

**YARRVA  
Module VIII Final Project**

Kevin Hetterscheid (s1490443)  
Jeroen Klein Brinke (s1472348)

Table of Contents

**1 – Summary   
*This is a short summary of the most important and honorable features of Yarrva.***

**1.1 Supported data types and operations**Yarrva supports the basic datatypes of Integers (*Doubloon*) and Booleans (*Orders*). You can use the basic arithmetic operations (*adding, subtracting, multiplying* and *divisions*) on Integers and you can compare two Booleans.

**1.2 Loops**Yarrva supports two ways of looping: a while-loop (*whirlpool*) and a for-loop (*navigate*). The while-loop checks one or more conditions and loops until the result of those conditions becomes false (*Nay*). The for-loop is a way to use a counter and index in your loop and loops from a given variable up to and/or including a given value.

**1.3 Functions**Yarrva supports the use of functions. In fact, Yarrva always requires a main function (*Flagship*). Above the main function you are able to declare an arbitrary number of other functions and you can call these functions from the main function or another function. Yarrva uses a function hierarchy in which a function can only call a function which has been previously declared. Yarrva also supports nested functions. Yarrva does also support nested functions. These functions follow the same hierarchy as the regular functions, but inside the scope of the function they are declared. Yarrva does not support recursive functions.

**1.4 Global variables**Yarrva supports the use of global variables. This means that you can declare an arbitrary number of variables before you declare the functions and the main function and you can use these in all scopes.

**1.5 Comments**Yarrva does support line comments (*\*\**) and block comments (*>> <<*). Block comments includes multiple line comments, but also mid-line comments.

**1.6 Increment and decrement**Yarrva has built-in ways to increment and decrement a value by one (*gift* and *plunder*). These can be used in the for-loop, but also in the declaration and assignment of an integer.

**1.7 Warnings and errors**Yarrva throws warnings and errors. The errors Yarrva can throw errors found in the tokenizer, parser and type checker. Yarrva also throws a warning when a variable is declared but never used.

**2 – Problems and Solutions**

**2.1**

**3 – Detailed language description**

**3.1 Yarrva code layout**Yarrva uses a very strict coding layout in order to be properly parsed. This is to ensure that (nested) functions and global variables work. There is a strong hierarchy: functions and global integers can only be used by other functions that are declared at a later time.  
 In *Exampe1* you can find the pseudocode for a typical Yarrva file and in *Example2* you can see *Example1* in working Yarrva code. The Yarrva keywords and syntax seen in *Example2* will be explained starting *Section 3.2*.

1) A Program {

2) A global Integer a = 5

3) A global Boolean b = True

4)

5) Function add(Integer x, Integer y) {

6) return x + y

7) }

8)

9) Function subtract(Integer x, Integer y) {

10) return x – y

11) }

12)

13) Main function() {

14) If (b) {

15) Print add(a, 5)

16) }

17) }

18) }

***Example1: Pseudocode showing the general layout of Yarrva code***

1. fleet Fleet {
2. doubloon a be 1, Arrr!

3) order b be Aye, Arrr!

4)

5) doubloonShip add(doubloon x, doubloon y) {

6) doubloon temp be x + y, Arrr!

7) avast temp, Arrr!

8) }

9)

10) doubloonShip sub(doubloon x, doubloon y) {

11) doubloon temp be x - y, Arrr!

12) avast temp, Arrr!

13) }

14)

15) flagship() {

16) doubloon five be 5, Arrr!

18) parley(b) {

19) parrot (add(a, five)), Arrr!

20) }

21) }

22)}

***Example2: Actual Yarrva code of Example1***

**3.2 Datatypes**In this section we will talk about the datatypes that Yarrva supports, namely the integer and the boolean.  
**3.2.1 Integers**An integer in Yarrva is called “doubloon”. The initialization of an integer in Yarrva can be seen in *Example3*. You can use the following operations on integers: addition (+), subtracting (-), multiplying (\*), dividing (/) and module (%).

1. doubloon a be 5, Arrr! \*\* Integer declaration
2. order b be Aye, Arrr! \*\* Boolean decleration

***Example3: An integer and boolean declaration in Yarrva.***

**3.2.2 Booleans**A boolean in Yarrva is called “order”. The initialization of a boolean in Yarrva can be seen in *Example3*. The value for True in Yarrva is *Aye*, and the value for False in Yarrva is *Nay*.

