Compiler construction

Harm Berntsen s4244931, Danny Hendrix s4241746

June 20, 2014

1 Introduction

In this report we will describe our SPL compiler. We have the following additional features:

1. Support for higher order functions and partial application of functions.
2. Support for multiple error messages with the location of the error in the input code (via a generated HTML page and console output). Only the semantic analysis supports those error messages. When the scanner finds an error it will immediately stop and give you the location of the wrong token in the source file.
3. Garbage collection for SSM code.
4. Tail call optimization for SSM. (LLVM's optimizer includes tail call optimization)
5. Optimization of static expressions.
6. Compilation to LLVM code.
7. Demonstration of SPL code running on an ATmega88 microcontroller.
8. A modified SSM interpreter that allows you to run SSM code without the user interface.

2 Grammar

We had to change the grammar slightly to solve some conflicts between rules. Our grammar parses the same programs as stated in the first assignment. The only addition is a new type that is separated by an arrow. This is used to specify types of functions. For example, the function `Bool foo(Int a, Int b)` has the type `Int -> Int -> Bool`. Using this notation, you can write programs like:

```
Int add(Int a, Int b) { return a + b;}
Void main() {
    Int -> Int add2 = add(2);
    Int five = add2(3); //yields 5
}
```

In Figure 1 we show our version of the grammar.

3 Parser

The parser transforms the raw SPL code in a processable parse tree. Part of the parsing is the tokenizer. In our compiler this is a straight forward implementation where we read characters one by one and use parser methods who, on success, save tokens in a monadic state.
The parser itself is a bottom-up LALR parser. At first we started creating the parse schema manually. We soon found this was a hard and error-prone task and therefore we decided, after consulting our teacher, to use a tool to create the parsing table automatically. This brought us to the GNU parser generator Bison. Figure 2 shows the parsing schema of our grammar, generated by Bison. In total our grammar of 61 rules consists of 143 states and 762 transitions. This clearly shows that if we would have done it manually: it would have cost too much time. To implemented the transition and reducing rules output from Bison, we made an automated script. This allowed us to extend and alter the grammer in later stages. This came to use when we decided to extend SPL with higher order functions where we had to extend the grammar.

```
Decl            : VarDecl
                | FunDecl

VarDecl         : Type "id" '=' Exp ';'

FunctionBegin   : VarDecl FunctionBegin
                | Stmt

FunDecl         : Type "id" '(' FArgsOpt ')' '{' FunctionBegin StmtStar '}'
                | "Void" "id" '(' FArgsOpt ')' '{' FunctionBegin StmtStar '}'

Type            : "Int"
                | "Bool"
                | '(' Type ',' Type ')'
                | '[' Type ']'
                | "id"
                | Type "->" Type
                | (Type "->" Type)

FArgs           : FArgs ',' Type "id"
                | Type "id"

FArgsOpt        : %empty
                | FArgs

Stmt            : '{' StmtStar '}'
                | "if" '(' Exp ')' Stmt
                | "if" '(' Exp ')' Stmt "else" Stmt
                | "while" '(' Exp ')' Stmt
                | "id" Field '=' Exp ';'
                | FunCall ';'
                | "return" ';'
                | "return" Exp ';'

StmtStar        : %empty
                | Stmt StmtStar

Exp             : "id" Field
                | Exp '+' Exp | Exp '-' Exp | Exp '*' Exp | Exp '/' Exp | Exp '%' Exp
                | Exp "==" Exp | Exp '<' Exp | Exp '>' Exp
                | Exp "<=" Exp | Exp ">=" Exp | Exp "! =" Exp | Exp "&&" Exp | Exp "||" Exp
                | Exp ':' Exp | '!' Exp | '-' Exp | "int" | "False"
                | "True" | '(' Exp ')' | FunCall | '[' ']' | '(' Exp ',' Exp ')'

Field           : %empty
                | '.' "hd" Field
                | '.' "tl" Field
                | '.' "fst" Field
                | '.' "snd" Field

FunCall         : "id" '(' ')'
                | "id" '(' ActArgs ')'

ActArgs         : Exp
                | Exp ',' ActArgs
```

Figure 1: The grammar that is used in our parser. In addition to some precedence rules, this is the input of bison.

