# PP Final Project

Bjorn Oude Roelink, S1988077
Gibson Vredeveld, S1944495

## Summary of the main features

Please note, features marked red are in the grammar, but not implemented in TypeChecker or SPRILGenerator (or both). This is because we would not get them to work (in time) for the delivery, so the decision was made to only do it if everything else was finished.

- Types: Integers, Booleans, Strings, Arrays of the previous types including nested arrays
- Variable declaration and assignment of the Types
- Comparisons with ==, !=, >=, <=, > and <
- Basic operations like addition, subtraction, and multiplication
- If statements with optional elif statements, and an optional else statement
- While loops
- For loops
- Function declarations with optional multiple input variables, and an optional return value
- Multithreading with Parallel and Sequential
    - The main thread is able to activate other threads but not yet working for nested parallels/ nested thread activation. We were not able to discover the source of this problem.
- Locking and unlocking of declared variables

## Problems and solutions

- **Problem**: Strings and arrays take more space than 4 bytes.
  **Solution**: Our solution to this problem would be to assign the literal calling to the variable. So Arr Int a = [5,6,7,8,9] would be implemented into the symboltables like a[1] with the heap location for 5 and a[2] with the heap location for 6. The thread starts at 1 to keep the first spot of the array as the size of the array. So a[0] would return 5. This way we can also lock the whole object by using the lock on the array. This way every element of the array has a specific lock and the array as a whole has a lock.

- **Problem**: How to implement multithreading in a way that is easy to use, and can be implemented.
  **Solution**: We solved this with the two keywords parallel and sequential. We first thought of making every line within the parallel block execute in parallel, with optional blocks of sequential to make multiple lines execute sequential on a separate thread. However, as this would mostly be useful when having access to methods, we decided to simplify our idea, and just make each sequential block inside a parallel block run on a separate thread.

- **Problem**: Some threads need access to variables that were declared in other threads
  **Solution**: For information on this solution, please see the “Sprockell intercommunication protocol” part later on in this report. But the short version is that the child thread already knows where the variable is stored and by which thread. So it makes an appropriately formatted call on the request variable channel.

- **Problem**: TypeChecker should be able to know the type inside arrays
  **Solution**: We decided to have the type in TypeChecker be implemented in the form of an integer array. The first integer declares the main type (Int, Bool, Str or Arr). If the main type is Arr, then the next indices of the array can be Arr, until an index in the array is Int, Bool or Str. Due to time constraints and other problems (see above) arrays are not fully implemented, so this solution did not really have to be implemented. However, since it is now, it will make implementing arrays later on be easier.

- **Problem**: Variable and function names could be overlapping
  **Solution**: We decided to have the variable names start with a lowercase letter, while function names start with a capital letter. This makes it so that the names cannot overlap, which would make checking types and syntax much easier. Since up to this point support for functions is not implemented, this solution was not necessary. However, it does mean that implementing functions later on will probably easier.

- **Problem**: variables should be able to be locked separately

**Solution**: For information on this solution, please see the “Sprockell intercommunication protocol” part later on in this report.

# Detailed language description

## Syntax and Usage

- **Types**: are always written with a capital as the first letter and shortened, e.g.
  *Int, Bool, Str, Arr Int, Arr Arr Int*, …
  Int consist of only digits, Bool is either true or false, Str elements are written with “ ” and arrays with [ ]. Strings cannot contain whitespace. Arrays cannot be empty, and each element in an array has to be of the same type. E.g.
  *12345, true, false, “string”, [1,2,3]*
  But not: *“this is not a string”, [ ], [1, false]*

- **Variable declaration and assignment**: variables are always declared with a type in front. Variable names consist of lower- or uppercase letters, and possibly include numbers, but are always started with a lowercase letter. Assignment is optional at declaration, but need to happen before usage of that variable. Assignment happens with the ‘=’ sign, with an element of the specified type afterwards. A variable declaration, assigned or not, always ends with a semicolon. E.g.
  Int i; Str s; Bool b; Arr Int;
  Int i = 17293;
  Str s = “string”;
  Bool b = true;
  Arr Int = [1,2,3];

- **Comparisons**: are written as *x comp y*, where x and y are elements of the same type, and comp is *<=, >=, <, >,* == or *!=*. Comparisons always result in a Bool. E.g.
  *4 <= 3*
  *“A” != “a”*

