Creation of the Potato Language

Mark van Doesum & Tim Sonderen

S1496468 & s1465252

Table of Contents

Table of Contents 2
Main Features 3
Problems and Solution 4
Detailed language description 5
Description of the software 11
Test plan and results 13
Conclusions 16
Appendices 18

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 followed by a dot, which makes it a perfectly normal English sentence.

Comments

Of course, you can also create comments in Potato code. The 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!

Recursion

Our tasks have built in support for recursion, however there is a minor issue with function calls and expressions which is discussed in the next chapter.

Arrays

In Potato, you can use arrays to store a list of values so you do not need 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 redeclare 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 as a node which can have an Alphabet and a list of sub-AST's. This seemed fine at first but turned into a major headache later on. 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 not everything was fully implemented.

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: Function calls & Expressions. Our language and code generation has built in support for recursion, but a few bugs keep it from being fully functional. Firstly, expressions do not handle function calls the way they should (e.g. Fib(n-1)+Fib(n-2) isn't guaranteed to work). We believe this is due to the fact that the result of a function call is stored in a register (Fib(n-1)), and then overwritten by the second function call in the expression (Fib(n-2)). In other words our language is not guaranteed to work if the second argument of an expression (the 'x' in the following expression: Fib(n-1)+x) is a function call. Due to time constraints we haven't had the time to evaluate if this is the actual problem, so in order to be save we discourage the use of function calls in expressions.

Solution: A possible solution for function calls in expressions is to simply avoid them by declaring variables for the function calls, and putting these variables in the expression instead. So in the Fibonacci example mentioned before: assign Fib(n-1) and Fib(n-2) to two separate variables and return the addition of these variables instead. Due to time constraints we were unable to fix this issue in our code generation.

Detailed language description

General

A few things that should be thought of in general; A line, that is; a declaration or an assignment of any sort (so the normal lines), always ends with a dot, otherwise a parse error will be thrown. Besides that, tabs are taboo, the compiler can't and won't handle them correctly, not even a decent error.

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:

```
program ExampleProgram:
    *1337 lines of code here*
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 has to start with a capital, just as is usual with names.

No SPRIL-code is generated for this feature.

Declaration

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

```
suppose integer x.
```

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

```
suppose integer x is 5.
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:

```
suppose [integer] l of length 3.
suppose [boolean] l is [true,true,false].
```

The array declaration using 'of length' takes an integer as last argument, no identifier. There was no time to fix this minor issue in time.

If you declare a variable, but do not assign it, it is automatically initialized with 0 for integer and false for boolean. The same goes for arrays without assign. They are given value 0 by default and thus length 1. Defeating the purpose of declaring an array without an assignment, since the length of the array cannot be edited after declaration.

The generation of the SPRIL-code for this function works as follows: first the expression part of the declaration is evaluated, this stores the result in the specified register. Then a new address in local memory is allocated for the variable and the result of the expression is stored at this address. In practice this looks as follows:

```
//Potato-code:
suppose integer integer x is 5 plus 3.
//SPRIL-code:
[Const 5 RegA, Const 3 RegB, Compute Add RegA RegB RegA, Store RegA (Addr 0)]
```

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:

```
suppose integer x.
x is 5.
x is 5 times 2.
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. However, it is possible to assign a value to an index which is out of bounds of the array. This would simply overwrite the value stored at that address. 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.

Assignment code generation is almost exactly the same as declaration, except that it doesn’t allocate a new memory address. Instead it looks up the address of the variable in a list and stores it at that address, overwriting the previous value:

```
//Potato-code:
k is 5 plus 3.
//SPRIL-code:
[Const 5 RegA, Const 3 RegB, Compute Add RegA RegB RegA, Store RegA (Addr 0)]
```

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:

```
suppose btw this is a comment. integer a.
a is 5. Btw this is also a comment.
```

Comments are deleted by the tokenizer and thus never reach code generation.

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 the different operators do as that is what this language intended.