**3.2.3 Integers and Booleans: grammar and target code**The declaration and assignment of both integers and booleans looks like the following in the grammar:  
 43. Assign 🡪 Opt [Var], idf, equalsKey, Expr  
The target code generated for an assignment/declaration is Sprill code and is generated in the following way: it first stores the given value (integer) or 1 (Aye) or 0 (Nay) (boolean) in a register, after which it will be stored in memory using Sprockell’s Store.

1) booty string be “Yo ho ho, me hearties!”, Arrr!

2)

3) doubloon[] intTreasure be [1,2,3,4], Arrr! \*\* Integer array declaration

4) order[] ordTreasure be [Aye, Aye, Nay], Arrr! \*\*Boolean array declaration

5) booty[] strTreasure be [“Yo”,”Ho”,”Ho”], Arrr! \*\*String array declaration

***Example4: String and Array declarations, what it would have looked like.***

**3.2.4 Almost completed types**Yarrva has two data types which were almost implemented, but due to time constraints aren’t made fully functional. These two datatypes are closely related: strings and arrays. A string in Yarrva would be called “booty” and an array in Yarrva is known as a “treasure”, though it is initialized differently. Both strings and arrays are tokenized, parsed and scope/type checked correctly, but the converter cannot convert it to Spril code yet. The array datatype would be initialized with a type and would only be able to contain the same types. An example of this and a string declaration are shown in *Example4*.

**3.3 Features**In this section we will talk about all the features that are useful in Yarrva. This subsection will not include the function declaration, these will be covered in section ***3.4 Functions***. Comments are also covered in another section, namely in section ***3.5 Comments***. The remaining keywords are covered in section ***3.6 Other keywords***.

**3.3.1 If(-else) statement**The if(-else) statement in Yarrva has two variants. It is possible to have an if block without an else block, but it is also possible to have an if block with an else block. The if-statement checks a condition, if this condition evaluates to true, it will execute in the if-block – if it evaluates to false, it will continue at the end of the if-block. The if-else-statement works the same, except for when it evaluates to false. If the if-else-statement evaluates to false, it will execute the else-block before continuing with the rest of the code.  
 In the grammar, an if-statement looks like the following:   
 4. Stat 🡪 IfExprKey, lpar, BoolExpr, rpar, Block, Opt [elseKey, Block]   
where the IfExprKey is parley, the BoolExpr holds a boolean value and the Block can be anything. You can also see in the grammar that there can be an optional elseKey (heave) with its own Block. In *Example5*, we have created a simple Yarrva program, containing a main function with an if-statement, which has a nested if-else-statement.

1) Fleet IfStatements {

2)

3) flagship() {

4)

5) order outerIf = Aye, Arrr!

6) order innerIf = Nay, Arrr!

7)

8) parley(outerIf) { \*\* *Outer If has no else-block*

9) \*\* *Do something here*

10) parley(innerIf) { \*\* *Inner if has an else-block*

11) \*\* *Will not enter here*  
12) } heave { \*\* *Else-block*

13) \*\* *Do something here*

14) }

15) }

16) }

***Example5: Example of two if-statements, one with and one without an else block.***

The target code generated for an if(-else) statement is Sprill code and is generated in the following way: it first evaluates the bool expression by either comparing (using Sprockell’s Compare) the two given values (in case of a comparison) or compare the value to 1 (in case of a Aye or Nay); it will then use Sprockell’s Branch to either execute the converted if-block, or it will jump to the end of the converted if-block. In case of an if-else block, the procedure is the same, except it adds two of Sprockell’s Jumps: if it evaluates to true, it will ignore the first jump, execute the converted if-block, then use the second jump to skip the converted else-block. If it evaluates to false, it will use the first jump to skip the converted if-block and will execute the converted else-block.