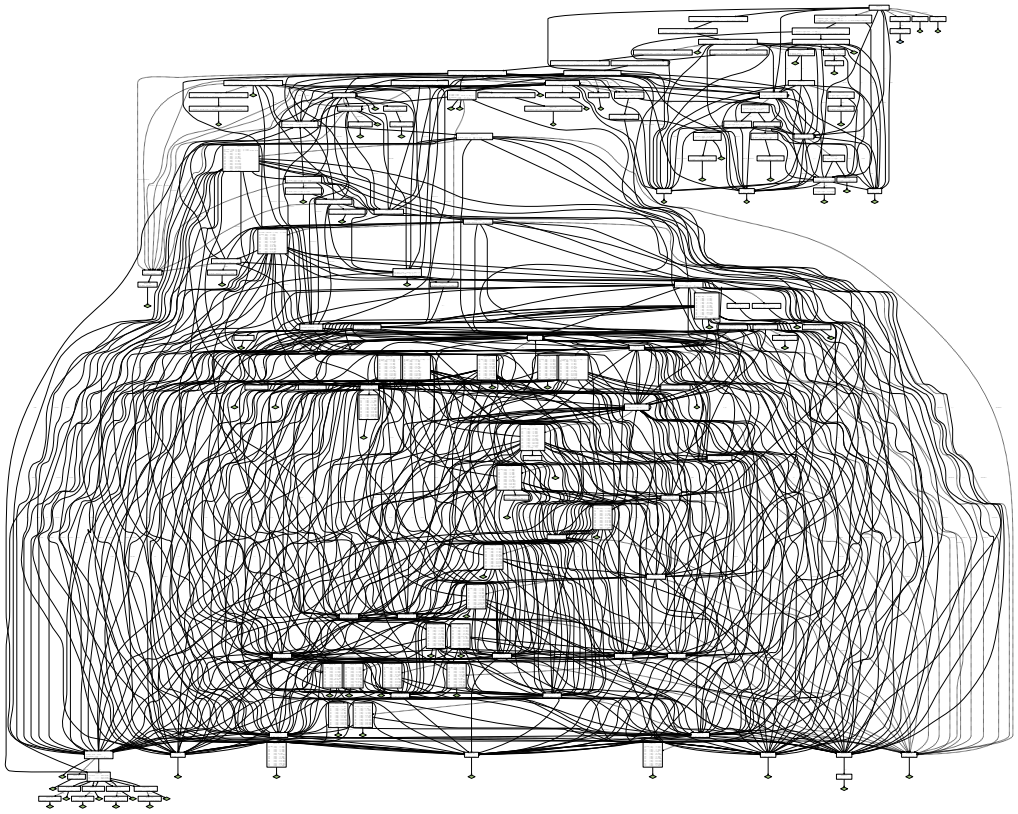

Figure 2: Schema of all the states and transitions in our parser

4 Semantic analysis

4.1 Scoping rules

Variable declarations and type checking is divided in 2 scopes: variables defined outside functions (globals) and variables declared inside functions (locals, so including arguments). Because we allow higher order functions, it is not allowed to have variables and functions with the same name. It is allowed to declare a global and local variable with the same name. This will produce a warning. The function arguments are not allowed to override. Otherwise you would be able to specify multiple arguments with the same name.
When you use lists, tuples or partial function application as local variables in a function, their memory will not be freed. This allows you to use those variables in the global variables. For example, you can create a tuple in a function and add that tuple to a global list. We included a garbage collector for SSM which can delete unreferenced objects from the heap.

4.2 Type rules

We applied monomorphism in our semantic analysis. As a result, when you have a function with a parameter `a -> a` you can only use this function to transform something of type a to a.
Type variables occuring on the global scope are defined as 'for all'. This means that that polymorphic functions like the identity function (`a->a`) can be used with different types. With variables declared within a function or parameters of a function, the type variable `a` can have only one meaning. So `a` cannot match with `b` or `Int`. When a global polymorphic function is called, the type variables in that global function will be quantified as 'for all'. In our semantic analysis we rename the 'for all' type variables to something unique so we can distinguish them. This means

that you can write global functions like the identity function that can be used with different types. In the code, the `insertGlobalTypeVar` function links those global type variables to types from within the function.
With the monomorphism rules, the semantic analysis is more strict than we actually need for our SSM code generator. For example, if you pass the identity function as argument to a function, calling the identity function in the argument can be done with different types multiple times and the program will still execute as expected. The semantic analysis will reject this though.

In order to be able to declare nested lists, we allow that the empty list expression can represent multiple nested lists. For example: `[[Int]] a = [];` is allowed. However, when you do `[[Int]] b = a : []` you get an error because the result of the right hand side is `[[[Int]]]`.

It is allowed to call a function before it has been declared. We first scan the source code for functions and store the types of each function before we type check the program. Polymorphic functions only have one implementation in the compiled code. Functions can use global variables that are declared later on in the file. We do the analysis in this order:

1. Store all the declared types of the global functions
2. Analyse each global variable
3. Analyse each function

When variable initializer calls a function that uses a global variable that has not been initialized yet the global variable will default to 0. So a bool is false, integer is 0 and lists are empty. For other situations you will get unexpected results since the tuple pointer is still a null pointer.

5 SSM code generation

5.1 Calling semantics

We have chosen for the 'Java approach'. Boolean and integer values will be copied as value when a function is called and other types will be passed by reference to an object on the heap.

5.2 Memory Management

Every block starts with an object header that defines the size of the block. Each block has at least the size 1, namely the object header. When we allocate new objects we first allocate the new block on the heap and then fill it with the correct values. To reduce heap memory usage we made a simple garbage collector (GC) which is explained in detail in chapter 6.3.

