Project Report
Programming Paradigms
Elective Module 2.4
University of Twente

Jesse van der Velden
1746308
Robert-Daniel Banu
1942727

1.Summary

The language described in this document is called AMv.

AMv is completely parsed, processed and ran in Haskell and Sprokell. Its features include : variable declarations , function declarations, function calling and using it's return, nested functions, concurrency (fork/join), fancy quick commands (*incr* ++, *decr* --, *quickadd* += , *quicksub* -=) and error handling. No arrays/ strings and

2.Problems and solutions

Division of labor:

Robert: Parsing, Code Generation Elaboration, Report

Jesse: Code generation, Report

Front-end Errors:

In the elaboration part of processing (in the front-end), initially, any illegal command or expression would simply 'throw' a Haskell error with a specific message, which was not satisfactory to AMv's standards. This is done by the file: *TreeWalker.hs*

The elaboration is done in two parts, Symbol Table creation and Type Checking. Initially , they would return the symbol table and a boolean respectively. To extend the functionality of elaboration, the TypeError data structure was declared and introduced as the return type of all Type-Checking methods which could test for errors (e.g.*typeCheckCondition, findinDb, checkDuplicant etc).*

The main Type-Checking method, *typeCheckProgram,* was changed so it returns an array of TypeError. If the array is empty, then the program being checked is correct, at least from the frontend's type perspective.

Afterwards, the *symbolTableBuilder* had to be adapted so it stores its errors , instead of "throwing" them and stopping the program. It also stores the Type-Checking errors that may arise while evaluating the AST. All the errors are stored for the purposes of displaying them, instead of displaying only the first one that it found.

The solution we came up with was to add another data type to the *DataBase* structure, namely Err ArgType TypeError. The ArgType is used to store where the error occurred and the TypeError stores the message of the error. In this way, when the the *symbolTableBuilder* encounters some type of error, either brought up by itself or by other tests within itself, it could store the error and proceed to find other possible errors.

If a program is correct then the Symbol Table would have no *Err* types within it and the *typeCheckProgram* would return an empty array.

3.Detailed language description

General:

AMv support **three basic types** boolean, integer and, for methods only, void.The two boolean constants in AMv are “ya” which is true and “nu” which is false. Both are parsed as boolean constants.

AST: Type = SimplyBol | SimplyInt | SimplyNull

In AMv there are blocks, which contain commands, such as while, assignment of a value to a variable, fork etc. An AMv program is basically a block , so each program has to start with “{“ and end with “}”.

AST: Bloc = Block [Commands]

A Bloc is a list of commands separated by semicolons.

Commands such as IfCommand, While and Function Declaration have blocks within themselves. The code generation will just evaluate all the commands from the given AST, while first looking for function declarations, followed by the main block’s code. It will create the AR, store the values in the local data area and a point to the previous AR.

No matter the command, it has to be followed by semicolon. Even the entire program must start and end in a block.

Some other AST building blocks:

ArgType = Arg Type String

Param = ByVal ArgType | ByRef ArgType

Condition : a Condition is a comparison between two Expressions. Boolean expression can only be checked if they are equal and Integer expressions can be: less ‘<’, greater ‘>’, greater or equal ‘>=’ , less or equal ‘<=’ or equal ‘==’. If the two expressions have different types or two booleans are compared with anything other than ‘==’, then an appropriate error will come up.

Expr : An expression can be one of many things, such as a constant integer, constant boolean, an IfExpression, an addition (2+3) etc. Using the information from elaboration ,the Type of any Expression can be determined.

Here is some code exemplifying some of the syntax of AMv.

```
{
int x = 0;
int y = 1;
bool z = bool ? (x>y) {ya} {nu} ;
y += y;
func int randomMethod(& int x){
```

```
            while ( x < 10 ){
                  x ++;
            };
            return x;
    }; ← notice the semicolon at the end of method declaration
print randomMethod(x); //prints 9
fork randomMethod(x);
}
```

Variable declarations: In AMv variables can be declared. The type first has to be specified than the variable name, followed, optionally, by its value. If no value is declared, the variable will be assigned the default, which is 0 for int and false for bool types.
A keyword “global” can be added at the beginning of a declaration to signify that variable is global.

