Compiler Construction

Lexical Analyses (Interim Report)

Marijn van Wezel (s1040392) Quinten Cabo (s1076992)

February 19, 2024

1 Introduction

This report outlines our progress with the SPL compiler.

The language we will be implementing is called SPL (Simple Programming Language). It is similar to C, but with support for polymorphic datatypes.

Some examples of SPL include:

```
fib(n : Int) : Int {
    if (n == 0) {
        return 0;
    }

    if (n == 1) {
        return 1;
    }

    return fib(n - 1) + fib(n - 2);
}
```

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

    return n * fac(n - 1);
}
```

We chose Haskell to implement our compiler, since it is especially well-suited for compiler construction. Mainly Haskell's support for algebraic data types and its well-suitedness for writing parsers makes it ideal for crafting compilers. It also provides a good learning opportunity for both of us, since we want to get better at Haskell.

2 Lexical analyses

This section describes the lexical analyses phase of the compiler (parsing).

2.1 Designing the abstract syntax tree

We designed the abstract syntax tree immediately the first week. It was the first thing that we did in the project and we spent quite some time on it. We used a bottom-up approach to construct the AST. We started with the simple constructs (e.g. literals), and worked up to larger constructs such as function declarations. The program is just a list of function declarations.

Since we did not have a grammar yet, we looked at the available examples in the GitLab repository to figure out the syntax of SPL.

2.2 The parser

We use a parser combinator for parsing. We started with building our own parser combinators. This was a good learning experiance. But we soon realised this effort would require an significantly amount of work and make it much more difficult to get good error messages. We decided that using more mature tools instead of trying to reinvent the weel would give us more time to work on cool extensions.

Therefore, we switched to the parser combinator library megaparsec, which is the (informal) successor of parsec.

To become more familiar with the libary, and for occasional tips, we used this excellent tutorial by Mark Karpov.

Parser combinators work through *function composition*, where we take very simple parsers (such as "parse a single character") and compose these together to create more complicated parsers (such as "parse the word 'parser' "). This composition usually happens through helper functions, defined by megaparsec, that take zero or more parsers, and construct a new (bigger) parser. For example, the `<|>` combinator takes two parsers, and constructs a new parser that first tries the parser on the left, and if it fails tries the parser on the right.

Since parsers in megaparsec are also part of a number of useful typeclasses, such as Monad, we can use a number of useful functions Haskell itself provides to compose parsers (e.g. `do` notation, `many`, `some`, `<$>`).

2.3 Handling associativity

Since we use the megaparsec library, we get quite a number of things for free, such as error handling and associativity. Megaparsec provides us with the helper function `makeExprParser` that 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]]
operatorTable =
    [ [ Prefix (UnaryOp Negate <$ L.tExcl)
      ]
    , [ InfixL (BinOp Mul <$ L.tStar)
      , InfixL (BinOp Div <$ L.tSlash)
      , InfixL (BinOp Mod <$ L.tPercent)
      ]
    , [ InfixL (BinOp Add <$ L.tPlus)
      , InfixL (BinOp Sub <$ L.tMin)
      ]
```

```
    , [ InfixR (BinOp Cons <$ L.tColon)]
    , [ InfixN (BinOp Gt <$ L.tGt)
      , InfixN (BinOp Gte <$ L.tGte)
      , InfixN (BinOp Lt <$ L.tLt)
      , InfixN (BinOp Lte <$ L.tLte)
      , InfixN (BinOp Eq <$ L.tDoubleEq)
      , InfixN (BinOp Neq <$ L.tExclEq)
      ]
    , [ InfixR (BinOp And <$ L.tDoubleAmpersand) ]
    , [ InfixR (BinOp Or <$ L.tDoublePipe) ]
    ]
```

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;
- `Prefix`: for prefix operators;
- `Postfix`: for postfix operators.

We chose the same precedence and associativity as Haskell (source).

2.4 Error handling

Error handling is handled by megaparsec, which, out of the box gives very 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 we have not implemented those yet.

2.5 Lexer

We do not have a seperate lexing step. Instead, we use a (what we call) just-in-time lexer. 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 keeping whitespace and comments into account.

This is (mostly) done using the `symbol` parser combinator, which 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 */`.

We use these lexer parsers all throughout the rest of the parser, in places where you would normally consume a token with a regular parser.

We chose this approach, since it is well supported by megaparsec. Eventhough megaparsec does work on arbitrary input streams (including user-defined tokens), you cannot use a large part of the predefined parsers created by the megaparsec community. We do not see a benefit of converting the whole input into tokens first when using parser combinators.

2.6 Pretty printing

We have not yet implemented the pretty printer. But we do know that with our current lexer design all the comments will be lost. To resolve this we could store the comments somewhere outside of the AST or actually have a lexer step, seperate from the parsing.

2.7 Testing

For testing, we use the testing library hspec, which has great support for megaparsec through the hspec-megaparsec library.

These tests really helped us during development, as we were able to find bugs quickly.

2.8 Problems

Since we started with writing the AST in Haskell based on the examples in the repo, we did not have any major problems during this phase. Having a well-defined AST really helped us make the parsers, as we were able to start from the most simple parsers (such as literals), and easily work our way upwards.

We initially did struggle with the left-recusivity of property access (e.g. `a.hd`), but fixed this eventually.

We also wrote many tests for our parsers from the start, which helped reduce bugs.

3 appendix

3.1 Grammar

Change the grammar to the one you actually used

```
Grammer
```