Compiler construction

Ivo Melse
s1088677

Erik Oosting
s1136456

May 8, 2025

Contents

Chapter 1

Introduction

In this report, we describe `HaSPL`, our compiler for the Simple Programming Language (SPL). The chapters are outlined as follows:

1. Introduction: We motivate our choice to write the compiler in Haskell, introduce SPL, and give some examples.
2. Parser: We explain the internals of our parser combinator system, and what we did to get the kinks out
3. Other chapters: TODO

1.1 Language choice: Haskell

We chose the a functional language because this class of languages enables us to define expressive operations over structures (such as Abstract Syntax Trees), which can be written down concisely. We chose Haskell in particular, because it has a convenient syntax for working with top-down parsers and parser combinators. In later sections, it will become clearer how exactly we used the language.

There was feedback for this, but we're not going to change it because the feedback was incorrect

1.2 Simple Programming Language (SPL)

The Simple Programming Language (SPL) is a programming language designed at Utrecht University. As the name implies, it is meant to be *simple*. The particular version of SPL that we use for `HaSPL`, has the grammar that we present in Appendix A

1.3 Example programs

We provide two small example programs in SPL, see Figure 1.1 and Figure 1.2.

```
fac(n) : Int {
    if (n == 0) {
        return 1;
    } else {
        return n * fac(n-1);
    }
}

main() : Void {
    print(fac(100));
}
```

Figure 1.1: Example SPL program that calculates the factorial of 100.

```
main() : Void {
    var l = 0:1:2:3;
    while (! isEmpty(l)) {
        print(l.hd);
        l = l.tl;
    }
    return 0;
}
```

Figure 1.2: Example SPL program that prints the elements of a list.

Chapter 2

Lexical analyses

We implement top-down parsing, using a parser combinator system with a custom parser monad. This monad is deterministic, and can do type safe error handling (that is, without the use of the `error` function, which crashes a haskell program with an error message).

2.1 Lexing

The lexing is a simple case of character recognition in a Haskell character list. During the lexing, location is also tracked, and locations of succesful scans are embedded with the token type into the resulting token stream.

Haskell pattern matching means that we want to start with more specific pattern matches, and have more general cases later on. We want to make sure we've not mistakenly recognised a keyword as an identifier, so identifiers are one of the last things we're trying to recognize, after pretty much all else fails.

The lexer is also a bit greedy. If an identifier starts with a keyword, we might naively split up the identifier into the keyword part, followed by the identifier part. We add in a non-exhaustive Haskell guards check in order to fix this:

```
checkKeyword :: String → Bool
checkKeyword [] = True
checkKeyword (c:_) = not $ isAlphaNum c || c ≡ '_'

tokenise (col, row) ('i':'f':xs) | checkKeyword xs = T IfToken col row : ...
tokenise (col, row) ('e':'l':'s':'e':xs) | checkKeyword xs = ...
...
```

Figure 2.1: Example of lexing with keywords

2.2 The parser monad

For parsing, we use a custom parser combinator:

```
newtype Parser a = P {runParser :: [Token] → Result [String] (a, [Token])}
```

This parser is pretty similar to the `StateT [Token] (Either [String]) a` monad, we can take advantage of this by using the `DerivingVia`[1] language extension to do half of our work for us. Of note is that we don't actually use `Either`, but rather a custom `Result` type. We'll elaborate on this in section 2.5.

Discussion on our custom implementation of `Alternative` (which is used to decide between different production rules) will also happen there. The general idea is that when we have a case where we need to choose `pa <|> pb`, then we choose `pa` if that parser succesfully parses something, and `pb` otherwise. This means that we want to put parsers that always succeed (`pure []`, for example), last, as other possible productions would get skipped, otherwise.

Indeed, the parser combinator library we've built up is order-sensitive (as most parser combinator libraries are). In practice, this turns out not to be a big problem.

Furthermore, during the parsing of `pa` in `pa <|> pb`, we do not track how many tokens `pa` consumed before reaching a failure state. Rather, it gives the exact same token stream that was given to `pa`, to `pb`. This makes our parser pessimistic (meaning that it always backtracks upon failure), which we need to take into account when doing expression parsing.

2.3 General parsing guidelines

Because of the previously stated order sensitivity, we try to put rules that we know can fail easily first, and put parsers that always succeed last. Furthermore, we have a lot of standard parser combinators (`option`, `chainl1`, `chainr1`, `sepyBy`, etc), that we try to use as much as possible. This is because naively using our parsing primitives can lead to a slow parser, which is undesirable. Having a parser take 5 minutes to parse "(((((1)))))" would be quite embarrassing. For more elaboration on how we get past this, we have to go to the expression parser.