- **Basic operations**: are written as x sign y, where sign is a *+,* - or * sig, and x and y are Int, except for addition (concatenation) where x and y can also be Str. For x and y, both types need to be the same. E.g.
  *5 + 3*
  *“X” + “y”*
  *3 - 2*
  *3 * 2*
  But not: *“wontWork” + 1*

- **If statements**: are always started with *if,* with a single expression that results in a Bool. After the expression, a *{* , afterwards (optional) multiple lines (distinguishable by the semicolon ending each line) and then *a }* . After this, optional multiple *elif* can happen,

which are typed the same as if only with *elif* instead of *if*. After that, a single optional *else* can happen, which again is typed like if, but without the expression. If, elif and else are all lowercase. If statements are closed by the final curly bracket, not a semicolon. E.g.
*if x == 2 { (...) } elif x == 3 { (...) } elif x == 4 { (..) } else { (...) }*

- **While loops**: always start with *while*, followed by a single expression which results in a Bool, then a *{* , then (otional) multiple lines, closing with a *}* . While is all lowercase, ending in a closing curly bracket, so no semicolon. E.g.
*while x == y { (...) }*

- **For loops**: are somewhat different from if and while. For loops start with *for*, followed by either the name of an already declared and assigned Int variable, or declaring such Int variable according to the variable declaration and assignment syntax. After that, a semicolon, and then an expression which results in a Bool, followed by another semicolon. That is followed by an already declared but not necessarily instantiated variable which is assigned according to the assignment syntax, followed by yet another semicolon. Then a *{* , multiple (optional) lines, and finally a *}* . For is all lowercase letters, ending in a closing curly bracket, so no semicolon. E.g.
*for Int i = 0; i < 3; i = i+1; { (...) }*

- **Function declarations**: start with Fun, and then a return type according to the Types syntax. After that, the function name which consists of lower- or uppercase letters and or digits, but with a capital letter as the first character. After that, there are optional variable names with types in front following the Types syntax. Then an opening curly bracket, followed by one or multiple lines. Then a possible *return* with an expression after it. And finally a closing curly bracket to end the function declaration. Return is types in all lowercase letters, and there is no semicolon after the closing curly bracket. E.g.
*Fun Int returnzero Int i { (...) return 0; }*

- **Multithreading**: parallel is written with *parallel*, followed by a *{* , followed by at least 1 sequential block and then ending in a *}* . The syntax for sequential is the same, but with *sequential* instead of *parallel*. Both sequential and parallel are written in all lowercase letters, ending in a closing curly bracket, so no semicolon. E.g.
*parallel { (...) }*
*sequential { (...) }*

- **Locking and unlocking**: locking is done with *lock*, followed by an already declared variable which does not have to be instantiated, ending in a semicolon. Unlocking is done with *unlock*, followed by an already declared variable which does not have to be instantiated, ending in a semicolon. Lock and unlock are both all lowercase letters. E.g.
*lock x;*
*unlock x;*

## Semantics

- **Types**: these are the basic types that are supported by the language.
  These will be executed for Int as it's representation in 32 bits (4 bytes). Bool will be executed as a 0 or 1 for false and true respectively in bits.

- **Variable declaration and assignment**: allows variables with names to be declared and used in the language. Variables can be assigned, reassigned and used in places where an element of the same type as the variable would also be allowed.
  For information on the execution, please see the "Code Generation" part later on in this report.

- **Comparisons**: allow comparisons between variables for instance in arguments for if. This feature checks the two variables with the comparator, and makes a Bool from it, either true or false.
  These will be executed by the same comparators in SPROCKELL, where it will result in a 0 or 1 respectively.

- **Basic operations**: basic operations allow the addition, subtraction and multiplication of integers. Not implemented yet is the Str type, else the + operator would allow for string concatenation.
  For Int, the execution will happen based on the same operators in the SPROCKELL language. These will give a result which is again an integer, which can be used for comparison or stored in a variable again if the code specifies this.

- **If statements**: if statements allow for checking an argument, and then doing different operations based on if the result of that argument is true or false. If statements also allow for elif and else parts, which allow for even more elaborate checking and only doing certain operations only if (multiple) conditions do(n't) hold.
  These are executed in SPROCKELL with branches that lead to the different parts of the code if certain conditions (don't) hold.

