My SPL Compiler

Quinten Cabo
s1076992

June 26, 2024

Contents

Chapter 1

Introduction

This is the final report for my Simple Programming Language (SPL) compiler.

1.1 Simple Programming Language

SPL is a C-style language, but with the following main differences:

- Support for polymorphic data types.
- Type inference
- No pointers
- Type safe operators, you get a type error with + on a chars for example
- Overloaded print and equality
- Arrays are quite different with the : (cons) operator and .tl and .hd access fields
- Variables should be defined at the top of a function.

Here are a couple examples of SPL.

Hello world

This example shows how to write a "Hello World" program in SPL. One can create a Hello World program by printing a list of characters. Canonically there is no literal list syntax in SPL but only is the : (cons) operator.

```
main() : Void {
    print('H':'e':'l':'o':' ':'W':'o':'r':'d':[]);
}
```

However, I added a literal string syntax that is parsed into cons syntax.

```
main() : Void {
    print("Hello world");
}
```

Just like c, SPL has if and else statements and while loops.

```
// This is one line SPL comment
/* Multi
line /* Nested */ comment */

countdown(x: Int) {
    Int zero = 0;
    while (x > zero) { // x is inferred as Int
        if (x == 5) {
            print("count is greater than 5\n");
        } else {
            print("Count is not greater than 5");
        }
```

```
        x = x - 1;
    }
}

main() : Void {
    countdown(10);
}
```

Because of the type inference you can actually leave out the types in SPL. This means that the following code:

```
var a = 3;

foo(x) {
    var zero = 0;
    var list = [];
    if (x) {print(x);}
    return a:zero:list;
}

main() {
    foo(a > 10);
    return;
}
```

is valid and will be inferred as:

```
Int a = 3;

foo(x : Bool) : [Int] {
    Int zero = 0;
    [t] list = [];
    if (x) {print(x);}
    return a:zero:list;
}

main() : Void {
    foo(a > 10);
    return;
}
```

Here is an example of a program with multiple type errors:

```
foo() {
    var list = [];
    3:list;
    return 'a':list; // Can not add a char anymore because list is inferred as [Int]
}

bar() {
    if (1) { // Not a boolean!
        return 3 == ([1] + 2); // Non matching types!
    }
}

fee(x) {
    if (x) {return 3;}
    return '3'; // Invalid return type!
}
```

This next program shows off tuple and list support as well as overloaded equals and printing.

```
var q = 137:[];

main() {
    var aa = ("Equal!", q.hd + 1);
    var zz = ("-Equal!", q.hd);

    var bb = (aa, aa);
    var yy = (zz, zz);

    var cc = (bb, bb);
    var xx = (yy, yy);

    print("cc: "); print(cc); print(); // Print without arguments print a new line
```

```
    print("xx: "); print(xx); print("\ncc and xx are ");
    if (cc == xx) { print(bb.snd.fst); } else { print("Not Equal!"); }

    aa.snd = bb.snd.snd - 1; // Make them the same
    zz.fst = zz.fst.tl;

    print("\n\ncc: "); print(cc); print();
    print("zz: ");   print(zz); print("\ncc and zz are ");
    if (cc == xx) { print(bb.snd.fst); } else { print("Not Equal!"); }
}
```

The type checked version of this program is:

```
[Int] q = 137:[];

main() : Void {
    ([Char], Int) aa = ('E':'q':'u':'a':'l':'!':[], q.hd + 1);
    ([Char], Int) zz = ('-':'E':'q':'u':'a':'l':'!':[], q.hd);
    (([Char], Int), ([Char], Int)) bb = (aa, aa);
    (([Char], Int), ([Char], Int)) yy = (zz, zz);
    ((([Char], Int), ([Char], Int)), (([Char], Int), ([Char], Int))) cc = (bb, bb);
    ((([Char], Int), ([Char], Int)), (([Char], Int), ([Char], Int))) xx = (yy, yy);

    print('c':'c':':':' ':[]);
    print(cc);
    print();
    print('x':'x':':':' ':[]);
    print(xx);
    print('\n':'c':'c':' ':'a':'n':'d':' ':'x':'x':' ':'a':'r':'e':' ':[]);
    if (cc == xx) {
        print(bb.snd.fst);
    } else {
        print('N':'o':'t':' ':'E':'q':'u':'a':'l':'!':[]);
    }
    aa.snd = bb.snd.snd - 1;
    zz.fst = zz.fst.tl;
    print('\n':'\n':'c':'c':':':' ':[]);
    print(cc);
    print();
    print('z':'z':':':' ':[]);
    print(zz);
    print('\n':'c':'c':' ':'a':'n':'d':' ':'z':'z':' ':'a':'r':'e':' ':[]);
    if (cc == xx) {
        print(bb.snd.fst);
    } else {
        print('N':'o':'t':' ':'E':'q':'u':'a':'l':'!':[]);
    }
    return;
}
```

The output of this program is:

```
cc: ((("Equal!", 138), ("Equal!", 138)), (("Equal!", 138), ("Equal!", 138)))
xx: ((("-Equal!", 137), ("-Equal!", 137)), (("-Equal!", 137), ("-Equal!", 137)))
cc and xx are Not Equal!

cc: ((("Equal!", 137), ("Equal!", 137)), (("Equal!", 137), ("Equal!", 137)))
xx: ((("Equal!", 137), ("Equal!", 137)), (("Equal!", 137), ("Equal!", 137)))
cc and xx are Equal!
```

1.2 Implementation language

The compiler is implemented in Haskell [6]. I made this choice since Haskell is a good fit for compiler construction. This is due to Haskell's support for algebraic data types, monads and its well-suitedness for writing parsers.

I also chose Haskell because I wanted to learn more about it. This made the first phases of the project more challenging because my Haskell knowledge was quite rusty. However, in the end I am pleased with how much I learned about Haskell. I gained an actual intuition for how to use applicative functors, monad and monoid.

Chapter 2

Lexical analysis

Lexical analysis, the first stage of the compilation process, involves reading the source code and transforming it into an Abstract Syntax Tree.

2.1 Abstract Syntax Tree

I initially designed the abstract syntax tree (AST) of SPL by looking at many provided SPL examples. I kept adding to the AST until I was that sure I captured everything expressed in those examples. When I later had to add source locations to the AST I decided to implement the Trees That Grow paper [10]. I will first explain the design of the AST without 'Trees That Grow' as this is more comprehensible.

Our abstract syntax tree is split up into four major inductive types, `Decl`, `Stmt`, `Expr` and `Literal`. The essential idea of the design is that a program is just a list of declarations.

```
type Program = [Decl]
```

Declarations A `Decl` can be a function declaration or a variable declaration. A variable declaration just has a name of type `String` and an expression of type `Expr`. A function deceleration has a name `String`, a return type of type `Type`, a list of arguments of type `[(Expr,Type)]`, a list of function variable declarations of type `[Decl]` and a function body of type `[Stmt]`. That gives

```
data Decl = FunDecl String Type [(String, Type)] [Decl] [Stmt]
```

For the function variable declarations I have actually reused the Decl type. This allowed me to reuse code that checks global variables for function variables. However, this also means that in theory nested function definitions would be valid in the AST. However, the parser will not parse this on purpose.

Statements The body of a function is just a list of statements. Possible statements include `if` with maybe an `else`, `while`, variable assignment like `a = 3;` but also `a.hd = 137;` and finally expression statement (`Expr;`). The expression statement allows calling functions without assigning to anything. This also allows expression statements that do not do anything like `3+3;` or `'a';`. But this is not a problem because all expression statements that do not have a function call expression inside are optimized away.
When I was designing the optimiser I also added the `BlockStmt`. This is a statement that is basically just a list of other statements. `BlockStmt` makes `Stmt` a monoid which I needed during the optimization step in order to remove or rewrite statements. The parser can not create these `BlockStmt`. This leaves the `Stmt` AST (before trees that grow) as:

```
data Stmt = AssignStmt Variable Expr
  | ReturnStmt (Maybe Expr)
  | IfStmt Expr [Stmt] (Maybe [Stmt])
  | WhileStmt Expr [Stmt]
  | ExprStmt Expr
  | BlockStmt [Stmt]
```

The `AssignStmt` has the `Variable` type. This type is there to support assigning to fields. For instance `a.hd = 3;` or `a.snd = [];` or `a.tl = 1:2:3:[];`. As such `Variable` looks like the following:

```
data Variable = Identifier String (Maybe Field)
data Field = HeadField | TailField | FirstField | SecondField
```

Expressions An `Expr` can be a unary operation like `-3`, a binary operation like `134 + 3`, a function call like `foo()` or `foo(1,2,3)`, a variable expression like `a` or `a.hd` and finally a `Literal` expression like `3` or `'a'` or `[]`. This gives the following data type:

```
data Expr
  = BinOpExpr BinOp Expr Expr
  | UnaryOpExpr UnaryOp Expr
  | FunctionCallExpr String [Expr]
  | VariableExpr Variable
  | LiteralExpr Literal

data UnaryOp = Negate | FieldAccess Field | Min
data BinOp = Mul | Div | Mod | Lt | Sub | Cons | Gt | Gte | Add | Lte | Eq |
             Neq | And | Or
```

Literal There are 7 types of `Literal` constructors. They speak for themselves in the data definition:

```
data Literal
  = TrueLit | FalseLit
  | IntLit Int
  | CharLit Char
  | TupleLit (Expr, Expr)
  | EmptyListLit
```

Types Finally there is the data type that I use to represent the possible types.

```
data Type
  = IntType | CharType | BoolType | VoidType
  | TupleType Type Type
  | ListType Type
  | TypeVar { typevarname :: String, rigid :: Bool }
  | FunType [Type] Type
```

The `VoidType` is only used for functions that do not return a value. A `TypeVar` is the most general type. Type variables are used when the compiler does not know the type of something yet. A `TypeVar` has a name so that the compiler can keep track of the identity of a type variable. A `TypeVar` is rigid when the programmer themselves specified that something as a type variable. For instance by writing:

```
id(x: a) : a {
   return x;
}
```

Here the programmer tells the compiler that the argument of the `id` function can be any type and that the return value of the function is the same as the type of the first argument. A rigid type variable is different from non rigid (or inferred) type variable because a rigid type variable can not be narrowed down further to a more concrete type like `Char` or `Int`. Non rigid type variables can be narrowed further to more concrete types. Only the programmer can specify rigid type variables. That means that after parsing no more rigid type variables are introduced. Any `TypeVar` that is introduced by the compiler during type checking is non rigid.
`FunType` is used to represent the type of functions. As the compiler does not allow functions as values, the `FunType` is not really used in the type checking. It is rather used to represent the types of `FunDecl`. The remaining types speak for themselves.