2.4 Expression parser

Expressions can be left-associative or right-associative. When left-associative, this causes a naive parser (`parseTerm = parseTerm <*> parseSubTerm <|> parseSubterm`) to be left-recursive, and be guaranteed to never terminate. When right-associative, a naive parser (`parseTerm = parseSubTerm <*> parseTerm <|> parseSubTerm`) will terminate, but if several layers deep it will take exponential time to find a solution to certain inputs (`(((((((1)))))))` tending to be one of them), since it parses the same left-hand-side subterm over and over again every time it encounters a failure.

To solve this, we first have the unfolding combinators `chainl1` and `chainr1`, which deal with left-associative and right-associative expressions respectively. They both take as arguments a parser that parses a subterm, and a parser that combines 2 parsed subterms into a bigger term. The implementation of `chainl1` was taken from the Haskell standard library[2]. `chainr1` is a logical derivation based off of `chainl1`. (Figure 2.2)

2.4.1 Precedence

Operators have a certain precedence. The precedence we take follows the one of python [3], with a few exceptions:

- Unary operators bind the most tightly

[1] `https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/deriving_via.html`
[2] Text.ParserCombinators.ReadP docs
[3] `https://www.geeksforgeeks.org/precedence-and-associativity-of-operators-in-python/`

- List consing is introduced. It is right-associative and has the lowest precedence.

All in all, we get the following precedence table 2.3. This table describes *how we actually parse our operators*. In theory some of these operators are non-associative, but in that case we still have to choose whether we parse it left- or right-associatively

A large number of these operators are binary and left-associative, this means that we can put them into a list and then fold the list with `chainl1`. We can then fold up the list expressions with `chainr1`, and handle all the unary operators in the high-precedence parser (`parseFactor`) all at the same time.

```
— for eliminating left recursion
chainl1 :: Parser a → Parser (a → a → a) → Parser a
chainl1 p sep = p ≫= rest
  where
    rest x =
        ( do
            f <- sep
            y <- p
            rest (f x y)
        )
            <|> return x

— for eliminating right recursion
chainr1 :: Parser a → Parser (a → a → a) → Parser a
chainr1 p sep = p ≫= rest
  where
    rest x =
        ( do
            f <- sep
            y <- p
            (f x) <$> rest y
        )
            <|> return x
```

Figure 2.2: Definition of `chainl1` and `chainr1`

Operators	**Associativity**
(:) (Cons)	Right
(||) (or)	Left
(&&) (and)	Left
(==) (equals) (<) (less than) (>) (greater than) (<=) (less-than-or-equal) (>=) (greater-than-or-equal (!=) (not-equal)	Left
(+) addition (-) subtraction	Left
(*) multiplication (/) division (%) modulo	Left
(!) boolean negation (.hd) list head selection (.tl) list tail selection	Right [4] Left [4]

Figure 2.3: Operator precedence table

[4]Slightly unusual, since all of these are unary operators, but even they have associativity

2.5 Error handling

The current state of error handling is that we can get one precise error, with a stack trace of failures that lead up to it. There is quite a clear path towards detecting multiple parsing errors (albeit only 1 per function definition/variable declaration), but we'll discuss this towards the end of the section.

2.5.1 The `Result` monad

As said in section 2.2, we use a custom version of the `Either` monad called `Result`. It looks like this:

```
data Result e a = Ok e a | Fail e deriving (Functor)
```

The main difference with `Either` is that we can carry errors in a successful parse. This has everything to do with the `pure` parser combinator. It can return a successful parse without parsing anything at all. If we were to use the `Either` monad, we would then lose access to any errors we found in the previous (failing) parser. To prevent this from happening, we define the `Monad` instance for `Result` as following:

```
instance (Monoid e) ⇒ Monad (Result e) where
    return = Ok mempty
    (Fail e) ≫= _ = Fail e
    (Ok es a) ≫= k = case k a of
        (Ok es' b) → Ok (es <> es') b
        (Fail e) → Fail (e <> es)
```

The `Applicative` instance should trivially follow from this.

2.5.2 Collecting & Pruning

If we continue this pattern with the `Alternative` instance of our parser, we would start getting way too many errors that actually just false positives (while we think there is nothing wrong with false positives, as lots of languages make them, we would like to not overwhelm the user with useless errors). Therefore, we do some pruning action here to keep the total amount of errors down:

```
pa <|> pb =
    P
        ( \s →
            case runParser pa s of
                Ok e (a, s') → Ok (if s ≡ s' then e else mempty) (a, s')
                Fail e →
                    case runParser pb s of
                        Ok e' b → Ok e' b
                        Fail e' → Fail $ (e <> e')
        )
```

When we have to choose between two parsers `pa` and `pb`, we first try to run `pa`. If that parser succeeds, and it consumed tokens, then we remove the errors we obtained up to then. Otherwise, we run parser `pb`. If that fails, we collect the errors of both parsers and return them.

