Programming Paradigms

Integration project

Pickle Cannon Programming Language

Done by:

Karolis Butkus s2700603

Group 14

2021

Table of Contents

1. Summary

Pickle Cannon is a simplistic programming language mostly intended for simple mathematical or logical calculations. The main features of the language are discussed below.

Firstly, language supports three data types in total. Two basic types - integers and booleans, and one compound type – array. Arrays can be only one-dimensional and store the values of one of the basic types. Language is strongly typed, thus, before each new declaration of the variable its type must be specified. Each declaration of the variable does not require a programmer to specify an initial value and it is assigned by default if it was not specified. *Pickle Cannon* also supports local and nested scopes which allow a programmer to re-declare variables with the same name in the newly opened scope.

Secondly, language supports simple mathematical and logical expressions. Addition, subtraction, negation, multiplication, soft-division and comparisons are all possible arithmetic operations that can be applied to integers. Logical negation, logical AND, logical OR and equality/inequality are all possible logical operations that can be applied to booleans (equality/inequality to all types).

Thirdly, language supports program control flow constructs. These two constructs are *if* and *while* statements. *if* construct may consist only of *if* statement or of *if-else* statement. *while* cycle will be executed until the condition is met and does not support any cycle-ending commands like a *break* or *continue*.

Fourthly, language supports simplistic concurrency mechanisms. *Pickle Cannon* allows a programmer to spawn and join threads using fork/join construct. Also, language syntax allows declaring shared variables that can be accessed across multiple threads. Moreover, language has one global lock which can be used to make changes to a shared object in a concurrently safe manner.

Lastly, language supports procedures. All procedures are declared before the main body, which in *Pickle Cannon* language starts with the *cannon* keyword. All procedures in the *Pickle Cannon* language are called *pickles*, thus the name of the language. Even though procedures must be declared before the main body, they still can call other procedures even if they are declared below them. As it may already be clear, language supports only procedures, so it is not possible to return value to the caller.

These are all main features supported by the *Pickle Cannon* language, which are discussed in greater detail in the *Detailed language description* section.

2. Problems and solutions

During the project, there were 5 main encountered problems. All of them are discussed below.

Concurrency

The first encountered problem was the management of shared data. To make type checking simpler, the definition of shared variables was embedded into the syntax of the language, so that all shared variables must be declared with the keyword 'shared'. But then another problem arose – what to do with the re-declaration of a shared variable inside the forked thread, how long should it live and so on. Thus, the restriction was imposed that shared variables can be declared only in the global outer scope of the main body. Another encountered problem was the synchronization of the threads in the Sprockell. The main question was how to start and stop threads from executing. This problem was solved by acquiring the maximum number of concurrently executing threads during the elaboration phase and then allocating one memory unit in shared memory space for each thread synchronization. This way each thread could know if others are still executing, and also if they are waiting, this space would be used to pass the number of their next instruction (to start a thread).

Memory management

Since *Pickle Cannon* language supports arrays, memory management complications were encountered. Arrays can take up a varying amount of space, thus storing the whole array in the registers was not the option. To solve this, the taken approach was to push all array values on the stack so that another procedure could then just pop the values. However, this means that during the execution of the program involving arrays, memory can get quite filled up.

Register allocation

One major difference from laboratory exercises with ILOC is the fact that register number is limited, thus careful register management is needed. One of the major difficulties is the fact only 6 registers are available for general use, and due to the fact that *Pickle Cannon* language supports procedures one register was needed to store ARP, thus only 5 general use registers were left. So to make sure that all calculations were possible each expression calculation after using the needed registers would free them as early as possible so that the registers limit would not be reached. The most demanding operations were soft-division and array storing. For example, a division operation would use up 4 registers as it would need 2 registers for expressions, 1 register for result accumulator and 1 for general values (such as storing comparison values or offsets). That is why after this operation is done, 3 registers must be freed immediately (1 register must remain to store the value) to allow other operations to execute normally.

Arrays

There were two encountered problems with the arrays. The first problem was array storing. As the array is a compound type it does not have any predefined size as it depends on the number of stored values. So to make the compilation and type checking process easier, language enforces the user to declare the size of each array (even in the procedure parameter definition) that must remain the same during the whole execution. Also, due to time limitations and trying to keep code cleaner and more understandable, multi-dimensional arrays were omitted. The second encountered

problem was the run-time errors of accessing array values out of bounds. Since the exception handling mechanism was not implemented, the taken approach is similar to the C language - it is to inform the programmer to carefully access the array values, he is intending to.

Procedures

The last encountered problem was concerning procedures. Due to the fact that procedures can call other procedures, it meant that procedure call type check could be executed only after all other elaboration steps have been done. The approach was to store all procedure calls in the list, and at the end of the elaboration phase check if they try to access the exiting procedures with correct parameters (language does not support nested procedures so all procedures are visible from the global scope).

3. Detailed language description

Before analyzing all language features in detail some it is worth knowing some basic language structure composition. *Pickle Cannon* language files must consist of the main body which starts with a keyword ‘cannon’ and opens/closes the scope with a braces ‘{‘/’}’. If the program does have the procedures, they are declared before the main body. More details about basic concepts can be seen in Table 1.

Table 1. Overall language structure

```
/* Procedure declarations go there */
cannon {
/* Main body statements go here */
/* Each statements ends with a semicolon except for those statements
that open a new scope. Those are the statements that end with opening
and closing braces ‘{‘/’}’*/
/* All keywords: int, bool, while, if, cannon, pickle, fork …, can be
declared in both uppercase and lowercase letters as long as keyword is
matched (‘Int’, ‘BooL’ are acceptable keywords) */
/* Comments start and end with slash ‘/’ and star ‘*’ symbols */
}
```

Basic types

As mentioned earlier *Pickle Cannon* language supports two basic types: integers and booleans that can be used in various calculations.

Syntax

Basic type variables are declared as follows. Firstly, the type is declared: *int* or *bool*. Secondly, if the variable is shared, the keyword ‘shared’ is added. Thirdly, the variable identifier is specified - identifiers start with any letter and then zero or more letters/numbers follow. Lastly, the variable can be initialized with a custom starting value that is assigned using ‘=’ sign and the expression, or this assignment can be omitted and then the variable is initialized with a default value: for integers it is *0* and for booleans it is *false*. The statement must end with a semicolon ‘;’. Example declaration can be seen in Table 2.

Table 2. Basic types example declarations

```
int a; /* Creates an integer ‘a’ with default value 0 */
bool b; /* Creates a boolean ‘b’ with default value false */
int a1 = 10; /* Creates an integer ‘a1’ with the value 10 */
bool bTrue = true; /* Creates a boolean ‘bTrue’ with value true */
int shared c = 3; /* Creates a shared integer ‘c’ with the value 3 */
bool shared d = true; /*Creates shared boolean ‘d’ with value true */
int a1exp = 5*10-3+4; /* Creates an integer ‘a1exp’ with value 51 */
bool e = false||(true!=false); /*Creates boolean e with value true*/
```

Usage

Basic types are one of the most important building blocks for all other remaining features as by manipulating these data types, results of various calculations can be achieved. For mathematical calculations integer type is used, for logical conditions boolean type is used. These basic types also are used in procedure parameters declaration. How they can be used is already shown in Table 2, but they can be used in many other ways as can be seen in later sections concerning expressions, assignments, arrays, control flow constructs, procedures.

Semantics

Basic types specify the type of the variable, which helps to ensure that language is strongly typed. These types act as a guarantee that executed calculations are performed with the expected types and undefined behavior such as *2+false* is avoided. If an undefined type would be encountered compiler would detect it during the elaboration phase and throw an error.

Code generation

Integers are generated as simple numerical values that are loaded using immediate value instruction. Booleans in the Sprockell are encoded also using numerical values: 0 – for false and 1 – for true. An example of code generation can be seen in Table 3. Here is one generation for the variable of integer type, and one for the variable of boolean type. For the integer variable, the first instruction loads the immediate value of 10 into the register and later three instructions are used to calculate the offset in the ARP and store the variable. For the boolean variable, the first instruction loads the immediate value of 1 which means 'true' and the later three instructions calculate the offset and store the variable.

Table 3. Basic types example code generation

cannon {
  int a = 10;
  bool b = true;
}
	[--other instructions
	, Load (ImmValue (10)) regB
	, Load (ImmValue (0)) regC
	, Compute Sub regA regC regC
	, Store regB (IndAddr regC)
	, Load (ImmValue (1)) regB
	, Load (ImmValue (1)) regC
	, Compute Sub regA regC regC
	, Store regB (IndAddr regC)
	, -- other instructions]

Arrays

Pickle Cannon language supports one compound type – array.