5.3 Compilation scheme

The program will first write the stack to the heap. The garbage collector then knows where the stack starts. The next thing a program has to do is creating the global variables. This is done by counting the number of `VarDecl`s and then creating a block on the heap with this size. Each variable is put there in the order it is declared. In order to access a global variable, we use the fact that global variables are declared first on the heap and the heap always begins at a static address: `0x000007d0`.

5.3.1 Memory allocation of types

The types *Int* and *Bool* both fit in a word so they can be placed directly on the stack or heap where needed as value. The empty list is encoded as null pointer value.
Value: Int, Bool and (possibly 0) pointer.

Tuple

The tuple (a, b) is encoded on the heap as:

Object header
a
b

Populated list element

Object header
value of one element
pointer to next element, 0 if end of list

Partial function application

Partial function application will be stored as:

Object header
last additional supplied argument
⋮
first additional supplied argument
function pointer(if this is the very first arg supplied to the function)
pointer to previous arguments, 0 if none

Every time you do partial function application we create a new block. We could have copied the previous arguments into this new block but that uses more heap space. Eventually when calling the function we can place all the arguments on the stack and call it as usual.

5.3.2 Function calls

Arguments are placed in reverse order on the stack. The function that supplies the final argument in a partial function application will have to remove the arguments from the stack. Due to this, each function places the number of arguments on the stack so we know how much arguments there need to be removed from the stack.

6 Extensions

6.1 Higher order functions

In our implementation of SPL we allow higher order functions. This means that every variable can be a function. To denote this we have introduced a new type in the grammar. Every function is of the type $A_0 \rightarrow ... \rightarrow A_n \rightarrow Ret$ where A are the arguments of the function and Ret is the return type of the function. For example $Int \rightarrow Bool \rightarrow Int$ is a function where the first argument is of type Int and the second argument is of type Bool. The return type of this function is of type Int. To type check function calls we have a method $resolveFunction$. This method does a lookup of the type of the function. It then compares the given argument types with the definitions type. For this it keeps track of declared type ids ($a \rightarrow a$ becomes $Int \rightarrow Int$ if the first argument was an integer). This method returns the remaining type of the function call. If the function was called with fewer arguments than the definition then the remaining type is a function type. A higher order function can be treated the same as a variable, so `id` is the same as `id()` when id is a function. When you want to do partial application, you can simply 'call' the function with less arguments than it has.

6.2 Error reporting

Our abstract syntax tree contains the location of every element. Each element has a `Context` record which contains the location. We added it as a record to allow us to easily add more information to an element later on. In the end it appeared that we did not need additional information in our abstract syntax tree so the record only contains the location. We created a special error module which allows us to easily create error messages and to nest them. By combining the state monad with the error monad we could create a system where we can throw errors and allow functions to catch them or add additional information to the error. For example, the code that analyses a variable wraps all the generated errors with something like "When analysing function x...". This will help getting a better sense of what went wrong. Errors in the semantic analysis and the parser are shown both as output in plain text and as HTML error report. The plain text gives an overview of the errors and shows an in-depth location with line, position and also the function in which the error occurs. The HTML error report shows a more sophisticated overview with a pretty-printed version of the input and in-line error messages and warnings. Examples of the output can be found with the samples, included in the submission of our compiler.

6.3 Garbage collection

The GC is written in SPL to demonstrate the use of our compiler. This also allows a more sophisticated implementation. The GC is only activated when the heap pointer reaches the maximum allowed heap-space. It then looks for heap blocks that have no reference on both the stack and the heap. For this we have a global variable that has the value of the start of the stack. This allows us to limit the search to only the stack and the heap. The size of the blocks that we allocate on the heap can be of any length. Tuples and list items have a length of 3 (for a tuple that is a header, the left value and the right value) but blocks for the higher order function can be of any length. The GC tries to find a gap that is big enough for the to be allocated block. It does this by "freeing" consecutive blocks with a combined size big enough to fit the new block. The downside of this is that it leaves holes on the heap if the freed size is bigger then the size of the new block. We expect however that these gaps are not too big since the most used blocks, the tuple and list items have only a size of 3 and can fit in most gaps. Overall this simple implementation of a GC does a good enough job. The SPL source code is included in the submission. The SSM output is slightly modified to make the labels unique and unoverwritable. The type checker does find errors in the GC SPL code. These are caused by default functions that were added to make the GC read direct values in the stack/heap.

6.4 Tail call optimization for SSM

We added tail call optimization for the SSM code we generate. Tail call optimization will happen when you are within a function that on return only calls a globally defined function with the same amount of parameters. We do not apply this optimization with higher order functions because when compiling you don't know how much arguments the function has. We also did not do this with functions with less arguments because each function stores the number of arguments on the stack so that the caller knows how much variables need to be removed from the stack. Calling a function with less arguments via tail call optimization will change that number.

6.5 Expression optimization

In our parser, we added optimizations for some of the expressions and statements. For example, if you write `1+2`, only the result, 3, will be stored. In our reduce function in the GrammarTable module we added additional pattern matches to do this. When an if statement is reduced to `if(false)`, we also remove that. This will help speed up the resulting SSM code. The LLVM optimizer is also optimizing the static expressions.