AST:GlobalVarDecl ArgType Expr | VarDecl ArgType Expr

```
{
int x = 2;
bool y = ya;
global int a;
global bool = nu;
z = 2; //this will throw an error as z is undefined
}
```

If the wrong type is assigned or if two variables have the same name within the same scope, or with a global variables, appropriate error messages will come up.

In the code generation, it will generate the code for it’s expression after the equal sign, and first wants to store it in the local data area, but can traverse to upper scopes if that is needed.

In AMv there are two types of If.

If Command, which is a condition which would lead to one of two paths/blocks.

AST: IfCom Condition Bloc Bloc

```
{ if (x >=2) { print x;}{ print 2;} ;}
```

If Expression which technically an expr, it has a type and returns something on both branches. If any of the two branches returns not the type specified, an error arises.

AST: IfExpr Type Condition Expr Expr

```
{
 int x; int z = 2; int y = 3;
 x = int ? (2 > 1) {z*y} {y+z};
}
```

While statements : it is a while loop, it has a condition and goes though the commands of its block after it tests if the condition is true

AST: While Condition Bloc

```
{
int x = 0;
while (x <= 10){ x ++;} ;
}
```

The code generation for the if condition and the while condition will just generate both the expressions to compare. Then stores that in registers and they will be compared afterwards, with a branch to go to the appropriate location. At the end of the while condition, it will loop back to the condition again.

Calculation Commands:

AMv supports commands of the type “++” or “-=”. If they are not applied to an integer type, then an error will arise.

AST: AddCom String Expr | MinCom String Expr | Decr String | Incr String

```
{
 int x = (1);
 x -- ;
 x += x;
 x -= x;
 x++;
 print x;  //prints 1
}
```

In the code generation it will just evaluate the expression and store that again in the right location.

Functions:

AMv is supposed to support function declaration and calling.The AST structure and the parsing of methods are complete.

The parameters of the method being declared can also be passed as a reference, in which case an “&’ procedes its declaration as a parameter. Methods can be one of 3 types: bool, int and void. Bool and Int methods have to have a return statement in their body and void methods are allowed to be called as a command:

AST:

FunDecl ArgType [Param] Bloc : Function declaration, the ArgType contains its type and name, the [Param] holds its parameters and the Block is its body.

FunCall String[Expr] : Function call , if the methods being called is not void, an error will arise. The string holds its name and the [Expr] are the method’s arguments.

Funct String[Expr]: function call of a method with return. It is categorised as an expression, so it must be either bool or integer type.The string holds its name and the [Expr] are the method’s arguments.

```
{
 func int fn(& int x,int y){
      while ( x < 10 ){
             x ++;
      };
      return y;
   };
 func void hello(){
     print ya;
 }

 hello();        //prints true
 int x;
 print fn(x,10); //prints 10
 print x;        //prints 9
}
```

In the Code Generation we filter out all the functions at the start of a block. This way you can actually call a function before it is declared in the code. The offsets are calculated and stored in a separate function table. Call by reference and call by value will be checked before the generation in the validation phase. A call will set the AR to the new scope. When the function is created, in the prologue (so before the actual function’s main code) it will make sure the arguments will be available in the local data area. Then it will just generate the main's function code. Finally, in the epilogue all the variables changed in the function will be updated and stored again. If the function call was done, by assigning the resulting return value to a variable, the return value will be pushed to the stack. It will be automatically popped if there is no such variable or entirely skipped if it is a void method. Nested

Concurrency:

AMv supports some simple concurrency commands: Fork and Join. The Fork expect a function call, which will be ran in a different Sprokell.

AST: Fork Expr| Join

```
{
 func void fn( int x) { print x; };
 fork fn(2);
 join;
}
```

The fork instruction will run a function concurrently, by letting the waiting sprockells know something can be done. Forks must run of course on multiple sprockells.
An additional sprockell, will first come into the thread loop. There it checks if the main thread is finished. Therefore there are some reserved spaces in the shared memory, like where to jump to (line number), arguments (amounts, values and where to store them) and read and write locks (just bits).

The join makes sure that every sprockell finishes its execution before continuing with the further code. It just checks the threads are currently busy (stored in shared memory) and only continues if they aren't set.

Print: pretty self explanatory, it generates the expression, push it to the stack and will be printed out using the WriteInstr.

AST : Print Expr

Error Structures:

The data type

TypeError = Er String| Ok | Crt Type

is used in type-checking.