2.1.1 Trees that grow

The previously described AST is missing source code location information and expressions do not have room for types. During different phases of the compiler has different amounts of information. For instance, initially many types are not specified and so the type of an expression should be a Maybe. At

first I thought that the only way to manage these different levels of information was to create separate, similar but distinct trees for every phase of the compiler. Like this example (with only literal and binary operation expression for brevity) of expressions for multiple phases of the compiler:

```
data ExprEmpty
= BinOpExprEmpty BinOp ExprEmpty ExprEmpty
| LiteralExprEmpty Literal

data ExprParsed
= BinOpExprParsed SourcePos (Maybe Type) BinOp ExprParsed ExprParsed
| LiteralExprParsed (Maybe Type) SourcePos Literal

data ExprTypechecked
= BinOpExprTypechecked SourcePos Type BinOp ExprTypechecked
| LiteralExprParsed Type SourcePos Literal
```

This quickly got tiring. This approach requires making modifications in many places to make changes to the AST, since you have to keep all the different data types in sync. Additionally, it forces you to use long names for the different data constructors and makes it impossible to make functions that work on generic ASTs (e.g. functions that are agnostic to the phase of the AST). I wanted a flexible system to annotate the AST with additional metadata which could be different at each phase of the compiler. To achieve this, I have implemented a variation of the Trees that Grow paper [10]. The approach allows to annotate a tree with different types of metadata depending on the phase of the compiler.

This approach to annotating the AST with metadata is based on type families. It works by adding a parameter to each (relevant) data type that determines the phase of the compiler. Using type families you can then create separate instances of data based on the phase of the compiler while keeping the general structure of the AST the same.

The trees that grow paper promises to make it easy to extend the AST with more phases in the future where you have different metadata needs. At the time I thought that without type families you would need to build a separate, similar but distinct tree for every phase of the compiler. However, at the end of the project I found out that this was not the case. You can achieve basically the same effect of type families by using polymorphic types in your data type definition. I reflect more on why trees that grow was not the best fit for this project in Section 2.1.1.

Implementing "Trees That Grow"

Now I will go into more depth about how I implemented trees that grow into the compiler.

Type families [3] can be seen as functions from types to types. These functions can take types as input and produce other types as output. The idea of trees that grow is to define general data types that take a compiler phase type as input to form another type.

By using the Haskell `DataKinds` language extension, the phases of the compiler can be represented as a data type. This is something I think is an improvement over the original paper as they used multiple data without constructors instead.

```
{- All the different phases of the compiler -}
data Phase
  = EmptyP          -- Empty phase, used for testing
  | ParsedP         -- Phase after parsing with location information
  | ReturnsCheckedP -- Phase after the returns have been checked
  | TypecheckedP    -- Phase after full type checking
```

Lets add the phases to binary operation expressions and literal expressions as an example.

```
data Expr (p :: Phase)
= BinOpExpr (BinOpExpr p) BinOp (Expr p) (Expr p)
| LiteralExpr (LiteralExpr p) Literal
```

The Expr data type takes a phase type as input. Two types take the phase as input: `BinOpExpr` and `LiteralExpr`. The type families for this example will then be:

```
type family BinOpExpr (p :: Phase) where
  BinOpExpr EmptyP = ()
  BinOpExpr ParsedP = SourceSpan
  BinOpExpr ReturnsCheckedP = SourceSpan
  BinOpExpr TypecheckedP = (Type, SourceSpan)

type family LiteralExpr (p :: Phase) where
  LiteralExpr EmptyP = ()
  LiteralExpr ParsedP = SourceSpan
  LiteralExpr ReturnsCheckedP = SourceSpan
  LiteralExpr TypecheckedP = (Type, SourceSpan
```

Note that these are closed type families because at this time I do not need to allow others to add compiler phases and closed type families give better type inference in Haskell.

The full AST using type families looks like this:

```
type Program (p :: Phase) = [Decl p]

data Decl (p :: Phase)
  = VarDecl (VarDecl p) String (VarDeclT p) (Expr p)
  | FunDecl (FunDecl p) String (FunDeclT p) [(String, FunDeclT p)] [Decl p]
            [Stmt p]

data Stmt (p :: Phase) =
  AssignStmt (AssignStmt p) Variable (Expr p)
  | ReturnStmt (ReturnStmt p) (Maybe (Expr p))
  | IfStmt (IfStmt p) (Expr p) [Stmt p] (Maybe [Stmt p])
  | WhileStmt (WhileStmt p) (Expr p) [Stmt p]
  | ExprStmt (ExprStmt p) (Expr p)
  | BlockStmt [Stmt p]

data Expr (p :: Phase) =
  BinOpExpr (BinOpExpr p) BinOp (Expr p) (Expr p)
  | UnaryOpExpr (UnaryOpExpr p) UnaryOp (Expr p)
  | FunctionCallExpr (FunctionCallExpr p) String [Expr p]
  | VariableExpr (VariableExpr p) Variable
  | LiteralExpr (LiteralExpr p) (Literal p)

data Literal (p :: Phase) =
  TrueLit
  | FalseLit
  | IntLit Int
  | CharLit Char
  | TupleLit (Expr p, Expr p)
  | EmptyListLit

data BinOp = Mul | Div | Mod | Add | Sub | Cons | Gt | Gte | Lt | Lte | Eq
           | Neq |  And | Or
data UnaryOp = Negate | FieldAccess Field | Min

data Type
  = IntType | CharType | BoolType | VoidType | TupleType Type Type
  | ListType Type | TypeVar { typevarname :: String, rigid :: Bool }
  | FunType [Type] Type
```

The actual type families are all quite similar. Every type family which name ends with `Expr` is the same as the two instances shown above in the example of `BinOpExpr` and `LiteralExpr`. Every type family ending in `Stmt` or `Decl` has instances like this:

```
type family IfStmt (p :: Phase) where
  IfStmt EmptyP = ()
  IfStmt ParsedP = SourceSpan
  IfStmt ReturnsCheckedP = SourceSpan
  IfStmt TypecheckedP = SourceSpan


type family VarDecl (p :: Phase) where
  VarDecl EmptyP = ()
  VarDecl ParsedP = SourceSpan
  VarDecl ReturnsCheckedP = SourceSpan
  VarDecl TypecheckedP = SourceSpan

-- etc same for AssignStmt, ReturnStmt, WhileStmt
```

Every type family which name ends with `DeclT` have the following instances:

```
type family VarDeclT (p :: Phase) where
  VarDeclT EmptyP = ()
  VarDeclT ParsedP = Maybe Type
  VarDeclT ReturnsCheckedP = Maybe Type
  VarDeclT TypecheckedP = Type

type family FunDeclT (p :: Phase) where
  FunDeclT EmptyP = ()
  FunDeclT ParsedP = Maybe Type
  FunDeclT ReturnsCheckedP = Maybe Type
  FunDeclT TypecheckedP = Type
```

Reflection on Trees That Grow

In the end, the trees that grow approach caused much more hassle then it was worth. I could have achieved the same by adding some polymorphic types to the data in the AST and making type synonyms to keep things short. At the time I added the type families from Trees that Grow I was not sufficiently experienced with polymorphic types and data types to come up with this. Instead I looked for a paper that solved our problem and implemented that instead. Ultimately just using polymorphic types in the data would have been much simpler and would have led to much less boilerplate. I also remember that I thought that the types of the annotations would differ much more than they ended up doing. In the end each data only needed the type of the Source Location and the type of the type of the node which was either Maybe Type or Type.

Another benefit besides the metadata that I thought this approach provided was enforcing the order of the phases of the compiler at the type level. What I mean with this is that the only way to get a `Program TypecheckedP` is from the type checker function. However, the type checker function only takes `Program ReturnsCheckedP` which you can only get by giving a `Program ParsedP` to the return checker. This way it is actually not possible to skip the return checking.

But this could also have been achieved using a data definition that just takes multiple polymorphic types as input. There was no need to involve the complexities of type families. Here is an example of that:

```
type Program meta types = [Decl meta types]
type ReturnsCheckedProgram = Program SourceSpan (Maybe Type)
type TypeCheckedProgram = Program SourceSpan Type
```

The type families led to a significant amount of boilerplate. For instance when deriving type classes like `Show` and `Eq` you have to derive it separately for every possible instance of the AST per class you want to derive. Here is an example of deriving `Show` and `Eq Expr`:

```
deriving instance Eq (Expr EmptyP)
deriving instance Eq (Expr ParsedP)
```

```
deriving instance Eq (Expr ReturnsCheckedP)
deriving instance Eq (Expr TypecheckedP)

deriving instance Show (Expr EmptyP)
deriving instance Show (Expr ParsedP)
deriving instance Show (Expr ReturnsCheckedP)
deriving instance Show (Expr TypecheckedP)
```

In theory it sounds good that you could decide not to derive some of these type classes for certain phases of the compiler. But in practice you need this for every phase anyway.

Another annoying thing is that Haskell just can't infer that two instances of the same type family are of the same type. Like for instance with:
`VarDeclT ParsedP = Maybe Type` and also `VarDeclT ReturnsCheckedP = Maybe Type`
Both of these type family instances are `Maybe Type` but Haskell still does not know. This means that you can not touch the contents of the nodes, and you can not convert them to another phase with identity. So a function with type `Literal ParsedP -> Literal ReturnsCheckedP` can not be id.

To solve this I made a type class to convert a data to the next phase if the transformation is trivial id in practice.

```
class Convertable n (p1 :: Phase) (p2 :: Phase) where
    upgrade :: n p1 -> n p2
```

You then recursively apply this to a node to upgrade it to the next phase. This used to be a lot of boilerplate but in the end I actually mostly do meaningful transformations. For instance when converting from `RetrunsChckedP` to `TypecheckedP` I actually have to replace the Nothing with type variables so the conversion can not be `id` anyways.

When you want to have a type class that can work on multiple phases of the compiler like:

```
class Prettier a where
    {-# MINIMAL pretty #-}
    pretty :: a -> String
    prettyBrief :: a -> String
    prettyBrief = pretty
```

You still need to implement it for every phase separately. Even though in reality the actual types of UnaryOpExpr p are all the same. At least here the upgrade function helps a little but because you can just upgrade most things to the latest phase (in this case TypecheckedP), and only implement pretty for that. Later I learned about unsafeCoerce which probably would have worked quite good as well.

2.2 Parsing

The source code is transformed into an AST instance using parser combinators. Parser combinators are higher order functions that allow you to combine parser functions in standard ways using monad and applicative functors. A parser function consumes a part of the input string and transforms this consumed string into parsed data. In this cased the parsed data are AST nodes. Instead of consuming a part of the input a parser function can also fail if the input is not what it expects.

