Compiler Construction

Lexical Analyses (Interim Report)

Marijn van Wezel (s1040392) Quinten Cabo (s1076992)

February 22, 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. We used our ast as a grammer for the most part but we did write an extended BNF grammer in the appendix.

Our abstract syntax tree is split up into four major inductive types, `Decl`, `Stmt`, `Expr` and `Literal`, with some helper inductives (e.g. for the types and binary operators) as well. These inductive types have the following meaning:

- `Decl`: for declarations (functions, global variables);
- `Stmt`: for statements (e.g. return, if, while);
- `Expr`: for expressions (e.g. binary expression, assignment, function call);
- `Literal`: for literals (e.g. numbers, chars, tuples).

We have decided to not allow global variable declarations at the moment, because global variables are evil. Instead, we have made variable declarations a statement. This also has the benefit of allowing variable declarations everywhere in the body of a function, and not just at the start. In the future, we might add global variables to make the compiler more flexible (and to make more of the examples in the repository functional).

2.2 The parser

We use parser combinators for parsing. We started with building our own parser combinators. While this was a good learning experience, we soon realised this would require a significant amount of work and would make it much more difficult to get good error messages. Therefore, we decided that using more mature tools (e.g. an existing parser combinator library) 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, the maintainer of megaparsec.

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. A large part of the internal mechanics of parser combinators, such as carrying error messages, are hidden away in the `Parser` type.

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

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 also 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 =
    [ [ Postfix (UnaryOp (FieldAccess HeadField) <$ try (L.tDot <* L.tHead))
      , Postfix (UnaryOp (FieldAccess TailField) <$ try (L.tDot <* L.tTail))
      ]
    , [ 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). We also use the `makeExprParser` for the field access operators (`.hd` and `.tl`), as this was the easiest solution.

2.4 Error handling

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 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 (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. In megaparsec, whitespace is only discarded at the end of tokens in the lexer. 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.

"Lexing" 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 and it just easily deals with any comments and whitespace as soon as you use the symbol parser. 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. ‘ ## Pretty printing

We have implemented a pretty printer. However, since comments are not included in the AST, our pretty printer strips comments, and can therefore not realistically be used for formatting. We have discussed creating a seperate data type that holds the locations of all the comments in the source code. We think a seperate datatype is a good idea because then we do not have to bloat all the AST nodes with comments. This seperate datatype could also be part of the program type so that it is still part of the AST but only in once place. Once this is implemented we could restore the comments in the pretty printer.

2.6 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.

2.7 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 early on 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 by making `.hd` and `.tl` 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.

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

3 Appendix

3.1 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* '}'

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

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

UnaryPrefix =
    | '!'
```

```
UnaryPostfix =
    | '.' Field

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

Variable = Identifier ['.' Field]

Field =
    | 'hd'
    | 'tl'

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

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