6.6 LLVM code generation

The Low Level Virtual Machine (LLVM) is a compiler infrastructure. In our compiler we convert SPL to LLVM's intermediate language. LLVM can optimize this intermediate language. This means that we initially don't need to focus on emitting optimized code since LLVM's optimizer can optimize a lot. For example, we wrote a recursive and non-recursive SPL program to calculate the greatest common divisor of two numbers. LLVM's optimizer fully calculated the result, only the code to print the result remained.

6.6.1 Memory management

LLVM's intermediate language has unlimited typed registers. Each typed register can only be assigned once. Using the alloca, load and store instruction you can modify data that is stored in a pointer. In our compiler we use those instructions to allow us to assign new values to variables in SPL. The `mem2reg` parameter of LLVM will remove all unnecessary allocate, load and store instructions. When the code is compiled to machine code for a specific architecture, LLVM will decide how the physical hardware registers will be used.
Because the variables(typed registers) have a type, our compiler needs to emit well typed code. This became a problem with polymorphic functions. The polymorphic functions need to be able to accept any type. When they return a type variable, the caller expects that return value to be a specific type. To solve the problem we could generate functions for each combination of types we need. This means that we need to recursively track all the function calls and make sure that we generate all needed combinations. This could lead to lots of different versions of functions, especially if you have polymorphic functions that call polymorphic functions again. We have chosen to encode type variables as `i8*`, LLVM's equivalent of the the `void *` from `C`. We then had to make the choice whether the pointer would point to the actual value or would contain the actual value(like we did in SPL). When you let the pointer point to the value you have the advantage that the size of a pointer does not limit your possibilities. For example, if we would want SPL to have 256bit integers, we could not fit those inside a pointer. We have chosen to encode the values of booleans and integers inside the pointer itself. Since we use 32bit integers we are sure we can fit those inside pointers on our 64 bit OS on our computers. This would also make it easier to generate code for partial function application: we then don't need to worry about pointers to stack memory that might not be valid anymore. The integers and booleans are encoded in the pointer, for other types the pointer points to some heap memory.
LLVM does not have a malloc instruction to allocate heap memory, you need to call the libc malloc function for that. When construction lists and tuples, we call this function to allocate the heap memory for them. We use a similar structure as in SPL except that we don't have a header that contains the size of the memory block. Like calls to polymorphic functions, the values stored in tuples and lists are converted to a pointer first. We do this using the `inttoptr` instruction. This instruction does nothing if the value already is a pointer, otherwise it converts an arbitrary int to pointer, adding zero bits or truncating when needed. Our compiler has conversions in a lot of places. For example, every time you assign a new value to an Int variable in SPL, we first do a `ptrtoint` conversion to make sure we are not trying to store an `i8*` in an `i32`. The LLVM typechecker does not accept that. We could have made our compiler smarter by remembering the type of each operand. LLVM's optimizer is able to remove the unnecessary conversions so in the end it doesn't matter.

6.6.2 Used packages

To convert SPL to LLVM's IR, we use the `llvm-general` and `llvm-general-pure` package[1]. The pure package contains an abstract syntax tree for LLVM's IR, the other contains code that uses LLVM's C++ API to generate the output or directly run the code using LLVM's JIT compiler. We had some trouble installing those packages. Danny uses Windows and couldn't get the packages

[1] https://github.com/bscarlet/llvm-general

to install there. On Linux the packages installed fine but couldn't be used in combination with GHCi version 7.6 that is shipped with Ubuntu. GHCi is an interactive environment where you can interactively evaluate expressions. We liked it and in order to get the llvm packages to work in it we compiled the latest GHC version from source. See Appendix B for an explanation how to set up your development environment to compile our code.
We based our code on Stephen Diehls tutorial[2]. He provided an excellent tutorial in building a compiler with Haskell. The LLVM part formed a great starting point for us.

6.6.3 Higher order functions

We have tried to implement higher order functions in the LLVM code generator. Hower because of time and complexity we were unable to finish this. Theoretically we can implement higher order functions as followed. The implementation can be similar to the implementation for SSM. Each function call can be stored in an array block that contains a reference to the initial function, the supplied arguments and an optional reference to a previous partial function. The parameters have to be converted to pointers in order to store them in the array. On execution of the final function we have to convert these parameters back to the correct type. In theory this concept should work but we were unable to fully implement it.

6.7 AVR

LLVM does not ship with a compiler to compile to AVR. We had to compile LLVM from source with an AVR patch. It looks like the AVR backend is originally hosted on sourceforge[3] but hasn't been updated for a while. We found code for LLVM 3.5 on GitHub[4]. After a fixing a few compilation issues we got it to work, sort of. Not all programs ran successfully on our Atmel ATmega88 chip and sometimes the LLVM to AVR compiler crashed with an error.
We were able to demonstrate the playback of a song via SPL. In our compiler we manually coded some functions that set up the hardware and allow us to access some hardware features from SPL.