This once again runs into the issue of `pure`. Once we have a parser that always succeeds, we still lose all the progress we made. This is especially true for the `many` parser combinator, which,

failing to consume any input, returns successfully with the empty list. To fix this, we implement `many` ourselves (and since `some` doesn't actually mutually recurse into `many`, likely because of compiler inlining reasons, we need to define that parser as well)

```
many p =
    P
        ( \s → case runParser (some p) s of
            Ok e a → Ok e a
            Fail e → Ok e ([], s)
        )
some p = (:) <$> p <*> (many p)
```

In fact, this isn't the only case where we have parsers that don't parse stuff and still return values. For the general case, we have the `<|?>` combinator, which doesn't throw away the errors of the first parser upon success of the second one.

```
(<|?>) :: Parser a → Parser a → Parser a
pa <|?> pb = P (\s → case runParser pa s of
                          Ok e a → Ok e a
                          Fail e →
                              case runParser pb s of
                                  Ok e' b → Ok (e <> e') b
                                  Fail e' → Fail $ e <> e')
```

This all, is sufficient to get a error report at the right spot (a true positive), but also it will still collect the errors that it gets from running out of alternatives when the parser backtracks. Conveniently this then provides a sort of "stacktrace" which tells us where the parser started going wrong, which helps us slowly narrow down the true issue.

2.5.3 What's left to do

Error collection isn't perfect. Currently, we have a parser that fails and halts when it has encountered 1 error. However, in the most common case, running the parser then actually just produces a `Right` with some tokens left over. We can take these tokens, and dig through them until we find a spot we can assume to be a function definition, and parse from there. This way, we can detect errors in multiple function declarations. However, since we're only retrying to parse after the initial "bad" function/variable declaration, this does mean that the error reporting can be a bit coarse, meaning that we may skip some parsing failures.

2.6 The Abstract Syntax Tree

Finally, we arrive at the AST. In general, the AST is not very interesting when comparing it to the grammar (see also appendix B). The most remarkable difference is the absence of `FExp`. We try to eliminate it as a parsing artifact and consider field selectors to either be:

- extensions of variable names, if they're on the left-hand side of an assignment
- unary operators, if they appear in the expression itself.

This eliminates the need to have "Field expressions" as a separate datatype, allowing us to nest into expressions more easily.

We also chose to make the `Ty` polymorphic. This allows us to do away with names when we have to go and inference/unify new types during the type checking phase. During the parsing phase, they are instantiated as `Ty String` Down the line the AST may be changed, most likely in the `Exp` definition. This is because, as it stands currently, the AST doesn't easily allow itself to be changed easily. We can however, separate `Exp` into it's fixpoint (`Term`, see below), and it's branching logic, `ExpF a` [5]

AST is now updated, update this text later

```
newtype Term f = In {out :: f (Term f)}
data ExpF a = BoolLit Bool
            | NumLit Integer
            | CharLit Char
            | UnOp Op1 a
            | BinOp Op2 a a
            ...
```

This allows us to then annotate the now non-recursive `ExpF` with stuff like types (`((Ty Int, ExpF a)`), or locations for better info on where each expression term is located (`(((Row, Col), ExpF a)`), or both.

It is unlikely that we need this for the statements, since those don't actually need their type inferred. They just need to have the expressions type-checked against some likely existing variable.

[5] See also: recursion-schemes

Chapter 3

Semantic analyses

For the semantic analyses we have developed a type system that is uniquely tailored to first-order function languages like SPL. The typechecker type checks polymorphic (but still simple) types using a number-based approach.

3.1 A new expression tree

As stated in the previous chapter, we have modified `Exp` such that we can split it up in `ExpF a` and `Term`. This allows us to have variations of the expression part of our AST. This however does come at the cost of our AST not being trivially represented as a string anymore (via `deriving Show`, that is). We use the fact that `ExpF a` is a `Functor`, and the fact that 2 Functors can form a new Functor under composition[1]

3.1.1 Locational expressions

We have amended the parser to show the location of each part of the expression. This will be handy for error reporting later. For this variation we compose the `ExpF a` Functor with the `((Column, Row),a)` Functor

```
type LocExpF = (Compose ((,) (Int, Int)) ExpF)
type LocExp = Term LocExpF
```

3.1.2 Typed expressions

In this case, we compose the `ExpF a` functor with the `((Column, Row), Ty Int, a)` Functor. At this point however, the composition becomes a bit more convoluted (especially considering that during development, we also composed it with the `Either String a` functor), so we'll create a newtype for it

```
newtype TypExpF a = TypExpF ((Int, Int), Ty Int, (ExpF a))
    deriving (Functor, Foldable, Traversable)

type TypExp = Term TypExpF
```

[1] See Data.Functor.Compose in the haskell base library

3.2 The typing environment

For tracking variables we have a central, global state called `Env`. There are 4 variables in `Env`:

- `vars`: These are the variables that have a known type beforehand. These types cannot be refined further, and it will be a type error if we try to
- `inferred`: These are the variables with no known type. They can be refined to more specific types as we wish.
- `fresh`: This is a source of new bottom types.
- `funcArgs`: This is a map of function names to the types of their arguments

We pass `Env` through various different `State`[2] monads in order to keep track of all these variables. The analysis (both binding time and typechecking, though type checking already implies the BTA) goes from top to bottom, putting each succesful type inference and type check into it's environment to help assist with later definitions.

