Creation of the Potato Language

Mark van Doesem & Tim Sonderen

S1496468 & s1465252



# Main Features

## Story Telling

Potato code is designed to look a lot like normal text, so you can literally tell the program what to do in relatively normal English. It works with capital letters for names and dots at the end of each line to simulate the feeling of writing normal text instead of code even more. For example a declaration is constructed with ‘suppose’ and a variable name and a task (function) is declared as ‘Task’ , then the variables it takes in correct comma and ‘and’ usage and then what it gives back.

## Comments

Of course, you can also create comments in Potato code. This comments are completely ignored by the compiler, but can be useful to explain what you code is supposed to do at a certain point. A comment is started with the keyword ‘btw’ and is ended with a dot. Quite a nice feature about these comments are that they can be put anywhere in the code. So for example you can put a comment after each argument of a task that explains what that argument should represent.

## Tasks

Potato works with tasks (procedures) so that you can work more efficiently because you do not have to duplicate any code if you want to reuse it. This way less lines are required for a more majestic piece of code. You can even define tasks in your tasks. Taskception!

## Arrays

In Potato, you can use arrays to store a list of values so you do not need to a separate variable for every value. You can use the elements of an array just like any other variable. This way you can for example loop over an array to check if each value in that array is greater than a certain number, which could be useful for checking test results.

## Increment

Potato also supports task increment. This task, of course, increments the given variable. Increment takes only a variable because incrementing an integer wouldn’t be saved anywhere. The Increment task can be very useful in a ‘while’ loop, for example to iterate over a list or to count something.

## Reusable variable names

In Potato you are able to re-declare variables if that variable is not yet declared in that scope. This way you won’t run out of variable names as fast. Of course you can’t re declare variable names in the same scope.

## Errors

The Potato compiler is able to give you several errors at once when you accidentally made a mistake in your code. It will try to compile as much as it can and then print a list of errors if there are any. This list is clearly ordered with the sentence number in front of each error so it is clear where and what went wrong.

# Problems and Solution

Problem: Defining the AST; we defined the AST just as a node which can have an Alphabet and a list of sub-AST’s. This seemed fine at first also because we did not really think ahead with this. Also, we completely forgot that that was not even the way to make a decent AST as we learned it. A bit later we discovered this, but by then it was already too late to change as we didn’t have enough time anymore.

Solution: As you might be able to tell, we were not really able to fix this, we just put it in the report as a sign that we know how it is supposed to be. We should have made a separate type of AST for each non-terminal and extra nodes if a non-terminal can have different kinds/amounts of sub AST’s.

Problem: Giving line numbers with the errors so you can see where the error is located. This was took more effort than we expected because we did everything in Haskell so we used an AST tree that did have any line numbers in it so if we found an error we would know what node it is at, but not what line that node corresponds to.

Solution: Define a function that calculates the line number. The function first calculates a list of integers that represent the path to the node that you want the line number of. Then it starts at the root of the AST and walks the path calculated before. For each step down, it calculates the amount of lines that correspond to each sub-AST left to the sub-AST you go to, until it gets to the node it wanted the line number of.

Problem: Too ambitious. We tried to add too many extra features for bonus points right at the start. This became a problem at the end because by then we had a bit of everything, but none of the objectives completely.

Solution: When we noticed this we just dropped a few objectives and focused on the others so we might be able to get some of the bonus points. We put functions and arrays as our main goals and dropped characters and with that, strings.

Problem: Recursion of tasks does not work after code generation. A function calling another function is fine, but a function calling itself doesn’t go very well.

Solution:

# Detailed language description

## Program

At the beginning of every program you have to start it by defining that it is a program and giving the name. Usage is very easy:

1. program ExampleProgram:
2. \*1337 lines of code here\*
3. stop.

As you can see, you start the program with the line ‘program \*name\*:’ and end it with the line ‘stop.’ Between these lines you can put your code. The program name always have to start with a capital, just as is usual with names.

## Declaration

To declare a new variable you can use the keyword ‘suppose’. An example for declaring the integer x would be:

1. suppose integer x.

You can also declare a variable and immediately assign it to a value or even an expression as seen in the following example:

1. suppose integer x is 5.
2. suppose boolean y is x equals 5.

Variable names should always start with a lower case character, in contrary to program- and task names. Also, a variable can only be declared if it is not yet declared in the same scope. It can be re-declared if it is in a lower scope than the declaration before it. You can declare integers, Booleans and arrays. Declarations of arrays are a little different as will be shown in this example:

1. suppose [integer] l of length 3.
2. Suppose [boolean] l is [true,true,false].