6.8 Modified SSM interpreter

In the `testPrograms` folder, we store `spl` test programs with their expected output. We modified the SSM interpreter such that it can print the output of the program directly to the console instead of launching the user interface. This allowed us to automatically test whether our compiler still produces the correct code. This came in handy when doing refactoring or writing additions like the garbage collection or tail call optimization. We could then automatically compile and run the programs and see whether our changes to the compiler broke anything.

7 Known bugs

1. We do not check whether a tuple is used before initialization, see Section 4.2. The program will have undefined behaviour when you do this.

2. The SPL code we generate might be so long that the source code ends up in the heap. Since the heap starts at a fixed address, using the heap will change the program code.

3. When compiling for LLVM, the semantic analysis accepts higher order functions and partial application. Our SPL to LLVM compiler or LLVM's typechecker will complain when you attempt to use them though.

[2] http://www.stephendiehl.com/llvm/
[3] http://sourceforge.net/projects/avr-llvm/
[4] https://github.com/sushihangover/llvm-avr

8 Code organisation

The compiler is written in Haskell. In the source tree, there are 3 folders:

- src, this contains the compiler source code
- test, this contains the code to run our integration tests
- testPrograms, the test code scans this directory for test programs to execute. You can place various files in here:
  - .spl, this contains your spl program you want to test.
  - .spl.msg, the expected error messages from our semantic analysis. Spl files without messages don't have a .msg file.
  - .spl.ssmout, the output that is expected from the SSM interpreter.
  - .spl.llvmout, the output that is expected when the code is executed with LLVM.

Our code is divided into several modules:

- CompilerArguments. Contains the code to handle the arguments that were given to the compiler
- ErrorMessage. The semantic analysis uses this module to generate error messages.
- ErrorShow. This module generates an HTML report of the error messages.
- Grammar. The scanner passes the tokens it has found to this module. This module converts the tokens to our abstract syntax tree.
- GrammarFunctions. This module contains some utility functions for use with our abstract syntax tree.
- GrammarShow. Code to pretty print the abstract syntax tree to valid SPL code.
- GrammarStructure. Contains the abstract syntax tree.
- GrammarTable. Contains the rules for our scanner to convert the tokens to the abstract syntax tree. Most of the code in this module is automatically generated using the XML output of Bison. The code that was used to convert Bison's XML to haskell can be found in the Bison subfolder. We changed the automatically generated reduce function to include simple optimizations of expressions.
- Label. This generic module can generate unique labels.
- Main. This is the main module for the SSM compiler.
- Scanner. The SPL source code is converted to tokens using the scanner. The scanner integrates with the Grammar module to generate our abstract syntax tree.
- SemanticAnalysis. This module analyses the abstract syntax tree and detects semantic errors. We do not use the results of the semantic analysis for code generation.
- Util. Generic utility functions.

The Bison subfolder contains the code to convert Bison's XML output to Haskell code. The code for the SSM and LLVM output each have their own subfolder. The SSM subfolder has the following modules:

- CodeGeneration. This module converts our AST to SSM code.

- GarbageCollection. The garbage collection can be enabled in this module by changing a constant. The majority of this module contains SSM code that is used for the garbage collection. This code is generated by our compiler with some manual modifications.
- Instructions. All the SSM instructions are encoded in a Haskell data structure. This file contains that data structure and the code to convert it to text.
- SSMRun. We modified the SSM interpreter such that it can be run from the command line. This module contains utility functions to easily run the generated SSM code.

In the LLVM subfolder we have the modules:

- AVR. AVR specific code we used for our presentation.
- Codegen. Makes using the `llvm-general` package easier. For example, it contains helper functions to create a state where we can easily save instructions in LLVM blocks.
- Emit. Where the Codegen module contains more generic functions to output LLVM code, Emit contains SPL specific code to convert our AST to `llvm-general-pure`'s AST.
- LLVMRun. Runs LLVM code using the `lli` program. The `lli` program executes the LLVM code using its internal just-in-time compiler.
- Main. Contains the main function of the `cc-llvm` program we generate.
- Types. Small module which converts SPL types to their `llvm-general-pure` equivalent.

9 Example code

We created 10 interesting sample programs which can be found in Appendix A. We have tried to choose examples that show a broad range of that what our compiler is capable of. Five of the examples contain typing errors which are given as output. Even more sample programs can be found in the testPrograms folder in the source folder. These programs are used for unit testing and contain more samples of LLVM programs.