Parser combinators work through function composition. You compose simple parsers, such as "parse a single character", together using parser combinators to create more complicated parsers (such as "parse the word 'parser' "). This composition usually happens through helper functions. For example, the `<|>` combinator takes two parsers, and constructs a new parser that first tries the parser on the left, and only if it fails tries the parser on the right.

Due to Haskell support for monads and applicative functors it is very suited to build parser combinators. Initially I built my own parser combinators. While this was a good learning experience, I soon realised this would require a significant amount of work to get it to be any good. For example it would be difficult to get good error messages with source code positions. I decided that using more mature tools (e.g. an existing parser combinator library) would give us more time to work on other parts of the compiler. Therefore, I switched to the monadic parser combinator library megaparsec [7], which is the (informal) successor of the popular parsec library [5]. To become more familiar with the library, and

for occasional tips, I used an excellent tutorial [4] by Mark Karpov, the maintainer of megaparsec to learn more about it. Megaparsec uses monads to hide a large part of the internal mechanics of parser combinators. For instance carrying error messages or saving corresponding source position, are hidden away in the megaparsec Parser type. This allows to use a number of useful combinators Haskell itself provides to compose parsers (e.g. do notation, `many`, `some`, `<$>` and `<*>`).

When using the megaparsec library, you get quite a number of things for free, such as error handling, associativity and also nested multi line comments.

2.2.1 Handling associativity

To deal with associativity Megaparsec provides the helper function `makeExprParser` which, given an operator table and a parser for terms (e.g. any expression that does not have an operator, such as literals), constructs a parser that has the specified operators in the specified order or precedence and with the specified associativity.

Our operator table is as follows:

```
operatorTable :: [[Operator Parser (Expr ParsedP)]]
operatorTable =
    [ [ Postfix (unary (FieldAccess HeadField) (try (L.tDot <* L.tHead)))
      , Postfix (unary (FieldAccess TailField) (try (L.tDot <* L.tTail)))
      , Postfix (unary (FieldAccess SecondField) (try (L.tDot <* L.tSnd)))
      , Postfix (unary (FieldAccess FirstField) (try (L.tDot <* L.tFst)))
      ]
    , [ Prefix (unary Negate L.tExcl)
      ]
    , [ InfixL (binary Mul L.tStar)
      , InfixL (binary Div L.tSlash)
      , InfixL (binary Mod L.tPercent)
      ]
    , [ InfixL (binary Add L.tPlus)
      , InfixL (binary Sub L.tMin)
      , Prefix (unary Min L.tMin)
      ]
    , [ InfixR (binary Cons L.tColon)]Semantic Analysis
    , [
        InfixN (binary Gte L.tGte)
      , InfixN (binary Gt L.tGt)
      , InfixN (binary Lte L.tLte)
      , InfixN (binary Lt L.tLt)
      , InfixN (binary Eq L.tDoubleEq)
      , InfixN (binary Neq L.tExclEq)
      ]
    , [ InfixR $ binary And L.tDoubleAmpersand  ]
    , [ InfixR $ binary Or L.tDoublePipe ]
    ]
 where
  unary def pSymbol = pSymbol $> \e -> UnaryOpExpr (exprSpan e) def e
  binary def pSymbol = pSymbol $> \e1 e2 -> BinOpExpr (srcSpan (startPos $ exprSpan e1)
                       (endPos $ exprSpan e2)) def e1 e2

pExpr :: Parser (Expr ParsedP)
pExpr = makeExprParser pTerm operatorTable

pTerm :: Parser (Expr ParsedP)
pTerm = choice [ try $ L.parens pExpr , try pFunctionCall , try pLiteralExpr,
                 try pVariableExpr , try pStringExpr ]
```

The operator table is ordered in decreasing precedence (i.e. the higher in the list, the greater the

binding strength of the set of operators). Any operators in the same sublist have the same precedence. Associativity can be modified using the constructors from the `Operator` datatype, which supports: `InfixN`: for non-associative infix operators; `InfixL`: for left-associative infix operators; `InfixR`: for right-associative infix operators and `Prefix` and `Postfix`. I chose the same precedence and associativity as Haskell [9]. I also use the `makeExprParser` for the field access operators (`.hd` and `.tl`) of lists and (`.fst` and `.snd`) of tuples as this was the easiest solution.

While using `makeExprParser` is pleasant when it works, it is also quite "magical". This means that it can be quite hard to fix when it breaks.

2.2.2 Parser Errors

Error handling is handled by megaparsec, which, out of the box, gives quite good error messages. For example:

```
test.spl:1:7:
  |
1 | (a + b
  |       ^
unexpected end of input
expecting "!=", "&&", "<=", "==", ">=", "||", '!',
'%', ''', ')', '*', '+', ',', '-', '/', ':', '<', '>', '_', or alphanumeric character
```

Megaparsec also supports custom error messages and error recovery, but I have not implemented that. I have added two custom errors to the parsing step and for those I just used the normal Haskell `error` function. These two errors are related to string parsing and number parsing.

Number Range Error

The first custom parse error is thrown when you try to parse numbers that are too large for Haskell. For instance the following program:

```
Int large = 9223372036854775808;
```

Will result in the following parse error:

```
Semantic Analysis error Integer is too large for the compiler!
9223372036854775808 > 9223372036854775807.  At examples/errors/too_large.spl:1:13
```

This error is accomplished by checking each integer parsed against Haskell's `intLowerBound` and `intUpperBound` functions before moving on with the parsing.

Empty String Error

The second parse error happens when an empty string is parsed. As can be seen in the Introduction Strings are just syntactic sugar in the parser for a large `Cons` expression. That means however that an empty string `""` would be parsed into an `EmptyListLiteral`. But that is not semantically equivalent because the type of `""` is `[Char]` the type of `[]` is `[a]`. To avoid this issue altogether I just raise the following error when encountering an empty string during parsing:

```
Empty string not allowed at examples/errors/empty_string.spl:1:10.
The string syntax is just syntactic sugar for a large cons expression.
But that means that "" == [] which is not ideal because the typeof "" is [Char]
while typeof [] is [a].
But the type checker can't know this anymore as this information is thrown away
with the desugaring.  So to prevent this confusion just no empty strings.
```

2.2.3 Lexer Step

I do not have a separate lexing step during parsing. Instead, I use a (what I call) just-in-time lexer (scannerless). Our lexer is just a collection of regular parsers that only parse simple tokens (as a regular lexer would), such as keywords, identifiers or symbols, while discarding whitespace and comments. I achieve this "lexing" using the `symbol` parser combinator. The `symbol` parser combinator takes any string and creates a parser that parses exactly that string, while throwing away comments and whitespace at

the end. For example, `symbol "a"` should parse `a`, but also `a /* comment */`. The `symbol` parser only consumes and discards additional whitespace after the end of the parsed token. This is why our main program parser has an additional `whitespace` parser at the start to discard comments and whitespace at the beginning of files as well. I use these `symbol` parsers throughout the other parsers in instances where you would typically consume a token from a lexer generated token list. I chose this approach, since it is well supported by megaparsec and it easily deals with any comments and whitespace. I do not see a benefit of converting the whole input into tokens first when using parser combinators. This can be done when it is actually needed.

2.3 Testing

During the stage of the project where I mainly worked on the lexical analysis I wrote a lot of tests using Hspec [11]. This was also why I added the `EmptyP` phase to be able to compare AST nodes by stripping all metadata. The tests did help in the beginning to find out if our parser still worked after making changes. However, later on I abandoned the test because things kept changing too much and updating the tests was too time consuming. Just running the compiler on complicated input and seeing if it pretty printed the expected result worked much more efficiently.

2.4 Problems

Since I started with writing the AST in Haskell based on the provided examples I did not have any major problems during this phase. Having a well-defined AST early on really helped with making the parsers. By using a bottom up approach starting from the most simple parsers (such as literals), and working upwards I managed to implement the parser without many problems.

I did initially struggle with the left-recusivity of property access (e.g. a.hd), but fixed this by making the fields `Postfix` operators in the `makeExprParser`. This fixes the left-recursion, as megaparsec will only parse things with lower precedence on the left, causing it to no longer be fully recursive.

Chapter 3

Semantic analyses

In the lexical analysis the compiler checks the syntax of the program. If there are no errors then at this point the compiler has a `Program ParsedP`. During the semantic analysis phase the compiler performs many checks to figure out if your code actually makes sense semantically. If any of these checks fails the programmer gets an error. I spend a lot of time making the errors helpful and easy to understand.

This phase is further divided into 5 other phases:

1. Preprocessing
   (a) Checking for empty functions
   (b) Checking the main function
   (c) Removing dead code
   (d) Hoist global variable declarations
2. Check for uniform returning (`ReturnsCheckedP` phase)
3. Check duplicate declarations
4. Type checking (`TypecheckedP` phase)
5. Optimisation

3.1 Preprocessing

The preprocessing step of the compiler consists of multiple smaller steps which prepare the `ParsedP` AST for the next phases.

3.1.1 Checking for empty functions

During this phase of the preprocessing step the compiler checks every function for an empty body. With an empty body I mean a function like this:

```
main() {}
```

Recall that the program is just a list of `Decl`. The compiler can find such functions by checking if there is any function declaration in the list with an empty list of statements. If the compiler finds such a function the following error is raised:

```
Semantic Analysis Error:  The 'foo' function declared at examples/errors/empty_function.spl:2:1 has
an empty body.  Empty functions are not allowed.  Please either add a body (just a 'return;' is enough)
or remove the function.
```

After this step the compiler has been established that every function in code has a body. However, for the remainder of rest of the examples the main function definition has often been left out to save space.

3.1.2 Check the `main` function

In this step of the preprocessing the compiler checks if the `Program ParsedP` contains a main function. This works in a similar way as checking for empty bodies. The compiler iterates the function definitions and checks if there is a function called main. If the compiler can not find a main function the following error is presented to the programmer:

```
Semantic Analysis Error: No main function in your program! Please add a main function to your program.
```

Using the same technique the compiler also checks if the main function does not have any arguments.
For instance like this:

```
main(aap, baap) {
    return;
}
```

If the compiler finds that the main function has arguments the programmer is presented with the following error:

```
Semantic Analysis Error: The 'main' function can not have any arguments. There is no way for you
to initialise them. Please remove the arguments.
```

I do not do specific checks on the return type of the main function. It is currently meaningless to return a value from the main function but the compiler does not deny it either.
After this phase it has been established that the program has a main function without arguments.