```
suppose integer a is 5.
suppose integer b is 2.
suppose boolean x is true.
suppose boolean y is false.
a is a plus b.            btw takes 2 integers, gives 1 integer.
a is a minus b.           btw takes 2 integers, gives 1 integer.
a is a times b.           btw takes 2 integers, gives 1 integer.
a is a divided by b.      btw takes 2 integers, gives 1 integer.
x is x and y.             btw takes 2 booleans, gives 1 boolean.
```

```
x is x or y.                btw takes 2 booleans, gives 1 boolean.
x is a equals b.            btw takes 2 integers or booleans, gives 1 boolean.
x is a is greater than b.   btw takes 2 integers, gives 1 boolean.
x is a is greater than or equal to b.   btw same as above.
x is a is smaller than b.               btw same as above.
x is a is smaller than or equal to b.   btw same as above.
```
Not giving the right type to an operator will result in a type error. And of course you can use parenthesis while combining these operators.

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

```
((a plus b) is smaller than or equal to b) or (x equals y)
a plus b plus a plus a times 5
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 given to a function, not used on top level.

The code generated for expressions works as follows: an expression either exists of (left-side, right-side and operator) or just one of the following [Integer, Boolean, Identifier or Function call]. In the first case, the left-side and right- side are expressions, so recursion is used to generate the code for the two separate expressions, each one storing its result in a different register. Then the operator is determined and applied to the registers, storing it in the predetermined register. This looks as follows:

```
//Potato-code:
(3 times 4) plus 5
//SPRIL-code:
Const 3 RegA, Const 4 RegB, Compute Mul RegA RegB RegA, Const 5 RegB,
Compute Add RegA RegB RegA
```

When

This is our equivalent of an if-statement, while we are aware that when and if have different meanings, we just couldn't stand blatantly copying 'if' which is used in almost every known programming language while we could come up with a decent alternative. A when-statement can have 2 bodies and must at least have one. 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 followed by 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:

```
suppose boolean b is true.
when b do:
    b is false.
stop.

suppose integer x is 0.
when x is smaller than or equal to 5 do:
    x is x plus 5.
stop.
otherwise do:
    x is 2.
stop.
```
To generate code for the 'when', the expression is evaluated first. The result of the expression gets XOR'd with 1 and is stored in a register (if the expression is true, the XOR returns false and the other way around). This is done for readability of the SPRIL code (e.g. the body of the when comes before

the otherwise). Next a calculation of where to branch to in order to skip the first body is made and the body of the when is generated followed by a calculation of where to jump to in order to skip the otherwise. Lastly the body of the otherwise is generated (can be completely empty and thus not visible in the SPRIL-code).

```
//Potato-code:
when true do:
    b is false.
Stop
//SPRILL-code
Const 1 RegA, Const 1 RegB, Compute Xor RegA RegB RegA,
Const (bodySize+4) RegB, Compute Add RegB PC RegB, Branch RegA (Ind RegB),
//BODY
Jump (Rel otherwiseSize+1)
//OTHERWISE
```

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. After the body it will return to the given expression and revaluate it before deciding to start the while loop once more, or be done with it. This results in the while looping over the body until the expression is no longer true. An example of a while-statement is:

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

The while code generation works in a similar manner as the when with a few alterations: the very first step is to push the program counter (PC) to the stack. The second step is to evaluate the expression, XOR it and create a branch for when the expression no longer holds. As a third step the body is generated. After the body generation, the PC is popped from the stack and a jump to that instruction is made. This means that after one iteration of the while, a jump is made back to the very first step: pushing the PC to the stack.

```
 //Potato-code:
while b do:
    b is false.
Stop
//SPRILL-code  b@ Addr 0
Calculate PC Zero RegA, Push RegA,
Load (Addr 0) RegA, Const 1 RegB, Compute Xor RegA RegB RegA,
Const (bodySize+4) RegB, Compute Add RegB PC RegB, Branch RegA (Ind RegB),
//BODY
Pop RegA, Jump (Ind RegA)
//OTHERWISE
```

Increment

The increment statement is a very simple statement. It is basically a function that just adds 1 to the given variable and stores it at the address of the variable. Usage is very simple as well:

```
suppose integer x is 0.
increment x.
btw x is now 1.
```

The code generation is also fairly straightforward: the identifier is loaded in a register, 1 is loaded in a different register, and the two get added and stored to the address belonging to the identifier:

```
//Potato-code:
Increment a.
//SPRILL-code a@ Addr0
Load (Addr 0) RegA, Const 1 RegB, Compute Add RegA RegB RegA,
Store RegA (Addr0)
```