**3.3.2 For loop**The for-loop in Yarrva contains of three parts separated by points: a starting value, a condition that must hold and an increase/decrease function. The for-loop is basically used to run from one value to another while providing a counter.   
 In the grammar, a for-loop looks like the following:  
 7. Stat 🡪 forKey, lpar, Assign, point, BoolExpr, point, Expr, rpar, Block  
where the forKey is navigate, Assign is assigning a new value to an existing variable or declaring a new variable, BoolExpr holds a boolean value, the Expr holds either an expression to decrease or increase the value or a standard function (gift, plunder – **3.6.6 Gift and Plunder**) and a Block, which is the code that will be executed within the for loop. In *Example6*, we have created working Yarrva code that walks from 0 to 5 and adds 1 to val every loop. This means that it basically adds 5 to 2, which corresponds to the output: 7.

1) fleet forLoopIncrease {

2)

3) flagship() {

5) doubloon val be 2, Arrr! *\*\* set val to 2*

6) navigate(doubloon i be 0. i be below 5. gift i) { *\*\* 0 -> 5*

7) parrot i, Arrr! *\*\* print current i*

8) gift val, Arrr! *\*\* val = val + 1*

9) }

10) parrot val, Arrr! *\*\* print val (=7)*

11) }

12)}

***Example6: Add 5 to 2 using a for loop.***

The target code generated for an if(-else) statement is Sprill code and is generated in the following way: it will first assign or declare the variable, then it will push the Program Counter (PC) to a register when it enters the for loop (using Compute Add PC Zero aRegister), after which it will evaluate the bool expression, much like the if-statement. It will then branch using Sprockell’s Branch: If it is true, execute the code; if it is false, jump to the end of the loop and continue with the rest. After the block in the for loop has been executed, it will decrease or increase the counter of the for loop. It will then pop the beginning of the for-loop (right before the boolean evaluation) of the stack and jump there to repeat the process.

**3.3.3 While loop**The while-loop in Yarrva contains of a single part: a boolean condition, which can either be a simple Aye or Nay, or a comparison between integers (for example, i be below 8). The while-loop is basically used to loop over a piece of code as long as the while-condition holds.   
In the grammar, a for-loop looks like the following:  
 8. Stat -> whileKey, lpar, BoolExpr, rpar, Block  
where the whileKey is whirlpool, BoolExpr holds a boolean value and a Block, which is the code that will be executed within the for loop. In *Example7*, we have created working Yarrva code that prints all integers between 0 and 8 using gift (**3.6 Other keywords**).   
 The target code generated is Sprill code and is generated in the following way: it will push the Program Counter (PC) to a register when it enters the for loop (using Compute Add PC Zero aRegister), after which it will evaluate the bool expression, much like the if-statement and for-statement. It will then branch using Sprockell’s Branch: If it is true, execute the code in the black; if it is false, jump to the end of the loop and continue with the rest. After the code inside the block has been executed, it will pop the return address from the stack, jump back there and evaluated the condition again.

1) fleet whilePrintNumbers {

2)

3) flagship() {

4) doubloon i be 0, Arrr! *\*\* Set i to 0*

5) whirlpool(i be below 8) { *\*\* while: i <= 8*

6) parrot i, Arrr! *\*\* print i*

7) gift i, Arrr! *\*\* i = i + 1*

8) }

9) }

10)}

***Example7: Print some numbers using a while loop***

**3.4 Functions**Yarrva uses a function hierarchy in which a function can only call a function which has been previously declared. Yarrva also supports nested functions. Yarrva does also support nested functions. These functions follow the same hierarchy as the regular functions, but inside the scope of the function they are declared. Yarrva does not support recursive functions.   
 Yarrva has two kinds of functions: doubloonShip(…) and orderShip(…). The doubloonShip returns either an integer (doubloon) or nothing (void) and the orderShip returns either a boolean (order) or nothing (void). To return a value, you use the keyword avast (**3.6 Other Keywords**).

fleet showDoubloonFunc {

doubloonShip addDiffNumbers(doubloon x, doubloon y) {

parley(x be y) { \*\* Check if x == y

avast 1, Arrr! \*\* return 1 if x==y

} heave {

doubloon sum be (x+y), Arrr!

avast sum, Arrr! \*\* return the sum

}

}

flagship() {

doubloon x be 7, Arrr!

doubloon y be 1, Arrr!

parrot addDiffNumbers(x, y), Arrr!

doubloon a be 2, Arrr!

doubloon b be 2, Arrr!

parrot addDiffNumbers(a, b), Arrr!