- **While loops**: while loops allow for repeating certain operations as long as a given argument does (not) hold.
  These will be executed in SPROCKELL with branches, that will lead back to the start of the loop if the condition is still true, or to the code after the while loop if the condition is not true anymore.

- **For loops**: for loops allow for doing certain operations multiple times, possibly for an amount of times that is not known before starting the for loop (with a variable that is also edited inside the for loop).

These would work based on branches in SPROCKELL, which will loop to the code inside the for if the condition holds, or outside of the for if the condition does not hold anymore.

- **Function declarations**: function declarations allow for reusing whole blocks of code by assigning them a function name. This function can then later on easily be called to execute the code inside the function.
  Function declarations are not implemented at this moment, so no final information of how this would be executed is available.

- **Multithreading**: multithreading allows for the utilization of multiple hardware cores at the same time, and it allows multiple parts of the code to run at the same time. This makes it possible for certain parts of the code to split up calculations, and then return the result which can be combined again. This can speed up the code significantly.
  For information on the execution, please see the “Sprockell intercommunication protocol” part later on in this report.

- **Locking and unlocking**: locking and unlocking allows for a certain variable to be locked by a thread, after which other threads cannot access that variable at the same time. In our codegenerator we have explicit checks to enforce the release of a locked variable at the end of a sequential block/ threads lifetime.This means that the variable cannot be changed or read by other threads while one thread holds a lock on it, which prevents concurrency issues.
  For information on the execution, please see the “Sprockell intercommunication protocol” part later on in this report.

## Code generation

- **Types**:
  The only working type are integers and booleans. The value is stored on the first empty spot on the heap. Keeping track of the heaps happens with the store method. This method reserve a spot for the lock on a variable and stores the value itself after it.

- **Variable declaration and assignment**:
  - Declaration:
    When there is a declaration the generator reserves a spot for the variable on the heap.
  - Assignment:
    To make sure that the assignment is valid, in the sense that it has previously been declared, is checked by the symboltable of code generation. In case it is NOT a valid assignment it will throw runtime errors that try to explain the reason the assignment is invalid. For valid assignments it stores the value of the expression on the heap location of that specific variable, which is retrieved from the symbol table.

- **Comparisons:**
  To do the comparisons, e.g. greater than, less than, etc. We used its counterpart listed as operator to be used in compute instructions. The typechecker already makes sure that the expression is a valid comparison,

- **Basic Operations**:
  Same as for the comparisons. These operations could all be implemented with their counterparts in a compute operation.

- **If statements**:
  To generate if statements we created a not utility method which negates the value of a register. So, for the if and elif cases we negate the value so it can proceed if it is true and branch otherwise. If there is an else

- **While loops**:
  While loops are implemented in a similar vein to if statements. The condition will be checked. If it does not compute then there is a jump to over the while loop. At the end of the body of the while loop there is a jump back to the start loop.

- **For loops**: not implemented, so no code generation for this as of yet.

- **Function declarations**: not implemented, so no code generation for this as of yet.

- **Multithreading**:
  For multithreading code generation, please see “Sprockell intercommunication protocol” later in the report.

- **Locking and unlocking**: For locking and unlocking code generation, please see “Sprockell intercommunication protocol” later in the report.

# Description of the software

**RunLanguage**:

- RunLanguage: this class is used for running programs in the language. It will do everything, from lexing, to type checking and using SPRILGenerator to generate the code in a Haskell file, which will then be run.

**codeGeneration**:

- SPRILGenerator: this class generates the SPROCKELL code which will eventually make the program run in Haskell. This class uses a tree visitor on the parse tree generated by ANTLR. It will generate the code that belongs to the instructions written in our language when calling the convertProgToFile method. This file is then Haskell code with the SPROCKELL code in it, which can be executed to run the program.

- Symtable: this class is used to store the heap location of a variable, with as key the variable name. Variables are put into this table at declaration, and can be referenced to obtain the heap location of a variable, but as well the sprockell id in which it is declared. This is used for the requests. Every time a new scope is opened in the code, also a new scope is made in this table. The variables in that and higher scopes can then be used, but not variables from other equal or deeper scopes.

**Grammar**:

- Grammar.g4: this is the grammar file that defines the language. ANTLR uses this grammar class to tokenize (lexer class) and parse the input code in the language, and make it into a parsetree. That parsetree can then be used later on by TypeChecker and SPRILGenerator.