3.3 Type Inference

In SPL, expressions always have a type. In fact, as show in section 3.1, we have a type for each node of the ASTs representing the expression. This allows us to set up incremental type checks along the way. Having a type at every layer may also be helpful when compiling the expressions down to assembly. While the implementation of it is fairly convoluted, the general idea can be split up into 3 parts: Type equality, unification guarding, and then finally type checking on terms. We start off by preparing the AST. We give every "layer" in the AST a unique bottom type (with the help of `fresh`). This means that if 2 parts of an expression tree are polymorphic, they are guaranteed to still be identifiable by their polymorphic type number. This all happens in a type checking monad `TC`, which is a mix of the `State` and the `Maybe` monad, and derives it's features accordingly

```
newtype TC a = TC {outTC :: Env → Maybe (a, Env)}
    deriving (Functor, Applicative, Alternative, Monad,
              MonadState Env, MonadFail)
    via (StateT Env Maybe)
```

[2] Control.Monad.State docs

3.3.1 Type equality

We need to check if 2 different types are the same, modulo polymorphism. If we can say that they are equivalent on the type level, we return the most specific type. If 2 polymorphic types are compared, we return the polymorphic type with the lowest number (since that one is likely to be bound by something already)

```
tyEq :: Ty Int → Ty Int → Maybe (Ty Int)
tyEq (TyList xs) (TyList ys) = TyList <$> tyEq xs ys
tyEq (TyBot x) (TyBot y) = Just $ TyBot (min x y)
tyEq (TyBot _) y = Just y
tyEq y (TyBot _) = Just y
tyEq x y
    | x ≡ y = Just x
    | otherwise = Nothing
```

3.3.2 Unification Checking

Note how we don't take the environment into account when checking type equality. If a user states that a certain variable is strictly polymorphic, we need to make sure the expression doesn't accidentally specialise the polymorphic expression into booleans. This is why we have two types of typed variables in our environment. If the type checker found it tried to specialize a variable in `vars`, it will give an error. If it found the variable in `inferred`, it will instead add this specialization to the map, overriding the old definition.

3.3.3 Type checking

Finally, we will start type checking the definitions. This is done by first checking for equality against either static terms, or against other types in that layer. Then we check if we can unify the new type with the types we put in. Below are some examples

```
— type-checks "e.hd" for some typed expression "e"
checkUnOp (Sel Hd) subty = do
    n <- newFresh
    (TyList x) <- inTC $ tyEq subty (TyList (TyBot n))
    checkCanUnify subty (TyList x)
    return x
— typechecks numerical operations such as +,-,/
checkNumeral :: Ty Int → Ty Int → TC (Ty Int)
checkNumeral l r = do
    checkCanUnify l TyInt
    checkCanUnify r TyInt
    return TyInt
— typechecks the (==) operation
checkBinop Equals lty rty = do
    newTy <- inTC $ tyEq lty rty
    checkCanUnify lty newTy
    checkCanUnify rty newTy
    return TyBool
```

For the comparison (< and >) operations we have decided to allow `Char` and `Int` types to be comparable. [3].

```
checkCompare :: Ty Int → Ty Int → TC (Ty Int)
checkCompare l r = do
    lty <- inTC $ tyEq l TyInt <|> tyEq l TyChar
    rty <- inTC $ tyEq r TyInt <|> tyEq r TyChar
    checkCanUnify l lty
    checkCanUnify r rty
    return TyBool
```

The rest of the code is mostly glue code to send stuff back to the statement part of the type checker.

3.3.4 Function calls

One particularly tricky part of this are the function calls. Functions have a certain type signature, which have a set of polymorphic types in that signature. When we perform a function call we need to match the polymorphic variables from the function signature to the concrete types of the expressions passed to function at the callsite. Typechecking a function call happens in 3 steps:

- Typecheck the expressions that we give to the functions.
- Match the inferred types of the expressions to the types given to the arguments in the function type signature
- Make a mapping of the given types to their inferred types and check if there are no conflicts in polymorphic type variable instancing.