Syntax

Arrays are declared similar to the basic types, only after the identifier the square brackets with the size follow. Primitive array values are specified between square brackets separated by commas. Individual values can be accessed by specifying the index of the already declared array (starting from 0). The example syntax can be seen in Table 4.

Table 4. Array example syntax

```
int a[3]; /*Creates int array with 3 elements of default value 0*/
bool b[2];/*Creates bool array with 2 elements of default value false*/
int c[2] = [4,5]; /*int array with 2 elements: 4 and 5*/
bool d[2] = [true,false];/*bool array with 2 elements: true and false*/
int shared e[5]=[3,2,0*6,10,-1]; /*shared int array with 5 elements*/
bool shared f[4]; /* shared bool array with 4 default bool values*/
print(e[3]); /* prints the 4 value value of array e – 10 */
print(e[1+1]); /* prints the 3 value of array e – 0 */
print(e); /* prints all array e values */
```

Usage

Arrays can be used to store collections of the same basic type values so that multiple variable creations can be avoided. However, due to time limitations, only one-dimensional arrays have been implemented and it is not possible to store arrays inside the array. Another imposed restriction is that array size must be specified during the variable declaration and it cannot be altered during the execution of the program. Array size must be a number (not an expression) and it must be larger than 0 because the array of size 0 or negative size would not make sense. Lastly, programmers should carefully assess the array's values that they are trying to get as there is no run-time protection for out-of-bounds access.

Semantics

Arrays similarly to basic types are mostly used in various mathematical and logical calculations. Most operations can be only on the individual array elements, as only equality/inequality and print operations can be used on the array as a whole. As the size of the array is known at the compile-time arrays are stored as the contiguous list in the data area of the ARP (or in shared memory if the array is shared), where the offset to the array points to the first value of the array. If primitive array declarations are used, values of that array are temporally pushed on the stack, so that the other method then can pop them. This of course temporally double arrays storing in the memory.

Code generation

Array code generation is mostly done in cycles so that instruction count would not depend on the array size (exception is primitive array declaration like [4,5,-10]). An example of code generation can be seen in Table 5. There are 3 types of generations in this example: default initialization, initialization with custom primitive values, and array index access.

For the default initialization first two instructions calculate the address of the first array value in the ARP, then array size -1 is loaded and the cycle is created that executes until all values are assigned. In the body of the cycle, the default value of 0 is assigned to the current array value pointer and then the pointer is moved by 1 to point to the next value.

For the initialization with the primitive values, firstly all those values have to be pushed on the stack. Thus, the first instructions calculate the values of the expressions and push them on the stack. Then, because values are pushed in the reverse order, instructions that calculate the address

of the last element are executed and the cycle to assign values is started. During the cycle, values are popped from the stack and the pointer is moved to the next element.

For the index access, the first instructions calculate the index value, then the address of the first array element is calculated. The index is subtracted from the array address (because the program uses a stack to store ARP that grows from maximum address) and the pointer to wanted value is achieved. Then the value is loaded to the register and the last instruction prints the value.

Table 5. Array code generation example

cannon {
bool a[3];
int b[3] = [4,5,-10];
print(b[1]);
}
	[-- other instructions
	, Load (ImmValue (0)) regC
	, Compute Sub regA regC regC
	, Load (ImmValue (2)) regB
	, Compute Sub regC regB regC
	, Compute Lt regB reg0 regD
	, Branch regD (Rel (5))
	, Store reg0 (IndAddr regC)
	, Compute Incr regC reg0 regC
	, Compute Decr regB reg0 regB
	, Jump (Rel (-5))
	, Load (ImmValue (4)) regB
	, Push regB
	, Load (ImmValue (5)) regB
	, Push regB
	, Load (ImmValue (10)) regB
	, Load (ImmValue (-1)) regC
	, Compute Mul regC regB regC
	, Push regC
	, Load (ImmValue (3)) regD
	, Compute Sub regA regD regD
	, Load (ImmValue (2)) regB
	, Compute Sub regD regB regD
	, Compute Lt regB reg0 regC
	, Branch regC (Rel (6))
	, Pop regC
	, Store regC (IndAddr regD)
	, Compute Incr regD reg0 regD
	, Compute Decr regB reg0 regB
	, Jump (Rel (-6))
	, Load (ImmValue (1)) regB
	, Load (ImmValue (3)) regC
	, Compute Add regC regB regC
	, Compute Sub regA regC regC
	, Load (IndAddr regC) regC
	, WriteInstr regC numberIO

	, -- other instrunctions]

Assignments

In *Pickle Cannon* language assignments are very similar to most languages where each declared variable can get a new value.

Syntax

Assignment syntax is really simple. Firstly, the name of the variable is specified and then the assigned value is given. An example can be seen in Table 6.

Table 6. Example syntax of assignments

```
a = 3;
/* 3 is assigned to previously declared int variable a */
b = false;
/*false is assigned to previously declared bool variable b*/
c[2] = 4;
/*4 is assigned to the 3 element of previously declared int array c */
c = [2,3,1];
/* [2,3,1] is assigned to the previously declared int array c */
```

Usage

The main requirement for all the assignments is that the variable must be previously declared and the value that is being assigned to is of the same type. For example, it is not allowed to assign a boolean value to an integer variable or assign an integer array of 2 elements to an integer array of 3 elements. However, as mentioned earlier, the programmer needs to be careful with specifying array index as run-time protection for out-of-bounds access is not implemented.

Semantics

Assignments are very useful when the variable value needs to be adjusted or overwritten. Each assignment firstly calculates the expression value that will be assigned, then loads the variable address in the memory and overwrites the currently stored value.

Code generation

Assignment code generation consists of two steps: calculation of the expression and the value storing in the memory. The example of generated code can be seen in Table 7. There are two different assignments: one for basic type and one for the array. For basic types instructions are quite simple as firstly the expression value is calculated (more about expression calculation in the Expressions section), then the address where a variable should be stored is calculated and the last instruction simply stores the value in the memory. For array assignments, the process is a bit more complicated. Firstly, the expression values are calculated and pushed on the stack. Then the address of the last array element is calculated and the cycle is started. During the cycle, values are popped and stored in the according to memory location.

Table 7. Assignment code generation example

cannon {
int a;
bool b[3];
a=2*8;
b = [true,false,true];
}
	[-- other instructions
	, Load (ImmValue (2)) regB
	, Load (ImmValue (8)) regC
	, Compute Mul regB regC regB
	, Load (ImmValue (0)) regC
	, Compute Sub regA regC regC
	, Store regB (IndAddr regC)
	, Load (ImmValue (1)) regB
	, Push regB
	, Push reg0
	, Load (ImmValue (1)) regB
	, Push regB
	, Load (ImmValue (1)) regB
	, Compute Sub regA regB regB
	, Load (ImmValue (2)) regC
	, Compute Sub regB regC regB
	, Compute Lt regC reg0 regD
	, Branch regD (Rel (6))
	, Pop regD
	, Store regD (IndAddr regB)
	, Compute Incr regB reg0 regB
	, Compute Decr regC reg0 regC
	, Jump (Rel (-6))
	, -- other instructions]

Expressions (with soft-division)

As in most programming languages, *Pickle Cannon* supports simple mathematical and logical expressions.

Syntax

Expression syntax is similar to most languages. Mathematical operators are as follows: addition '+', subtraction and negation '-', multiplication '*', division '/' and parenthesis are '(' and ')'. Logical operators are as follows: logical AND '&&', logical OR '||' and logical negation '!'. Lastly, there are comparison operators are: equality '==', inequality '!=', greater than '>', greater than or equal '>=', less than '<' and less than or equal '<='. Besides all the operators, primitive values and variables can be used. An example of syntax can be seen in Table 8.

Table 8. Example expression syntax

```
int a = -2+(3-2)*5-4/2; /* Expression result is 1 */
bool b = true||false; /* Expression result is true */
bool c = true&&false; /* Expression result is false */
bool d = !true==false; /* Expression result is true */
int e = [2,3,1];
```

```
int f = [2,3,2];
bool g = e==f; /* Expression result is false */
bool h = e==[2,3,1]; /* Expression result is true */
bool i = e[2]>=1; /* Expression result is true */
```

Usage

All mathematical operators can be used with integers, and logical operators can be used with booleans. Equality and inequality operators can be used with any type, whereas greater/less comparisons can be only used with integers. Parenthesis also can be used with any type, however parenthesis after the identifier means a procedure call (like 'p1()') and not any arithmetic or logical operation. As mentioned above language uses division, the division is implemented using cycle and subtraction, thus there is no protection for division by 0 which results in the infinite cycle. That is why it is recommended to make sure that expressions do not contain division by 0.