}

}

***Example8: Add two numbers only if they are different, if they are not different, the function returns 1.***

**3.4.1 doubloonShip**As mentioned before, the doubloonShip must return either an integer or nothing at all. You can, however, give a doubloonShip a boolean value. In *Example8*, two numbers will only be added if they are not the same, else the function will return 1.

**3.4.2 orderShip**This was not implemented due to time constraints.

**3.4.3 Nested functions**Yarrva does support nested functions. However, as mentioned before, they have the same hierarchy as all functions, but inside the scope of the function in which they are declared. In *Example9*, we have created a function that adds two numbers and a constant using a nested function.

1) fleet showNestedFunction {

2)

3) doubloonShip upNumber(doubloon x, doubloon y) {

4) doubloonShip addThree(doubloon c) {

5) doubloon temp be c + 3, Arrr!

6) avast temp, Arrr!

7) }

8) doubloon result be x + y, Arrr!

9) avast addThree(result), Arrr!

10) }

11)

12)

13) flagship() {

14) doubloon x be 3, Arrr!

15) doubloon y be 2, Arrr!

16) parrot upNumber(x, y), Arrr!

17) }

18) }

***Example9: Add x and y first, then add 3 after using a nested function***

**3.4.4 Functions: grammar and target code**The declaration of a function looks like the following in the grammar:   
 11. Stat 🡪 Function, idf, lpar, FValues, rpar, Block   
As can be seen, a function requires an identifier (name) and an arbitrary number of expressions (like doubloon d or order o).   
The call of a function looks like the following in the grammar:   
 44. Func 🡪 idf, lpar, Opt [Expr, Rep0 [NoCat comma, Expr]], rpar  
As can be seen, a function call can receive an integer, a boolean or an identifier. However, due to time constraints and call-by-reference, a direct input of an integer or boolean is not supported. The FValues contain the arguments.  
 The target code generated is Sprill code and has two different parts: one part for the generation and one part for the usage. The generation is as follows: it will just build the code as usual, but it will store the Program Counter (PC) in memory and put a jump right after it. The jump is there that so when the code is executed, it will skip the function until it is actually called in the main or from another function. The usage of a function is as follows: it will compute the address of where to return (right after the function) and push that on to the stack, after which it will push the function arguments on the stack, load the function address from memory and then jump to the function to execute it.

**3.5 Comments**Yarrva supports three different kind of comments. The supported comments are: line comments (which comments the remainder of a line), in-line comments (comment part of a line) and block comments (comment multiple lines). In-line comments and block-comments share the same syntax. Comments are left out by the tokenizer and thus not a part of the grammar.

1) fleet someProgram {

2) doubloon d be 5, Arrr! *\*\* This is a line comment*  
3) order o be Aye, Arrr! *\*\* This is another line comment*  
4) *>> Rest of the program would be here <<****Example10: Example of line comments***

**3.5.1 Line comments**Line comments can be used to comment the remainder of a line. You start a line comment with \*\*.You can see an example in *Example10*.

1) fleet someProgram {

2) doubloon d *>> You can put a comment here <<* be 5, Arrr!   
3) order o be Aye >> Or here! <<, Arrr!   
4) *>> Rest of the program would be here <<****Example11: Example of in-line comments***

**3.5.2 In-line comments**In-line comments can be used to comment part of a line. You start a line comment with >> and end it with <<. The arrows basically point towards the code that will be commented. You can see an example in *Example11*.

1) fleet someProgram {

2) doubloon d be 5, Arrr!

3) *>> You can start a block comment here*  
4) *write some stuff*  
5) *and end it here <<*   
6) order o be Aye, Arrr!   
7) *>> Rest of the program would be here <<****Example12: Example of block comments***

**3.5.3 Line comments**Line comments can be used to comment the remainder of a line. You start a block comment with >> and end it with <<. The arrows basically point towards the code that will be commented. You can see an example in *Example12*.

**3.6 Other keywords**In this section we will talk about the keywords that do not belong to any section, belong to multiple sections or are just general (required) keywords.