[4]

FIX THIS

3.4 Type Checking the rest

Equipped with an algorithm to infer the types of expressions, we now need to put them into an environment where those expressions are typed correctly. We do this by modifying the environment as we typecheck the statements and function definitions in our language.

3.4.1 Numbering our types

Types in SPL can be generic ("polymorphic" if we want to be formal). However, coming up with unique text variables is tough, and require us to constantly keep tabs on which variables are shadowed, and which ones are not. For this reason, we turn all our variables into numbers, and bind the fresh variable generation to the `fresh` function in our environment very early already. This way, when we know that 2 polymorphic types should be the same, we can just give them the same number.

[3] As seen in the previous examples, `==` is fully polymorphic. This is because all the data you can express with our type system is data you can compare for equality (functions are not expressed as part of the type system). This is important for making section 3.4.3 work

[4] As it stands right now, we don't unify types within function calls based on the types of the function arguments we're given. This is possible (and necessary), but we haven't gotten around to it yet

3.4.2 Function and Variable declarations

The most important part of declarations is that they get inserted into the typechecking environment. Currently, this is done in lock-step with the type-checking of other statements, but it can also easily be re-programmed to allow for forward definitions if that turns out to be a hard requirement. [5]

3.4.3 Statements

When typechecking function declarations, we define a state variable for the return type, along with whether the procedure is guaranteed to have returned already. This allows for a few things:

actually implement this after thursday

- First, it allows us to give certain control structures a "type". This is important for checking that certain branches always return a certain type.
- It also allows us to mark "unreachable code". If we are found still typechecking stuff while we are guaranteed to have returned
- Finally, it allows us to carry this information about the function we're currently typechecking to any possible return statement. While it is possible to type return statements without this, it is pretty painful

Below we'll describe some of the individual statement types we encounter and what we do with them.

Variable assignment

For variable assignment, we first typecheck the expression we're assigning the value of. If this succeeds, we typecheck the expression `varname == expression`, to check if the type of the expression can match the variable's type. Since `==` is expressable over all possible types, this can work.

Return statements

As said previously, we check the expression against the given returntype state. If the returntype state is `Nothing`, we add it in. Because the Return statement means that we stop executing further code in the branch, we set the termination flag in the state to "true".

While statements

This part is pretty straight-forward. The only thing is that we need to do is make sure that we can equate the loop condition to a boolean . We can use the `tyEq` function for this.

we forgot this part in the code. todo for after thursday

If statements

For the condition, we do the same as while statements. This type of statement is the only one that produces multiple branches. If either one of the branches terminates with a return statement, it updates the state accordingly. If the type of that return statement was not `TyVoid`, we require that the rest of the function should also return with an expression of the same type.

[5] One of the hard requirements is overloading of built-in functions. This will probably be implemented once we get to the compilation of built-ins themselves. As said in section 3.3 we do have a limited form of polymorphism for types that have to resolve to either `Int`s or `Char`s, which you can see as some form of overloading I suppose

If both branches end with a return statement, we check the 2 types for equality. Naturally this implies that both branches need to start of with the same return type state.

3.4.4 Errors

As opposed to parsing, finding multiple errors while typing the program is actually fairly easy. Because errors can occur at every node of our expressions, we can have multiple errors in the same expression if it is constructed sufficiently badly. Type checking expressions is also independent per expression, so every expression that reports type errors gets thrown into the total list of errors. This may have an unfortunate side-effect if the type checking affects the typechecking state such that there is more confusion down the line, but we'll just note that down as a small imperfection.

Chapter 4

Code Generation

4.1 Semantics

For the majority of the programming constructs, it is quite clear what the semantics should be, therefore we do not explain them here. The most interesting part is the calling semantics.

4.1.1 Scoping and shadowing

In our version of SPL, there are three scope levels:

1. The local scope (variables declared inside of the current function). We call this the highest scope.
2. The current function's parameters
3. The global scope. We call this the lowest scope.

If a variable from a lower scope is re-declared by a higher scope, then the new value is used within the current function. Re-declaration of a global variable does not affect this global variable outside of the current function.

See Figure 4.1 for an example program that demonstrates the scoping rules.

4.1.2 Calling convention

In our version of SPL, simple types (integers, booleans and characters) are simply copied when a function is called. Modifying them inside of the function does not affect the variable outside of the function. If you have a function $f(x)$, and you have a call to f with parameter y, then assigning to x within f does not affect y.

Lists are implemented as linked lists. They are passed as heap pointers. They follow the same rule that we just discussed, except in the case of assignments to a list head or tail. If you use `hd` or `tl` to assign to a list directly, then y is affected at the caller side. For example, `x.hd = 77` sets the head of x to 77 within f. It sets the head of y to 77 for the caller.

If you assign to global variables within a function, the global variable is always changed globally. This is true for both simple variables and lists.