Semantics

Expressions allow performing various mathematical or logical calculations whose results can be stored in the variables or used in the condition statements. Mathematical expressions are executed according to mathematical laws where multiplication/division is executed before addition or subtraction and parenthesis have the highest precedence. Lastly, language supports soft-division, thus only the integer part of the real number is returned. For example, 4/3 equals 1 and -5/2 equals -2.

Code generation

Expression code generation mostly consists of two steps: calculating the values of child expressions and applying operation on those values. In Table 9 there is a given example of generated code for 5 different expressions.

The first two expressions are relatively simple as they only perform one operation on integers. Both instructions firstly load the number values in the registers, then calculate the appropriate operation and store the value in the specified address.

The third instruction is a lot more complicated compared to the previous ones as few evaluations are needed to be done first. Firstly, both numbers are loaded into the registers. Then the evaluation about the sign of the end result is made. Each number is inspected if it is negative or not and if it is negative, it is multiplied by -1 to become positive. Then if both of them are positive or negative the end result will be positive and 1 is pushed onto the stack, if they are of different signs then -1 is pushed onto the stack. After this evaluation is done, a cycle is started that decreases the dividend by the divisor value until the dividend becomes smaller than the divisor. Lastly, the accumulated value is multiplied by the number that was pushed on the stack (1 or -1) and the result is saved at the specified address.

The fourth expression demonstrates a boolean operation. It is really simple, as it loads true value (as 1), uses zeros register for false value and calculates the result, which is later stored at the specified address.

The fifth expression is also quite complicated as it has to compare array values individually. Firstly, both arrays are pushed on the stack. Then the cycle is started where one value is retrieved by popping it from the stack and another is accessed by adding the size of the array to the stack pointer. All values are compared individually and added to the accumulator. In the end, the stack pointer is moved by array size because only one full array has been popped, and the result is stored in the specified address.

Table 9. Example code generation for expressions

```
cannon {
      int a = 2+3;
      int b = 3*8;
      int c = 5/2;
      bool d = true||false;
      bool e = [2,3,4]==[2,3,4];
}
```

```
[ -- other instructions
 , Load (ImmValue (2)) regB
 , Load (ImmValue (3)) regC
 , Compute Add regB regC regB
 , Load (ImmValue (0)) regC
 , Compute Sub regA regC regC
 , Store regB (IndAddr regC)
 , Load (ImmValue (3)) regB
 , Load (ImmValue (8)) regC
 , Compute Mul regB regC regB
 , Load (ImmValue (1)) regC
 , Compute Sub regA regC regC
 , Store regB (IndAddr regC)
 , Load (ImmValue (5)) regB
 , Load (ImmValue (2)) regC
 , Compute GtE regB reg0 regD
 , Branch regD (Rel (3))
 , Load (ImmValue (-1)) regD
 , Compute Mul regB regD regB
 , Compute GtE regC reg0 regE
 , Branch regE (Rel (3))
 , Load (ImmValue (-1)) regE
 , Compute Mul regC regE regC
 , Compute Mul regD regE regD
 , Push regD
 , Load (ImmValue (-1)) regD
 , Compute Incr regD reg0 regD
 , Compute GtE regB regC regE
 , Compute Sub regB regC regB
 , Branch regE (Rel (-3))
 , Pop regE
 , Compute Mul regD regE regD
 , Load (ImmValue (2)) regB
 , Compute Sub regA regB regB
 , Store regD (IndAddr regB)
 , Load (ImmValue (1)) regB
 , Compute Or regB reg0 regB
```

	, Load (ImmValue (3)) regC , Compute Sub regA regC regC , Store regB (IndAddr regC) , Load (ImmValue (2)) regB , Push regB , Load (ImmValue (3)) regB , Push regB , Load (ImmValue (4)) regB , Push regB , Load (ImmValue (2)) regB , Push regB , Load (ImmValue (3)) regB , Push regB , Load (ImmValue (4)) regB , Push regB , Load (ImmValue (1)) regB , Load (ImmValue (3)) regC , Compute LtE regC reg0 regD , Branch regD (Rel (9)) , Pop regD , Load (ImmValue (2)) regE , Compute Add regSP regE regE , Load (IndAddr regE) regE , Compute Equal regD regE regD , Compute And regB regD regB , Compute Decr regC reg0 regC , Jump (Rel (-9)) , Load (ImmValue (3)) regC , Compute Add regSP regC regSP , Load (ImmValue (4)) regC , Compute Sub regA regC regC , Store regB (IndAddr regC) , -- other instructions]

Local, nested scopes

Pickle Cannon language also supports local and nested scopes.

Syntax

Local and nested scopes are declared each time opening and closing braces (‘{‘, ‘}’) combination is used. New scopes can be easily opened using this combination. Each procedure and the main body itself must use braces combination which by default opens a new scope (exception is for procedure parameters which are also considered to be part of the procedure main global scope). Language allows variables to be re-declared with the same name in different scopes. An example of scope opening and closing can be seen in Table 10.

Table 10. Local, nested scopes syntax example

```
cannon {
    int a=2;
    {
       bool a=true;
       print(a); /* Prints true */
    }
    print(a); /* Prints 2 */
    if(true){
          int a=3;
          print(a); /* Prints 3 */
    }
    else{
          int a=4;
          print(a); /* Prints 4 */
    }
    print(a); /* Prints 2*/
}
```

Usage

Scopes can be opened many times inside other scopes, the only requirement is that they also must be closed. Variables with the same name can be re-declared in different scopes (they can be of different types). If this is done, variables are thought to be two different instances and don't have anything common besides the name. If the variable is used in the assignment or expression, then the closest declaration (most inner declaration) is used. However, there is one restriction to prevent undefined behavior. The re-declaration after use (example can be seen in Table 11) is prohibited as it would result in quite strange behavior in the context of the scope. Lastly, there are restrictions regarding shared variables as they can be only declared in the main body global scope, but more about that in the Concurrency section.

Table 11. Forbidden declaration example

```
cannon {
    int a = 3;
    {
        a=2;
        bool a; /* This declaration is not allowed */
    }
}
```

Semantics

Local and nested scopes allow variable declaration in different levels which allows structuring code better. Scopes are executed sequentially and after the scope is closed all the declared data in

that scope is considered not to be used anymore. However, memory is still allocated in the ARP for all possible variable definitions and dynamic local data management is not supported.

Code generation

Opening and closing scopes do not result in any additional Sprockell instructions (procedure nesting is not supported), they are mostly used during the elaboration phase to type check every variable and calculate its offset in the memory.

Control flow constructs: *if* and *while*

Pickle Cannon language supports two control flow constructs: *if* and *while*.

Syntax

if and *while* statement syntax is similar to most languages. *if* statements start with the *if* keyword, then the condition between parenthesis and ends with opening/closing braces. Optionally, the *if* statement can have an *else* part, then after *if* braces *else* keyword follows and then another combination of braces. *while* statements start with a *while* keyword, then condition between parenthesis and ends with opening/closing braces. An example of syntax can be seen in Table 12.

Table 12. Example syntax for if and while statements

```
if(a>5){
    print(true); /* Prints true (in Sprockell 1) if a>5 */
}

if(b<0){
    print(true); /* Prints true (in Sprockell 1) if b<0 */
}
else{
    print(false); /* Prints false (in Sprockell 0) if b>=0 */
}

while(c>0){
    print(c); /* Prints c while c>0 */
    c=c-1;
}
```

Usage

As seen above if statements can be declared in two ways: with the *else* part or without it. *else if* functionality is not supported. All control flow constructs open new scopes as can be seen by their use of braces, thus any variable declared inside them is not visible in the outer scope. Both constructs require boolean value as the condition, otherwise the compiler throws an error.

Semantics

if and *while* statements are very useful when program order is concerned. *if* statements enter the *if* scope only when a certain condition is met, otherwise it continuous with the next operation or

enters the *else* scope if it was declared. w*hile* cycle executes its scopes operations while the condition is met, when it becomes false the cycle is stopped.