**TypeChecker**:

- SymbolTable: this class is used to store the type of a variable as a value belonging to a key, which is the name of that variable. Variables are put into this table at declaration, and can be checked for existing and the right type when a variable is used further on in the code. Every time a new scope is opened in the code, also a new scope is made in this SymbolTable, which means that variables declared in higher scopes can be used, but variables used in deeper scopes cannot.

- TypeChecker: this class makes sure all the types are correct in the given code, according to what is allowed in the language. It does this by using the parsetree generated by ANTLR together with tree listeners. When a basic type is reached by a tree listener, the type will but put in the parse tree as a parse tree property. This parse tree property will be taken out of the parse tree again by other tree listeners to check types. The method checkProgram in this class should be used (except for JUnit testing) for checking if the code written in our language is typed correctly. If the code is typed correctly, the method in TypeChecker will return true. If the code is not correctly typed, the method will return a ParseException, with a description of what is wrong, as well as the (first) part of the code that is not correctly typed and it's location.

## Test plan and results

**tests**:

- **LexerTest**: this test class tests for the lexing of ANTLR on our grammar. These tests can easily be run by opening the LexerTest class located in the tests folder, and clicking the double triangles to the left of `public class LexerTest {`
  The following methods in LexerTest test these things:
  - functionTest(): since functions are not currently implemented, this test method will not be explained in detail.
  - returnTest(): this is an extension on functionTest() which tests for the same things, but then with returns.
  - ifTest(): will test for correctness of if, for instance for position of elif and else
  - forTest(): since for loops are not currently implemented, this test method will not be explained in detail.
  - whileTest(): will test for correct lexing of while with conditions and curly brackets.
  - parallelTest(): will test for correct lexing of parallels, including the curly brackets and minimum of one line inside the parallel.
  - sequentialTest(): will test for correct lexing of sequentials, including the curly brackets and minimum of one line inside the curly brackets.
  - TypeAndAssign(): will test for the correct lexing of types, variable declarations and variable assignments. These all need to be specified according to the language specifications in order to be correct.
  - declTest(): will test for just declaring variables. Also tests for correct declaration.
  - assignTest(): will test for lexing of the assignment of variables that have been declared before.
  - lockTest(): will test for the correct lexing of locking and unlocking according to the language specifications.
  - exprTest(): will test for the correct lexing of expressions, which includes addition, multiplication, comparisons, constants, arrays, variables and parentheses.

- **sprilTest**: please note, not all tests function as they should. The tests that pass will be explained here. For the tests that do not pass, please see the discussion in later parts of the report, for instance at “Sprockell intercommunication protocol”. This class tests for the correctness of the SPROCKELL code that is generated from the language. This class will not test for things that should be rejected by TypeChecker or LexerTest. These tests can be run by opening sprilTest located in the tests folder, and clicking the double triangles to the left of `public class sprilTest {`
  The following methods in sprilTest test these things:
  - LockTest(): this test will test for locking a variable multiple times, which is not allowed.
  - unlockTest(): this test will test for unlocking a variable that is not locked, which will fail.
  - addError(): this test will make sure that declarations of a variable with the same name in the same scope (redeclaration) is not allowed.
  - typeError(): this test will make sure the type of a variable that is already declared cannot be changed during runtime.
  - asgnError(): this test will make sure that no variables can be assigned if they have not been declared.
  - ifTest(): this test will make sure that ifs behave as expected, and that variables outside of the if can be used and assigned from inside of the if.
  - whileTest(): this test will make sure that while loops behave as expected and give the right output. It also tests that variables declared outside of the while can be used and assigned from inside of the while loop.
  - comExprTest(): this test will test for combinations of if and while. It makes sure that a while loop repeats and executes the if, and that if changes the variables in the expected way so that the while will propagate. The outputs should be as expected for the test to succeed.
  - concurTest(): this will test for ways of concurrency using parallel and then blocks of sequential. It will test for variables from higher scopes to be used in the lower, multithreaded scopes in different sequential blocks
  - properJointTest(): this will test to make sure that variables that are used and reassigned/changed inside sequential blocks inside parallel blocks, can be used outside of those blocks again as long as the variable was first declared outside of that block. It will test this for up to 4 threads in the test, but more are possible. However, because the multithreading is not fully complete yet, the variable from outside the thread will be changed in the thread itself. It will not be changed to be visible outside the thread. That was the idea, but it is not possible for now.
  - failedNestedParallelsTest(): shows that our generated code fails for nested parallel statements. It is most likely caused by forgetting a lock. However, with the given time constraints we were not able to resolve a lot of concurrency issues.