Er String is an error with a message, Ok is simply correct and Crt Type is correct Type, as this was needed in some methods that had to pass down a Type.

typeCheckProgram is ran on a list of commands (and the DB generated by that list of commands) and it returns a list of TypeError , which is empty if there are no errors. Some possible error messages are:

```
{
 int x = ya;          -> Er "VarDecl x wrong type assigned"
 x = ya*2;       -> Er "Multiplication elements are not the same
type in Assigment x"

 x = int ? (ya >= ya) {2}{ya};
-> Er "Bools cannot be >= in IfExpr and first expr is wrong type
in Assigment x"

 func int fib(int x) { int y; return y;};
 int z = fib(ya);
     -> Er "Function's arguments are not correct in FuncExpr in
VarDecl z"
}
```

This is the data structure used for the Symbol Table entries:
 DataBase =
        DB ArgType Int Int | DBF ArgType [Param] Bloc | Err ArgType TypeError

The Err is used to keep track of error that arise outside of the Type-Checking.

```
{
 int x =2;
 bool x = ya;              -> Err (Arg SimplyInt  x ) (Er
"Duplicate declaration in same scope")

 func int fib (int x) { print x;}; -> Err (Arg SimplyInt fib) (Er
"Method has no return" )

 func int other (int x) { return ya;}; ->  Err (Arg SimplyInt
other) (Er "Return type of methods not same type and methods type"
)

 fib (2); ->Err (Arg SimplyNull ) (Er "Could not find Method in DB
in Funcall fib")  //this is because fib was erroneous when it was
declared
}
```

When running a program, all it’s errors, including the Type-Checking errors, are stored in the Symbol Table. All types of errors have appropriate message to help the user fix the issues.

If Symbol Table has no Err types inside it, then the program is correct and it the Code generation starts in the AST generated by the parsing.

4.Description of the software:

Extra stack dependencies:
We used Parsec for parsing, and Sprockell of course for running our generated spril code. Besides we had made some sprockell extensions**:**
ComputeI op reg val reg: that does an operation on a register and a ImmValue.
BranchX reg target: because sometimes you want to jump on 0 values. Just saves time in not XOR'ing it first.

See also the Readme.md file for an entire overview of all the files.
src/Structure.hs: This file contains the data structures used in the parsing of text and creation of the AST. It also contains some *from* methods e.g. *fromBlock* or *stringArtgType* which take information out of the defined data structures. The data structure for *TypeError* is declared there too, with some helper methods to boot.

src/BasicParsers.hs: this file contains the list of reserved words and some basic parsers such as *identifier, reserved, parens* etc, already defined by Parsec.

src/ParseExpr.hs contains all the parsing methods related to the parsing of expression, e.g. constans, function calls, arithmetical operations and if expressions.

src/Parser.hs consists of methods used in the parsing of commands, such as method declaration, if and while commands, variable declarations, increase, decrease etc. They structure of the AST they are creating can be found in the Structure.hs file. The entire parsing is done by parsec.

src/TreeWalker.hs is the module in charge of elaboration. It is split in two parts, DataBase creation and Type checking:
DataBase/Symbol Table creation is the group of methods that create the DB of the program being parsed. It stores the offset, scope and type for variables declared and type parameter and body of declared methods. While creating the DB , it check for errors such as double declarations in the same scope, int/bool methods without a return command etc and stores the errors in the Symbol Table, if any.
The Type-Checking part consists of the methods used in the type checking of the program being parsed. It contains a methods *typeCheckProgram* which takes a list of commands, a DataBase array(which is the Symbol table) and return an array of TypeError. If the array is empty, it means that all the commands given were correct from a type correctness perspective.

src/TreeWalkerTests.hs is a fill full of tests which were initially in TreeWalker.hs. They were moved TreeWalker readability reasons.

src/Generator.hs: the file which makes the code generation. You pass an AST and then it runs the TreeBuilder which outputs an symboltable. Now it is able to generate the code based on the AST. It contains lots of helper methods to get the offset of each set of instructions so it is able to reference back to functions when they are called.

app/Compiler.hs: contains the main method of the compiler, so you can specify its arguments. When it goes to compile it will run the generated code of the specified file with the specified amount of threads. See the Readme.md file on how to run it.