Code generation

An example code generation for if-else and while statements can be seen in Table 13. For *if* statements code consists of branching where the calculated value is compared to false and if it is equal, program counter jumps over the scope, otherwise enters it. For the *while* statement, the cycle is created. Firstly, the condition is evaluated and if it is met cycle body is entered, if it is not, then the body is jumped over. At the end of the cycle jump back to condition evaluation is implemented.

Table 13. Example if and while code generation

```
cannon {
      if(true){
            print(true);
      }
      else{
            print(false);
      }

      int c=3;
      while(c>0){
            print(c);
            c=c-1;
      }
}
```

```
[ -- other instructions
 , Load (ImmValue (1)) regB
 , Compute Equal regB reg0 regC
 , Branch regC (Abs (12))
 , Load (ImmValue (1)) regB
 , WriteInstr regB numberIO
 , Jump (Abs (13))
 , WriteInstr reg0 numberIO
 , Nop
 , Load (ImmValue (3)) regB
 , Load (ImmValue (0)) regC
 , Compute Sub regA regC regC
 , Store regB (IndAddr regC)
 , Load (ImmValue (0)) regB
 , Compute Sub regA regB regB
 , Load (IndAddr regB) regB
 , Load (ImmValue (0)) regC
 , Compute Gt regB regC regB
 , Compute Equal regB reg0 regC
 , Branch regC (Abs (38))
 , Load (ImmValue (0)) regB
 , Compute Sub regA regB regB
 , Load (IndAddr regB) regB
 , WriteInstr regB numberIO
 , Load (ImmValue (0)) regB
 , Compute Sub regA regB regB
 , Load (IndAddr regB) regB
 , Load (ImmValue (1)) regC
 , Compute Sub regB regC regB
 , Load (ImmValue (0)) regC
 , Compute Sub regA regC regC
 , Store regB (IndAddr regC)
 , Jump (Abs (18))
 , Nop
```

	, -- other instructions]

Concurrency

Pickle Cannon language allows programmer to implement simplistic threading possibilities.

Syntax

Pickle Cannon language uses the fork/join model to support the concurrency. A new thread is spawned using the *fork* keyword and the thread body is declared between the braces. Threads are joined using the *join* keyword. Threads that are joined are all the spawned children and their descendant threads. Threads can be nested and be spawned from already spawned threads using the same *fork* keyword. Critical section mechanism is provided using *sync* keyword and the critical instructions are specified between the braces. An example of concurrency syntax can be seen in Table 14.

Table 14. Example syntax for concurrency instructions

```
cannon{ /* Main thread */
   int shared a=3;
   fork{ /* Thread 2 */
       /* ... */
       fork{ /* Thread 3 */
          /* ... */
          fork{ /* Thread 4 */
            /* ... */
            sync{ /* Acquires the global lock */
               a=2; /* Rewrites shared value */
            }
          }
         /* ... */
         join; /* Joins all children and their descendant threads */
         /*Thread 3 and Thread 4 joined into Thread 3 */
       }
       /* ... */
   }
   join; /* Joins all children and their descendant threads */
   /*Thread 1,2,3 joined into Thread 1 (Thread 4 already joined 3)*/
   a=a+1;
   print(a);
   fork { /* Thread 2 */
      /* ... */
   }
}
```

Usage

Thread spawning can be done only in the main body and cannot be done inside the while loops, procedures, or if-else statements. Critical section mechanism *sync* allows only one thread to execute the code inside the *sync* scope and acts as a global lock. Individual locks are not implemented. Shared variables can be declared only in the main body most outer scope. This is done because all data is stored in the shared memory area and ambiguity between variable data names is wanted to be avoided, also as data is shared it is assumed it should be accessible by all threads.

Semantics

Spawned threads are executed concurrently and can do their code executions in parallel. When threads are joined, the thread that called the join waits until all its child and their descendants ended their execution before continuing with the next instruction. This allows a programmer to speed up the calculation process and join the threads only when it is needed. To share data between threads, shared memory is used. If a programmer wants to perform the update in the critical section, it can use a global *sync* lock that ensures that only one thread has access to it at the time.

Code generation

The example generated code can be seen in Table 15. During the elaboration phase compiler calculates the maximum number of concurrent threads and thus spawns all of them at the start as the Sprockell processor allows.

First instructions are used to set up procedure ARP for each thread. Then only the main thread starts its execution while other threads spin until they receive the instruction code in their reserved memory spot to jump to. The main thread firstly allocates the local data in the ARP (more about it in the Procedures section), then the following instructions save the data in shared memory for variable ‘a’. After that, the new thread is spawned by writing the instruction number for that thread’s specialized shared memory area. Then main thread jumps other spawned thread instructions and encounters join statement instructions which make the main thread spin until all children threads set their shared memory special area to 0 to indicate that they finished their execution.

The newly spawned thread firstly allocates its local data area in the ARP and then goes to its first instruction which is to spawn another thread. Thread writes the instruction address in the specified shared memory location and jumps over the new thread body. Then it enters global lock where the thread tries to change the global lock value from 0 to 1 to indicate that lock is taken. If the thread fails, the process is repeated until it successes.

The third thread firstly allocates its local data area in the ARP and then goes to its first instruction which is *sync* statement. Like the other thread, he tries to obtain the lock. When the thread ends it writes 0 to a special shared memory area to indicate that it finished and stops its execution.

Table 15. Concurrency example code generation

cannon {
  int shared a=3;
	prog = [
	  Jump (Abs (1))

```
      fork {
            fork {
                  sync {
                        a=a+1;
                  }
            }
            sync {
                  a=a+2;
            }
      }
      join;
      print(a);
}
```

```
 , Push regSP
 , Pop regA
 , Compute Decr regA reg0 regA
 , Branch regSprID (Rel (2))
 , Jump (Rel (6))
 , ReadInstr (IndAddr regSprID)
 , Receive regB
 , Compute Equal regB reg0 regC
 , Branch regC (Rel (-3))
 , Jump (Ind regB)
 , Load (ImmValue (0)) regB
 , Compute Sub regSP regB regSP
 , Load (ImmValue (3)) regB
 , WriteInstr regB (DirAddr (3))
 , Load (ImmValue (18)) regB
 , WriteInstr regB (DirAddr (1))
 , Jump (Abs (51))
 , Load (ImmValue (0)) regB
 , Compute Sub regSP regB regSP
 , Load (ImmValue (23)) regB
 , WriteInstr regB (DirAddr (2))
 , Jump (Abs (38))
 , Load (ImmValue (0)) regB
 , Compute Sub regSP regB regSP
 , TestAndSet (DirAddr (0))
 , Receive regB
 , Branch regB (Rel (2))
 , Jump (Rel (-3))
 , ReadInstr (DirAddr (3))
 , Receive regB
 , Load (ImmValue (1)) regC
 , Compute Add regB regC regB
 , Load (ImmValue (3)) regC
 , WriteInstr regB (IndAddr regC)
 , WriteInstr reg0 (DirAddr (0))
 ,   WriteInstr   reg0   (IndAddr
regSprID)
 , EndProg
 , TestAndSet (DirAddr (0))
 , Receive regB
 , Branch regB (Rel (2))
 , Jump (Rel (-3))
 , ReadInstr (DirAddr (3))
 , Receive regB
 , Load (ImmValue (2)) regC
 , Compute Add regB regC regB
```

```
 , Load (ImmValue (3)) regC
 , WriteInstr regB (IndAddr regC)
 , WriteInstr reg0 (DirAddr (0))
 ,   WriteInstr   reg0   (IndAddr
regSprID)
 , EndProg
 , ReadInstr (DirAddr (1))
 , Receive regB
 , ReadInstr (DirAddr (2))
 , Receive regC
 , Compute Or regB regC regB
 , Branch regB (Rel (-5))
 , ReadInstr (DirAddr (3))
 , Receive regB
 , WriteInstr regB numberIO
 , EndProg
 ]

main = run [prog,prog,prog]
```

Procedures

Pickle Cannon language also supports procedures, that can be called from the main body or other procedures.

Syntax

Procedures in the *Pickle Cannon* language are declared before the main body (*cannon* block). Each procedure declaration starts with a keyword *pickle* and then the procedure identifier follows (applies the same rules as for variable identifiers). After that, procedure parameters are specified between parenthesis. Each parameter has to have its type and identifier declared. Parameters are separated by the commas. Lastly, the procedure body is opened with braces. Procedure call consists of a procedure identifier and parameters values specified between the parenthesis. Procedure call values have to match the types of parameters specified in the procedure declaration. An example of syntax can be seen in Table 16.