Task

A task is a piece of code with a name, a few arguments and it can potentially give you a value back. Using this you can easily reuse code. A task can return a variable which, 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. Also, you have to define a variable **in** the task in order to be able to 'give' it. Another slight problem is our implementation of arrays. Since there is no way to obtain the length of an array (we need to know how many values we need to pop/push from/to the stack), our tasks have no support for arrays as input or output of a task. It is probably easiest to just show how it works so here is an example:

```
task ExampleTask takes integer a, integer b and integer c and gives integer
after:
    suppose integer d.
    d is a plus b plus c.
give d.

task ExampleTask2 takes integer a and gives nothing after:
    btw do something with variables from a higher scope here.
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.

The code generation works as follows: firstly two calculations are made to determine where to jump in order to start the task and where to jump in order to skip the task. The start is stored in memory and a jump to the instruction after the task is made. Secondly the actual body of the task is generated: the arguments get popped of the stack, the rest of the body is generated and finally the return address is popped. If the function is supposed to return something, the return value is pushed before a jump to the return address is made. This looks as follows:

```
//Potato-code
task ExampleTask takes integer b and integer c and gives integer after:
    //Body
give c.
//SPRILL-code
Const 3 RegA, Compute Add RegA PC RegA, Store RegA (Addr 0),
Jump (Rel bodySize), Pop RegA, Store RegA (Addr 1),
Pop RegA, Store RegA (Addr 2)
//BODY
Pop RegB, Load (Addr 2) RegA, Push RegA, Jump (Ind RegB)
```

Function Call

A function call is actually an expression but since the code generation of a function call is rather different the decision was made to give it its own heading. The syntax is fairly straightforward:

```
funcName(arg,arg,…arg)
Fib(5)
```

As mentioned, it is an expression type and can thus be used the same way as booleans and integers depending on the type of the function. This feature will execute a specified set of statements defined in the task it calls, which can possibly return something.

Code generation for function calls works as follows: First all the known address are pushed to the stack, then the return address is calculated and pushed followed by pushing all the arguments. Then the instruction of the start of the function is located and a jump to it is made. In case the function returns something, the return value is popped to a register, followed by the popping and restoring of all the known addresses using a different register. Only after this is done, the return value is stored to memory.