- **TypeCheckerTest**: this test class tests for pieces of code that should or should not pass the TypeChecker, even for some things that are in the grammar but not currently in use because no code can be generated for it yet. These tests can easily be run by opening the TypeCheckerTest class located in the tests folder, and clicking the double triangles to the left of `public class TypeCheckerTest {`
  The following methods in TypeCheckerTest test these things:
  - typesTest(): all types are tested, these should all be accepted, except Arr Arr which does not define a type for the array.
  - exprTest(): all basic expressions are tested here, but probably also in tests that are “higher up” because they need exprs. Parentheses are tested, multiplication is tested for strange types, addition same thing. Subtraction is also tested for strange types, and arrays are tested for all same type of elements, variable length and also for nested arrays.
  - lineTest(): all lines are tested here. All different possibilities for if are tested, a single if, an if with one or multiple elif, an if with one or multiple elif and else, and an if with just an else. Also tested are just elif, just an else, and ifs without a boolean argument. Whiles are tested for arguments that are not boolean. Parallel and sequential are tested for the curly brackets and at least one line inside. Type declarations and assignments are tested in a lot of forms. Lock and unlock are tested for having a variable (these will be tested more in program). And finally comparisons are tested for correctness.
  - programTest(): combinations of tests before are tested here, so that things that can only happen over multiple lines can also be tested. Tested here are variable declarations and assignments that are then used in if, while, locks, and so on. Also tested here are usage of variables that are not instantiated or declared, as well as reassignment of variables. And finally, if conditions in while loops as well as sequential and parallel statements are also tested in this method.

# Sprockell intercommunication protocol

To get sprockell intercommunication to work we gave each global memory address a dedicated function (the **bold name is also used in the codegenerator to avoid errors and enhance readability**).

## Basic definitions

**Main thread** is the thread which will be started on a non-empty program. In case this main thread has a parallel block, it becomes a parallel-handler. When all the child threads have terminated, the parallel-handler will join. In case the parallel-handler was the main thread, it will join and perform the lines after the parallel block.

The **child threads** referred to hereafter specify a thread that is created for a sequential block. Such a sequential block is only able to request variables from outer scopes, so parent/spawner threads and parents of that, which of course includes the global scope.

A **parallel-handler** will respond to all requests for this thread. The parallel-handler knows which calls to respond to by matching for its sprockellID. In the case the main thread instantiates threads, so calls parallel, it effectively becomes a parallel-handler.

## Division of global memory

The division of shared memory locations is as follows:
Shared mem0: (**req_Lock**)
This is the lock needed to write to the next 3 global memory locations. Basically, this set of global memory addresses work as a channel for this thread and the thread it invites, by writing the other threads sprockellID to the **req_sprilllD** global memory location**.**

Shared mem1: (**req_Type**)
This register contains the type of request a (child) thread wants to make when it has obtained the channel lock. This can either be a pure request, such as just asking current value of a variable, orr it can be a lock or unlock request.

Shared mem2: (**req_SprilllD**)
**var_locked:2 ,** loaded into shared mem2 by the parallel-handler thread in case a child thread requests a locked variable. If the variable is not locked and the type of request requires another push from the child-thread, the parallel-handler will push **readyToReceiveUpdatedVal** (-3).

Shared mem3: (**heaplocOrValue**)
This location is used by a child thread to first indicate the location of the variable on the heap. Depending on the type of request there are different interactions. Some simple examples below.

Shared mem4: (**terminateLoopingStartSprockells**)
This global memory location is used to terminate threads that have not been started and will otherwise loop indefinitely. So, this location is set to 1 at the end of the main thread execution. This location is unfortunately not yet added to the locations that the child thread will poll and properly react on.

Shared mem5: (**startprogLock**)
This is the lock to needed to write to shared memory addresses 6 and 7.

Shared mem6: (**ProgID**)
This address is polled by the main thread to read if a child thread has terminated and the child threads use this to read their activation. This is done, respectively, by the child thread writing the parallel-handler thread id (known at compile time) and by the parallel-handler pushing the child thread sprockell id (also known at compile time).