3.1.3 Removing dead code

In this step the compiler removes dead code by looking for certain patterns in the body of functions. Recall that the body of a function is lists of statements or `[Stmt]`.

The compiler recursively checks the body of each function declaration and looks for two specific patterns that indicate dead code. These two patterns are code behind a return statement and code behind an if else statement where every branch returns. As an example the following code has a print statement that will never be reached:

```
main() {
    return;
    print("Dead");
}
```

This code is rewritten by the compiler into:

```
main() {
    return;
}
```

and the programmer is presented with the following warning:

```
WARNING: Removing dead code after return at examples/warning/deadcode.spl:3:5
```

In this next example the function will always return in the if and else statement. Because of this the code behind the if else statement will never run.

```
foo(aap) {
    if (aap > 3) {
        return 'Q';
    } else {
        return 'u';
    }
    // we will never get here
    print("Dead");
    return '-';
}
```

Any code after such an if else is considered dead code and removed by the compiler during this step. As such the compiler will turn the above SPL is converted into:

```
foo(aap) {
    if (aap > 3) {
        return 'Q';
    } else {
        return 'u';
    }
}
```

and the programmer is presented with the following warning:

```
WARNING: Removing dead code after if at examples/warning/deadcode_if.spl:2:5
```

The if else dead code removal works by checking if every branch in the if else actually returns. The body of an

if and an `else` are both `[Stmt]`. This allows recursively checking all return paths of an if else statement. This allows the compiler to detect any arbitrarily nested if else that always returns. This step uses the same function later used in the Check return statements step. For this reason I will expand on it more there instead of here. But by using the same method the compiler is still able to detect the dead code in the following nested code:

```
foo(aap) {
    if (aap) {
        return;
    } else {
        if (aap) { return; } else { return; }
    }
    print("Never");
    return;
}
```

In this example the `print("Never")` and final `return` line will be removed by the compiler.

As a final feature of this step the compiler will print how much code was removed during this step (if any). In the last example the compiler would print: `Pruned 55% of tree by removing dead code.` 55 percent might seem high but this is due to the "Never" string which is actually a chained cons expression.

3.1.4 Hoist global variable declarations

Recall that the an SPL program to the compiler is just a list of declarations or `[Decl]`. This list has both function declaration and variable declarations. In the upcoming type checking step the compiler will check the types of the program from top to bottom. I want to provide a semantics where the global variables are always defined before the functions. Let's illustrate this with an example. Consider the following program:

```
var global_var1 = 3;

main() {
    print(global_var1, global_var2);
}

var global_var2 = global_var1 * 2;
```

Without doing anything the compiler would raise a variable not defined error:
`Semantic Analysis Error:  Undefined variable "global_var2" at examples/globals.spl:5:24.`

To prevent these errors the compiler moves all the global variable decelerations to the top of the program (the start of the list of `Decl` list). This step will transform the above code into:

```
var global_var1 = 3;
var global_var2 = global_var1 * 2;

main() {
    print(global_var1, global_var2);
}
```

This is achieved by sorting the list of `Decl` with the following Haskell sorting code

```
compareDecl :: Decl p1 -> Decl p2 -> Ordering
compareDecl (VarDecl {}) (FunDecl {}) = LT
compareDecl (FunDecl {}) (VarDecl {}) = GT
compareDecl (FunDecl {}) (FunDecl {}) = EQ
compareDecl (VarDecl {}) (VarDecl {}) = EQ

{- Sorts the program, putting the global vars at start -}
hoistGlobalVars :: Program ParsedP -> Program ParsedP
hoistGlobalVars = sortBy compareDecl
```

The major advantage of using sorting to move global variables to the top of the program is that the definition order is not changed. This means that the programmer can still have global variables depending on the previous ones like they would expect. For instance in the example `global_var2` depends on `global_var1` to exist.

This step provides the programmer with the ability to define global variables throughout their program while not having to worry that those global variables will be available to them in their functions.

3.2 Check the ways of returning

In this compiler step the compiler checks if all the branches of the program return in the same way. This step relieves the type checker from the burden of checking if each code path returns in the same way so that the type checker can focus only on the return types. This phase could potentially have been achieved by the type checker

but by making it a separate phase I can generate much more descriptive error messages. This goal of this phase is to reject programs like this:

```
foo(aap) {
    if (aap) { return 137; } // Returns something
    return; // Does not return anything!
}
```

Because of this phase this example will present the programmer with the following error:

```
Semantic Analysis Error:  Invalid returns in if condition at examples/errors/invalid_return1.spl:2:5
The true case of the condition returns at examples/errors/invalid_return1.spl:2:14 but a return statement
later at examples/errors/invalid_return1.spl:3:5 does not.  Each branch should return the same way.
You should either always return a value or never.  Either add a return value to the return statement
at examples/errors/invalid_return1.spl:3:5 or remove the value from the return statement in the if
at examples/errors/invalid_return1.spl:2:14
```

There are 3 ways a function can return.

1. With a value: `return 137;` which is `ReturnStmt Just (...)`
2. Without a value: `return;` which is `ReturnStmt Nothing`
3. No return statement. The base case if no return statement is found in a list of statements.

To capture all these cases I defined a data type:

```
data WayOfReturn = WithValue SourceSpan | WithoutValue SourceSpan | No
```

There are four rules when checking the way a function returns:

1. `No` and `WithoutValue` are compatible. That means that if one branch returns without a value and later there is no return value then this is not an error. This is due to the implicit `return;` statement at the end of each function if no other return statement is found.
2. `WithoutValue` is incompatible with `WithValue`. That means that if one branch returns with a value and another without then this is an error.
3. `WithValue` is only incompatible with `No` if the `No` happens later in the list of statements.
4. Each way of returning is compatible with itself.

To provide more clarity on rule 3 here is a case where `WithValue` and `No` is are compatible:

```
foo(aap) {
    if (aap) { print(aap); } /* No */ else { print(! aap); } /* No */
    return aap; // WithValue
} // Fine because WithValue happens after No
```

But in this next example they are not compatible because the `WithValue` happens before the `No`:

```
foo(aap) {
    if (aap) { print(aap); } /* WithValue */ else { print(! aap); } /* No */
    // No
}
```

This example will present the programmer with the following error:

```
Semantic Analysis Error:  Invalid return.  In the if at examples/errors/return2.spl:2:5 you return
a value at examples/errors/return2.spl:2:14 but nowhere else you return a value.
Each branch should return the same way.  You should either always return a value or never.  Either
remove the return statement at examples/errors/return2.spl:2:14 or add a return statement with a
value at the end of the function.
```

To implement these 4 rules I defined the following Haskell type class. Its `returns` function can be used to ascertain how a list of something returns.

```
class ReturnCheck a where
  returns :: [a] -> Either String WayOfReturn
```

The `returns` function either returns an error message (`Left`) or the way in which something returns (`Right`). I implemented this class for `Decl ParsedP` and `Stmt ParsedP`.

The `Decl` implementation just calls `returns` on the body of each function. If at any moment a `Left` is encountered the error string is raised as an error. Recall that the body of functions has type `[Stmt]`.

The `returns` implementation for `[Stmt]` recursively finds out how a statement returns. The idea is to check how the current statement returns and how the rest of the list of the statements return. The ways of returning are then compared using pattern matching. There are quite some patterns to match so instead of showing all the cases I will only show a few examples. Here is the implementation of `ReturnCheck` for `Stmt ParsedP`. I replaced the pattern matching to check the rules with `...` to save space.

```haskell
instance ReturnCheck (Stmt ParsedP) where
  returns [] = Right No
  returns (ReturnStmt meta Nothing : _) = Right (WithoutValue meta)
  returns (ReturnStmt meta (Just _) : _) = Right (WithValue meta)
  returns (IfStmt meta _ consequent Nothing : later) =
    case (returns consequent, returns later) of ...
  returns (IfStmt meta _ consequent (Just alternative) : later) =
    case (returns consequent, returns alternative, returns later) of ...
  returns (WhileStmt meta _ body : later) =
    case (returns body, returns later) of ...
  returns ((AssignStmt {}) : later) = returns later
  returns ((ExprStmt _ _) : later) = returns later
  returns (BlockStmt _ : _) = error "Encountered block statement in the return check phase!!!"
```

Because `returns` is recursive it will also work for deeply nested statements. Like an if else statement that has while loops that have if statements. I will now discuss pattern matching cases for the different types of statements.

If with else The `if` statement with an else statement is the most complicated as this needs to compare 3 things. The way the consequent returns, the way the alternative returns and the way the rest of the statements return. Here are some examples.

```haskell
(Right (WithValue m1), Right No, Right No) -> Left "Detailed error" -- Not Fine Rule 3
(Right (WithoutValue _), Right No, Right (WithoutValue m)) -> Right (WithoutValue m) -- Fine Rule 1 and 4
(Right (No _), Right No, Right (WithoutValue m)) -> Right (WithoutValue m) -- Fine Rule 1
(Right (WithoutValue m1), Right (WithValue m2), _) -> Left "Detailed Error" -- Rule 2
(Right (WithValue m1), Right (WithoutValue  m2), _) -> Left "Detailed Error" -- Rule 2
```

These last example presents the programmer with this error:

```
Semantic Analysis Error:  Invalid returns in the if condition at examples/errors/returns3.spl:2:5
The true case of the if condition returns a value at examples/errors/returns3.spl:2:17 while the
else case does return but without a value at examples/errors/returns3.spl:2:34.  Each branch should
return the same way.  You should either always return a value or never.  You probably either forgot
a return value in the else or you forgot to remove the return value in the true case of the if condition.
Either add a return value at examples/errors/returns3.spl:2:34 or remove the return statement at
examples/errors/returns3.spl:2:17.
```

While and if without else The while and if without an else cases only need to compare two ways of returning. The way they return and the way the code after them returns. Here are some of the cases:

```haskell
(Left err, _) -> Left err -- Propagate Left
(_, Left err) -> Left err -- Propagate Left
(Right No, Right No) -> Right No -- Rule 4
(Right (WithoutValue _), Right (WithoutValue m)) -> Right (WithoutValue m) -- Rule 4
(Right (WithValue _), Right (WithValue m)) -> Right (WithValue m) -- Rule 4
(Right (WithoutValue _), Right No) -> Right No -- Rule 1
(Right (WithValue m), Right No) -> Left "Detailed Error" -- Rule 3
(Right No, Right (WithValue m)) -> Right (WithValue m) -- Rule 3
(Right (WithoutVAlue m1), Right (WithValue m2)) -> Left "Detailed Error" -- Rule 2
```

3.2.1 Assigning the `VoidType` return type

As the returns checking step happens before the type checking step all the types are a `Maybe`. The types that are there have been specified by the programmer. If after checking a function declaration, the way of returning is `No` or `WithoutValue` you know that the return type of the function has to be `VoidType`. You also know that when the way of returning is `WithValue` that the return type can not be `VoidType`. These two things are actually checked during this phase by comparing the return type specified by the programmer with the way the function actually returns. If there is no return type specified by the programmer (e.g. the return type in the `Decl` is `Nothing`) and the function does not return a value, the compiler will replace the return type in the `Decl` with `Just VoidType`.