```
//Potato-code
Fib(3)
//SPRILL-code
Load (Addr 0) RegA, Push RegA, //push all other known addresses
Const 6 RegA, Compute Add RegA PC RegA, Push RegA
Const 3 RegA, Push RegA, Load (Addr 0) RegA, Jump (Ind RegA)
```

Description of the software

DatatypesEtc

The file DataTypesEtc contains all the data types of our project. We put these in one file in order to make the project more modular. This way all other files can just import DataTypesEtc and then they have all the data types they need. Otherwise imports would have to loop back and forward and Haskell doesn't accept that.

Grammar

The file Grammar contains only the grammar we used.

Parse

The file Parse contains the tokenizer, the parser, a function to make an AST from the tree you get from the parser and a function Parse0 which is the main function of the file. It takes a string and returns the AST it would correspond to, given that there are no errors of course.

The tokenizer first adds spaces around dots, comma's, parenthesis, etc. so that they can be recognized as separate tokens. Then the comments are filtered out, that is, a function takes elements from a list, then if it sees the word 'btw' it stops taking tokens but continues looking until it sees the keyword '.'. Then for the rest of the words it is determined what they are. First is checked if it starts with a capital letter, then it would be a function name. Then it is compared to true and false. After that we check if it may be an integer or a character (we don't use those anymore though). Lastly we check if it can be a token from the list of tokens we defined (which are the keywords) and if all else fails it is an identifier (var name).

The resulting token list is put in the parser we got from Jan Kuuper. This parser builds us a tree using the grammar, which we then convert to an AST so the checker can use it.

Checker

The Checker file contains the part where the AST is checked for type errors and scope errors. In this file, the error printing is also included. The main function is check, which takes an AST and checks it for type and scope errors. If it finds any, it will print them and throw a Haskell error so the compiling stops.

The function that checks the errors works with pattern matching so it is basically a tree visitor, besides that, the function is recursive so it is a synthesised attribute. The checker just runs over a list of nodes, then checks its sub trees, adds those errors to the error list, then checks the rest of its 'brothers' and add that to the error list too. Because the variables list is passed on in the recursion too, every time you go down a node, it is basically a new scope. Just in some cases such as program, task, when and while a special tuple is added to be able to see when the new scope starts for later checking if a variable may be re-declared.

In a declaration and task node, new variables are defined so then the function to make a new tuple for the variable list is called and the tuple is added to the variable list.

There is also a function that checks an expression and then returns the type the expression would result in. We couldn't build this in the main checker function because this wouldn't work with the return types. The main checker now just lets a separate function evaluate the expression so it just has to check if the types match.

Main

The main file is actually very simple. This is the only impure file we have. Its main function is compile, which puts the whole project together. It takes a string that represents a file path, this should be a .txt. compile reads the complete file as a string and puts it in the parse0 function from the file Parse, which gives back an AST. Then compile puts that AST into the checker and if that doesn't throw an error it will continue with the code generation. At the end it will print the sprockel code.

In this file there is also a function that works the same with the file reading and then creates an AST from that string. Then instead of checking and generating code, it will show the tree using the 'standard webpage 2' from the practical lab sessions.

TreeWalker

The TreeWalker file is the file in which all the SPRIL-code generation takes place. The writeToFile function adds a bunch of strings together which are necessary to create a file which can be run by the Sprockell. It also calls the most important function of the file: walkTree. This function is a tree visitor, it visits each node in the AST tree and generates SPRIL-code for it. All of the other functions in the file merely exist to avoid repetition of code, massive lines of pattern matching or exist to make our job easier. The only other notable function is evalExpr, which is used to evaluate all expression nodes in the tree (declaration, assign, when and while all use expressions).

The walkTree function visits a node and uses pattern matching to determine which course of action to take. For each node it creates one or more lists of instructions and concats all of them together to form a single, large list of SPRIL instructions which can be written to a file. Most of how each node is handled can be found in the previous chapter 'Detailed language description'.

The result of this file can be written to a file directly, which in turn can be executed on the Sprockell.

Testing

The testing file includes the functions used to tests all the different parts of the compiler. How to use it is discussed more in the "Test plan and results" chapter.

Test plan and results

In order to test our program, the file Testing is used. In this file three main functions can be found for the three different types of tests: syntax, context and semantics. Each of which will be discussed in a separate heading.

Syntax testing

For syntax testing the function testSyntax is used. This function takes a file path to a file in which a program is defined in potato-code and the AST tree we expect it to give. In the “src/input/syntax” directory a couple of test programs can be found. Two files for each feature mentioned in the chapter “Detailed language description”: one correct, the other faulty (Fail as part of the name). In file src/TestOutcomes.hs the expected tree for each of these files is provided. For ease of use we also added a shortcut to run the syntax test for each separate file which can be found at the bottom of Testing. Simply call those commands in order to reproduce our results.

For each of the faulty files, the following error should be thrown:

```
"*** Exception: ParseError
```

While all of the correct test should print:

```
"No syntactical errors found"
```

A third option is:

```
"No syntactical errors found but the AST's are not the same"
```

Which is printed when the parser didn’t throw an exception, but the generated and expected AST’s do not match. This should option should never occur unless one edits the test files/TestOutcomes.hs.

The functions drawTree and drawTreeFromAST are also provided in order to visualise the AST. drawTree takes a filepath, the same as testSyntax, and draws the tree in a web browser using Standard Webpage v2. This only works if the input file does not give a ParseError exception. drawTreeFromAST can be used to visualize the provided correct AST’s for each file.

Context testing

The function testContext is used for, guess what, context testing! It takes a filepath as input, which is similar to the syntax test. The test files can be found in “src/input/context” and consist of the following: IdfContext shows the correct use of identifiers including where they can be used while IdfContextFail gives a couple of examples of how NOT to use them. The result from IdfContext should be the string:

```
"no contextual errors found"
```

While the faulty test should display the following message:

