of an expression in the grammar.

expression-statement:
    ‘;’
    expression ‘;’

5.5.3 Compound Statement

A sequence of statements grouped together in curly braces to indicate a set of statements.

compound-statement:
    '{' '}'
    '{' statement-list '}'

5.5.4 Selection Statement

When one of two possible statements is executed based on the value that an expression takes, it is represented in the form of a selection statement where 'if' and 'else' are keywords.

selection-statement:
    'if' '(' expression ')' statement
    'if' '(' expression ')' statement 'else' statement

5.5.5 Iteration Statements

Used to model a loop where a statement is repeatedly executed based on the value that an expression takes. There are two kinds of iteration statements in our grammar corresponding to two kinds of loops-repeat and until. The repeat loop repeats for a certain number of times(as specified in the expression) and the until loop checks a condition and repeats until it's true.

iteration-statement:
    'repeat' '(' expression ')' statement
    'until' '(' expression ')' statement

5.5.6 Break Statements

A break statement comprises of the keyword 'break' followed by a semi-colon and is used to indicate exit from the closest looping structure.

break-statement:
    'break' ';'

5.5.7 Return statements

Return statements are usually placed at the end of a function body to indicate exit from the function. Optionally they may also return an expression to the calling function.

return-statement:
    'return' ';'
    'return' expression ';'

6. Grammar

The full grammar for iWAL is shown below.

translation-unit:
    external-declaration
    translation-unit external-declaration

external-declaration:
    function-definition
    statement

function-definition:
    type ‘def’ ID ‘(‘ parameter-list ‘)’ ‘{‘ statement-list ‘}’
    type ‘def’ ID ‘(‘ parameter-list ‘)’ ‘{‘ ‘}’

statement:
    declaration-statement
    expression-statement
    compound-statement
    selection-statement
    iteration-statement
    return-statement
    break-statement

declaration-statement:
    type ID ‘;’
    type ID ‘=’ assignment-expression ‘;’

compound-statement:
    ‘{‘ ‘}’
    ‘{‘ statement-list ‘}’

statement-list:
    statement
    statement-list statement

expression-statement:
    ‘;’
    expression ‘;’

expression:
    assignment-expression

assignment-expression:
    ID ‘=’ assignment-expression
    logical-OR-expression

logical-OR-expression:
    logical-AND-expression
    logical-OR-expression ‘||’ logical-AND-expression

logical-AND-expression:
    equality-expression
    logical-AND-expression ‘&&’ 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:
    primary-expression
    multiplicative-expression ‘*’ primary-expression
    multiplicative-expression ‘/’ primary-expression

parameter-declaration:
    type ID
    type ID ‘=’ assignment-expression

type:
    int
    double
    char

key
String

primary-expression:
ID
STRING
NUMBER
‘(‘ expression ‘)’

function-expression:
ID ‘(‘ parameter-list ‘)’

parameter-list:
parameter-declaration-optional
parameter-list ‘,’ parameter-declaration

iteration-statement:
‘repeat’ ‘(’ expression ‘)’ statement
‘until’ ‘(’ expression ‘)’ statement

selection-statement:
‘if’ ‘(‘ expression ‘)’ statement
‘if’ ‘(‘ expression ‘)’ statement ‘else’ statement

return-statement:
‘return’ ‘;’
‘return’ expression ‘;’

break-statement:
‘break’ ‘;’