See Figure 4.2 for an example program that demonstrates the calling convention.

```
var x = 42;

foo(y) {
    var x = 6;
    var y = 9;
    printLn(x);
    printLn(y);
}

main() {
    foo(0);
    printLn(x);
}
```

Figure 4.1: Example program that demonstrates the scoping rules. The output of this program is "`6 9 42`" (where newlines are replaced by spaces).

```
var ys = 0:[];

foo (x: Int) {
    x = x + 5;
}

bar (xs: [Int]) {
    xs = 42 : [];
    ys = 42 : [];
}

door (xs: [Int]) {
    xs.hd = 77;
    printLn(xs.hd);
}

main () {
    var x = 5;
    var xs = 1:[];
    foo(x);
    bar(xs);
    printLn(x);
    printLn(xs);
    printLn(ys);
    door(xs);
    printLn(xs);
}
```

Figure 4.2: Example program that demonstrates the calling convention. The output of this program is "`5 [1, ] [42, ], [77, ]`" (where newlines are replaced by spaces).

```
f(x) {
    printLn(x);
}

g(x) {
    f(x);
}

main() {
    g('c');
    g(42);
}
```

Figure 4.3: Example program that demonstrates polymorphism. The output of this program is "`c 42`" (where newlines are replaced by spaces).

4.1.3 Polymorphsim

In our version of SPL, users are allowed to define and use polymorphic functions, and to reference another polymorphic function from within the current polymorphic function. See Figure 4.3 for an example.

4.1.4 Standard library

print and printLn

There are two overloaded functions in the standard library, namely `print` an `printLn`. They can print any type. Lists are always printed comma-separated, e.g. `[1, 2, 3,]`, except for lists of characters (strings), which are printed without any separation or brackets.

Printing integers and characters should be self-explanatory. Booleans are printed as strings "True" or "False".

Equality

There are two overloaded operators: `==` and `!=`. They can compare any two elements of the same type to each other. In the case of lists, an element-wise comparison is used.

The rest of the standard library should be self-explanatory.

4.2 Representation of data

There are two important distinctions to make in representing data. First, there is a distinction between simple types and lists. Then there is also a distinction between global variables, local variables and parameters.

4.2.1 Simple values and lists

Simple values (integers, booleans and characters) are simply loaded on the stack and copied directly. Lists are allocated on the heap. When lists are passed around, only the pointer is

copied (shallow copy).

Lists are implemented as linked lists. When created, they are stored in the heap using a series of node nodes, that each have a pointer to the next node and a value. The empty list is represented by a pointer to the address 0.

4.2.2 Scope

Global variables are stored on the heap. Before the main function executes, special space for global variables is reserved there. If you have a list as a global variable, then there is a pointer on the heap that points to a list node further on the heap.

Local variables and parameters are stored on the heap, relative to the MP.

4.3 Compilation scheme

We generate SSM code using the following high-level steps. Code is generated completely sequentially.

1. Generate code that allocates space on the heap to store local variables.
2. Scan the AST for polymorphic function declarations and save them in the compiler's state.
3. Generate code that initializes the global variables. Remember all calls to polymorphic functions.
4. Generate code that jumps to the label main.
5. Generate the function declarations. Remember all calls to polymorphic functions.
6. For each call to a polymorphic function, generate a monomorphic version. Once again, remember all calls to polymorphic functions. Continue doing this until there are no more unresolved polymorphic function calls.

We will discuss each of these in more detail in the next section.

4.4 Implementation

4.4.1 Haskell Lenses

Generating SSM code is a process that requires at least some notion of state, and it is also generally more intuitive as an imperative program. In functional languages like Haskell however, stateful programming can be difficult and tedious.

This is why we extensively use Haskell Lenses. Briefly speaking, this language extension enhances the Haskell State with a lot of helper function that make stateful programming easier. It is not very important to understand how Lenses work to understand our compiler. For more information, we refer to The Haskell Wikibook (Lenses).

4.4.2 File structure

The following files are relevant for the code generation.

- In `app/Codegen/GenState.hs`, we define the state monad that we use for the code generation, using Haskell lenses. The fields of the `GenEnv` record will be explained when they are relevant in further sections.

- `app/Codegen/Codegen.hs` contains most of the definitions that are used for code generation, including `genProgram`.
- In `app/Codegen/Helpers.hs`, some helper functions for the code generations are defined. The most important definitions.
- `app/Codegen.hs` contains IO that calls the code generation for a specific target file. The output is saved in `out.ssm`. The standard library is also inserted here.
- `stdlib.spl` contains the definitions of the standard library functions in our version of SPL. This is further explained in the section about overloading.

4.4.3 The state

The state monad that is used during code generation is called `GenState`. As aforementioned, it is implemented using Haskell lenses. The most important fields are the following.