```
===========================================================
- Sentence 2:
Variable b not in scope
- Sentence 5:
Variable k not in scope
- Sentence 6:
Variable k not in scope
- Sentence 8:
Function Test not in scope
===========================================================
*** Exception: Errors found, printed them above this line.
```

Quick reminder: sentence numbers may be incorrect, this is a known bug as mentioned before.

TypeContext is a type checking test where all the types are correctly used while TypeContextFail, of course, contains faulty code. The correct test should once again give:

```
"no contextual errors found"
```

While the faulty code generates the following error:

```
============================================================
- Sentence 0:
"x" is of TypeBool not TypeInt.
- Sentence 0:
"b" is of TypeInt not TypeBool.
- Sentence 4:
length of array should be an integer
- Sentence 7:
length of array should be an integer
- Sentence 8:
"y" is of TypeBool not TypeInt.
- Sentence 9:
"a" is of TypeInt not TypeBool.
- Sentence 10:
"k" is array of TypeInt not of TypeBool.
- Sentence 11:
When statement should contain a boolean expression
- Sentence 12:
While statement should contain a boolean expression
- Sentence 0:
"c" is of TypeInt not TypeNothing.
- Sentence 13:
Task Test takes arguments of type ["TypeInt"], not of type ["TypeBool"]
- Sentence 0:
"d" is of TypeBool not TypeInt.
============================================================
*** Exception: Errors found, printed them above this line.
```

Once again, shortcuts where created which can be found at the bottom of the Testing file.

Semantic testing

testSemantic is the function used for testing semantics. It once again takes a filepath as input. The test files can be found in src/input/semantic: infiniteLoop, divisionByZero and fib. Shortcuts are provided at the bottom of Testing.

InfiniteLoop should not give an error, nor should it ever stop unless it is interrupted. Let it run for a few minutes and take a look at the code to see that it does indeed never stop.

divisionByZero is a simple algorithm which divides all values in an array by the value that comes after it. An example: the program has an array [1,2,3] and creates a new array [1/2,2/3,3]. The last value of an array doesn't have a follow up value and is thus ignored. Now, what happens when we introduce a zero somewhere in the input array? This is done in the provided test file which gives the following error:

```
*** Exception: divide by zero
```

Fib is discussed in Appendix B.

For all three files, the SPRIL-code be found in the src/Example directory where some comments are added to give the reader some idea of how the generated code works.

Conclusions

Tim

About the project: In total I am pretty satisfied with the programming language we made. At least the idea of it (text based, as natural as we could make it). The execution was not as good as we were over ambitious and couldn't finish everything we wanted. This was mainly because we were actually very excited about the project because it sounded (and was) very interesting and fun to make. We definitely learned how much work goes into making even such a simple programming language. I think it would be a great project to do along the whole line of the module so that instead of doing countless of roughly the same exercises we can do it as a part of the project. This way it is more motivating to do the exercises and you can make the project a bigger thing, implement more features in the language.

About the module: Overall I found the module quite interesting. And in general it was organized pretty well. There are some things that didn't go very well tough. For one, as you probably already know, CC took way too much time. (Also) because presence was mandatory, this pushed aside the other lines of the modules. Mainly CP because this wasn't mandatory. Also, CP was in my opinion pretty interesting, but the problem was the way it was given. I know the way the tutorials work are a pretty good way to learn the matter, but for me, it is highly demotivating to go when I have to do the homework before the tutorial and then 'present' my try. Especially when I am this busy with CC.

Another point is the fact that we had several occasions that we had a deadline and a few days (project) or even hours (CC HW 2) before the deadline the assignment was (minimally) changed. Even though there were no major changes, this is very annoying and can for some people be a source of a lot of stress.

FP I found very interesting and I have absolutely no complaints about how this was given. (Might be why we did the full project in Haskell).

Mark

I am fairly satisfied with our language as a whole. We achieved our goal of making it a spoken language based programming language and the most important features (arrays and functions) are also supported. It is however quite a shame that our implementation of arrays is so crappy, resulting in some key uses for arrays not being supported. If we had like one more week, we could probably have rewritten it in such a way that Potato fully supports arrays. I am however very happy that recursion is finally working. I spend quite a while on its code generation but somehow I never got it to work properly. I kept checking the SPRIL-code, but just couldn't figure out where the mistake was located. Eventually I just gave up. A couple days later, after some fixes in type checking and adding 5 additional registers (we needed 6 registers for one of the tests) it suddenly worked! I still don't really understand why, since the code doesn't use more than 3 registers as far as I'm aware.

I liked the setup of this project quite a bit. It is one of the first projects which really forces you to implement everything you learned during the module. The project provided a nice and interesting challenge. However, it was kind of annoying that the project description wasn't ready at the start of the project and that it kept being filled in while we were already working on it. On top of that, there were bits and pieces of the description which were written for the people working with ANTLR, not for the people whom did everything in Haskell. An example would be tests: automated test resulted in an additional 1.5 points bonus, but we weren't taught anything about testing in Haskell, which made it quite challenging and pretty much impossible to create automated tests. I feel like this project has a lot of potential and with the project manual now done, it should be even better next year. I really enjoyed making this project.

The module as a whole was also setup quite nicely, with a lot of interaction between CC and FP. It was kind of sad to see that CP didn't join in on this. For me that meant that I discarded CP a bit until one week before the exam. FP was well organized and fun. I don't really have any recommendations for that subject. CC took a lot more hours than it was supposed to, another reason for me to skip out on CP. There has already been a lot of discussion about this, so I'll leave it at that. I hope the workload is better for next year's students. Then there were the problems with the CC homework exercises. Please, get this is order for next year. Updating the exercises so close to the deadline is just unreasonable. This was probably the biggest issue I had with this module. The unfinished documents that had to keep being updated while we were already working with them. I really hope all of that gets fixed next year.

Overall I quite enjoyed this module as a whole. There are still some issues that need some attention, but nothing unfixable in my opinion. A nice way of ending the year!

Appendices

Appendix A – Grammar specification:

