Cobalt Specification

Version 0.1.0-alpha

Alexandre Charoy

28.10.2019

1. Introduction

1.1. Target Platform

Cobalt is not designed for a specific target platform.

1.2. Grammars

In order to define the lexical and syntactic structure of a Cobalt program, context-free grammar definitions are given throughout this specification. Those definitions are written in the Extended Backus-Naur Form. You can find a full grammar definition in Appendix A.

2. Lexical Structure

2.1. Unicode

Cobalt programs are written using the Unicode character set. All keywords, separators, operators, identifiers and primtive literals are formed from ASCII characters. Comments can contain non-ASCII Unicode characters.

2.2. Line Termination

Line terminators can be the ASCII `LF` character, the ASCII `CR` character, or the ASCII `CR` character followed by the ASCII `LF` character. The `CRLF` combination is therefore considered being one line terminator.

2.3. Whitespace

Space and horizontal tab (ASCII characters `SP` and `HT`) are considered whitespace and are of no consequence, except as delimiters to certain keywords.

2.4. Case Sensitivity

Cobalt is case-sensitive in all aspects.

2.5. Comments

Cobalt supports end of line comments: All text from the ASCII characters `//` to the end of the line is ignored. This means that an end of line comment needs to be terminated by any of the allowed line terminators (see section 2.2).

2.6. Identifiers

Identifiers can contain any letter (a-z and A-Z / ASCII codes 97-122 and 65-90) or digit (0-9 / ASCII codes 48 to 57), as well as underscores (`_` / ASCII code 95), but have to start with a letter. See listing 2.1 for a formal grammar definition for identifiers.

```
identifier = letter , { letter | digit | "_" } ;
letter = "A" | "B" | "C" | "D" | "E" | "F" | "G"
       | "H" | "I" | "J" | "K" | "L" | "M" | "N"
       | "O" | "P" | "Q" | "R" | "S" | "T" | "U"
       | "V" | "W" | "X" | "Y" | "Z" | "a" | "b"
       | "c" | "d" | "e" | "f" | "g" | "h" | "i"
       | "j" | "k" | "l" | "m" | "n" | "o" | "p"
       | "q" | "r" | "s" | "t" | "u" | "v" | "w"
       | "x" | "y" | "z" ;
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
```

Listing 2.1: Identifier Grammar

2.7. Keywords

The following multi-letter words are reserved keywords in Cobalt: `bool`, `int`, `float`, `true`, `false`, `stdin`, `stdout`, `def`

3. Primitive Types

3.1. The Boolean Type

The boolean type is specified by using the `bool` keyword. Boolean values can be `true` or `false`.

3.2. Number Types

Cobalt has integer and floating point types. The minus sign for negative numbers is written using the tilde character (`~`).

3.2.1. The Integer Type

The integer type is specified by using the `int` keyword. Integer values are 32-bit signed two's-complement integers: `-2147483648` to `2147483647`, inclusive.

3.2.2. The Float Type

The floating point number type associated with single-precision 32-bit values is specified using the `float` keyword.

3.3. Literals

Listing 3.1 gives a grammar of all possible literal values. Note that `.0` and `0.` are not valid floating point literals according to this definition, as at least one digit is expected both before and after the dot.

```
1 literal = integer | float | "true" | "false" ;
2 integer = ["~"] , digit , { digit } ;
3 float = ["~"] , digit , { digit } , "." , digit , { digit };
```

Listing 3.1: Literals Grammar

4. Operators

Cobalt has unary and binary operators. Cobalt expressions are written in infix notation, meaning that binary operators are placed between operands.

4.1. Logical Operators

Logical operators operate on boolean operands and evaluate to a boolean value.

4.1.1. Unary Logical Operators

Cobalt has the following unary logical operators:

Operator	Operand	Evaluates to	Description
!	bool	bool	Negates a boolean value.

4.1.2. Binary Logical Operators

Cobalt has the following binary logical operators:

Operator	Operands	Evaluates to	Description
&	bool*bool	bool	Compares two boolean values and evaluates to true if both are true.
|	bool*bool	bool	Compares two boolean values and evaluates to true if at least one of them is true.

4.2. Comparison Operators

Comparison operators operate on number types, and evaluate to boolean values. Cobalt has the following comparison operators:

Operator	Operands	Evaluates to	Description
==	number*number bool*bool	bool	Compares two numbers or booleans and evaluates to true if they are equal.
!=	number*number bool*bool	bool	Compares two numbers or booleans and evaluates to true if they are not equal.
<	number*number	bool	Compares two numbers and evaluates to true if the left-side operand is strictly smaller than the right-side operand.
<=	number*number	bool	Compares two numbers and evaluates to true if the left-side operand is smaller or equal to the right-side operand.
>	number*number	bool	Compares two numbers and evaluates to true if the left-side operand is strictly greater than the right-side operand.
>=	number*number	bool	Compares two numbers and evaluates to true if the left-side operand is greater or equal to the right-side operand.

4.3. Arithmetic Operators

Arithmetic operators operate on and evaluate to number values. Cobalt has the following comparison operators:

Operator	Operands	Evaluates to	Description
+, -, *	int*int	int	An addition, substraction or multiplication of two integers evaluates to an integer.
+, -, *	number*float float*number	float	With one or more floating point numbers, an addition, substraction or multiplication evaluates to a floating point number.
/	number*number	float	Divisions always evaluate to floating point numbers.