Table 16. Example syntax of the procedures

```
pickle p1(){  /* Procedure p1 */
    print(1);
}

pickle p2(int a, bool b, int [3] c){ /* Procedure p2*/
    p1();
    if(a>c[1]&&b){
        print(2);
```

```
    }
    else{
        print(3);
    }
}

cannon {
    p2(3,false==true,[2,0,9]); /* p2 procedure call*/
}
```

Usage

As mentioned above procedures can be only declared before the main body. Also, procedures cannot have the same names, thus language does not support overriding procedures. Moreover, parameter types have to be strongly specified - it is not allowed to not specify the size of an array. *Pickle Cannon* supports only procedures, so no return values are produced. An important point is that each procedure creates its own activation record when it is called, so it is important to not create large recursive procedures that can very quickly fill up all the memory space due to multiple recursive calls. Each procedure ARP consists of the return address, caller's ARP and parameter, local data areas. The activation record structure can be seen in Table 17 (ARP points to the start of parameters area).

Table 17. Activation record structure

Local data area
Parameter area
Caller's ARP
Return address

Semantics

Procedures allow writing repetitive instructions only once, thus decreasing the size of code. Of course, procedures do not produce return values, thus mostly should be used for independent code blocks that do not need to return anything. Procedures are executed when they are called. After the call, the procedure body is executed with the given values and the procedure call is ended.

Code generation

Procedure code generation complications mostly lie in the precall, prologue, epilogue and postcall sections. An example of code generation can be seen in Table 18. Each program's first instruction is a jump to the main body, since procedures are declared before the main body. Then at the start of main body execution, the correct activation record pointer value is set in register A. It is done by retrieving the current stack pointer value, as AR allocation is stack-based. Then local data area is allocated for the main body by moving the stack pointer to the end of the local data area. This way all necessary preparations are done. When the procedure is called, the caller firstly executes a precall. Firstly, during precall return address and caller's ARP are pushed on the stack, then the parameter values are pushed and lastly, the ARP is moved to the start of the parameter area and

jump to the procedure start is executed. During the prologue, the procedure moves the stack pointer to allocate the memory for local variables. After that, the procedure body is executed. When the procedure ends its execution, the epilogue is started. During the epilogue, the procedure retrieves the caller's ARP, return address and moves the stack pointer to the end of the caller's local data area. Retrieved ARP is written into the ARP register (regA) and at the jump to return address is executed. Because the procedure does not return any value and cannot be used in any mathematical or logical expressions, no prologue is needed as no register saves or return value retrievals need to be done.

Table 18. Example code generation for the procedures

```
pickle p1(int a) {
      print(a+1);
}

cannon {
      p1(5);
}
```

```
prog = [
   Jump (Abs (16))
 , Load (ImmValue (0)) regB
 , Compute Sub regA regB regSP
 , Load (ImmValue (0)) regB
 , Compute Sub regA regB regB
 , Load (IndAddr regB) regB
 , Load (ImmValue (1)) regC
 , Compute Add regB regC regB
 , WriteInstr regB numberIO
 , Compute Incr regA reg0 regB
 , Compute Incr regB reg0 regC
 , Compute Add regC reg0 regSP
 , Compute Incr regSP reg0 regSP
 , Load (IndAddr regB) regA
 , Load (IndAddr regC) regC
 , Jump (Ind regC)
 , Push regSP
 , Pop regA
 , Compute Decr regA reg0 regA
 , Load (ImmValue (0)) regB
 , Compute Sub regSP regB regSP
 , Load (ImmValue (29)) regB
 , Push regB
 , Push regA
 , Compute Decr regSP reg0 regB
 , Load (ImmValue (5)) regC
 , Push regC
 , Compute Add regB reg0 regA
 , Jump (Abs (1))
 , EndProg
 ]
```

4. Description of the software

The whole compiler is written purely in Java, thus the whole project is a Java project where scanning and parsing are done by ANTLR tool and code generation is made using Java language. The project is divided into 7 packages in total. Each package is discussed below (all packages can be found in the *src* folder of the project).

Grammar package

This package contains two ANTLR files that describe the grammar of the language. In the file named *Vocab.g4* the language tokens are specified and in the file *PickleCannon.g4* the language context-free grammar rules are described.

Checker package

This package contains the classes needed to perform the compiler elaboration phase. Firstly, in the file *TypeKind.java* enumerated types kinds are specified (*INT, BOOL, ARRAY, PROC*). In the file *Type.java* possible types are specified with their methods (all method documentation can be found in generated JavaDoc resources that are located at the top project level). Then in the file *Scope.java* language scope class is defined. Each scope is used to track information about local data of each function, thread, shared memory region or main body. The scope contains information about the declared variables, their types, offsets and nested levels. The terminology may become confusing but scopes refer to the procedures scopes, main body scope, thread scopes and nested levels refer to opening/closing braces in the same main body or procedure scope (an example explanation can be seen in Table 19). In the file *SymbolTable.java* the symbol table is described. The symbol table contains information about opened scopes, the scope for shared memory region, and various methods to put or retrieve variables from the current scope. One of the most important files in this package is the *Result.java* class which stores the information retrieved during the elaboration phase. The class contains 4 parse tree properties to store information about parse tree node types, offsets, information are they shared, procedure local data size and a list to store thread call counts. However, the most important file in this package is *Checker.java* class which is responsible for the elaboration. *Checker* class is the tree listener that extends generated *PickleCannonBaseListener*. *Checker* class uses *SymbolTable* to keep track of already declared variables that allows to type check all variable uses and declarations to ensure that the program does not contain errors. If an error is detected, it is added to an error list and at the end of elaboration, all existing errors are thrown as *ParseException* class object (more about *ParseException* class in the Compiler package). Because language uses procedures *Checker* cannot evaluate function calls right away they are encountered. That is why the *Checker* class stores all these calls in a list as *FunctionCall* class objects (*FunctionCall* is an inner class of *Checker* class) that are type-checked at the end of elaboration.

Table 19. Difference between scope and nested level

```
pickle p1() /* p1 procedure scope */
{
  {
    /* p1 procedure nested level */
  }
```

```
}
cannon {   /* Main body scope */
  {
    /* Main body nested level */
  }
}
```

Generator package

This package contains all the classes needed to perform code generation after elaboration has been done. Firstly, the package contains 4 classes to describe possible Sprockell instruction argument types. These classes are as follows: *Addr.java* (describes address type arguments), *Operator.java* (describes possible mathematical and logical operators such as addition, subtraction and so on), *Reg.java* (describes register type arguments) and *Target.java* (describes jump target type arguments). All classes contain constructor methods to create their respective type objects and methods to retrieve their properties or print them as a string. *Operand.java* is an enumerator that lists all these possible types. Then there are 2 classes to describe Sprockell instructions. *OpCode.java* is an enumerator that lists all possible instructions and the types of arguments it receives. Class *Instr.java* is used to create Sprockell instructions, its constructor receives one of the operation codes from the *OpCode* enumerator and zero or more operands that are of the type listed in the *Operand* enumerator. *Instr* class also contains methods to retrieve these arguments and a method to print the instruction in Sprockell instruction format. Sprockell program itself is defined in *Program.java* class. This class contains methods to add new instructions or update them, also a method to write the program into the specified file in the output package. Lastly, the most important class of this file is a *Generator.java* class. This class is a tree visitor that extends *PickleCannonBaseVisitor* and is used to generate Sprockell code. *Generator* visits all tree nodes and generates the code as described in the Detailed language description code generation sections. For code generation, it takes full use of the *Result* class object that was created by the *Checker* class during the elaboration. To manage the available registers, the class uses an array of registers to indicate which registers are taken and which are free. Also, because there may be more than one thread a list of these register arrays is created. To make jumps possible instruction count variable is used to keep track of already inserted instructions. Lastly, all instructions are written to the *Program* class object which is the returned result of the code generation.

Compiler package

This package contains 3 files. *ParseException.java* is a class that defines *Exception* class object that should be thrown when the error during scanning, parsing or elaboration phases were encountered. *ErrorListener.java* is a class that defines error listener for the scanning, parsing and elaboration phases. This listener collects all the errors and has a method to throw all the errors if it has any. The last class is *Compiler.java*. This class is the main class to run the compiler. It has a main method in which an argument can be passed to compile the file and output the result in the output package.