```
grammar nt = case nt of

        Program -> [[prog, FuncName, ProgBody]]

        ProgBody    -> [[semi, Rep0 [Line], stop, dot]]

        Line    -> [[Decl]
                    ,[Assign]
                    ,[FuncCall]
                    ,[Incr]
                    ,[When]
                    ,[While]
                    ,[Task]]

        Decl    -> [[suppose, Opt [global], Type, Idf,
                    Alt [ofK, lengthK, Expr] [Opt [is, Expr]], dot]]

        Assign  -> [[Idf, is, Expr, dot]]

        FuncCall    ->  [[FuncName, lPar, Rep0 [Expr, Opt [comma]], rPar,
                         Opt [dot]]
                        ,[FuncName, lPar, Rep0 [Expr, Opt [comma]], rPar]]

        Incr    -> [[inc, Idf, dot]]

        When    -> [[when, Expr, doK, Body, Opt [otherwiseK, doK, Body]]]

        While   -> [[while, Expr, doK, Body]]

        Task    -> [[task, FuncName, takes, Args, gives, Type, after, Body]]

        Args    -> [[Rep0[Arg]]]

        Arg     -> [[Type, Idf, Alt [comma] [andK]]]

        Body    -> [[semi, Rep0 [Line], Alt [stop, dot] [give, VIA, dot]]]

        Expr    -> [[VIA, Op, Expr]
                    ,[lPar, Expr, rPar, Op, Expr]
                    ,[lPar, Expr, rPar]
                    ,[VIA]]

        VIA     -> [[Value]
                    ,[FuncCall]
                    ,[Idf]
                    ,[Array]]

        Op      -> [[plus]
                    ,[minus]
                    ,[times]
                    ,[DividedBy]
                    ,[equals]
                    ,[is]
                    ,[NotEqual]
                    ,[GreaterThan]
                    ,[GreaterThanEq]
                    ,[SmallerThan]
                    ,[SmallerThanEq]
                    ,[andK]
                    ,[orK]]

        NotEqual    -> [[is, notK, equal, to]]
        DividedBy   -> [[divided, by]]
        GreaterThan -> [[is, greater, than]]
```

```
GreaterThanEq -> [[is, greater, than, orK, equal, to]]
SmallerThan -> [[is, smaller, than]]
SmallerThanEq -> [[is, smaller, than, orK, equal, to]]

FuncName -> [[funcName]]

Type    -> [[TypeBool]
           ,[TypeInt]
           ,[TypeChar]
           ,[TypeArray]
           ,[TypeNothing]]