10 Work division

The work was not divided upfront. We found it was easier not to work on the compiler at the same time to avoid conflicts. In general we have both worked on every individual part of the compiler and fixed, when necessary each others code. We usually divided some time slots and each person continued where the other stopped. A general overview of the work division of part 2 of the assignment:

10.1 Harm

Phase 2:

- Researched and implemented monadic state by combining the state and error monad with monad transformers. The state contains error messages, a map from identifier to type, the current return type and compiler arguments.
- Error message system that allows nesting of errors
- Testing framework and a part of the integration tests
- Checking of the presence of a main function
- Analysis of (global) variables

- Function call type checking/matching
- Parser support and helper functions of the arrow type
- Basic structure where each `Decl` element is analysed separately to get lazily retrieve the error messages
- (Incomplete) type inferencing system which we removed due to complexity
- Variable field resolving

Phase 3:

- Re-implemented type checking of type variables and function calls (now with monomorphism).
- Setting field values
- Code structure of code generation
- AST of SSM code in Haskell
- Modified SSM Interpreter for integration tests
- Created integration tests
- Heap memory allocator (version when GC is disabled)
- Code generation of functions, expressions, statements
- Tail call optimization
- Fixed fields that were parsed in the wrong order (and thus the generated code didn't work correctly when those were nested)

Phase 4:

- All initial code structure(based on the tutorial)
- Testing different LLVM packages and getting it all to work with GHCi.
- Set-up of virtual machine for development
- Global variables
- Code generation for the functions
- Print and isEmpty function
- Tuples and Lists via malloc
- Automated tests for the LLVM code
- Function calls
- Conversion of types(e.g. in expressions and statements)
- AVR related things
- Optimization of expressions and statements in the parser

10.2 Danny

Phase 2:

- Type checking of function declarations
- Checking return types in functions (if the functions contains a return statement and if the return statement returns the correct type)
- Type checking of statements
- Type checking of the Op1 and Op2 operator
- Type checking of function calls
- Polymorphic type checking (If the type is $a \to a$ and the first argument is of type Int then a must always be of type Int)
- Higher order type checking
- Keeping track of variable and function types in the monad state
- Default SPL functions: `print` and `isEmpty`
- Lots of (unit)testing and fixing

Phase 3:

- Type checking fixes with higher order polymorphic functions
- Default functions: `print` and `isEmpty`
- Fields (`fst`, `snd`, `head` ..)
- Implementation of lists and the `:` operator
- `op1` and `op2` operators
- Garbage collection

Phase 4:

- HTML error report
- If statements
- While statements
- Binary operators
- Higher order functions/partial application for LLVM
- improved GC

A Sample code

A.1 sample0.spl

This sample demonstrates declaring a list and using it in a polymorphic function which accepts a function as parameter. There are no compilation errors.

```
The errors:

The input:
1. a foldl(a->b->a f, a z, [b] list) {
2. if(isEmpty(list)) {
3. return z;
4. } else {
5. return foldl(f, f(z,list.hd),list.tl);
6. }
7. }
8.
9. Int div(Int a,Int b) {
10. return a / b;
11. }
12.
13. Void main() {
14. [Int] b = 4:2:4:[];
15. print(b);
16. print(foldl(div, 64, b));
17. }
18.
The parsed:
1. a foldl(a->b->a f, a z, [b] list)
2. {
3.
4. if(isEmpty(list))
5. {
6. return z;
7. }
8. else
9. {
10. return foldl(f, f(z, list.hd), list.tl);
11. }
12. }
13. Int div(Int a, Int b)
14. {
15.
16. return (a / b);
17. }
18. Void main()
19. {
20. [Int] b = (4 : (2 : (4 : [])));
21. print(b);
22. print(foldl(div, 64, b));
23. }
```

A.2 sample1.spl

An empty list can be a list of lists of lists of integers. The polymorphic function print can have a list of Int as parameter. There are no compilation errors.

```
The errors:
```

```
The input:
1. Void main()
2. {
3. [[[Int]]] c = [];
4. print(c);
5. }
6.
The parsed:
1. Void main()
2. {
3. [[[Int]]] c = [];
4. print(c);
5. }
```

A.3 sample2.spl

This sample shows the usage of higher order functions in global variables. There are no compilation errors.

```
The errors:

The input:
1. a test(a aap, a uil)
2. {
3. return aap;
4. }
5.
6. Int->Int a = test(7);
7.
8. Void main()
9. {
10. Int b = a(4);
11. print(b);
12. }
The parsed:
1. a test(a aap, a uil)
2. {
3.
4. return aap;
5. }
6. Int->Int a = test(7);
7. Void main()
8. {
9. Int b = a(4);
10. print(b);
11. }
```

A.4 sample3.spl

This sample shows that a polymorphic function can be used multiple times with different types. There are no compilation errors.

```
The errors:

The input:
1. Bool isEven(Int x) {
2.     return x % 2 == 0;
3. }
```

```
4.
5. [t] reverse ( [t] list )
6. {
7. [t] accu = [] ;
8. while ( ! isEmpty ( list ))
9. {
10. accu = list.hd : accu ;
11. list = list.tl;
12. }
13. return accu;
14. }
15.
16. [(l,r)] zip ([l] a,[r] b) {
17.     return (a.hd,b.hd) : zip (a.tl,b.tl);
18. }
19.
20. a id(a a) {return a;}
21.
22. [b] mapReverse (a->b f, [a] l)
23. {
24. [b] accu = [];
25. while(! isEmpty(l)) {
26. accu = f(l.hd) : accu;
27. l=l.tl;
28. }
29. return accu;
30. }
31.
32. Void main()
33. {
34.     [Int] ints = mapReverse(id(), 1 : 2 : 3 :4 : []);
35.     [Bool] evens = mapReverse(isEven, ints );
36.
37.     print(ints);
38. }
39.
```