Shared mem7: (**terminatedOrStart**)
Written to in order to indicate whether a child thread should terminate or start. For now it is always terminate, since there was no time to finish this feature before the deadline.

## Handling and making requests

One of the problems we had was that the main thread saw the initial values of the shared memory as a match to its sprockellID. This, of course, is not meant to be because it leads to deadlocks and so forths. So the first execution of the main thread is to load -1 into shared memory position req_sprillID and progID. So it is only done by the **main thread.** Also, the lock of every variable is located at the location of the variable on the heap **-1**.

**Types of requests**

Only the child threads can make requests. These 3 types of requests can be made when the thread has obtained the channel lock on requests (so obtained **req_Lock**). These possible type requests are:

**purerequest** = 0:

With this type of request one just requests the most recent value of a variable. Thread specific tasks will be elaborated.

**Updaterequest** = 1:

We were not able to complete the code generation for this type of request. In theory one could update a variable by first locking it then modifying it and then unlocking it. Like this: Int a = 10;parallel {sequential {lock x; x = 20; unlock x;}}

**requestAndlock** = 2:

This type of request is initiated by a child thread when it wants to lock a variable. This will also load the most recent value into the local memory of the thread.

**unlockAndwriteBack** = 3

This type of request is done when a variable will be unlocked by a thread. It will also write it back to the thread that “hosts” the variable.

**Parallel-Handler**

First of all, throughout the execution the parallel-handler has 2 dedicated registers in which it stores the amount of child threads left to be activated and the amount of child threads that are still left to be terminated. Henceforth referred to as **regToStart** and **regToTerminate**, respectively.

The parallel-handler does the following operations in order until all child threads have terminated, in which case a jump will go beyond the loop:

1. It pushes the ids of the child threads onto the stack
   Start of loop:
2. It will try to obtain the lock **startProgLock**, when obtained it will check whether it regToStart > 0 then pop id from stack and load to **ProgID,** decrement regToStart and write 1 **to terminatedOrStart**. If all threads are already activated it releases the lock and jumps to the line where it starts checking for variable requests.
3. Check whether a child thread has terminated, if so decrement **regToTerminate**. If that register is 0 then jump to after the loop, otherwise got to 4.

**Below is for checking type requests**

4. Start of checking variable request. If it is a request for this sprockell, so matches sprockell iid, proceed with 3 and otherwise jump back to the start of the loop.
5. Read the shared memory addresses **heaplocOrValue** and **req_Type.** First check whether the variable is locked. The parallel-handler will then checked whether it is locked (if that heap location is 1 then the var is locked). The only valid **req_Type** while locked is **unlockAndWriteBack.** If that is the type request write **readyToReceiveUpdatedVal** to indicate that the other thread can write back the updated value to global memory location **heaplocOrValue.** In the meantime the parallel-handler will poll until **heaplocOrValue** is changed, so the updated value is placed there.

**We unfortunately discovered that we should actually let the child thread write a different value to req_sprillid rather than heaplocOrValue, however too late to be able to modify it. Because if it so happens that the updated value from the child thread is the same as the heap location to store it at, the parallel-handler will poll indefinitely.**

When the variable is locked it will write **locked_var** to the **req_sprillID,** after which the child thread will terminate the request and reset the needed global memory addresses.

**variable unlocked**

Because of the fact that the code generator keeps track of the variables that have been locked and unlocked and checks its correctness: ie first unlocking and then unlocking already throws a runtimeError(), as well as unlocking or locking twice etc. There are only 2 (excluding **updatereq**) possible requests in case the variable is unlocked, namely:

**Purerequest**: when this is the case it will just write the value of the requested heap location onto the **heaplocOrValue** shared memory location. When that is done it will change the **req_sprillID** to **readyToReceiveUpdatedVal** to indicate that the value is ready to be read.

**requestAndLock:** when this is the case it will write the value of the requested heap location to the **heaplocOrValue** shared memory location. When that is done it will change the **req_sprillID** to **readyToReceiveUpdatedVal** to indicate that the value is ready to be read. Also it will set the lock of the variable to 1, which is located at the heaplocation-1.

**Child thread**

When finishing a request for variables the child thread makes sure that the locks and other locations of the global memory will be reset properly. Furthermore, when a child thread does a request and it signaled that the variable is locked, it will release the lock and reset the proper shared memory locations. However, the child thread will block until its request has been properly addressed by the requested parallel-handler thread. This of course also means that forgetting to unlock a variable will result in a guaranteed deadlock.