- `localVars`, `globalVars`, `params2`. These are all Maps from identifiers to a memory address. These adresses are relative to the MP in the case of local variables or parameters, and relative to the bottom of the heap in the case of global variables.
- `lines2`. Stores the program that has been generated so far as a list of Strings. This variable is never directly referenced, but always through the function `emit`.
- `address`. Keeps track of the next free address. Used when declaring variables.
- `scope`. Keeps track of the current scope, which can be global, local, or main.

4.4.4 Code generation

The function `genProgram` generates the whole program. It caries out the steps outlined in the previous section by calling `genGlobalVarSpace`, `genGlobalVars`, `genDecl`, and `resolvePolyFunctions`. Each of these is explained in more detail in the next sections.

4.4.5 genGlobalVarSpace (step 1-2)

This function does two things.

1. Allocate space for the global variables on the heap. In other words, the heap pointer is offset by however many global variables we come across. The location of these global variables is `not yet` stored in the compiler's state.
2. All functions declarations on the AST with polymorphic parameters are found, and they are stored in a map `polyFunctions`. In order to resolve function calls later, we need to know which functions are polymorphic in advance.

4.4.6 genGlobalVars (step 3)

The global vars are initialized. This means that the expression that they are assigned gets evaluated, and the result will be stored in the appropriate spot on the heap. After this step, the location of the variable in memory is stored in the compiler's state, in the map `globalVars`. This means that the compiler will know where to reference global variables later.

Global varriable declarations are separated from the other declarations because they need to be initialized before branching to main.

4.4.7 genDecl (step 5)

There are two kinds of declarations: Variable declarations and function declarations. All global variable declarations are skipped because they are already handled by `genGlobalVars`.

Local variable declarations

The local variable declarations are handled in a similar way as the global variables before. A free spot (this time relative to the MP) is chosen, and this spot is stored in the map `localVars`.

Function declarations

Function declarations are more interesting. First of all, functions that have polymorphic arguments are skipped completely in this stage. This is because we do not know what monomorphized functions we will need for the function in question. This will be resolved later.

If this is not the case, we set the map `param2`, such that it correctly references the function parameters. We generate a label name so that the function can actually be called. This label name consists of the function and the types of the argument. For example, the function `foo` that takes as arguments an integer and a character, will get the label "foo_ic". This is essential for monomorphization later.

An interesting detail to mention is that a return statement is always inserted at the end of a function, except in the main function where a halt is inserted instead.

4.4.8 genStmt (step 5)

Called by genDecl.

If and while

If statements and while statements are not that interesting. A state variable `if_count` is used to generate a fresh if or while label when needed.

Assignments

Assignments are implemented by retrieving the location that was stored in `globalVars`, `localVars` or `params2` and generating code that loads this address on the stack.

Function calls with effects

In our AST, we have a statement `EffCall`, which is a function call that is purely executed for the side effects.

Jump to the label using 'bsr', and clean the arguments from the stack using 'ajs'. If this was a function call to a polymorphic function, save it in `polyFunctionCalls`. More specific cases related to overloading are explained later.

4.4.9 genExpr (step 5)

Called by genDecl and genStmt.

Identifiers

Identifiers are loaded of the stack. The compiler knows in which memory address the variable is stored thanks to `globalVars`, `localVars` and `params2` in the state.

Function calls

Function calls are implemented by calling `genStmt` with a corresponding `Effcall`, and reading the result from the return register.

4.5 Overloading and polymorphism (step 6)

As aforementioned, we have polymorphic functions, and we allow programmers to write their own. We have implemented this using monomorphization: we keep track of all calls to polymorphic functions, and then we generate code for instances of these functions that take the type of these calls. For example, in the SSM code generated in Figure 4.3, we generate code with labels `f_i`, `g_i`, `f_c`, and `g_c` for the instances of `f` and `g` with integers and characters respectively.

This is implemented using the following loop

1. Let $f(y)$ be a function with polymorphic argument y. Let $f(x)$ be a call to $f(y)$ with $x : t$ such that t is known on compile time.
2. Generate a monomorphic instance of f, where the type of y is set to the type of x. Use type inference for the monomorphization of the AST for f.
3. Remember any new calls to polymorphic functions that were encountered while generating the monomorphization of f.
4. If there are no more unresolved calls to polymorphic functions, then we are done.

When a function call is generated, the compiler should always know what the types of the arguments are on the side of the caller, otherwise it will not know what to do. The only exception to this is the empty list. Currently, it is only possible to print the empty list.

4.5.1 Printing

Printing is implemented using overloading. There are four special cases.

- `Int`. The value on the stack is printed as an integer.
- `Char`. The value on the stack is printed as a character.
- `Bool`. Call `printBool` (defined in the standard library)
- `List`. Call `printList` (defined in the standard library)

Note that `printList` is polymorphic (type [a]), and it may introduce another overloaded instance of print (type a).

