Manchester Metropolitan University

Coffee Language Specification

Contents

Coffee Grammar (Backus-Naur Form)

$program$ -> { <imports> | <global> | <method> | <block> }* (see footnote[1])

<comment> -> { // .*? <\n> | /* .*? */ } -> skip (see footnote[2])

<whitespace> -> { <\n> | <\r> | <\f> | <\t> | <\n> } -> *skip* (see footnote[3])

<imports> -> **import** <id> { , <id> }* ;

<global> -> <var_def>

<id> -> <alpha> <alphanumeric>*

<alpha> -> **a** | **b** | **c** | ... | **z** | **A** | **B** | **C** | ... | **Z** | **_**

<alphanumeric> -> <alpha> | <number>

<number> -> **0** | **1** | **2** | ... | **9**

<local> -> <var_def>

<var_def> -> <data_type> <variable> <assign>? { , <variable> <assign>? }*

<assign> -> = <expr>

<data_type> -> **int** | **float** | **bool**

<variable> -> <id> | <id> **[** <integer> **]**

<integer> -> <number>+

<method> -> <return_type> <id> **(** { <data_type> <id> { , <data_type> <id> }* }? **)** { <block> | <expr> }

<return_type> -> **void** | <data_type>

<block> -> **{** { <var_ def> | <block> }* **}**
| <statement>

<statement> -> <method_call> ;
| <location> <assign_op> <expr> ;
| **if (** <expr> **)** <block> { **else** <block> }?
| **for (** <loop_var> **in** { <id> | <limit> } **)** <block>
| **while (** <expr> **)** <block>
| **return** <expr>? ;
| **break** ;
| **continue** ;
| ;

<loop_var> -> <id>

<method_call> -> <id> **(** { <expr> { , <expr> }* }? **)**

<expr> -> (<expr>)
| (<data_type>) <expr>
| - <expr>
| ! <expr>
| <expr> <op> <expr>
| <expr> ? <expr> : <expr>
| <literal>
| <location>
| <method_call>

<op> -> <assign_op> | <arithmetic_op> | <relation_op> | <equal_op> | <condition_op>

<assign_op> -> =

<arithmetic_op> -> * | / | % | + | -

<relation_op> -> > | >= | < | <=

<equal_op> -> == | !=

<condition_op> -> && | ||

<literal> -> <integer> | <float> | <bool> | <string> | <char>

<float> -> <number>+ . <number>* | <number>* . <number>+

<bool> -> true | false

<string> -> " <character>* "

<char> -> ' <character>? '

<character> -> a valid C character (see footnote[4])

<location> -> <id> | <id> [<expr>]

<limit> -> [<low>? : <high>? { : <step> }?]

<low> -> <expr>

<high> -> <expr>

<step> -> <expr>

[1]tokens of the form $name$ represent the root node of the grammar.

[2]the wildcard sequence is denoted . (i.e. dot). A repeating wildcard (i.e. star) should be followed by a question mark – ANTLR syntax for "match the shortest string" – it prevents the parser from matching the whole input file with the wildcard. *skip* indicates that text which matches the expression to its left should be ignored (i.e. not produce a token).

[3]tokens of the form <\n>, <\f>, etc, represent their respective character codes used in text formatting. <space> represents the space ascii character.

[4]a valid <character> is any single character which is not a newline, tab, form feed, double or single quote. The following double characters (i.e. not their respective single character codes) are allowed: \n \t \f \r \" \'

Operator Precedence

Operator	Description
(<data_type>)	type cast
-	unary minus
!	logical not
* / %	multiply, divide, modulo
+ -	add, subtract
< <= > >=	relational
== !=	equality
&&	conditional and
||	conditional or

Type Precedence

Data Type	Description
float	double precision float
int	infinite precision (memory limits) integer
bool	Boolean

Scoping

1. All methods including main have their own scope

2. For loops have their own scope

3. All code blocks have their own scope

Code Interpreter

Code instructions are executed using native Python 3 directly from the parse tree or AST / computation graph.

Semantic Rules

1. Variables must be declared before use

2. Variable declarations must have unique identifiers in a scope

3. Method declarations (including imported methods) must have unique identifiers in a scope

4. Method calls must refer to a declared method with an identical signature (return type, and number and type of parameters)

5. Method calls referring to imported methods must produce a warning to check the argument and return types match that of the imported method

6. Void methods cannot return an expression

7. Non-void methods must return an expression

8. The main method does not require a return statement, but if it has one, it must be of type **int**

9. Branch statements (**if-else**) containing return statements do not qualify a method as having a return statement and a warning must be issued unless they appear in both the main branch and the else branch

10. Loops containing return statements do not qualify a method as having a return statement and a warning must be issued

11. The expression in a branch statement must have type **bool**

12. The expression in a while loop must have type **bool**

13. The <low> and <high> expressions in a limit must have type **int**

14. Arrays must be declared with size greater than 0

15. The <id> in a **for**-loop must reference a declared array variable

16. Arrays cannot be assigned during declaration

17. Char expressions must be coerced to **int**

18. The expression in an assignment must have type **bool**, **int** or **float**

19. Locations of the from <id> **[** <expr> **]** must refer to a declared array variable

20. In a location, array indices must have type **int**

21. The expression in unary minus operation must have type **int** or **float**

22. The expression in logical not operation must have type **bool**

23. The expression(s) in an arithmetic operation must have type int or **float**

24. The expression(s) in a logical operation must have type **bool**

25. Singular expressions in a block provide a valid return value for a method without requiring the **return** keyword

26. Methods returning **void** cannot be used in an expression

27. Break and continue statements must be contained within the body of a loop.