5. Expressions

Expressions may contain literal values, variables, operators and parentheses, and evaluate to a single value and type. The listing 5.1 contains a grammatical definition of expressions.

```
1 expression = ["!"] , term { ("&" | "|") , ["!"] , term } ;
2 term = sum { comparison operator , sum } ;
3 sum = product { ("+" | "-") , product } ;
4 product = factor { ("*" | "/") , factor } ;
5 factor = literal | identifier | "(" expression ")" ;
```

Listing 5.1: Expressions Grammar

5.1. Evaluation Order

When an expression of the form `operand operator operand` is evaluated, the left operand is evaluated first, then the right one, and finally the expression as a whole is evaluated. The type that the expression evaluates to depends on the operator, see section 4.

5.2. Operator Precedence

The following table lists operators in order of highest to lowest precedence:

Operator Type	**Operators**
negative number sign	`~`
multiplicative arithmetic	`*`, `/`
additive arithmetic	`+`, `-`
comparison	`==`, `!=`, `>`, `>=`, `<`, `<=`
unary logical	`!`
binary logical	`&`, `|`

6. Variables

6.1. Declaration

Variables are declared using the `def` definition keyword, followed by an identifier, the `:` type setting operator and optionally a type keyword. This can optionally be followed by the `=` assignement operator and an expression. The statement is then closed wiht a semicolon. The matching grammar definition can be seen in listing 6.1.

```
1 variable declaration = "def" , identifier , ":" ,
2         (type [, "=" , expression] | "=" , expression) , ";" ;
```

Listing 6.1: Variable Declaration Grammar

6.1.1. Variable Type

A variable always has an associated type, which is determined at declaration. The variable type can be explicitely declared with a type keyword, or inferred from an assigned expression. In the case of inference, the type of the expression is first determined, and then assigned to the variable. For more information on expression type, see section 5. If both an explicit type keyword and an expression are used when declaring a variable, the type of the expression needs to match the explicitely declared type of the variable.

6.2. Assignment

An already declared variable can be assigned a value. If the variable already had a value, it is replaced. An assignment starts with the variable name, followed by the `:=` assignment operator and an expression. The matching grammar definition can be seen in listing 6.2.

```
1 assignment = identifier , ":=" , expression , ";" ;
```

Listing 6.2: Variable Assignment Grammar

6.3. Scope

When a variable is declared, it is declared inside the local scope. It will not be accessible from superordinate scopes. If a variable of the same name is declared in a superordinate scope, the local variable will overshadow the variable of the superordinate scope. Two variables of the same name can therefore exist in different hierarchical scopes, and hold

different values. A variable can't be declared, if a variable of the same name has already been declared in the local scope.

6.3.1. Program Scope

A Cobalt program has a `program scope` as highest scope. It can have any number of nested subordinate scopes. Any variable declared is declared in a scope, which is the program scope if is not declared in a subordinate scope.

6.4. Usage

6.4.1. Lookup

A variable needs to be declared in the local or a superordinate scope in order to be used. When a variable is used, it will first be looked up in the local scope. If no variable of that name exists in the checked scope, it will be looked up in the next superordinate scope, until the program scope is reached. If it is not found at all, compilation should fail.

6.4.2. Value

A variable needs to be assigned a value in order to be used. When attempting to use a variable that has no assigned value, compilation should fail.

7. Input/Output

Cobalt only supports simple standard input/output of single values through the `stdin` and `stdout` keywords. Depending on the platform being targeted, a compiler needs to appropriately translate these.

7.1. Input

An input statement consists of the `stdin` keyword followed by a variable identifier. It is attempted to read a value of the type of that variable from standard input. On success, the varible is set to that value. On fail, program execution should abort. The matching grammar definition can be seen in listing 7.1.

```
1 input = "stdin" , identifier , ";" ;
```

Listing 7.1: Input Statement Grammar

7.2. Output

An output statement consists of the `stdout` keyword followed by an expression. The value the expression evaluates to is written to standard output. The matching grammar definition can be seen in listing 7.2.

```
1 output = "stdout" , expression , ";" ;
```

Listing 7.2: Output Statement Grammar

Appendix Appendix A.

Full Grammar

A full grammar definition of Cobalt 0.1.0 can be found in the following listing Appendix A.1.

```
program = { statement } ;

statement = variable declaration | assignment | input | output ;

variable declaration = "def" , identifier , ":" ,
        (type [, "=" , expression] | "=" , expression) , ";" ;
assignment = identifier , ":=" , expression , ";" ;
input = "stdin" , identifier , ";" ;
output = "stdout" , expression , ";" ;

expression = ["!"] , term { ("&" | "|") , ["!"] , term } ;
term = sum { comparison operator , sum } ;
sum = product { ("+" | "-") , product } ;
product = factor { ("*" | "/") , factor } ;
factor = literal | identifier | "(" expression ")" ;
comparison operator = "==" | "!=" | ">" | "<" | ">=" | "<=";

identifier = letter , { letter | digit | "_" } ;
type = "int" | "bool" ;

literal = integer | float | "true" | "false" ;
integer = ["~"] , digit , { digit } ;
float = ["~"] , digit , { digit } , "." , digit , { digit };

letter = "A" | "B" | "C" | "D" | "E" | "F" | "G"
       | "H" | "I" | "J" | "K" | "L" | "M" | "N"
       | "O" | "P" | "Q" | "R" | "S" | "T" | "U"
       | "V" | "W" | "X" | "Y" | "Z" | "a" | "b"
       | "c" | "d" | "e" | "f" | "g" | "h" | "i"
       | "j" | "k" | "l" | "m" | "n" | "o" | "p"
       | "q" | "r" | "s" | "t" | "u" | "v" | "w"
       | "x" | "y" | "z" ;
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
```

Listing Appendix A.1: Literals Grammar