Return ways not matching specified return type errors

Here are the errors the programmer is presented with if the return type they specified does not match with the way the function actually returns.

In the case of `WithValue` and a specified return type of `VoidType` like this:

```
foo(): Void { return 3; }
```

the programmer is presented with this error:

```
Semantic Analysis Error: Unexpected return value in function 'foo' defined at
examples/errors/returns4.spl:1:1. Function 'foo' has a specified return type of Void which means
no value should be returned. However at examples/errors/returns4.spl:1:14 you are returning a value.
Here are some things to try:
- If you do not want to return a value, remove the returned value at examples/errors/returns4.spl:1:14
- If you do want to return a value, change the return type annotation of the function to another
type that is not Void
- If you do want to return a value but you don't know what type (yet), just remove the return type
annotation of the function all together and let the compiler figure it out.
```

In case of `No` or `WithoutValue` and a specified type that is not `VoidType` like this

```
foo(): Int { return; } // WithoutValue
bar(): Int { print("Not returning"); } // No
```

the programmer is presented with the following error:

```
Semantic Analysis Error: Missing return statement in function 'foo' defined at
examples/errors/returns5.spl:1:1. Function 'foo' is specified to return a value of type Int but
no value is being returned from the 'foo' function. Either add a return value (of type Int) or change
the return type of the function to Void.
```

After completing this step the AST has been transformed from a `Program ParsedP` into a `Program ReturnsCheckedP`.

3.3 Check duplicate declarations

In this step the compiler looks for duplicate declarations. This could be duplicate variable declarations or duplicate function names.

I decided to not allow shadowing of global variables. This not only makes things much simpler to implement but also I believe that it leads to more clear code. This means that as soon as you declare a global variable with name `aap` no other global variable, function argument or function variable in the program can be called `aap` anymore. There is however local scope between functions. You can reuse variable names across functions. So something like this is fine:

```
raise(number) {
    var result = 0;
    result = number * number;
    return result;
}

power(number, by) {
    var result = 0;
    while (by > 0) {
        result = result + raise(number);
        by = by - 1;
    }
    return result;
}
```

Naturally within one function variables still can not overlap.

The checking works by recursively iterating over the `Program` and passing along a `Map String SourceSpan`. For every name the compiler performs a lookup in the map. If the result is `Nothing` the compiler adds the name to the map with the `SourceSpan` of the name. If the result is `Just ...` the name has already been declared before and the error is created.

3.3.1 Shadow Error Examples

If the programmer tries to shadow a global variable name with the name of a function argument like like this:

```
var aap = 137;
foo(aap) {return aap;}
```

they will be presented with the following error:

```
Semantic Analysis Error: The argument name 'aap' of the 'foo' function has already been defined
earlier. The first time at: examples/defined.spl:1:1 and the second time at: examples/defined.spl:3:1
Names used for global variables and argument names should not overlap. You could consider renaming
```

```
the argument to aap' (adding a ' behind the name) or foo_aap (adding the function name as a prefix).
```

A similar error is generated by this code:

```
foo(aap, aap) {
    return;
}
```

If the programmer tries to shadow a global variable name with the name of a function variable like like this:

```
foo() {
    var aap = 731;
    return aap;
}

var aap = 137;
```

they will be presented with the following error:

```
Semantic Analysis Error:  Variable with name 'aap' is defined two times!
The first time at:  examples/defined2.spl:6:1 and the second time at:  examples/defined2.spl:2:5.
```

Even though the variable is defined after the function it is being mentioned in the error as first because it was moved to the start of the file in the hoist globals step.

The last error is also produced for the following two programs:

```
var aap = 137;
var aap = 731;
```

```
foo(aap) {
    var aap = 137;
}
```

3.4 Type checking

During type checking the compiler checks if the code written by the programmer actually made sense given the types. For example, in SPL the `+` operation can only be performed with two Int. As such writing `True + 137` does not make sense because you can not do the `+` operation with a Bool and an Int. This means that when the compiler encounters a `+` operation it has to check if both of the operands can be Int types. This is type checking.

Because the operands of `+` have to be Int the compiler can assume that `a` and `b` in `a + b` are of type Int. This is type inference. If the compiler later encounters information which contradicts this. For instance `a && b`, an operation that can only be performed on two Boolean's, the compiler has to generate a type error.

The type system is based on the classic Hindley-Milner [8] system. The AlgorithmW step by step tutorial paper [2] was very helpful during the implementation of the type checker.

Fixing the `Maybe Type`

At the start of the type checking the compiler only has the type information it was given by the programmer. Many types might be missing. For this reason the type of the nodes in the AST that hold the type information are `Maybe Type`. The compiler removes `Maybe` by replacing all the `Nothing` with a fresh type variable. After this in theory every part of the AST has a type. But many of the types are type variables.

Narrowing the type variables

Type variable are the most general type and represent the least amount of information. Each type variable has a name. The idea of the type checking phase is to create a substitution that substitutes type variables with more concrete types. Essentially the substitution is a map from type variable names to types. Hopefully a more concrete type but a type variable can also be substituted with another type variable. That means that a type variable has the same type as another type variable.

The substitution is created by collecting information about how variables are used. If the compiler manages to create a substitution for the entire program without generating an error you know that the program is correctly typed. After obtaining a substitution for the entire program the compiler applies it to the AST to add the obtained type information to the AST.

3.4.1 Unification

Every time the compiler infers something it can turn this information into a substitution using unification. Unification is the process of making two types equal by finding a suitable substitution. For instance if you unify an `Int` with `Int` you get an empty substitution because the types are already equal. There are no type variables to substitute. If you try to unify *Int* with a type variable `a` you can make them equal with the $\{a \rightarrow \texttt{Int}\}$ substitution. If you try to unify `Int` with `Bool` you get $\perp$ because there is no way to make `Int` and `Bool` equal. Here is my unification function implemented in Haskell:

```
unify :: Type -> Type -> TI Subst
unify (TypeVar u1 False) t2@(TypeVar _ False) = return $ Map.fromList [(u1, t2)]  -- U (, ) = id
unify (TypeVar u False) t = gets currentMeta >>= \meta -> varBind meta u t -- bind the type var!
unify t (TypeVar u False) = gets currentMeta >>= \meta -> varBind meta u t
unify ty1@(TypeVar name1 True) ty2@(TypeVar name2 True) =
  if name1 == name2 then return nullSubst else unifyError ty1 ty2
unify ty1@(TypeVar _ True) ty2 = unifyError ty1 ty2
unify ty1 ty2@(TypeVar _ True) = unifyError ty1 ty2
unify (ListType t1) (ListType t2) = unify t1 t2
unify (TupleType t1 t2) (TupleType t1' t2') = do
  s1 <- unify t1 t1'
  s2 <- unify (apply s1 t2) (apply s1 t2')
  let s = s1 `composeSubst` s2
  applySubToTIenv s
  return s
unify IntType IntType = return nullSubst
unify BoolType BoolType = return nullSubst
unify CharType CharType = return nullSubst
unify VoidType VoidType = return nullSubst
unify t1 t2 = unifyError t1 t2

varBind :: SourceSpan -> String -> Type -> TI Subst
varBind meta u t
    | u `Set.member` typevars t =
        throwError $
            red "Occurs check:" ++ " cannot construct the infinite type"
                    ++ u ++ " ~ " ++ show t ++ " at " ++ showStart meta
    | otherwise = return (Map.singleton u t)
```

Once the compiler gets a substitution from unification it has to be merged with the larger substitution of the entire program. Two substitutions can be composed using the `composeSubst` function. For instance merging $\{a \rightarrow \texttt{Int}\}$ with $\{b \rightarrow \texttt{Bool}\}$ becomes $\{a \rightarrow \texttt{Int}, b \rightarrow \texttt{Bool}\}$.

Unify Errors

When the `unify` function can not unify two types it calls the `unifyError` function. I tried to make the unification error very helpful to the programmer. As an example the following code:

```
main() { return 1 + True; }
```

Will produce this error:

```
Semantic Analysis Error:  Inside function main;
Types do not unify:
Int vs.  Bool at examples/errors/unify.spl:1:20 until line 1 column 24

learn more about unification errors here:
https://en.wikipedia.org/wiki/unification_(computer_science) or here https://cloogle.org/#unify%20error
```

The error generated by the `unifyError` function can also make suggestions if there are other variables in scope that would be the correct type. For instance with this code:

```
var three = 3;
var fff = False && False;

foo(a_bool: Bool)
{
    var ttt = True;
    var two = 2;
    return 1 + True;
}
```

The following lines would be added to the error:

```
Potential variables of type Int in scope include 'three' and 'two'
Potential variables of type Bool in scope include 'a_bool', 'fff' and 'ttt'
```

3.4.2 TI Monad

In the unify function the result is not a `Subst` but a `TI Subst`. The `TI` type is a Monad. The TI monad is a State monad wrapped by an `ExceptT` Monad transformer. This monad gives a convenient way to manage state in the type checker and gain the ability to throw and catch errors using `throwError` and `catchError`. The definition of the TI monad is as follows:

```
type TI a = ExceptT String (State TIState) a
```

The definition of TIState is a record type:

```
type TIState = TIenv
data TIenv = TIenv {currentFunEnv :: FunEnv, -- Map String Scheme
                    currentVarEnv :: VarEnv, -- Map String Type
                    currentTypeVarCounter :: Int, -- Used to generate fresh type vars
                    currentFunName :: Maybe String,
                    currentFunType :: Maybe Type,
                    currentMeta :: SourceSpan,
                    currentFunArgNames :: Maybe [String],
                   }
```

3.4.3 Type Inference

Type inference is determines the type of something while type checking checks if the type of something conforms to a certain type. Type inference gives you a type and a substitution while type checking only gives you a substitution. If you only implement one of these operations you can get the other one for free. I took full advantage of this with a `Typecheck` type class:

```
class Typecheck a where
    {-# MINIMAL tc | ti #-}
    tc :: a -> Type -> TI Subst
    tc a t = do
        (s1, inferredT) <- ti a
        s2 <- unify t inferredT
        let s = s1 `composeSubst` s2
        applySubToTIenv s
        return s
    ti :: a -> TI (Subst, Type)
    ti a = do
        t <- newTyVar
        s <- tc a t
        applySubToTIenv s
        return (s, apply s t)
```

I chose to implement type inference for everything and rely on the derived `tc` for where it was needed. I did this because I found type inferencing easier and more intuitive to implement than type checking.