1) doubloon d be 5, Arrr! *\*\* ‘Be’ used as a declaration*2) *>> doubloon <<* e be 3, Arrr! *\*\* ‘Be’ used as an assignment*3) parley(e be d) *>> if statement <<\*\* ‘Be’ used a a comparison*

***Example13: All the uses of be***

**3.6.1 Be**In Yarrva the keyword be is an important keyword: it is used for assigning and declaring variables, comparing two variables and it can be combined with above (**3.6.2 Above**) and below (**3.6.2 Below**) into be above (**3.6.3 Be above**) and (**3.6.3 Be below**).   
 In the grammar, the keyword be is shown as equalsKey, even during an assignment or declaration.  
 In the converter, the only special case for be is during Sprockell’s Comp, since it is looked at to see if it must be NEq or Equal.  
 In *Example13*, there are examples for all the uses of be.

1) parley(5 below 6) \*\* example of below   
2) parley(6 above 5) \*\* example of above

***Example14: Examples of above and below***

**3.6.2 Below and Above**In Yarrva the keywords above and below are used for comparing to values. above means strictly greater than and below means strictly smaller than.  
 In the grammar, the keywords above and below are shown as greaterKey and lesserKey, respectively.  
 In the converter, the only special cases for above and below are during Sprockell’s Comp, since it is looked at to see if it must be GtE, Lt, LtE or Gt.  
 In *Example14*, there are examples for the use of below or above.

**3.6.3 Be below and Be above**In Yarrva the keywords be above and be below are used for comparing to values. be above means greater or equal to and be below means smaller or equal to.  
 In the grammar, the keywords be above and be below are shown as gequalsKey and lequalsKey, respectively.  
 In the converter, the only special cases for above and below are during Sprockell’s Comp, since it is looked at to see if it must be GtE, Lt, LtE or Gt.  
 In *Example15*, you can add a be before below or above and it still holds.

1) doubloon a be 2, Arrr!

2) a be a + 4, Arrr!

3) avast a, Arrr! \*\* Used to return the value

4)

5) avast a + 4, Arrr! \*\* Alternative of the above

***Example15: Examples of avast***

**3.6.4 Avast**In Yarrva, the keyword avast is used to return a variable from a function to the function who originally called the function.   
 In the grammar, the keyword avast is shown as returnKey.  
 In the converter, avast is used to push a variable to the stack so the original caller can use it.  
 In *Example14*, you can see the the use of avast.

1) doubloon y be 2, Arrr!

2) parrot y, Arrr! \*\* Print 2 to the screen

***Example16: Example of parrot***

**3.6.5 Parrot**In Yarrva, the keyword parrot is used to print a a variable on screen. Due to time constraints, the print is only limited to print a value between 0 and 9.   
 In the grammar, the keyword parrot is shown as printKey.  
 In the converter, parrot is used to make Sprockell’s write to write to the stdio.  
 In *Example16*, you can see an example of parrot.

**3.6.6 Gift and Plunder**In Yarrva the keywords gift and plunder are used for comparing to values. The uses of gift and plunder are pretty simple: gift increases the value of an integer by one and plunder decreases the value of an integer by one.  
 In the grammar, the keywords gift and plunder are shown as incKey and decKey, respectively.  
 In the converter, the only special cases for gift and plunder are when they are called. In which, 1 will be loaded into a register and either be subtracted from or added to the current value using Sprockell’s Compute. It will then store the value again using Sprockell’s Store.   
 In *Example17*, there are examples for the use of gift and plunder.

1) doubloon y be 5, Arrr! *\*\* Initialize an integer with 5*  
2) parrot y, Arrr! *\*\* Prints 5 to the screen*

3) plunder 5, Arrr! *\*\* Decreases 5 by 1*

4) parrot y, Arrr! *\*\* Prints 4 to the screen*

5) gift 5, Arrr! *\*\* Increases 5 by 1*

6) parrot y, Arrr! *\*\* Prints 5 to the screen again*

***Example17: Examples of gift and plunder***

**3.6.7 Fleet**In Yarrva, the keyword fleet is used to declare a program. This was added because we wanted to add Objects (we were very optimistic at the start), in which case you would need to name it.   
 In the grammar, the keyword fleet is shown as progKey.  
 In the converter, the keyword fleet is used to import Sprockell.System, start the list of Sprockell instructions, run End Prog and create a main to run the file.  
 In *Example2*, you can see a clear use of fleet.