Idf     -> [[idf, Opt [lBracket, Expr, rBracket]]]

Value   -> [[Boolean]
           ,[Integer]
           ,[Character]]

Array   -> [[lBracket, Rep0 [ArrayVal], rBracket]]
ArrayVal    -> [[VIA, Opt [comma]]]
TypeArray   -> [[lBracket, Type, rBracket]]

Boolean -> [[Alt [TrueK] [FalseK]]]
TrueK   -> [[trueK]]
FalseK  -> [[falseK]]
TypeBool -> [[typeBool]]

Integer -> [[int]]
TypeInt -> [[typeInt]]

Character -> [[char]]
TypeChar -> [[typeChar]]

TypeNothing -> [[nothing]]
```

Appendix B – Extended test program

As an extended test we decided to implement Fibonacci, since it uses quite a few of our implemented features: Declarations, assigns, when/otherwise, tasks, task calls and recursion.

```
program Fibonacci:
    task Fib takes integer n and gives integer after:
        suppose integer k.
        when (n equals 0) or (n equals 1) do:
            k is 1.
            stop.
        otherwise do:
            suppose integer l is Fib(n minus 1).
            suppose integer r is Fib(n minus 2).
            k is l plus r.
            stop.
        give k.
    suppose integer g is Fib(13).
    stop.
```

As seen in the potato-code above, this is an implementation of Fibonacci with Fib(0) and Fib(1) = 1. We are aware most people use Fib(0) = 0 and Fib(1) & Fib(2) = 1, but this is the way we had to implement it before, thus we will continue doing so.

```
1)  {-# LANGUAGE RecordWildCards #-}
2)  module Output.Fib where
3)  import Sprockell.System
4)  prog:: [Instruction]
5)  prog = [Const 3 RegA
6)          ,Compute Add RegA PC RegA
7)          ,Store RegA (Addr 0)            -- start of Fib is at line 9
8)          ,Jump (Rel 86)                  -- skip task Fib: line 94
9)          ,Pop RegA
10)         ,Store RegA (Addr 1)            -- arg is popped from stack and
    assigned to n
11)         ,Const 0 RegA
12)         ,Store RegA (Addr 2)            -- k = 0
13)         ,Load (Addr 1) RegA             -- RegA <-- n
14)         ,Const 0 RegB                   -- RegB <-- 0
15)         ,Compute Equal RegA RegB RegA   -- RegA <-- n == 0
16)         ,Load (Addr 1) RegB             -- RegB <-- n
17)         ,Const 1 RegC                   -- RegC <-- 1
18)         ,Compute Equal RegB RegC RegB   -- RegB <-- n == 1
19)         ,Compute Or RegA RegB RegA      -- RegA <-- n == 0 || n == 1
20)         ,Const 1 RegB
21)         ,Compute Xor RegA RegB RegA     -- RegA <-- !(n == 0 || n == 1)
22)         ,Const 5 RegB
23)         ,Compute Add RegB PC RegB       -- RegB <-- instruction line 28
    (otherwise)
24)         ,Branch RegA (Ind RegB)         -- go to otherwise if the
    expression is false
25)         ,Const 1 RegA
26)         ,Store RegA (Addr 2)            -- k = 1
27)         ,Jump (Rel 63)                  -- skip otherwise
28)         ,Load (Addr 3) RegA
29)         ,Push RegA
30)         ,Load (Addr 2) RegA
31)         ,Push RegA
32)         ,Load (Addr 1) RegA
33)         ,Push RegA
34)         ,Load (Addr 0) RegA
35)         ,Push RegA                      -- push known addresses
36)         ,Const 8 RegA
37)         ,Compute Add RegA PC RegA
38)         ,Push RegA                      -- push return address: 45
39)         ,Load (Addr 1) RegA
40)         ,Const 1 RegB
```