If you declare a variable, but do not assign it, it is automatically initialized with 0 for integer and false for boolean.

## Assignment

Assigning a variable is rather easy. You just state the variable name, then what it is should be with the keyword ‘is’ in between as seen in the following example:

1. suppose integer x.
2. x is 5.
3. x is 5 times 2.
4. x is x minus 7.

Assignment of arrays is pretty much the same and with arrays you can also assign a value on a certain position in that array. So in the following example line 2 shows assignment of a complete array and line 4 and 5 assignment of a certain index of an array:

1. suppose [integer] list of length 5.
2. list = [1,2,3,4,5].
3. suppose integer i is 2.
4. list[i+1] is 12.
5. list[2] is list[1].

Of course, you can assign only the appropriate type to a certain variable e.g. integer to an integer, boolean to a boolean etc. and you can only use variables that are already declared.

## Comment

A comment is a piece of code that is completely ignored by the compiler. It is started with the keyword ‘btw’ and is ended with a dot. This comment can be placed anywhere in the code. Of course it may not split words in half. Example:

1. suppose btw this is a comment. integer a.
2. a is 5. Btw this is also a comment.

## Expression

An expression is basically anything that has a value. It can contain a value (integer, boolean or array), a variable, a task call and any combination of those with operators. Of course, everything you use should be declared in the lines above it, including tasks you call. In the following example all the operators are shown. It is supposedly pretty clear what they do as that is wat this language intended.

1. suppose integer a is 5.
2. suppose integer b is 2.
3. suppose boolean x is true.
4. suppose boolean y is false.
5. a is a plus b. btw takes 2 integers, gives 1 integer.
6. a is a minus b. btw takes 2 integers, gives 1 integer.
7. a is a times b. btw takes 2 integers, gives 1 integer.
8. a is a divided by b. btw takes 2 integers, gives 1 integer.
9. x is x and y. btw takes 2 booleans, gives 1 boolean.
10. x is x or y. btw takes 2 booleans, gives 1 boolean.
11. x is a equals b. btw takes 2 integers or booleans, gives 1 boolean.
12. x is a is greater than b. btw takes 2 integers, gives 1 boolean.
13. x is a is greater than or equal to b. btw same as above.
14. x is a is smaller than b. btw same as above.
15. x is a is smaller than or equal to b. btw same as above.

Not giving or taking the right type to an operator will result in a type error. And of course you can use brackets while combining these operators.

So now a few examples of what an expression could be:

1. ((a plus b) is smaller than or equal to b) or (x equals y)
2. a plus b plus a plus a times 5
3. TestingTaskZero(a,b). btw this is a task call.

Note that these are no legit Potato code lines, expressions should be used to assign a variable or give to a function.

## When

A when-statement can have 2 bodies and must at least have 1. It uses a boolean expression to see which of the 2 bodies it should execute. If the expression is true, it will do the first one, if it is false it will do the second one. If the second body is not present it will do nothing. A when-statement is defined by the keyword ‘when’, then a boolean expression and then the keyword ‘do:’ to start the first body. Then you can type as many lines of code as you want and to stop the body use the keyword ‘stop.’. If you want to use a second body you can add the keywords ‘otherwise do:’, then type some lines of code and end the body again with the keyword ‘stop.’. A few examples of a when statement are:

1. suppose boolean b is true.
2. when b do:
3. b is false.
4. stop.
6. suppose integer x is 0.
7. when x is smaller than or equal to 5 do:
8. x is x plus 5.
9. stop.
10. otherwise do:
11. x is 2.
12. stop.

## While

A while-statement is created pretty much the same as a when-statement. It is started with the keyword ‘while’, then a boolean expression followed by the keyword ‘do:’ to start the body. Unlike a when-statement, a while-statement only has 1 body. It will check if the expression is true, then execute the body. Then it will repeat itself so it will keep looping over the body until the expression is not true anymore. An example of a while-statement is:

1. suppose boolean tooSmall is true.
2. suppose integer x is 0.
3. while tooSmall do:
4. x is x plus 1.
5. tooSmall is x is smaller than or equal to 5.
6. stop.

## Increment

The increment statement is a very simple statement. It is basically a function that just adds 1 to the given variable and then returns it. Usage is very simple also:

1. suppose integer x is 0.
2. increment x.
3. btw x is now 1.