# Conclusions

For the language, we wanted to do much more than we eventually implemented and got to work. We started about one week late with the project since other lab exercises still had to be signed off. This made it so that we immediately were behind on schedule.
We started by implementing the grammar with all the features we intended to put in our language. This grammar in our opinion is very nice, and we would feel quite good if we had implemented that all. We tried to avoid a lot of problems by explicitly avoid them by making them invalid. This took us quite some time and definitely avoided some problems along the way. Unfortunately, we quickly noticed we would not have enough time to put all the intended features in our language.
For that reason, we started to first do the mandatory and easy features, and left for instance strings, arrays, functions and function calls and so on for if we finished the other things before the deadline. Soon, however, we noticed the complexity of concurrency and sprockell intercommunication. We started to race through tasks, and even got a bit of extra time (two days). However, with the trouble implementing multithreading and the whole report still to do, we were still not able to get everything to work.
For this reason, some things like spawning threads from threads that aren't the main thread and locking and unlocking variables are still not working as they should.
We feel that our language is quite okay single threaded, however we also feel that the multithreading implementing was really hard and not nice at all. For example the documentation lacked adequate examples for multithreading. Besides that we had trouble understanding the workings and how to use IndAddr. Unfortunately the debugging tool wasn't very helpful in figuring out how to work with sprockell instructions either. We figured it out eventually but those things only further delayed our progress. Our language, in our opinion, is because of those reasons incomplete to fully perform  multithreading tasks.

For the module as a whole, we found the compiler construction interesting to get a better understanding of how programming languages and compilers work underneath.
Concurrent programming was also useful, as everything needs to be a lot more multithreaded nowadays, and probably even more so in the future. It is then good to know what are good practices when multithreading and what should not be done. It gives a good idea of what might be the problem if a concurrent program behaves strangely, but it will also help prevent that behaviour in the first place. We also think that it was good knowledge to see what good patterns are, but also what anti-patterns are for multithreading. It's in our opinion a nice skill to be able to quickly spot them. Furthermore, the fact that the concurrency strand worked mostly on concepts rather than 100% concrete examples definitely made it easier to figure it out for sprockell programs.
We do however think that, for compiler construction, teaching a lot of things in ILOC but then require us to write SPROCKELL for the final project is a very strange decision, and it made the project a whole lot more difficult because of some differences in the languages that are in our opinion not very well documented or taught during lectures. To contrast, we had an entire

framework and extensively documented methods to generate the code code with ILOC instructions. So, while the concepts of ILOC where definitely helpful the gap in regards to documentation and framework assistance where for us too large to easily switch to generating sprockell code.

For concurrent programming, we feel it is strange to have us do concurrency in SPROCKELL for the project, while that is something that is very different from the things that we learned during cp. We know that it probably has to be put in the project, but then at least teach us a bit more how concurrency works in such low level languages. Especially since the documentation for multithreading in SPROCKELL was very confusing and minimal with only one meagre example. Also, it required quite some time crafting a good protocol when several ta's told us to either put global variables on the global memory. When mentioning this solution is not scalable the remark was to first try that and see how far we got. This definitely led us forced us on the wrong path in regards to solving concurrency for multiple sprockells and further delayed our progress.

For functional programming, we found that a bit different from everything else in the module. It did have a relation with the compiler construction, and it is a whole different way of thinking in comparison to other programming languages. But that also makes it quite hard, and we felt that we just had too little time, especially for the later exercises for fp.
For logic programming, it was not too difficult and it did not really cost us too much time to understand, and we felt it was interesting how this language is again very different from others. We think it might be very useful in certain situations (especially when programming, well, logic problems). However we also feel that logic programming was the odd one out this module. In our opinion, it was just randomly there for two weeks and a small project, and we missed most of the ways to optimize as it was all just condensed in a few quick lectures.