Tests package

This package contains all JUnit4 tests for automatic testing and all sample testing programs. It is divided into 4 folders. One for syntax testing, one for context testing, one for semantic testing and one for test suite that runs all three types of tests. More about the testing can be found in the Test plan and results section.

Output and sample packages

Output package is a package that is used to contain all the compiled programs. All resulting test and *Compiler* class Sprockell files can be found here. The sample package is a convenience folder to store sample *Pickle Cannon* language programs that can be later compiled.

Other folders

There are 3 other folders that come with this project. The *lib* folder contains external JUnit4 and ANTLR libraries, a *generated-sources* folder that stores all ANLTR generated classes and a *doc* folder that stores generated JavaDoc documentation.

5. Test plan and results

Testing was done in 3 parts. Syntax testing, contextual testing and semantic testing. All tests are located in the *src/tests* package and are split into 3 parts.

Syntax testing

Syntax testing is done to evaluate the correctness of the scanner and parser. In this testing, all features syntax was tested to make sure that the compiler detects syntax errors such as incorrect identifier, keyword naming or wrongly structured statements. For convenience purposes two methods were created: *accepts* and *rejects*. *accepts* method expects the test program to pass and only then the test passes, while the *rejects* method expects the test program to fail and only then the test passes. All features were tested in short string lines of code as there was no need to have large test programs that contain one small syntax error which may not be very visible. Each feature was tested using correct and incorrect programs to eliminate the risk of the compiler accepting all programs, or rejecting all programs. Moreover, there were done a few tests to check the structure of the parse tree. These tests check if the generated parse tree has the expected shape. For these tests, some children's nodes were investigated and compared to the expected values. All these syntax tests can be found in the *src/tests/syntax/SyntaxTest.java* file. This file can be run as a JUnit4 test to re-run these syntax tests. The results of these syntax tests can also be found in the *Test_Results.pdf* file submitted with a project (located at the top level).

Contextual testing

Contextual testing is done to evaluate the correctness of the compiler elaboration phase. In this testing all features context was tested to make sure that the compiler detects context errors such as use before the declaration, double declaration variable declaration in the same scope, wrong initial values and so on. For convenience purposes, like in syntax testing *accepts* and *rejects* methods were created. All features were tested in short string lines of code as well as files with moderate-size programs were tested. This was done to firstly make sure that small feature errors were detected and then files were tested to make sure that errors do not go missing when the program is more complex. Each feature was tested using correct and incorrect programs to eliminate the risk of the compiler accepting all programs, or rejecting all programs. Tests mainly focused on the variable declaration, use, type, initial value rules, also on thread spawning rules. All these context tests can be found in the *src/tests/context/ContextTest.java* file. This file can be run as a JUnit4 test to re-run these context tests. Source code for the test programs is located in the *src/tests/context/testSources* folder. The results of context tests can also be found in the *Test_Results.pdf* file.

Semantic testing

Semantic testing is done to evaluate the correctness of the compiler at run-time. In this testing, all features semantic was tested to make sure that the compiler-generated code runs as expected. There are no tests to catch run time errors as functionality to throw them was not implemented. If the program manages to pass scanning, parsing and elaboration phases then the code will be generated and ran. For example, accessing values outside array bounds would not throw any error and this has to be prevented by a programmer (similar to C language). For convenience purposes, two methods were created: *check* and *runFile*. *check* method is created to check if the generated code

from the source program after running it would produce the expected output. *runFile* is a method that is used by *check* to run files. It creates a process that runs the generated code in the *ghc* and returns the output. Firstly, some simple features were tested using short lines of strings to make sure that essential features are generated as expected and later moderate programs with general algorithms were ran to make sure that the compiler is functioning correctly. Tested algorithms were: bank example for concurrency, Peterson's example, nested threads example, February days calculation, prime detection, array concurrent sum calculation, Fibonacci calculation, finding a maximum element in the array, checking if all boolean array elements are true. Lastly, two tests were done to check if the program enters an infinite loop when a loop with always true condition is created, or when division by 0 occurs. All these semantic tests can be found in the *src/tests/semantics/SemanticTest.java* file. This file can be run as a JUnit4 test to re-run these semantic tests (take in mind that semantic tests due to multiple file executions take about 1 minute to complete). However, it is important to note that infinite loop tests create *ghc* process that still executes after the test is done and this process has to be killed manually through the task manager on Windows, or terminal on Linux using the *kill* command. Source code for the test programs is located in the *src/tests/semantics/testSources* folder. The generated code can be found in the *src/output* folder. The results of semantic tests can also be found in the *Test_Results.pdf* file.

Automatic testing

There is a possibility to run all three parts of tests at once (takes about 1 minute due to multiple file executions in semantic testing). In the file *src/tests/combined/TestSuite.java* a JUnit test suite is created. This file itself is a JUnit test that runs all test classes specified in it. That is why running the *TestSuite.java* as JUnit test will run syntax, context and semantic tests at once. However, do not forget to kill infinite loop *ghc* processes spawned by the semantic test.

Testing conclusion

All tests that have been described above were run and all of them passed (see Figure 1), thus it provides more confidence that the compiler is functioning correctly as formal verification is impossible due to infinite possibilities of written code.

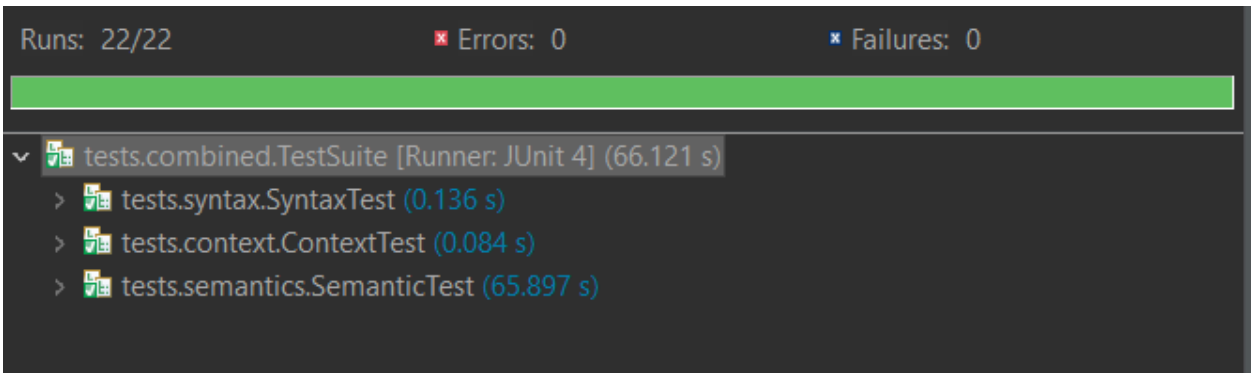

Figure 1. Testing results

6. Conclusions

Language evaluation

Pickle Cannon language in my personal opinion is quite simplistic, yet powerful language. In 3 weeks' time, I managed to define a language and write a compiler that supports mathematical and logical expressions, can use one-dimensional arrays, has procedures and supports concurrency. Of course, it would have been nicer if language would have an exception handling mechanism as now the programmer has to be careful when working with arrays, also next extension that language would mostly need would be support for characters and strings. However, language allows the program to write general algorithms and can be used as the language for moderate complexity problems. Writing this language helped me to better understand the whole process behind language definition and compilation as well as find out what complex problems need to be solved to write sophisticated, optimized compilers.

Module evaluation

Firstly, I would like to talk about the structure of the module. Personally, I think it would be better if the integration project would start earlier and would be done concurrently with the module. For example, after learning about scanning and parsing project could start where we firstly would define the grammar of our language, then after learning about elaboration we would continue with the elaboration phase of the project and so on. This way I think the information would remain fresher and less stressful work would need to be done at the end of the module. Also, I think this would help with the preparation for the compiler construction exams.

Secondly, I would like to talk about each module component individually.

Compiler construction was the module that I enjoyed the most. In this module, the presented information was well structured and quite easy to follow with the exception of the last block. In the last block, the explained function implementation is very theoretical and examples of practical implementation would be more appreciated. As at least to me, it took a whole weekend to understand how to write manual ILOC code for procedure implementation. And I think when you understand how to write that code manually it is not that difficult to write a program that does it automatically. But this block was the only exception, everything else I really enjoyed.

