Compiler construction

Ivo Melse
s1088677

Erik Oosting
s1136456

February 20, 2025

Contents

Chapter 1

Introduction

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

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

1.1 Language choice: Haskell

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

1.2 Simple Programming Language (SPL)

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

1.3 Example programs

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

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

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

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

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

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

Chapter 2

Lexical analyses

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

2.1 Lexing

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

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

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

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

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

Figure 2.1: Example of lexing with keywords

2.2 The parser monad

For parsing, we use a custom parser combinator:

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

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

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

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

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

2.3 General parsing guidelines

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

2.4 Expression parser

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

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

2.4.1 Precedence

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

- Unary operators bind the most tightly

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

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

All in all, we get the following precedence table 2.3.

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

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

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

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

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

Figure 2.3: Operator precedence table

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

2.5 Error handling

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

2.5.1 The `Result` monad

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

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

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

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

The `Applicative` instance should trivially follow from this.

2.5.2 Collecting & Pruning

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

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

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

This once again runs into the issue of `pure`. Once we have a parser that always succeeds, we still lose all the progress we made. This is especially true for the `many` parser combinator, which, failing to consume any input, returns successfully with the empty list. To fix this, we implement `many` ourselves (and since `some` doesn't actually mutually recurse into `many`, likely because of compiler inlining reasons, we need to define that parser as well)

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

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

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

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

2.5.3 What's left to do

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

2.6 The Abstract Syntax Tree

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

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

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

We also chose to make the `Ty` polymorphic. This allows us to do away with names when we have to go and inference/unify new types during the type checking phase. During the parsing phase, they are instantiated as `Ty String`

Down the line the AST may be changed, most likely in the `Exp` definition. This is because, as it stands currently, the AST doesn't easily allow itself to be changed easily. We can however, separate `Exp` into it's fixpoint (`Fix`, see below), and it's branching logic, `ExpF a` [5]

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

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

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

[5]See also: recursion-schemes

Chapter 3

Semantic analyses

- New Abstract Syntax Tree? Decorate existing Abstract Syntax Tree?
- Error messages?
- Polymorphism? Inference? Overloading?
- Problems?
- ...

Chapter 4

Code Generation

- Compilation scheme?
- How is data represented? Lists?
- Semantics style, call-by-reference, call-by-value?
- How did you solve overloaded functions?
- Polymorphism?
- Printing?
- Problems?
- ...

Chapter 5

Extension

Describe your extension in detail

Chapter 6

Conclusion

What does work, what does not etc.

6.1 Reflection

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

Appendix A

Grammar

Change the grammar to the one you actually used

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

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

Appendix B

Haskell definition of the abstract syntax tree

```
type Prog a = [Decl a]

data Decl a
    = Var (VarDecl a)
    | Fun (FunDecl a)

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

data Exp
    = Ident String
    | Binop Op2 Exp Exp
    | UnOp Op1 Exp
    | LitInt Integer
    | LitChar Char
    | LitBool Bool
    | Funcall String [Exp]
    | EmptyList

data Op1 = Sel Field | Not | Negate | Minus

data Field = Hd | Tl

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

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

data FunDecl a = FunDecl String [(String, Maybe (Ty a))] (Maybe (Ty a)) [VarDecl a] [Stmt]
```