# Appendices

## Grammar specification

```
grammar Grammar;
program: (function|line)+ EOF;

function: FUN (types)? FUNNAME (types VARNAME)* OCUR line+ (RETURN expr)? CCUR;

sequential: SEQUENTIAL OCUR line+ CCUR;
line
 : IF expr OCUR line* CCUR
   (ELIF expr OCUR line* CCUR)*
   (ELSE OCUR line* CCUR)?                    #ifLine
 //for is not supported at this moment
 | FOR (VARNAME | INT VARNAME ASGN expr) SCOL
   expr SCOL
   VARNAME ASGN expr SCOL
   OCUR line* CCUR                            #forLine
 | WHILE expr OCUR line* CCUR                 #whileLine
 | PARALLEL OCUR sequential+ CCUR             #parallelLine
  | types VARNAME
     (ASGN expr | ASGN functioncall)? SCOL    #declLine
 //function calls not supported yet
  | target ASGN (expr | functioncall) SCOL    #asgnLine
  | (LOCK | UNLOCK) VARNAME SCOL              #lockLine
 //function calls not supported yet
  | functioncall SCOL                         #funcallLine
  ;

expr
  : OPAR expr CPAR                           #parensExpr
 | expr comp expr                            #compExpr
 | expr TIMES expr                           #multExpr
 | expr (PLUS|MIN) expr                      #addorsubExpr
 | (NUM | TRUE | FALSE | STRING)             #constExpr
 | VARNAME OSQR expr CSQR                    #arrExpr
 | VARNAME                                   #varExpr
 //arays are not fully supported at this moment
 | OSQR expr (COM expr)* CSQR                #listExpr
 ;
```

```
comp
  : EQ
  | NEQ
  | LT
  | GT
  | GET
  | LET
  ;

target
  : VARNAME                                  #varTarget
  //arrays are not fully supported at this moment
  | VARNAME OSQR expr CSQR                   #arrayTarget
  ;

functioncall: FUNNAME expr*;

types
  : INT                                      #int
  | BOOL                                     #bool
  | STR                                      #str
  //arrays are not fully supported at this moment
  | ARRAY types                              #array
  ;

FUN: 'Fun';
RETURN: 'return';
LOCK: 'lock';
UNLOCK: 'unlock';
PARALLEL: 'parallel';
SEQUENTIAL: 'sequential';

OCUR:  '{';
CCUR:  '}';
OPAR:  '(';
CPAR:  ')';
OSQR:  '[';
CSQR:  ']';
COL:   ':';
SCOL:  ';';
COM:   ',';
ASGN:  '=';
```

```
NOT:'!';

EQ:'==';
NEQ:'!=';
LT:'<';
LET:'<=';
GT:'>';
GET:'>=';

PLUS:  '+';
MIN:   '-';
TIMES: '*';

INT:   'Int';
BOOL:  'Bool';
STR:   'Str';
CHAR:  'Char';
ARRAY: 'Arr';
IF:        'if';
ELIF:  'elif';
ELSE:  'else';
WHILE: 'while';
FOR:   'for';
TRUE:  'true';
FALSE: 'false';

fragment LOWERCASE: [a-z];
fragment UPPERCASE: [A-Z];
fragment LETTER: UPPERCASE | LOWERCASE;
fragment DIGIT: [0-9];
fragment COMP: EQ | NEQ | LT | LET | GT | GET;

FUNNAME: UPPERCASE (LETTER | DIGIT)*;
VARNAME: LOWERCASE (LETTER | DIGIT)*;
NUM: DIGIT+;

//no whitespace allowed in strings!
STRING: "" [\u0000\u0021\u0023-\uFFFE]* "";
// CHARACTER: '\'' . '\'';

WHITESPACE: [ \t\r\n]+ -> skip;
```

## Extended test program

We were unable to get the concurrency to work as a whole. We were able to properly implement the part of

- Main thread spawning child threads
    - Int a = 10; OutNumber 10; parallel {sequential {OutNumber 20;} sequential {OutNumber 30;}} a = 40; OutNumber a;
    - Executing the previous program clearly shows that spawning threads works appropriately.
- Child threads spawning threads not working. For some reason the nested child does not get activated while the program code does definitely contain activation code for the thread. This is most likely due to some  error with
    - Int a = 10; OutNumber 10; parallel {sequential {OutNumber 20;} sequential {OutNumber 30} sequential {parallel {sequential {OutNumber 10;}}}
      } a = 40; OutNumber a;

But because the locking and unlocking does not work at this moment, we could not write “an implementation of Peterson’s algorithm for mutual exclusion”. Nor could we write a working test for “an elementary banking system, consisting of several processes trying to transfer money simultan-eously”. And because these tests would not really work and thus are not there, we can also not give the generated code for that program and show the correct functioning. We would like to, but it sadly just does not properly work at this moment.