Functional programming was the module that was the hardest for me to overcome at the start. As I'm an exchange student it was my first time encountering a functional programming paradigm and as I understood Twente students have encountered functional programming in their first year. Thus, at the start, I had to complete exercises late in the evening to make sure I do not fall behind. But after the 1 and a half weeks I managed to keep up with a tempo. I think this paradigm is taught in a very short time and a lot of information is presented in one lecture. Thus, maybe throwing away the logic programming paradigm and extending functional programming to 4 weeks instead of 3 would be more beneficial. Laboratory exercises are not the simplest and teach you quite a lot about functional programming and I think it is a good sign as you become a lot more confident in your functional programming skills.

Concurrent programming was the module that I feel was the most theoretical one. During the lectures, plenty of different concepts were explained, but for me personally, it felt quite theoretical.

During labs, we would try to implement some of them, but most of the time they would be pretty simple and isolated. Personally, I would like to work more on general use programs that have to support concurrency and where the problems are not so obvious or implementation is more difficult.

Logic programming was one paradigm too much for the module. It was very short and not a lot of information was presented. Also, I did not really see the popular use for this paradigm and it felt as it was added to the module just to present that it exists. As I mentioned earlier, I think it would be better if instead of this paradigm the functional programming would be extended.

Lastly, the module as said at the start is work-intensive and it is very important to maintain the pace. In the tutorials at the start of the module, I worked with a partner and due to fact that it was hard to maintain the pace, I think he decided not to continue the module. However, this module taught me a lot about programming languages and different programming styles and I'm thankful for that. I think this knowledge is really useful to any programmer as it helps to understand the behind-the-scenes work of programming.

7. Appendices

Grammar specification

Table 20. Language grammar rules PickleCannon.g4

```
grammar PickleCannon;

import Vocab;

/** Program (procedures are declared before main program) */
program: proc* CANNON block EOF
       ;

/** Procedure declaration */
proc: PICKLE ID LPAR (pars (COMMA pars)* )? RPAR block
    ;

/** Procedure parameters */
pars : type ID                          #varPar
     | type ID LSQ NUM RSQ              #arrayPar
     ;

/** Block */
block: LBRACE stat* RBRACE
     ;

/** Statements */
stat: type (SHARED)? ID (ASSIGN expr)? SEMI                 #simpleVarStat
    | type (SHARED)? ID LSQ NUM RSQ (ASSIGN expr)? SEMI     #arrayVarStat
    | target ASSIGN expr SEMI                               #assignStat
    | IF LPAR expr RPAR block (ELSE block)?                 #ifStat
    | WHILE LPAR expr RPAR block                            #whileStat
    | FORK block                                            #forkStat
    | JOIN SEMI                                             #joinStat
    | SYNC block                                            #syncStat
    | block                                                 #blockStat
    | PRINT LPAR expr RPAR SEMI                             #printStat
    | ID args SEMI                                          #callStat
    ;

/** Target of an assignment */
target: ID                      #idTarget
      | ID LSQ expr RSQ         #arrayTarget;

 /** Arguments of a call. */
args: LPAR (expr (COMMA expr)*)? RPAR
    ;

/** Expression */
expr: prfOp expr                        #prfExpr
    | expr multOp expr                  #multExpr
    | expr plusOp expr                  #plusExpr
    | expr compOp expr                  #compExpr
    | expr boolOp expr                  #boolExpr
    | LPAR expr RPAR                    #parExpr
```

```
    | ID                                  #idExpr
    | NUM                                 #numExpr
    | TRUE                                #trueExpr
    | FALSE                               #falseExpr
    | ID LSQ expr RSQ                     #indexExpr
    | LSQ expr (COMMA expr)* RSQ          #arrayExpr
    ;

/** Prefix operator */
prfOp: MINUS | NOT;

/** Multiplicative operator */
multOp: STAR | SLASH;

/** Additive operator */
plusOp: PLUS | MINUS;

/** Boolean operator */
boolOp: AND | OR;

/** Comparison operator */
compOp: LE | LT | GE | GT | EQ | NE;

/** Data type */
type: INT  #intType
    | BOOL #boolType;
```

Table 21. Language vocabulary Vocab.g4 file

```
lexer grammar Vocab;


// Keywords
BOOL:   B O O L;
INT:    I N T;
ELSE:    E L S E ;
FALSE:   F A L S E ;
FORK:    F O R K;
IF:      I F ;
JOIN:    J O I N;
PRINT:   P R I N T;
PICKLE:  P I C K L E;
CANNON:  C A N N O N;
SHARED:  S H A R E D;
SYNC:    S Y N C;
TRUE:    T R U E ;
WHILE:   W H I L E ;



// Operators
AND:    '&&';
OR:     '||';
ASSIGN: '=';
```

```
COMMA:  ',';
EQ:     '==';
GE:     '>=';
GT:     '>';
LE:     '<=';
LBRACE: '{';
LPAR:   '(';
LSQ:    '[';
LT:     '<';
MINUS:  '-';
NE:     '!=';
NOT:    '!';
PLUS:   '+';
RBRACE: '}';
RPAR:   ')';
RSQ:    ']';
SEMI:   ';';
SLASH:  '/';
STAR:   '*';

// Content-bearing token types
ID: LETTER (LETTER | DIGIT)*;
NUM: DIGIT (DIGIT)*;


fragment LETTER: [a-zA-Z];
fragment DIGIT: [0-9];

// ignore whitespace
WS : [ \t\n\r] -> skip;
// ignore comments
COMMENT: SLASH STAR .*? STAR SLASH -> skip;

fragment A: [aA];
fragment B: [bB];
fragment C: [cC];
fragment D: [dD];
fragment E: [eE];
fragment F: [fF];
fragment G: [gG];
fragment H: [hH];
fragment I: [iI];
fragment J: [jJ];
fragment K: [kK];
fragment L: [lL];
fragment M: [mM];
fragment N: [nN];
fragment O: [oO];
fragment P: [pP];
fragment Q: [qQ];
fragment R: [rR];
fragment S: [sS];
fragment T: [tT];
fragment U: [uU];
fragment V: [vV];
```

```
fragment W: [wW];
fragment X: [xX];
fragment Y: [yY];
fragment Z: [zZ];
```

Extended test program

Table 22. Extended test program extendedProgram.pickle

```
pickle positiveArray(int a[6]){
	int i=0;
	while(i<=5){
		if(a[i]>0){
			print(a[i]);
		}
		i=i+1;
	}
}

cannon {
	int shared total = 0;
	int shared a[6] = [4,-2,1,-3,10,9];
	fork {
		int i=0;
		int sum = 0;
		while(i<3){
			if(a[i]>0){
				sum=sum+a[i];
			}
			i=i+1;
		}
		sync {
			total = total + sum;
		}
	}
	fork {
		int i=3;
		int sum =0;
		while(i<6){
			if(a[i]>0){
				sum=sum+a[i];
			}
			i=i+1;
		}
		sync {
			total = total + sum;
		}
```

```
      }
      join;
      positiveArray(a);
      print(total);
}
```