Literals

In general, type inferencing for literals is trivial. The type of an integer literal is just `IntType` and so on. However, there are two special cases: tuples and lists.

For tuple literals you have to infer the types of the two tuple members and compose the obtained substitutions.

The empty list type is special because it is impossible to know the type of the elements that will be added to an empty list in the future. As such every empty list starts out with type `[t]` where `t` is a fresh (non rigid) type variable.

Because the implementation is not very long I included the entire `Typecheck` instance for `Literal`.

```
instance Typecheck (Literal TypecheckedP) where
  ti (IntLit _) = return (nullSubst, IntType)
  ti TrueLit = return (nullSubst, BoolType)
  ti FalseLit = return (nullSubst, BoolType)
  ti (CharLit _) = return (nullSubst, CharType)
  ti EmptyListLit = do
    var <- newTyVar
    return (nullSubst, ListType var)
  ti (TupleLit (e1, e2)) = do
    (s1, t1) <- ti e1
    (s2, t2) <- ti e2
    let s = s1 `composeSubst` s2
    applySubToTIenv s
    return (s, TupleType t1 t2)
```

Expressions

Expressions result in a value. When type inferencing an expression you try to figure out the type of this value.

Simple Binary Operation I define simple binary operations as operations that can only be applied to a single type. Because the simple operations can only be applied to a single type they do a good job of narrowing types. Before you can infer the type of a binary operation you have to check if the operands of the operation are the correct type and propagate the substitutions properly. To help with this I created the `tcBinOp` function.

```
{- Check if the type of 2 expr can be CheckType return the obtained substitution if they can -}
tcBinOp :: Type -> Expr TypecheckedP -> Expr TypecheckedP -> TI (Subst)
tcBinOp checkType e1 e2 = do
  s1 <- tc e1 checkType
  s2 <- tc e2 checkType
  let s = s1 `composeSubst` s2
  applySubToTIenv s
  return s
```

Then to infer the type of integer binary operations I created the `tiBinaryIntOp` function which builds on.

```
tiBinaryIntOp :: Expr TypecheckedP -> Expr TypecheckedP -> Type -> TI (Subst, Type)
tiBinaryIntOp e1 e2 ty = do
  s1 <- tcBinOp IntType e1 e2
  s2 <- unify ty IntType
  let s = s1 `composeSubst` s2
  applySubToTIenv s
  return (s, IntType)
```

Using `tiBinaryIntOp` I was able to make the implementation of `ti` of integer operations quite terse.

```
ti (BinOpExpr (ty, meta) Mul e1 e2) = replaceMeta meta >> tiBinaryIntOp e1 e2 ty
ti (BinOpExpr (ty, meta) Mod e1 e2) = replaceMeta meta >> tiBinaryIntOp e1 e2 ty
ti (BinOpExpr (ty, meta) Add e1 e2) = replaceMeta meta >> tiBinaryIntOp e1 e2 ty
ti (BinOpExpr (ty, meta) Div e1 e2) = replaceMeta meta >> tiBinaryIntOp e1 e2 ty
ti (BinOpExpr (ty, meta) Sub e1 e2) = replaceMeta meta >> tiBinaryIntOp e1 e2 ty
```

I implemented type checking for the boolean binary operations in the same way.

Equality Expression The equality and non equality expressions are special because they are actually overloaded functions. That means that they can be applied to any type. However, I did restrict this by making the type of equality $==:: a \to a \to Bool$. These a represent rigid type variables. What this means is in my implementation of SQL the programmer can not just compare an Int to a Bool or an Int to a Char. In other words the type types you compare have to be unifiable.

Greater than and smaller than expressions The >, >=, < and <= operations are special because they work on two types: Int and Char. I solved this by first checking if both the expressions unify with an Int and if that throws an error to try a Char type. Because this could be confusing for the programmer I added an extra line to the usual unification error for this case.

```
tcBinOpCharOrInt :: BinOp -> Expr TypecheckedP -> Expr TypecheckedP -> TI (Subst)
tcBinOpCharOrInt op e1 e2 =
    get >>= \env -> tcBinOp IntType e1 e2 `catchError`
             \_ -> put env >> tcBinOp CharType e1 e2 `catchError`
              \_ -> do
                (_, t1) <- ti e1 -- Get the type for the error
                (_, t2) <- ti e2
                meta <- gets currentMeta
                unifyError t1 t2
                   `catchError` \errorTail -> throwError $ "Invalid operation at " ++ show meta
                                                        ++ "\nArguments to" ++ pretty op ++
                                                        "should be either Int or Char but you gave "
                                                        ++ show t1 ++ pretty op ++ show t2 ++ "\n"
                                                        ++ errorTail
```

Cons The cons operation (:) is special because you have to check that the left hand side is of type list and that the right hand side unifies with the type of the element inside the list. When the programmer makes a mistake with cons the compiler will create specific errors. These errors include

```
Semantic Analysis Error:  You tried to cons Int with Int but that does work.  The right of a cons
should always be a list type.
```

```
Semantic Analysis Error:  You tried to cons Int with [Char], but this is not legal.
```

Here is the code for inferring Cons expressions. I truncated the code that generates the additional error to keep the code more readable.

```
  ti (BinOpExpr (ty, meta) Cons e1 e2) = do
    replaceMeta meta
    (s1, t1) <- ti e1
    (s2, listty) <- ti e2
    s3 <- case listty of -- If t2 is a list type we have to unify with the type inside the list
            ListType u1 -> unify t1 u1  `catchError` \err -> throwError $ "Additional error" ++ err
            _ -> throwError $ red "Additional Error 2"
    let inferredType = ListType t1
    s4 <- unify ty inferredType -- Check the type of this node and save the result
    let s = s1 `composeSubst` s2 `composeSubst` s3 `composeSubst` s4
    applySubToTIenv s
    return (s, apply s inferredType)
```

Unary Operation As all unary operations can only be applied to a single type inferring the type of a unary operation works basically the same as with the binary expressions.

Field Operation Expressions Fields are special because the inferred type comes from the structure that it is applied to. There are 4 fields in SPL: `tl` and `hd` for lists and `fst` and `snd` for tuples. For each field the compiler has to check that it is applied to the right type. I added special errors for each. Here is one example:

```
main() {
    var a = [].snd;
    return;
}
```

```
Semantic Analysis Error:  You accessed the snd field on a [Typevar f] at cons2.spl:2:13.  But that
is invalid you can only access the snd field on tuple types.
Maybe you meant .tl?
```

Variable Expressions The types of all defined variables are stored in the Variable environment in the TI monad. This environment stores the type of each defined variable. Thus, to infer the type the compiler can just look in the variable environment. If a variable is not there a "variable is not defined" error is thrown. Here is the code that checks variables:

```
  ti (VariableExpr (ty, meta) (Identifier var Nothing)) = do
    replaceMeta meta
    sigma <- lookupVarType var -- Throws error if not defined
    s <- unify ty sigma
    applySubToTIenv s
    return (s, sigma)
```

Statements

It would seem but I decided to make type inference on statements yielding the type that the statement returns. This means that the type of an if statement with a return statement that returns a value of type Int, is inferred as an Int. A list of statements must return the same type. If it does not, the programmer is presented with an error. The only other things to check during the type inference of statements is the types of expressions in conditions. The types of the condition expressions should be boolean.

This code is a good example of statement type checking as it demonstrates both narrowing an expression to Boolean and the return type checking.

```
foo(aap) {
    if (aap) {
        return aap;
    }
    return 137;
}
```

If the programmer would compile this code they would be presented with the following error:

```
Types do not unify:
Bool vs.  Int at examples/invretty.spl:6:12 until line 6 column 13
Potential variables of type Bool in scope include 'aap'.

learn more about unification errors here:
https://en.wikipedia.org/wiki/unification_(computer_science) or here https://cloogle.org/unify%20error
```

Declarations

The declarations are at the top level of the program. Inferring variable declarations is not harder than inferring expressions. But the special thing about inferring variable declarations is that they add themselves to the variable environment. Function declarations keep track of the variables that are added to the environment before type checking the body. After the body is checked they remove the local and argument variables again after the body is checked. Type checking a function declaration is special because it generates a FunType from the argument types and the return type obtained from checking the statements in the body. This FunType can then be applied to a function `Decl` AST node to update it accordingly.

```
checkDecls :: Program TypecheckedP -> TI (Program TypecheckedP)
checkDecls [] = return []
checkDecls (decl:future) = do
    (sub, ty) <- ti decl
    solved <- checkDecls future
    return $ apply sub decl : solved
```

3.4.4 Polymorphism & Function call expressions

Function call expressions look up the type of the function from the TI monad. The result of that monadic action will either be a FunType or if the function is not defined an error. Build in function are hard coded into the function environment. After obtaining the type of the function the compiler unifies the types of the expressions with the types of the arguments in the FunType.

There is also a special check to make sure that you actually give the same number as arguments. If that does not happen you will get an error.

```
hi(a:Int, b:Bool, c:Char) {
    return;
}

main() { hi(2); }
```

will generate:

```
Semantic Analysis Error: Function call at examples/errors/much.spl:7:5 is missing 2 arguments
```

When calling a function the compiler checks if this function is the same function the compiler is currently checking (recursive call) or another function. In the first case the function call should narrow the type of the function that is being called in the function environment. In the second case the function call should not narrow the type of the function being called in the function environment. This effectively enables polymorphism because the polymorphic arguments will just unify with the type of the expression given as input without affecting any other calls to the polymorphic function.

When checking function types something special happens with the rigid type variables. If two arguments are defined as the same rigid type variable the types of those expressions have to be unifiable with each other. If the return type of the function is a rigid type variable that is also used as one of the argument types, the type of the expression of that argument will be the inferred type of the function call expression. This way you can make the return type of a function depend on the type of an argument. However, this is not incredibly useful as Rigid type variables can not be narrowed inside of functions. So you can only use rigid type variables if you don't touch them. But this does cause the compiler to know that in this next example the type of `aap` is Int.

```
id(value: a): a {
    return a;
}

main() {
    var aap = id(137);
    print(a);
}
```

3.4.5 Problems

The fact that the types were a `Maybe` actually caused a lot of hassle. I was trying to resolve the Maybe in the same function that was type checking the `Decl` and this was a major source of hidden complexity because the type checking function only returns a substitution and not a new AST node. Only quite late I realised that this was the source of the complexity. The right thing to do was to first instantiate the entire tree with type variables and only then start type checking.