The parsed:

```
1. Bool isEven(Int x)
2. {
3.
4. return ((x % 2) == 0);
5. }
6. [t] reverse([t] list)
7. {
8. [t] accu = [];
9. while(!isEmpty(list))
10. {
11. accu = (list.hd : accu);
12. list = list.tl;
13. }
14. return accu;
15. }
16. [(l, r)] zip([l] a, [r] b)
17. {
18.
19. return ((a.hd, b.hd) : zip(a.tl, b.tl));
20. }
21. a id(a a)
```

```
22. {
23.
24. return a;
25. }
26. [b] mapReverse(a->b f, [a] l)
27. {
28. [b] accu = [];
29. while(!isEmpty(l))
30. {
31. accu = (f(l.hd) : accu);
32. l = l.tl;
33. }
34. return accu;
35. }
36. Void main()
37. {
38. [Int] ints = mapReverse(id(), (1 : (2 : (3 : (4 : [])))));
39. [Bool] evens = mapReverse(isEven, ints);
40. print(ints);
41. }
```

A.5 sample5Fail.spl

This sample shows that you cannot divide a `Bool` by an `Int`. It also shows the notation of the messages. The first part of a message is the type of message: Info, Warning or Error. After that you have the position which is notated as `[line:colon]`. The length of the arrows determines to what 'parent' error it belongs. It also shows the optimization of expressions, $4 + 2$ is converted to 6 in the pretty printed version.

```
The errors:
[Info][1:1] About the function 'main'
->[Info][3:2] About variable a
-->[Error][3:18] Operator type mismatch, Bool / Int not allowed
--->[Error][3:14] Type Bool does not match expected type@[3:18]: Int
The input:
1. Void main()
2. {
3. Int a = 4+2+True/5*3;
4. print(a);
5. }
6.
The parsed:
1. Void main()
2. {
3. Int a = (6 + ((True / 5) * 3));
4. print(a);
5. }
```

A.6 sample6Fail.spl

This sample shows that in all paths of a function there needs to be a return statement. The `if(1>2)` is automatically removed because the code can never be reached. Our parser does not have the messaging system of the semantic analysis so it was hard to give warnings about the dead code. The if statement of the testOk function is removed because it is always true.

```
The errors:
[Info][1:1] About the function 'test'
->[Error][1:1] Missing return statement in function test
The input:
```

```
1. Int test()
2. {
3. if(1 > 2)
4. return 5;
5. }
6.
7. Int testOk() {
8. if(2 > 1)
9. return 4;
10. }
11.
12. Void main()
13. {
14. test();
15. }
16.
The parsed:
1. Int test()
2. {
3.
4. {
5.
6. }
7. }
8. Int testOk()
9. {
10.
11. return 4;
12. }
13. Void main()
14. {
15.
16. test();
17. }
```

A.7 sample7Fail.spl

In this program we compare lists that contain different types which is rejected by the compiler.

```
The errors:
[Info][6:1] About the function 'compare'
->[Error][9:2] When analyzing the statement return (sort(as) == sort(bs));
-->[Error][9:18] Operator type mismatch, [Int] == [Bool] not allowed
--->[Error][7:3] Type Int does not match expected type@[8:3]: Bool
The input:
1. [a] sort([a] lijst)
2. {
3. return lijst;
4. }
5.
6. Bool compare() {
7. [Int] as = 1 : 2 : 3: [];
8. [Bool] bs = True : [];
9. return sort(as) == sort(bs);
10. }
11.
12. Void main() {
13.     compare();
14. }
```

```
15.
The parsed:
1. [a] sort([a] lijst)
2. {
3.
4. return lijst;
5. }
6. Bool compare()
7. {
8. [Int] as = (1 : (2 : (3 : [])));
9. [Bool] bs = (True : []);
10. return (sort(as) == sort(bs));
11. }
12. Void main()
13. {
14.
15. compare();
16. }
```