Finally, test/Tests.hs: This file contains our automated Haskell frontend test. It contains a data structure

data ParseTest = ParseT { testName :: String, testParse :: Bool }

and an array of ParseT.

Each of the ParseTest entries is a test method that can be found in one of the other files, under the methods it is testing
There are more test methods in our files than there are ParseT entries in Tests.hs.

5.Test plan and results

A vast majority of parsing methods in the front-end have test methods right under them. Due to the fact Type-Checking and Symbol Table builder methods are many and interconnected, individual tests for helper methods would have been a headache to set up, so only larger, more encompassing tests were created.

Most front end methods are tested in the same way, their results are compared to a hard coded answers, which is what the method is expected to return, see also src/TreeWalkerTests.hs

The tests/Tests.hs file contains a series of automated tests, which make use of individual test methods previously mentioned. It contains random tests from other files and runs them, returning an appropriate error message if one of the tests fails. To run all the front-end tests automatically: type in your terminal: `` `stack test` ``. If there is an error there will be shown an exception, but it should say:

```
PP-AMV-Lang-0.1.0.0: Test suite amv-test passed.
```

Even our so called *wrong* tests will be tested. These consists of several syntax and contextual tests that should return an error in the frontend. These error messages are then compared to the ‘hardcoded’ error messages.

The example codes can be found in the /examples folder. These can be run with Haskell stack. First set this up with: `` `stack build` ``. The generated Spril code can also be found in the same folder.

src/banking.amv: with three threads: ‘`stack exec -- amv c examples/banking.amv 3`’.Showing pass by reference, pass by value, concurency

Expected output:

```
Sprockell 0 says 1000
Sprockell 0 says 5000
Sprockell 0 says 99999
Sprockell 0 says 1100
Sprockell 0 says 5100
Sprockell 0 says 100000
Sprockell 0 says 1090
Sprockell 0 says 5150
Sprockell 0 says 103000
Sprockell 0 says 590
Sprockell 0 says 5550
Sprockell 0 says 102925
Sprockell 0 says 540
Sprockell 0 says 5560
Sprockell 0 says 102915
Sprockell 0 says 340
Sprockell 0 says 5760
Sprockell 0 says 102865
```

src/peterson.amv: See the appendix

src/functest.amv: ‘`stack exec -- amv c examples/functest.amv 1`’, showing pass by reference.

Expected output:

```
Sprockell 0 says 1000
Sprockell 0 says 1003
Sprockell 0 says 1003
```

src/functest2.amv: ‘`stack exec -- amv c examples/functest2.amv 1`’, showing pass by reference and value.

Expected output:

```
Sprockell 0 says 1000
Sprockell 0 says 123
Sprockell 0 says 3123
```

src/functest3.amv: ‘`stack exec -- amv c examples/functest2.amv 1`’, showing pass by value and return type.

Expected output:
Sprockell 0 says 4

src/functestnested.amv: ‘`stack exec -- amv c examples/functestnested.amv 1`’, showing nested functions.
Expected output:
Sprockell 0 says 2

src/iftest.amv: ‘`stack exec -- amv c examples/iftest.amv 1`’ showing just a simple if statement.
Expected output:
Sprockell 0 says 1000
Sprockell 0 says 9999

src/iftest2.amv: ‘`stack exec -- amv c examples/iftest2.amv 1`’ showing just a simple if statement with booleans.
Expected output:
Sprockell 0 says 1

src/whiletest.amv: ‘`stack exec -- amv c examples/whiletest.amv 1`’ showing just a while loop, and some calculations.
Expected output:
Sprockell 0 says 10
Sprockell 0 says 9
Sprockell 0 says 8
Sprockell 0 says 7
Sprockell 0 says 6
Sprockell 0 says 5
Sprockell 0 says 4
Sprockell 0 says 3
Sprockell 0 says 2
Sprockell 0 says 1
Sprockell 0 says 0

6.Conclusions

Robert:

I believe our language, AMv, turn out alright. With more time and better state of mind, we would have for sure made it better, but alas, so little time because of exams. We ended up working on it for two weeks, not three, as we both studied for our exam. I wish we had implemented arrays, but I also feel like what we managed to implement was a lot anyway.