**3.6.8 Flagship**In Yarrva, the keyword flagship is used to declare a program. This was added because we wanted to add functions, in which case you would need a main function.   
 In the grammar, the keyword flagship is shown as mainKey.  
 In the converter, the keyword flagship. This is because all the functions declared before it will jump over their own code until called, so the program will always start at the main after it has initialized the global scope variables.  
 In most of the previous *Example*s, you can see the use of flagship.

**3.6.8 , Arrr!**In Yarrva, the keyword , Arrr! is used as an endmark.  
 In the grammar, the keyword , Arrr!is shown as endmark.  
 In the converter, the keyword , Arrr! has no use. This is because the endmark is only useful in tokenizing the program.   
 In all of the above *Example*s (except for *Example14*), you can see the use of , Arrr!.

**3.6.9 Not working keyword**Yarrva has four keywords that can be tokenized and parsed, but cannot be converted. These are or, ‘n, God’s speed and belay. or and ‘n would have been the boolean operations or and and, respectively. God’s speed would have been used to continue a for- or while-loop, without executing the block and belay would have been used to break out of a for- or while-loop.

**4 – Description of the software**

**5 – Test plan and results**

**6 – Conclusions**

**6.1 Project conclusion**

**6.2 Module conclusion**

Appendices

|  |  |  |  |
| --- | --- | --- | --- |
| 1. | Program | 🡪 | progKey, idf, PBlock |
| 2. | Stat | 🡪 | Assign, endmark |
| 3. |  | | | Array |
| 4. |  | | | IfExprKey, lpar, BoolExpr, rpar, Block, Opt [elseKey, Block] |
| 5. |  | | | printKey, Expr, endmark |
| 6. |  | | | Alt [decKey] [incKey], Type, endmark |
| 7. |  | | | forKey, lpar, Assign, point, BoolExpr, point, Expr, rpar, Block |
| 8. |  | | | whileKey, lpar, BoolExpr, rpar, Block |
| 9. |  | | | Func, endmark |
| 10. |  | | | returnKey, Opt [Expr], endmark |
| 11. |  | | | Function, idf, lpar, FValues, rpar, Block |
| 12. | Array | 🡪 | arrayKey, Type, idf, equalsKey, lbra, ArrayList, rbra, endmark |
| 13. | Block | 🡪 | lcbr, Rep0 [Stat], rcbr |
| 14. | PBlock | 🡪 | lcbr, Rep0 [GlobalVars], Rep0 [FuncBlock], MainFunc, rcbr |
| 15. | GlobalVars | 🡪 | Var, idf, equalsKey, Expr, endmark |
| 16. | FuncBlock | 🡪 | Function, idf, lpar, FValues, rpar, Block |
| 17. | BoolExpr | 🡪 | BoolExpr2 |
| 18. |  | | | Bool |
| 19. |  | | | idf |
| 20. |  | | | BoolExpr2, Alt [orKey] [andKey], BoolExpr |
| 21. | BoolExpr2 | 🡪 | Expr, Alt [equalsKey] [Alt [Alt [lesserKey] [lequalsKey]] [Alt [greaterKey] [gequalsKey]]], Expr |
| 22. | Expr | 🡪 | Alt [Expr 2] [lpar, Expr2, rpar] |
| 23. | Expr2 | 🡪 | ArrayOp |
| 24. |  | | | Type, Opt [Op, Type] |
| 25. |  | | | Alt [incKey] [decKey], Type |
| 26. | Op | 🡪 | plus |
| 27. |  | | | minus |
| 28. |  | | | times |
| 29. |  | | | divide |
| 30. |  | | | notSym |
| 31. | Type | 🡪 | Nmbr |
| 32. |  | | | Bool |
| 33. |  | | | idf |
| 34. |  | | | String |
| 35. |  | | | Func |
| 36. | ArrayOp | 🡪 | idf, lbra, ArrayIndex, rbra |
| 37. | ArrayIndex | 🡪 | Alt [Nmbr] [idf] |
| 38. | Bool | 🡪 | Alt [trueKey] [falseKey] |
| 39. | ArrayList | 🡪 | ArrayElem, Rep0 [comma, ArrayElem] |
| 40. | ArrayElem | 🡪 | Alt [Type] [idf] |
| 41. | FValues | 🡪 | Opt [FuncVal, Rep0 [NoCat comma, FuncVal]] |
| 42. | FuncVal | 🡪 | Var, idf |
| 43. | Assign | 🡪 | Opt [Var], idf, equalsKey, Expr |
| 44. | Func | 🡪 | idf, lpar, Opt [Expr, Rep0 [NoCat comma, Expr]], rpar |