```
41)         ,Compute Sub RegA RegB RegA
42)         ,Push RegA                        -- push argument n-1
43)         ,Load (Addr 0) RegA
44)         ,Jump (Ind RegA)                  -- jump to task Fib
45)         ,Pop RegA                         -- RegA <-- return value
46)         ,Pop RegB
47)         ,Store RegB (Addr 0)
48)         ,Pop RegB
49)         ,Store RegB (Addr 1)
50)         ,Pop RegB
51)         ,Store RegB (Addr 2)
52)         ,Pop RegB
53)         ,Store RegB (Addr 3)              -- pop known addresses and
    restore them
54)         ,Store RegA (Addr 3)              -- l = return value
55)         ,Load (Addr 4) RegA
56)         ,Push RegA
57)         ,Load (Addr 3) RegA
58)         ,Push RegA
59)         ,Load (Addr 2) RegA
60)         ,Push RegA
61)         ,Load (Addr 1) RegA
62)         ,Push RegA
63)         ,Load (Addr 0) RegA
64)         ,Push RegA                        -- push known addresses
65)         ,Const 8 RegA
66)         ,Compute Add RegA PC RegA
67)         ,Push RegA                        -- push return address line 74
68)         ,Load (Addr 1) RegA               -- RegA <- n
69)         ,Const 2 RegB                     -- RegB <- 2
70)         ,Compute Sub RegA RegB RegA       -- RegA <- n-2
71)         ,Push RegA                        -- push argument n-2
72)         ,Load (Addr 0) RegA
73)         ,Jump (Ind RegA)                  -- jump to task Fib
74)         ,Pop RegA                         -- pop return value
75)         ,Pop RegB
76)         ,Store RegB (Addr 0)
77)         ,Pop RegB
78)         ,Store RegB (Addr 1)
79)         ,Pop RegB
80)         ,Store RegB (Addr 2)
81)         ,Pop RegB
82)         ,Store RegB (Addr 3)
83)         ,Pop RegB
84)         ,Store RegB (Addr 4)              -- pop and restore known
    addresses
85)         ,Store RegA (Addr 4)              -- store return value
86)         ,Load (Addr 3) RegA               -- RegA <-- l
87)         ,Load (Addr 4) RegB               -- RegB <-- r
88)         ,Compute Add RegA RegB RegA       -- l+r
89)         ,Store RegA (Addr 2)              -- k = l+r
90)         ,Load (Addr 2) RegA               -- RegA <-- k
91)         ,Pop RegB                         -- RegB <-- return address
92)         ,Push RegA                        -- push return value k
93)         ,Jump (Ind RegB)                  -- jump to return address
94)         ,Load (Addr 1) RegA               -- AFTER task Fib
95)         ,Push RegA
96)         ,Load (Addr 0) RegA
97)         ,Push RegA                        -- push all known addresses
98)         ,Const 6 RegA
99)         ,Compute Add RegA PC RegA
100)         ,Push RegA                        -- push return address line 105
101)         ,Const 4 RegA
102)         ,Push RegA                        -- push argument 4
103)         ,Load (Addr 0) RegA
104)         ,Jump (Ind RegA)                  -- jump to task Fib
105)         ,Pop RegA
106)         ,Pop RegB
```

```
  ,Store RegB (Addr 0)
  ,Pop RegB
  ,Store RegB (Addr 1)
  ,Store RegA (Addr 1)
   ,EndProg]
main = run 1 prog >> putChar '\n'
```

This code was generated using the compile function in the Main file:

compile “input/semantic/fib.txt”

This generates the given output file in the folder output. Since a second call of the function would overwrite the file, it was copied to the Examples folder where the exact file can be found. This file can be run, however, a debug function or some other form of output checking will be required. Instead of running this file, the testSemantic function in testing can be used:

testSemantic “input/semantic/fib.txt”

This function checks if the value stored at local memory address 1 is the same as the one hard coded in the function testSemanticTest. This value will have to be edited in order to reproduce our test, along with the value in the fib.txt file. Below a table with a couple of our test results can be found:

n	Function call in .txt	Value testSemanticTest	Pass/Fail
1	Fib(1)	1	Pass
2	Fib(2)	2	Pass
3	Fib(3)	3	Pass
4	Fib(4)	5	Pass
7	Fib(7)	21	Pass
13	Fib(13)	377	Pass
13	Fib(13)	376	Fail
13	Fib(13)	255	Fail

As seen in the table above, it doesn’t work to just put in random values in the testSemanticTest, the test will only pass if the correct value is inserted.