The Module is a lot. In the last three weeks I found it almost impossible to study,I guess that's what you'd call burnout. By the 7th week i did feel like I learned a whole lot , esp in FP and CC, not so much FP as I felt like I already had an okay grasp on it. LP was weird, i do not understand why it was taught.

Jesse: The entire module is the heaviest I came across my life on uni in terms of the amount of work. You start wondering what the real amount of work for 15 EC is, if you do some modules easy peasy, this was something entirely different. However, I felt like we learnt a lot, and I'm glad that I learnt Haskell, and Rust during this module. Overall it felt very satisfying when everything comes together and it finally makes sense, (but that didn't count for LP for me too). Now I have a better overview of things accomplished in the field of computer science together with the previous module, and what early steps had to be made in order to make easy programming languages. There are still loads of optimisations that can be made. Overall, it is smart to make it mandatory sign off everything, otherwise I think a lot of people just wouldn't make it through the module.

Appendix: Grammar

Grammar is written in ANTLR style.

```
block : '{' (command ';')* '}' ;

command: declr
        | functDecl
        | IDENTIFIER '=' expr
        | IDENTIFIER ('--' | '++' )
        | IDENTIFIER ( '+=' | '-=' ) expr
        | while
        | ifcom
        | FORK block
        | JOINT
        | PRINT expr
        | RETURN expr
        | fucntionCall
        ;

declr: GLOBAL? Type IDENTIFIER ('=' expr)?;

functDecl : Func Ftype LPar ('&'? Type IDENTIFIER)* RPar block ;

while : WHILE condition block;

ifcom : 'if' condition block block;

expr: term Add expr
        | term Div expr
        | term ;

term : factor Mult term
        | factor;

factor : NUBER | IDENTIFIER | BOOLVAL
        | fucntionCall | LPar expr RPar |
        | Type '?' condition '{' expr '}' '{' expr '}';

fucntionCall : IDENTIFIER LPar expr* RPar;

condition: LPar expr CmpSign expr RPar;
```

```
IDENTIFIER : LETTER+ ;
RETURN : 'return';
GLOBAL : 'global' ;
NUBER: DIGIT+ ;
JOIN : 'join' ;
FORK : 'fork' ;
PRINT : 'print' ;
WHILE : 'while' ;
BOOLVAL : 'ya' | 'nu';
fragment DIGIT: [0-9];
fragment LETTER: [a-zA-Z];
fragment Type : 'int' | 'bool' ;
fragment Ftype: Type | 'void' ;
fragment Func : 'func' ;
fragment CmpSign: '==' | '<' | '>' | '>=' | '<=';
fragment Mult: '*' ;
fragment Add: '+' ;
fragment Div: '-' ;
fragment LPar : '(' ;
fragment RPar : ')' ;
fragment SemiCol : ';' ;
fragment IfExpr: '?' ;
```

Appendix: Extended test program: Peterson

```
{
global bool flag0 = nu;
global bool flag1 = nu;

global int turn = 0;
global int counter = 0;

func void pZero() {
    flag0 = ya;
    turn = 0;

    bool flag1AndTurn1 = ya;
    while (flag1AndTurn1 == ya) {
      if (flag1 == nu) {
            if (turn == 0) {
                  flag1AndTurn1 = nu;
            } {};
      } {};
    };

    // critical section
    counter--;
    counter--;
    counter--;
    counter--;
    counter--;

    // end of critical section
    flag0 = nu;
};

func void pOne() {
    flag1 = ya;
    turn = 1;

    bool flag0AndTurn0 = ya;
    while (flag0AndTurn0 == ya) {
      if (flag0 == nu) {
            if (turn == 1) {
                  flag0AndTurn0 = nu;
            } {};
```

```
        } {};
    };

    counter++;
    counter++;
    counter++;
    counter++;
    counter++;
    // end of critical section
    flag1 = nu;
};

int i = 10;
while (i > 0) {
    fork pZero();
    fork pOne();
    join;
    print counter;
    i--;
};
}
```

Run: `stack exec -- amv c examples/peterson.amv 3`

Expected output:

```
Sprockell 0 says 0
Sprockell 0 says 0
Sprockell 0 says 0
Sprockell 0 says 0
Sprockell 0 says 0
Sprockell 0 says 0
Sprockell 0 says 0
Sprockell 0 says 0
Sprockell 0 says 0
Sprockell 0 says 0
```

Spril code: see examples/peterson.spril