The more and more I worked on the type checker the simpler the code became. But this simplicity is very hard to reach in one go.

3.5 Optimisation

The last step of the semantic analysis is the optimization step. During this phase the compiler prunes and evaluates parts of the AST so that less code has to be generated. Quite some optimisations have been implemented:

- Literal evaluation for binary and unary operations.
- Lazy boolean expressions, meaning that if in an expression one of the bools is true I just replace it with a True lit.
- Remove while (false)
- Remove if (false) and always run the else
- Comparing variables with the same name always yields true

In the last two cases I replace the statements with a `BlockStatement` first which I later merge into the main statement list.
This next example shows how a program can be optimised by the compiler.

```
main() {
    var a = 3 + 3 * 2;
    var b = (10,"Long string").fst;
    var ttt = isEmpty([]);
    var c = (10:[]).hd;
    var d = c == c;

    if (True) {
        print("1");
    }

    if (False) {print("gone");} else {print("2");}

    while (False) {print("gone");}
}
```

Is optimised into:

```
main() : Void {
    Int a = 9;
    Int b = 10;
    Bool ttt = True;
    Int c = 10;
    Bool d = True;

    print('1':[]);
    print('2':[]);
    return;
}
```

You can skip this phase by using the `--skip-optimizer` flag.

Chapter 4

Code Generation

During the code generation phase I generate code for the simple stack machine (SSM) [1]. This virtual machine was originally developed at Utrecht university to teach about code generation and compilers. However, in this project I use a Radboud University fork found at: https://gitlab.science.ru.nl/compilerconstruction/ssm.

4.1 Code generation infrastructure

I started out by implementing every SSM instruction as a data type. Simply calling `show` on this datatype would print the assembly code. I also defined the `Code` type which is just a list of SSM instructions.

```
data Instr
    = STR Reg | STL Int   | STS Int   | STA Int         -- Store from stack
    | LDR Reg | LDL Int   | LDS Int   | LDA Int         -- Load on stack
    | LDC Int | LDLA Int | LDSA Int | LDAA Int        -- Load on stack, for local always start with 1
    | LDML Int Int   | STML Int Int   | LDMS Int Int   -- Load/Store local multiple,
    | BRA Int | Bra String                             -- Branch always (relative/to label)
    | BRF Int | Brf String                             -- Branch on false
    | BRT Int | Brt String                             -- Branch on true
    | BSR Int | Bsr String                             -- Branch to subroutine
    | ADD | SUB | MUL | DIV | MOD                      -- Arithmetical operations on 2 stack operands
    | EQ  | NE  | LT  | LE  | GT  | GE                 -- Relational   operations on 2 stack operands
    | AND | OR  | XOR                                  -- Bitwise      operations on 2 stack operands
    | NEG | NOT                                        --                operations on 1 stack operand
    | RET | UNLINK | LINK Int | AJS Int                -- Procedure utilities, Ret = Return from
    | SWP | SWPR Reg | SWPRR Reg Reg | LDRR Reg Reg  -- Various swaps
    | JSR | TRAP Int | NOP | HALT                      -- Other instructions
    | LABEL String                                     -- Pseudo-instruction for generating a label
    | LDH Int | STH | STMH Int                         -- Heap variables
    deriving Show

type Code = [Instr]
```

Another important thing to implement was the `codeSize` function. This is calculates how many memory cells a `Code` takes up. This is useful when calculating how far the program counter has to jump.

The code generation works by implementing the GenSSM typeclass for every node in the AST.

```
generate :: a -> Env Code
```

As you can see the `generate` function returns `Code` inside an `Env` monad. This monad is a state monad and keeps track of all information relating to the code generation:

```
data Location =  LocalVar  Int Type | GlobalVar Int Type
              deriving (Eq, Ord, Show)
newtype Key = Var String
              deriving (Eq, Ord, Show)

data Info = Info {
   -- genEnv has the data needed to generate code like variables and where they are
   genEnv :: Map.Map Key Location,
```

```
    globalVarCounter :: Int,
    needsUnlink :: Bool,
    needsOutOfBoundsRuntimeExeptionCode :: Bool,
    programArgs :: Args
}

type Env = State Info
```

The most important thing in this monad is the `genEnv`. It is a map that tells the code generator where variables can be found in memory. The `needsOutOfBoundsRuntimeExeptionCode` variable remembers if the array out of bounds runtime error runtime should be included or not. It is only included when the .hd field is used. I implemented monoid for `Env Code` so that I could combine instances of the `Env Code` without having to get the code out of the monad.

```
instance Semigroup (Env Code) where
      m1 <> m2 = m1 >>= (\code -> m2 >>= \code2 -> pure $ code <> code2)

instance Monoid (Env Code) where
      mempty = pure []
```

4.2 Data Representations

The data is represented in such a way that all data types only take up a single memory cell on the stack. Arrays are stored on the stack as a pointer to the heap where they are implemented as linked lists. The linked lists are achieved by using the `STMH 2` instruction. The end of a list is indicated by two zero values on the heap. Those zero values get there because an empty list literal generates `[LDC 0, LDC 0, STMH 2]`. A tuple is a pointer on the stack that points to the first value of the tuple stored in the heap. If you load this address with an offset of 1 you get the second value. The rest of the literals are quite trivial.

```
instance GenSSM (Literal TypecheckedP) where
      generate TrueLit = pure [LDC (-1)] -- There is also a True and False but its just a bit pattern
      generate FalseLit = pure [LDC 0]
      generate (IntLit int)  = pure [LDC int]
      generate (CharLit char)  = pure [LDC $ ord char]
      generate EmptyListLit = pure [LDC 0, LDC 0, STMH 2] -- address of 0 marks the end of the array!
      -- A tuple always has two values. These do not have to be other tuples.
      -- So lets make it [value1, value2], one of these can be an address but you don't know.
      generate (TupleLit (e1, e2)) = generate e1 <> generate e2  <> pure [STMH 2]
```

4.2.1 Global variables

Global variables are loaded into the heap at the start of the program. By loading them into the heap they can be assessed from anywhere. However, the heap pointer moves away after the start of the program so how can I get the values? I solved this by allocating the fifth register to store the starting address of the heap. This way whenever you want to load a global variable you have to push the register 5 value onto the stack and load with the offset. The offset of global variables can be obtained from the `genEnv`.

```
  generate (VarDecl _ name _ expr)  = do
    globalvarcount <- increaseGlobalVarCount
    modify $ \env@(Info {genEnv = genenv}) ->
                env {genEnv = Map.insert (Var name) (GlobalVar globalvarcount (getType expr)) genenv}
    generate expr -- The program generator generates the actual saving cause we do it in one go
```

4.2.2 Function Calls

When a function is called I use the 'Bsr' instruction with a label to branch to it. This pushes the current program counter onto the stack and jumps to the label.

I implemented function locals by using the special smm `LINK` and `UNLINK` instructions. These instructions make it simple to deal with locals and nested function calls. However, if a function does not have arguments and no return value I optimise and do not generate these instructions.

The arguments of a function call are passed through the stack. I managed to load the arguments and store them as locals in just two instructions. This works by using a load local multiple instruction with a negative argument to load values from before the mark pointer. Here is the code that generates these instructions:

```
[LDML (-(argcount + 1)) argcount | argcount > 0] ++ [STML 1 argcount | argcount > 0]
```

The local variables of a function are inserted as expressions at the start of the function code. As each expression leaves only a single value on the stack we can load all the values into locals with a single multiple instruction.
Because everything only takes up a single cell in memory I can pass the return values through the return register.

4.3 Polymorphism

The fact that all types only take up a single memory cell in the stack has the great advantage that it deals with polymorphism for free because the code generation can assume that the result of an expression always leaves a single value on the stack. If a function has 3 arguments it always has to load 3 memory cells from the stack. Because the program type checked we know that a polymorphic function will not perform any operation on its argument that will cause trouble.

4.4 Overloading

As can be seen from the example in the introduction both overloaded functions have been implemented. SPL has two build in overloaded functions. Printing `print` and `equality`.

Overloaded functions are treated as a special case in the code generator. This is possible because during the semantic analysis phase every expression was annotated with a it's type. This allows the code generator to generate specific code for each version of the overloaded functions.

4.4.1 Printing

I started out by implementing special code for the basic types like Char and Int. These types have special system call instructions to print them. The Boolean type is a bit more complicated because it needs a runtime condition to check whether a True or False should be printed. From this I moved on to lists and tuples. Lists were especially complicated because they need to iterate till the end of the list during runtime and also compile in the right print function for their argument. I achieved this by generating the print code recursively and jumping the right amounts. Implementing these specialisations was a very precise job. To give some idea of that I put a couple of the specialisations below but there are more.

```
generatePrint :: Type -> Code
generatePrint ty = case ty of
  CharType -> [TRAP 1] -- Print adds a newline
  IntType -> [TRAP 0] -- Print does not add a newline, for a newline call print without arguments
  (TypeVar tyname False) -> printStringCode ("Non rigid TypeVar " ++
    show tyname ++ ", idk how to print this!\n")
  (TypeVar tyname True) -> printStringCode ("Rigid TypeVar" ++ show tyname ++ ", idk how to print this!\n")
  BoolType ->
    let trueCode = printStringCode "True"
        branchSize = instrSize (BRF undefined)
        falseCode = printStringCode "False"
     in [LABEL "'printBool", BRF (codeSize trueCode + branchSize)] <> trueCode <>
        [BRA (codeSize falseCode)] <> falseCode
  ListType IntType ->
    [ LABEL "'printIntList",
      LDC (ord '['),
      TRAP 1,
      LDS 0, -- Remember adress of cons
      LDA 0, -- Load the adress
      BRF 20, -- If its empty list skip to the end
      LDS 0, -- Remember the cons cell for the next LDA
      LDA (-1), -- Otherwise load the value
      TRAP 0, -- Print it
      LDA 0, -- Move to the next cons cell
      LDS 0, -- Check if we should end it or print a comma and continue
      LDA 0,
      BRF 6,
      LDC (ord ','), -- print comma
      TRAP 1,
      BRA (-20), -- Jump back to the copy after the inital emtpy check
      LDC (ord ']'),
```

```
      TRAP 1,
      AJS (-1)
    ]
  ListType CharType ->
    [ LABEL "'printChrList",
      LDS 0, -- Save the adress of the cons cell
      LDA 0, -- Load the first adress
      BRF 10, -- If its empty list skip to the end
      LDS 0, -- Remember the cons cell for the next LDA
      LDA (-1), -- Load the value
      TRAP 1, -- Print it
      LDA 0, -- Move to the next cons cell
      BRA (-16),
      AJS (-1) -- Jump back to the brf 12
    ]
  ListType itemType ->
    let printItemCode = generatePrint itemType
        printItemCodeSize = codeSize printItemCode
     in [ LABEL "'printListList",
          LDC (ord '['),
          TRAP 1,
          LDS 0, -- Check for empty list once
          LDA 0,
          BRF (18 + printItemCodeSize),
          LDS 0, -- Remember the cons cell for the next LDA
          LDA (-1) -- Otherwise load the value
        ]
          <> printItemCode
          <> [ LDA 0, -- Load the next adress
               LDS 0, -- Remember adress
               LDA 0, -- Load the next cons cell to check for empty
               BRF 6, -- If its empty list skip to the end
               LDC (ord ','),
               TRAP 1,
               BRA ((-20) - printItemCodeSize + 2), -- Jump back to the brf 12
               LDC (ord ']'),
               TRAP 1,
               AJS (-1) -- These prints leave 1 value on the stack always just clean that up
               -- ListType (ListType a) ->
             ]
  TupleType a b ->
    let printAcode = generatePrint a
        printBcode = generatePrint b
     in [ LABEL "'printTuple",
          LDC (ord '('),
          TRAP 1,
          LDS 0, -- Copy for the second value
          LDA (-1)
        ]
          <> printAcode
          <> [LDC (ord ','), TRAP 1, LDC (ord ' '), TRAP 1, LDA 0] -- Print `, ` and load next value
          <> printBcode
          <> [LDC (ord ')'), TRAP 1]
  ty' -> printStringCode ("Error: Can not print value of type " ++ showTypeWithoutColor ty' ++ "\n")
```