Grammar

|  |  |  |  |
| --- | --- | --- | --- |
| 45. | Var | 🡪 | intKey |
| 46. |  | | | boolKey |
| 47. |  | | | stringKey |
| 48. | Function | 🡪 | intFunction |
| 49. |  | | | boolFunction |
| 50. |  | | | strFunction |
| 51. |  | | | arrFunction |
| 52. | MainFunc | 🡪 | mainKey, lpar, rpar, Block |
| 53. |  |  |  |
| 54. | progKey | 🡪 | Keyword “fleet” |
| 55. | functionKey | 🡪 | Keyword “ship” |
| 56. | intFunction | 🡪 | Keyword “doubloonShip” |
| 57. | boolFunction | 🡪 | Keyword “orderShip” |
| 58. | strFunction | 🡪 | Keyword “bootyShip” |
| 59. | arrFunction | 🡪 | Keyword “treasureShip” |
| 60. | mainKey | 🡪 | Keyword “flagship” |
| 61. | returnKey | 🡪 | Keyword “avast” |
| 62. | equalsKey | 🡪 | Keyword “be” |
| 63. | lesserKey | 🡪 | Keyword “below” |
| 64. | greaterKey | 🡪 | Keyword “above” |
| 65. | lequalsKey | 🡪 | Keyword “be below” |
| 66. | gequalsKey | 🡪 | Keyword “be above” |
| 67. | trueKey | 🡪 | Keyword “Aye” |
| 68. | falseKey | 🡪 | Keyword “Nay” |
| 69. | intKey | 🡪 | Keyword “doubloon” |
| 70. | boolKey | 🡪 | Keyword “order” |
| 71. | stringKey | 🡪 | Keyword “booty” |
| 72. | arrayKey | 🡪 | Keyword “treasure” |
| 73. | ifExprKey | 🡪 | Keyword “parley” |
| 74. | elseKey | 🡪 | Keyword “heave” |
| 75. | breakKey | 🡪 | Keyword “belay” |
| 76. | printKey | 🡪 | Keyword “parrot” |
| 77. | continueKey | 🡪 | Keyword “God’s speed” |
| 78. | whileKey | 🡪 | Keyword “whirlpool” |
| 79. | forKey | 🡪 | Keyword “navigate” |
| 80. | orKey | 🡪 | Keyword “or” |
| 81. | andKey | 🡪 | Keyword “’n” |
| 82. | incKey | 🡪 | Keyword “gift” |
| 83. | decKey | 🡪 | Keyword “plunder” |
| 84. | endmark | 🡪 | Keyword “, Arrr!” |
| 85. | lpar | 🡪 | Symbol “(“ |
| 86. | rpar | 🡪 | Symbol “)” |
| 87. | lbra | 🡪 | Symbol “[“ |
| 88. | rbra | 🡪 | Symbol “]” |
| 89. | lcbr | 🡪 | Symbol “{“ |
| 90. | rcbr | 🡪 | Symbol “}” |
| 91. | eq | 🡪 | Symbol “=” |
| 92. | lt | 🡪 | Symbol “<” |
| 93. | ge | 🡪 | Symbol “>” |
| 94. | plus | 🡪 | Symbol “+” |
| 95. | minus | 🡪 | Symbol “-“ |
| 96. | times | 🡪 | Symbol “\*” |
| 97. | divide | 🡪 | Symbol “/” |
| 98. | notSym | 🡪 | Symbol “~” |
| 99. | colon | 🡪 | Symbol “:” |
| 100. | point | 🡪 | Symbol “.” |
| 101. | comma | 🡪 | Symbol “,” |