Table 23. Generated extended test program code extendedProgram.hs

```
import Sprockell

prog :: [Instruction]
prog = [
   Jump (Abs (52))
 , Load (ImmValue (6)) regB
 , Compute Sub regA regB regSP
 , Load (ImmValue (0)) regB
 , Load (ImmValue (6)) regC
 , Compute Sub regA regC regC
 , Store regB (IndAddr regC)
 , Load (ImmValue (6)) regB
 , Compute Sub regA regB regB
 , Load (IndAddr regB) regB
 , Load (ImmValue (5)) regC
 , Compute LtE regB regC regB
 , Compute Equal regB reg0 regC
 , Branch regC (Abs (44))
 , Load (ImmValue (6)) regB
 , Compute Sub regA regB regB
 , Load (IndAddr regB) regB
 , Load (ImmValue (0)) regC
 , Compute Add regC regB regC
 , Compute Sub regA regC regC
 , Load (IndAddr regC) regC
 , Load (ImmValue (0)) regB
 , Compute Gt regC regB regC
 , Compute Equal regC reg0 regB
 , Branch regB (Abs (34))
 , Load (ImmValue (6)) regB
 , Compute Sub regA regB regB
 , Load (IndAddr regB) regB
 , Load (ImmValue (0)) regC
 , Compute Add regC regB regC
 , Compute Sub regA regC regC
 , Load (IndAddr regC) regC
 , WriteInstr regC numberIO
```

```
 , Jump (Abs (34))
 , Nop
 , Load (ImmValue (6)) regB
 , Compute Sub regA regB regB
 , Load (IndAddr regB) regB
 , Load (ImmValue (1)) regC
 , Compute Add regB regC regB
 , Load (ImmValue (6)) regC
 , Compute Sub regA regC regC
 , Store regB (IndAddr regC)
 , Jump (Abs (7))
 , Nop
 , Compute Incr regA reg0 regB
 , Compute Incr regB reg0 regC
 , Compute Add regC reg0 regSP
 , Compute Incr regSP reg0 regSP
 , Load (IndAddr regB) regA
 , Load (IndAddr regC) regC
 , Jump (Ind regC)
 , Push regSP
 , Pop regA
 , Compute Decr regA reg0 regA
 , Branch regSprID (Rel (2))
 , Jump (Rel (6))
 , ReadInstr (IndAddr regSprID)
 , Receive regB
 , Compute Equal regB reg0 regC
 , Branch regC (Rel (-3))
 , Jump (Ind regB)
 , Load (ImmValue (0)) regB
 , Compute Sub regSP regB regSP
 , Load (ImmValue (0)) regB
 , WriteInstr regB (DirAddr (3))
 , Load (ImmValue (4)) regB
 , Push regB
 , Load (ImmValue (2)) regB
 , Load (ImmValue (-1)) regC
 , Compute Mul regC regB regC
 , Push regC
 , Load (ImmValue (1)) regB
 , Push regB
 , Load (ImmValue (3)) regB
 , Load (ImmValue (-1)) regC
 , Compute Mul regC regB regC
 , Push regC
 , Load (ImmValue (10)) regB
```

```
 , Push regB
 , Load (ImmValue (9)) regB
 , Push regB
 , Load (ImmValue (4)) regD
 , Load (ImmValue (5)) regB
 , Compute Add regD regB regD
 , Compute Lt regB reg0 regC
 , Branch regC (Rel (6))
 , Pop regC
 , WriteInstr regC (IndAddr regD)
 , Compute Decr regD reg0 regD
 , Compute Decr regB reg0 regB
 , Jump (Rel (-6))
 , Load (ImmValue (95)) regB
 , WriteInstr regB (DirAddr (1))
 , Jump (Abs (164))
 , Load (ImmValue (2)) regB
 , Compute Sub regSP regB regSP
 , Load (ImmValue (0)) regB
 , Load (ImmValue (0)) regC
 , Compute Sub regA regC regC
 , Store regB (IndAddr regC)
 , Load (ImmValue (0)) regB
 , Load (ImmValue (1)) regC
 , Compute Sub regA regC regC
 , Store regB (IndAddr regC)
 , Load (ImmValue (0)) regB
 , Compute Sub regA regB regB
 , Load (IndAddr regB) regB
 , Load (ImmValue (3)) regC
 , Compute Lt regB regC regB
 , Compute Equal regB reg0 regC
 , Branch regC (Abs (148))
 , Load (ImmValue (0)) regB
 , Compute Sub regA regB regB
 , Load (IndAddr regB) regB
 , Load (ImmValue (4)) regC
 , Compute Add regC regB regC
 , ReadInstr (IndAddr regC)
 , Receive regC
 , Load (ImmValue (0)) regB
 , Compute Gt regC regB regC
 , Compute Equal regC reg0 regB
 , Branch regB (Abs (138))
 , Load (ImmValue (1)) regB
 , Compute Sub regA regB regB
```

```
 , Load (IndAddr regB) regB
 , Load (ImmValue (0)) regC
 , Compute Sub regA regC regC
 , Load (IndAddr regC) regC
 , Load (ImmValue (4)) regD
 , Compute Add regD regC regD
 , ReadInstr (IndAddr regD)
 , Receive regD
 , Compute Add regB regD regB
 , Load (ImmValue (1)) regC
 , Compute Sub regA regC regC
 , Store regB (IndAddr regC)
 , Jump (Abs (138))
 , Nop
 , Load (ImmValue (0)) regB
 , Compute Sub regA regB regB
 , Load (IndAddr regB) regB
 , Load (ImmValue (1)) regC
 , Compute Add regB regC regB
 , Load (ImmValue (0)) regC
 , Compute Sub regA regC regC
 , Store regB (IndAddr regC)
 , Jump (Abs (105))
 , Nop
 , TestAndSet (DirAddr (0))
 , Receive regB
 , Branch regB (Rel (2))
 , Jump (Rel (-3))
 , ReadInstr (DirAddr (3))
 , Receive regB
 , Load (ImmValue (1)) regC
 , Compute Sub regA regC regC
 , Load (IndAddr regC) regC
 , Compute Add regB regC regB
 , Load (ImmValue (3)) regC
 , WriteInstr regB (IndAddr regC)
 , WriteInstr reg0 (DirAddr (0))
 , WriteInstr reg0 (IndAddr regSprID)
 , EndProg
 , Load (ImmValue (167)) regB
 , WriteInstr regB (DirAddr (2))
 , Jump (Abs (236))
 , Load (ImmValue (2)) regB
 , Compute Sub regSP regB regSP
 , Load (ImmValue (3)) regB
 , Load (ImmValue (0)) regC
```

```
 , Compute Sub regA regC regC
 , Store regB (IndAddr regC)
 , Load (ImmValue (0)) regB
 , Load (ImmValue (1)) regC
 , Compute Sub regA regC regC
 , Store regB (IndAddr regC)
 , Load (ImmValue (0)) regB
 , Compute Sub regA regB regB
 , Load (IndAddr regB) regB
 , Load (ImmValue (6)) regC
 , Compute Lt regB regC regB
 , Compute Equal regB reg0 regC
 , Branch regC (Abs (220))
 , Load (ImmValue (0)) regB
 , Compute Sub regA regB regB
 , Load (IndAddr regB) regB
 , Load (ImmValue (4)) regC
 , Compute Add regC regB regC
 , ReadInstr (IndAddr regC)
 , Receive regC
 , Load (ImmValue (0)) regB
 , Compute Gt regC regB regC
 , Compute Equal regC reg0 regB
 , Branch regB (Abs (210))
 , Load (ImmValue (1)) regB
 , Compute Sub regA regB regB
 , Load (IndAddr regB) regB
 , Load (ImmValue (0)) regC
 , Compute Sub regA regC regC
 , Load (IndAddr regC) regC
 , Load (ImmValue (4)) regD
 , Compute Add regD regC regD
 , ReadInstr (IndAddr regD)
 , Receive regD
 , Compute Add regB regD regB
 , Load (ImmValue (1)) regC
 , Compute Sub regA regC regC
 , Store regB (IndAddr regC)
 , Jump (Abs (210))
 , Nop
 , Load (ImmValue (0)) regB
 , Compute Sub regA regB regB
 , Load (IndAddr regB) regB
 , Load (ImmValue (1)) regC
 , Compute Add regB regC regB
 , Load (ImmValue (0)) regC
```

```
 , Compute Sub regA regC regC
 , Store regB (IndAddr regC)
 , Jump (Abs (177))
 , Nop
 , TestAndSet (DirAddr (0))
 , Receive regB
 , Branch regB (Rel (2))
 , Jump (Rel (-3))
 , ReadInstr (DirAddr (3))
 , Receive regB
 , Load (ImmValue (1)) regC
 , Compute Sub regA regC regC
 , Load (IndAddr regC) regC
 , Compute Add regB regC regB
 , Load (ImmValue (3)) regC
 , WriteInstr regB (IndAddr regC)
 , WriteInstr reg0 (DirAddr (0))
 , WriteInstr reg0 (IndAddr regSprID)
 , EndProg
 , ReadInstr (DirAddr (1))
 , Receive regB
 , ReadInstr (DirAddr (2))
 , Receive regC
 , Compute Or regB regC regB
 , Branch regB (Rel (-5))
 , Load (ImmValue (259)) regB
 , Push regB
 , Push regA
 , Compute Decr regSP reg0 regB
 , Load (ImmValue (0)) regC
 , Load (ImmValue (6)) regD
 , Compute GtE regC regD regD
 , Branch regD (Rel (8))
 , Load (ImmValue (4)) regD
 , Compute Add regD regC regD
 , ReadInstr (IndAddr regD)
 , Receive regD
 , Push regD
 , Compute Incr regC reg0 regC
 , Jump (Rel (-9))
 , Compute Add regB reg0 regA
 , Jump (Abs (1))
 , ReadInstr (DirAddr (3))
 , Receive regB
 , WriteInstr regB numberIO
 , EndProg
```

```
  ]

main = run [prog,prog,prog]
```