## Task

A task is a piece of code with a name, a few arguments and it will give you a value back. With this you can easily reuse code. A task can return a variable which then of course has a type, or it can return nothing, in which case you can’t assign it to a variable either.

To create a task use the keyword ‘task’, then the task name which should start with a capital letter, followed by the keyword ‘takes’ and the arguments it will take followed by the keyword ‘gives’ and the type of value it will give back. Then the keyword ‘after:’ to start the body. To stop the body use the keyword ‘give’ and the variable that should be returned or ‘stop.’ to return a ‘nothing’ type. It is probably easiest to just show how it works so here is an example:

1. task ExampleTask takes integer a, integer b and integer c and gives integer after:
2. suppose integer d.
3. d is a plus b plus c.
4. give d.
6. task ExampleTask2 takes integer a and gives nothing after:
7. btw do something with variables from a higher scope here.
8. stop.

You can also assign tasks in tasks and even deeper, but keep in mind that in order to be able to call a task, it has to be defined in the same scope, before you call it.

# Description of the software

# Test plan and results

# Conclusions

# Appendices

## Appendix A – Grammar specification:

1. grammar nt = case nt of
2. Program -> [[prog, FuncName, ProgBody]]
4. ProgBody -> [[semi, Rep0 [Line], stop, dot]]
6. Line -> [[Decl]
7. ,[Assign]
8. ,[FuncCall]
9. ,[Incr]
10. ,[When]
11. ,[While]
12. ,[Task]]
13. Decl -> [[suppose, Opt [global], Type, Idf, Alt [ofK, lengthK, Expr] [Opt [is, Expr]], dot]]
15. Assign -> [[Idf, is, Expr, dot]]
17. FuncCall -> [[FuncName, lPar, Rep0 [Expr, Opt [comma]], rPar, Opt [dot]]
18. ,[FuncName, lPar, Rep0 [Expr, Opt [comma]], rPar]]
20. Incr -> [[inc, Idf, dot]]
22. When -> [[when, Expr, doK, Body, Opt [otherwiseK, doK, Body]]]
24. While -> [[while, Expr, doK, Body]]
26. Task -> [[task, FuncName, takes, Args, gives, Type, after, Body]]
28. Args -> [[Rep0[Arg]]]
30. Arg -> [[Type, Idf, Alt [comma] [andK]]]
32. Body -> [[semi, Rep0 [Line], Alt [stop, dot] [give, VIA, dot]]]
34. Expr -> [[VIA, Op, Expr]
35. ,[lPar, Expr, rPar, Op, Expr]
36. ,[lPar, Expr, rPar]
37. ,[VIA]]
39. VIA -> [[Value]
40. ,[FuncCall]
41. ,[Idf]
42. ,[Array]]
44. Op -> [[plus]
45. ,[minus]
46. ,[times]
47. ,[DividedBy]
48. ,[equals]
49. ,[is]
50. ,[GreaterThan]
51. ,[GreaterThanEq]
52. ,[SmallerThan]
53. ,[SmallerThanEq]
54. ,[andK]
55. ,[orK]]
57. DividedBy -> [[divided, by]]
58. GreaterThan -> [[is, greater, than]]
59. GreaterThanEq -> [[is, greater, than, orK, equal, to]]
60. SmallerThan -> [[is, smaller, than]]
61. SmallerThanEq -> [[is, smaller, than, orK, equal, to]]
63. FuncName -> [[funcName]]
65. Type -> [[TypeBool]
66. ,[TypeInt]
67. ,[TypeChar]
68. ,[TypeArray]
69. ,[TypeNothing]]
71. Idf -> [[idf, Opt [lBracket, Expr, rBracket]]]
73. Value -> [[Boolean]
74. ,[Integer]
75. ,[Character]]
77. Array -> [[lBracket, Rep0 [ArrayVal], rBracket]]
78. ArrayVal -> [[VIA, Opt [comma]]]
79. TypeArray -> [[lBracket, Type, rBracket]]
81. Boolean -> [[Alt [TrueK] [FalseK]]]
82. TrueK -> [[trueK]]
83. FalseK -> [[falseK]]
84. TypeBool -> [[typeBool]]
86. Integer -> [[int]]
87. TypeInt -> [[typeInt]]
89. Character -> [[char]]
90. TypeChar -> [[typeChar]]
92. TypeNothing -> [[nothing]]

## Appendix B – Extended test program