4.4.2 Equality

Equality is implemented in much the same way as printing. It generates the correct equality instructions for every type. I started with simple types and moved on to more complicated types like lists that recursively call the equality code generation of the simpler type.
If you compare polymorphic types like this:

```
foo(a, b) {
    return a == b;
```

```
}
```

you will get this warning.

```
 WARNING: You are comparing two type variables:  Typevar d and Typevar d at examples/warning/polly.spl:3:12.
These two type variables share the same name, which means they will have the same type.  However,
this comparison will be a simple reference comparison, not a value comparison.

Examples of reference comparisons:
- 1 == 1 -> True
- 1 == 2 -> False
- "Hi" == "Hi" -> False (These strings have the same value but different memory locations)
- a == a -> True (Always true, since it's comparing the same variables and thus the same memory location)

In summary, this comparison will work for primitive types like Int or Bool, but not for complex types
like Lists, Tuples, or Strings unless they refer to the same memory location.  Ensure this behaviour
is intended for your code.
```

4.5 Runtime Error

The implementation `GenSSM` for `Program TypecheckedP` generates the code for all the declarations but also a runtime. This runtime includes declaring the global, branching to main and halting but it also contains a runtime error that prints an error when you do an out of bound array index.

Each time code is generated for a .hd field access the `headaccess` function is called. This function loads the address from the heap and checks if the address to the next cons cell is 0. If it is then it was an out out bounds array access and the code jumps to `'outOfBoundExpection`.

```
headaccess :: SourceSpan -> Env Code
headaccess meta = includeOutOfBoundsRuntimeExceptionCode >> pure (checkBounds <> [LDA (-1)])
  where
    checkBounds1 = [LDS 0, LDA 0]
    checkBounds2 = [LDC (startCol meta), LDC (startLine meta), Bra "'outOfBoundExpection"]
    jumpSize = codeSize checkBounds2
    checkBounds = checkBounds1 <> [BRT jumpSize] <> checkBounds2
```

Only if `needsOutOfBoundsRuntimeExeptionCode` is true in the environment (`headaccess` sets this to true) will the `'outOfBoundExpection` function be included into the runtime. The first time that this boolean is set to true the compiler will also emit the following warning.

```
WARNING: Including list out of bounds runtime
```

If you actually do an out of bounds array access like this:

```
foo() {
    var a = [];
    return a.hd;
}

main() { foo(); return; };
```

the code will print:

```
Array out of bounds runtime expection:
Trying to access .hd on an empty array.
Caused by line 12 column 4.
```

As you can see it remembers where the error happened in the source code during the runtime.

4.6 Patching SSM

During this project I extended the SSM virtual machine with two new instructions. `TRAP 2` to print null terminated character arrays on the stack and the ability to have function names that start with a ' character. This is useful for generation labels that the programmer can not create.

4.7 Problems

I did not experience any major problems implementing this phase. I made good foundations with the state monad. It was rather tedious however to get all the jump distances correct for the overloaded functions.

Chapter 5

Extension

As an extension I added the following things to the SPL language:

- Tuples
- String syntax in the parser
- Good error messages and compiler warnings
- Variable names can end with ' like in Haskell
- The optimiser step
- Runtime out of bounds checking for lists
- The following build in functions:
  - `getChar` - Asks a single char input and puts it into the stack
  - `printIntAsChar` - A function that will print an int as a unicode char
  - `exit` - This function will hald the program

I was unable to add WASM code generation to the compiler as my partner left during the project. This meant that there was simply no time. Had he stayed involved I am sure it would have been implemented.

Chapter 6

Conclusion

6.1 Reflection

Besides learning a great deal about compilers, I learned two valuable lessons that made me a better programmer. "it is more important for code to be easy to break than easy to write" and "design for the hardest thing."

For "Easy to break over easy to write" imagine changing the type of a date value in Python to an int from a string. Next month you might still have bugs coming from this change while in Haskell it is immediately clear which part of the code broke due to your change. This is well worth the increased difficulty in writing Haskell over Python.

With "Design for the hardest thing" I mean that when solving a problem you should always aim to solve the most difficult version of it you know. Otherwise you might create a solution that works for a lot of cases but when you want to solve a more difficult case you have to rewrite a large part of the code. This happened multiple times during this project where supporting a final difficult case actually required a large rewrite. As an example at some point I could type check all the expressions except for function calls. I had to make a lot of changes to also support function calls. I feel like compilers are especially susceptible to this. With compilers you really have to choose the right abstractions otherwise it just will not work.

6.2 Conclusion

The journey of constructing this compiler has been an enlightening and challenging experience which will stay with me for the rest of my life. Throughout this project, I encountered and overcame numerous technical challenges, which changed my perspective on what I want to do with my life and that is working on complicated things.

One of the most significant aspects of this project was the deep dive into Haskell and functional programming. Initially, my understanding of Haskell was rudimentary, stemming from a course in my bachelor's degree. However, the demands of this project, particularly in implementing the type checker, caused me to explore advanced concepts in greater depth. Finally I can truly say I really understand advanced concepts such as Monads and Monoids.

In conclusion, this project has been a profound learning experience, pushing the boundaries of my technical knowledge and skills. The lessons I learned from this project will undoubtedly greatly improve my future endeavours in computer science and software development.
The code of the compiler can be found at https://github.com/tintin10q/SPL-compiler.

6.3 Acknowledgements

I started this project together with Marijn van Wezel (s1040392). We collaborated on Chapter One of this work. After completing most of the work in Chapter One together, Marijn decided to discontinue his involvement.

Bibliography

[1] Atze Dijkstra. Simple stack machine (ssm), 2019. Accessed: 2024-06-24.

[2] Martin Grabmüller. Algorithm w step by step, 2006.

[3] Haskell Wiki. GHC/Type families, 2024. Accessed: 2024-06-25.

[4] Mark Karpov. A gentle introduction to parsing with megaparsec. https://markkarpov.com/tutorial/megaparsec.html, 2024.

[5] Daan Leijen et al. *Parsec*, 2013. Version 3.1.9.

[6] Simon Marlow et al. Haskell 2010 language report. *Available online http://www. haskell. org/(May 2011)*, 2010.

[7] MegaParsec contributors. Megaparsec. https://hackage.haskell.org/package/megaparsec-5.1.1/docs/Text-Megaparsec.html, 2024. Version 5.1.1.

[8] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[9] Rosetta Code. Operator precedence in haskell, 2024. Accessed: 2024-06-25.

[10] Simon Peyton Jones Shayan Najd. Trees that grow. *Journal of Universal Computer Science*, 23(1):42–62, 2017.

[11] Hspec Team. Hspec - a testing framework for haskell, 2024. Accessed: 2024-06-25.

Appendix A

Grammar

```
| Symbol  | Meaning                                |
|---------|----------------------------------------|
| `|`     | or                                     |
| `'if'`  | literal string                         |
| `'\''`  | literal single quote                   |
| `[ a ]` | should appear 0 or 1 time              |
| `a*`    | should appear 0 or more times          |
| `a+`    | should appear 1 or more times          |

Program = Decl+

Type =
    | 'Int'
    | 'Char'
    | 'Bool'
    | 'Void'
    | '(' Identifier ',' Identifier ')'
    | '[' Identifier ']'
    | Identifier  -- a

Decl =
    Identifier '(' [Identifier [':' Type]
                ',' Identifier [':' Type] ]* ] ')' [':' Type] '{' Stmt* '}'
    [Type] Identifier '=' Expr ';'

Stmt =
    | 'return' [Expr] ';'
    | 'if' '(' Expr ')' '{' Stmt* '}' [else '{' Stmt* '}']
    | 'while' '(' Expr ')' '{' Stmt* '}'
    | Variable '=' Expr ';'
    | Expr  ';'
    | 'var' [Type] Identifier Expr  ';'

Expr =
    | Expr BinOp Expr
    | UnaryPrefix Expr
    | UnaryPostfix Expr
    | Identifier '(' ')'
    | Identifier '(' Expr [',' Expr]* ')'
    | Variable
    | Literal

UnaryPrefix =
    | '!'
    | '-'

UnaryPostfix =
```

```
    | '.' Field

BinOp =
    | '*'
    | '/'
    | '%'
    | '+'
    | '-'
    | ':'
    | '>'
    | '>='
    | '<'
    | '<='
    | '=='
    | '!='
    | '&&'
    | '||'

Variable = Identifier ['.' Field]

Field =
    | 'hd'
    | 'tl'
    | 'snd'
    | 'fst'

Literal =
    | 'true'
    | 'false'
    | Int
    | Char
    | '('Expr ',' Expr')'
    | '[]'

Char = ''' UnicodeChar '''
Float = [ '-' | '+' ] Int '.' Int
Int = [ '-' | '+' ] digit+
Identifier = (alphaNumLower | '_') (alphaNum | '_') '\''*
```