4.5.2 Equality

Equalilty is implemented in a very similar way.

- `Int`, `Char`, `Bool`. Do a simple value comparison in place.
- `List`. Call `ListEquals` (defined in the standard library)

4.6 Testing

We have written a lot of small test programs during the development of the code generator. They are not implemented in the CI yet due to time constraints. We will look if we can test the output of programs inside of the CI.

4.7 Problems

- The most difficult/annoying problem that we still want to solve for the code generation is the type inference in step 2 under 4.5. We would like to re-use our type checking code for this, but currently it is only defined for the untyped AST. If we can get this to work properly, then our version of SPL will be impressively polymorphic.
- This also ties into the fact that the overloading for equality doesn't currently work as it should. This is a direct consequence of the bad type checking: because of this, the code generator can't derive the type of the arguments of `!=` in line 15 in `stdlib.spl`. For now we have implemented it such that a simple comparison is used when the compiler fails to derive the type of the arguments. This means that if you compare nested lists, the result is currently incorrect. (It compares pointers on the nested level).
- Assigning to a list's head or tail does not work yet because we forgot about it.
- If you create a (polymorph) empty list, then the only thing you can currently do with it is print it out. If you try to pass it as an argument, you will get a compiler error. We are not sure if this is worth improving, because this could lead to really difficult type checking problems, for example:

```
f(x, xs) {
    return x : xs;
}

main ()
{
    var x = 1;
    var xs = [];
    f(x, xs);
}
```

Chapter 5

Extension

Describe your extension in detail

Chapter 6

Conclusion

What does work, what does not etc.

6.1 Reflection

- What do you think of the project?
- How did it work out?
- How did you divide the work?
- Pitfalls?
- ...

Appendix A

Grammar

Change the grammar to the one you actually used

```
SPL       = Decl+
Decl      = VarDecl
          | FunDecl
VarDecl   = ('var' | Type) id  '=' FExp ';'
FunDecl   = id '(' [ FArgs ] ')' [ ':' RetType ] '{' VarDecl* Stmt+ '}'
RetType   = Type
          | 'Void'
Type      = BasicType
          | '[' Type ']'
          | id
BasicType = 'Int'
          | 'Bool'
          | 'Char'
FArgs     = [ FArgs ',' ] id [ ':' Type ]
Stmt      = 'if' '(' FExp ')' '{' Stmt* '}' [ 'else' '{' Stmt* '}' ]
          | 'while' '(' FExp ')' '{' Stmt* '}'
          | id [ Field ] '=' FExp ';'
          | FunCall ';'
          | 'return' [ FExp ] ';'
FExp      = Exp [ Field ]
Exp       = id
          | FExp Op2 FExp
          | Op1 FExp
          | int
          | char
          | 'False' | 'True'
          | '(' FExp ')'
          | FunCall
          | '[]'
Field     = '.' 'hd' | '.' 'tl'
FunCall   = id '(' [ ActArgs ] ')'
ActArgs   = FExp [ ',' ActArgs ]
Op2       = '+'  | '-' | '*' | '/'  | '%'
```

```
          | '==' | '<' | '>' | '<=' | '>=' | '!='
          | '&&' | '||'
          | ':'
Op1       = '!'  | '-'
int       = [ '-' ] digit+
id        = alpha ( '_' | alphaNum)*
```

Appendix B

Haskell definition of the abstract syntax tree

```
type Prog ty exp = [Decl ty exp]

data Decl ty exp
    = Var (VarDecl ty exp)
    | FunDecl String [(String, Maybe (Ty ty))] (Maybe (Ty ty)) [VarDecl ty exp] [Stmt exp]

— Strings for parsing, Ints for typechecking
data Ty a
    = TyInt | TyBool | TyChar | TyVoid
    | TyList (Ty a) | TyBot a

data ExpF a
    = Ident String
    | Binop Op2 a a
    | UnOp Op1 a
    | LitInt Integer
    | LitChar Char
    | LitBool Bool
    | Funcall String [a]
    | EmptyList
newtype Term f = In { out :: f (Term f)}

type Exp = Term ExpF
data Op1 = Sel Field | Not | Negate | Minus

data Field = Hd | Tl

data Op2
    = Add | Sub | Mul| Div | Mod
    | Equals | Lt | Gt | Le | Ge | Ne
    | And | Or | Cons
```

```
data VarDecl a exp = VarDecl (Maybe (Ty a)) String exp

data Stmt exp
    = If exp [Stmt exp] [Stmt exp]
    | While exp [Stmt exp]
    | Assign String (Maybe Field) exp
    | Effcall String [exp]
    -- same as a FunCall, but as a statement (has to be effectful to do something)
    | Return (Maybe exp)
```