A.8 sample8fail.spl

This example shows multiple errors in the main function. With the resFail2 variable: In the zipf2 function, the third argument is a c so all instances of c must be an Int in zipf2. This matches with the isZero function. The error is that in the main function we expect the return type to be a tuple of Ints but in fact it is a tuple of Bool since the isZero function returns a Bool. With the resFail1 expression we pass a Boolean as third argument while the isZero function is Int-¿Bool. This gives an error about overlapping type variables because c should be the same type but an Int and a Bool is given in the function arguments.

```
The errors:
[Info][12:1] About the function 'main'
->[Info][15:2] About variable resFail1
-->[Error][8:1] Type Bool does not match expected type@[15:8]: Int
-->[Error][8:1] Type Bool does not match expected type@[15:4]: Int
-->[Info][15:25] When analyzing the function call zipf2((isZero : []), (isZero : []), True)
--->[Error][15:58] Type Bool does not match expected type@[8:13]: Int
->[Info][13:2] About variable resFail2
-->[Error][8:1] Type Bool does not match expected type@[13:8]: Int
-->[Error][8:1] Type Bool does not match expected type@[13:4]: Int
The input:
1. [(a,b)] zipf2([c->a] calist, [c->b] cblist, c param) {
2.         c->a calisthd = calist.hd;
3.         c->b cblisthd = cblist.hd;
4.         [(a,b)] rest = zipf2(calist.tl,cblist.tl,param);
5.         return (calisthd(param),cblisthd(param)) : zipf2(calist.tl,cblist.tl,param);
6. }
7.
8. Bool isZero(Int i) {
9. return i == 0;
10. }
11.
12. Void main() {
13. [(Int,Int)] resFail2 = zipf2 (isZero : [], isZero : [], 3);
14. [(Bool,Bool)] resOk = zipf2 (isZero : [], isZero : [], 4);
15. [(Int,Int)] resFail1 = zipf2 (isZero : [], isZero : [], True);
16. print(resOk);
17. }
18.
```

```
The parsed:
1. [(a, b)] zipf2([c->a] calist, [c->b] cblist, c param)
2. {
3. c->a calisthd = calist.hd;
4. c->b cblisthd = cblist.hd;
5. [(a, b)] rest = zipf2(calist.tl, cblist.tl, param);
6. return ((calisthd(param), cblisthd(param)) : zipf2(calist.tl, cblist.tl, param));
7. }
8. Bool isZero(Int i)
9. {
10.
11. return (i == 0);
12. }
13. Void main()
14. {
15. [(Int, Int)] resFail2 = zipf2((isZero : []), (isZero : []), 3);
16. [(Bool, Bool)] resOk = zipf2((isZero : []), (isZero : []), 4);
17. [(Int, Int)] resFail1 = zipf2((isZero : []), (isZero : []), True);
18. print(resOk);
19. }
```

A.9 sample9fail.spl

This sample shows resolving of the return type in polymorphic functions. The return type should be of type `a` but a type `b` is returned.

```
The errors:
[Info][2:1] About the function 'caller'
->[Error][5:2] When analyzing the statement return var(arg1);
-->[Error][5:9] Incorrect return type b
--->[Error][4:5] Type b does not match expected type@[2:1]: a
->[Info][4:2] About variable var
-->[Error][2:10] Type a does not match expected type@[4:5]: b
The input:
1. g test(g g, c b) {return g;}
2. a caller(a arg1)
3. {
4. a->b var = test(arg1);
5. return var(arg1);
6. }
7.
8. Void main() {{}}
9.
The parsed:
1. g test(g g, c b)
2. {
3.
4. return g;
5. }
6. a caller(a arg1)
7. {
8. a->b var = test(arg1);
9. return var(arg1);
10. }
11. Void main()
12. {
13.
14. {
15.
```

```
16. }
17. }
```

A.10 sample10fail.spl

This sample shows a warning when a a local variable overrides a global one.

```
The errors:
[Info][3:1] About the function 'main'
->[Info][4:2] About variable a
-->[Warning][4:2] Overwrites the existing declaration of 'a' at [1:1]
The input:
1. Int a = 3;
2.
3. Void main() {
4. Int a = 4;
5. print(a); // should print 4
6. }
7.
The parsed:
1. Int a = 3;
2. Void main()
3. {
4. Int a = 4;
5. print(a);
6. }
```

B Environment setup instructions

We do not know the exact steps we took to set up our environment. We work on (X)Ubuntu 14.04. To be able to compile our program you are probably fine with executing the following in a terminal:

```
sudo apt-get install haskell-platform g++ llvm ghc-dynamic
cabal update
cabal install cabal-install
echo "PATH=~/.cabal/bin:\$PATH" >> ~/.bashrc
#should be version 1.20.0.1
cabal -v

#now go to the directory of our compiler source code
cabal sandbox init
cabal install
```

It should now compile the executables. If you want to use GHCi, you need a newer version of GHC. Make sure to have the new cabal version using the steps above. The following steps will help you installing a newer version of ghc:

```
sudo apt-get install git autoconf make libghc-ncurses-dev
git clone git://git.haskell.org/ghc.git
cd ghc
VERSION=7.8.2
git checkout -b ghc-${VERSION} ghc-${VERSION}-release
./sync-all --no-dph get
./sync-all checkout -b ghc-${VERSION} ghc-${VERSION}-release
perl boot
./configure
```

```
make
sudo make install
#ghc versie should be 7.8.2
ghc --version

sudo apt-get autoremove ghc
sudo apt-get install libgmp-dev libmpfr-dev libbsd-dev libgmpxx4ldbl zlib1g-dev happy

rm ~/.cabal/config
rm -rf ~/.ghc
cabal update
cabal install cabal-install --enable-shared

#cd go to the source directory
cabal sandbox init
cabal configure --enable-shared
cabal install llvm-general --enable-shared -fshared-llvm llvm-general
cabal install --enable-shared
cabal repl cc-llvm
```