Compiler Construction

Semantic Analyses (Interim Report)

Marijn van Wezel (s1040392)   Quinten Cabo (s1076992)

April 7, 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 Semantic analyses

This section describes the semantic analyses phase of the compiler (typechecking, etc). We are currently behind on schedule with our compiler, and have therefore not fully implemented this phase yet. We will try to catch up in the coming weeks. In this section, we explain what we have done so far, and the problems that we are currently tackling.

3.1 Trees that grow

We wanted a flexible system to annotate trees with additional metadata that extends to the multiple phases of the compiler. To achieve this, we have implemented a variation of Trees that Grow (Najd and Jones 2016).

The approach to annotate our AST is based on type families. This works by adding a parameter to each (relevant) `data` type that determines the phase of the compiler. Using type families you can create seperate instances of `data` based on the phase of the compiler while keeping the general structure of the AST the same. Without type families you would need to build a seperate similar but distint trees for every phase of the compiler.

By using the `DataKinds` language extension, the phase is also a `data` type:

```
data Phase
  = EmptyP       -- Empty phase, used for testing
  | ParsedP      -- Phase after parsing with location information
  | TypecheckedP -- Phase after full typechecking
```

Next, we add to each constructor of the `data` type a new field that has a type family with the phase as its typed. Since type families are functions from type to type, we can control the actual type of that field through the phase.

To explain this concept fully, we will look at how to implement this approach for a simplified version of expressions in SPL, where only binary expressions and literals are allowed. We start with the naive approach of using separate `data` types for each phase, then we introduce type families.

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

data ExprParsed
  = BinOpExprParsed SourcePos BinOp ExprParsed ExprParsed
  | LiteralExprParsed SourcePos Literal

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

The above approach is clumpsy, and requires us to make modifications in many places to make changes to the AST, since it requires us to keep all these different `data` types in sync. Additionally, it forces us 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).

Trees that Grow (Najd and Jones 2016) solves this by introducing a type family, which is a function from type to type, for each field of the `data` type. The only argument to these type families will be the phase of the AST. The type families for this example will therefore be:

```
type family BinOpExpr (p :: Phase)
type family LiteralExpr (p :: Phase)
```

Any fields specific to a particular phase of the compiler will then be replaced by a field of the associated type family, like so:

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

Finally, for each phase we can create an instance of the type family that determines the actual type for that field:

```
type instance BinOpExpr EmptyP = ()
type instance BinOpExpr ParsedP = SourcePos
type instance BinOpExpr TypecheckedP = (SourcePos, Type)

type instance LiteralExpr EmptyP = ()
type instance LiteralExpr ParsedP = SourcePos
type instance LiteralExpr TypecheckedP = (SourcePos, Type)
```

This gives us a lot of control over the actual contents of the AST for each type, since it allows us to assign a different type for each constructor in the AST for each phase of the compiler, while keeping the overall structure of the AST identical. Furthermore, it allows us to write functions that work on any phase by simply ignoring the field with the type family.

3.2 Error messages

In the typecheck phase we have to unify the types that variables have with the types of expressions and the uses of the variables. This unification operation may fail, and as such our `unify` function returns an `Either`.

Because we have the location for each node in the AST, we can use this location to generate decent error messages that point to a specific location in the source code. However, since we have not yet finished the typechecking phase completely, we have not yet implemented generation of these error messages. Currently, the error is a simple string that explains *how* the unification failed (e.g. “Cannot unify `Int` with `Bool`.”).glorious .

3.3 Polymorphism, inference and overloading

We are planning on supporting polymorphism, inference and overloading. However, since we have not fully implemented typechecking yet, we cannot talk about this yet.

3.3.1 Type variable instantiation

We have already made the choice to have type variables be local to their (function) declaration, meaning that the “same” type variable used in different declarations will actually be different during typechecking. For example, in the following code, the `a` in `f1`’s signature is different from the `a` in `f2`’s signature:

```
id1(v: a): a {
  return v + v;
}

id2(v: a): a {
  return v;
}
```

This means that when we determine the `a` in `id1` to be an `Int`, we do **not** update the `a` in `id2`. In practice, this is implemented by having a phase that instantiated the tree with *unique* type variables first.

This mirrors the functionality of Haskell.

3.3.2 Polymorphism

We have not yet implemented polymorphism.

3.3.3 Inference

We have not yet implemented inference.

3.3.4 Overloading

We have not yet implemented overloading.

3.4 Problems

In this section, we explain the problems that we are currently tackling.

3.4.1 Missing type information (`InstantiatedP` phase)

The first problem that we faced was missing type information. In the AST, we used `Maybe Type` for annotating the AST with type information during parsing. For example you can say `f() { return 4; }`. The return type of this function declaration is (during parsing) `None`.

We added the `InstantiatedP` phase to deal with this. In the `InstantiatedP` phase we replace all the `Nothing` types with a unique type variable.

When adding fresh type variables in the `InstantiatedP` phase we make sure that the type variables of function arguments and the return value refer to the same types, but only in the context of that function. This idea is best explained with an example:

```
foo(arg1 : a, arg2 : b) : a {}
bar(arg1 : b, arg2 : b) : a {}
```

Will be converted into:

```
foo(arg1 : f0, arg2 : f1) : f0 {}
bar(arg1 : f2, arg2 : f2) : f3 {}
```

This means that if you use the same polymorphic type variable name in multiple functions (like arg2 in both `foo` and `bar`) then the type variable only corresponds to the same type within that same function (see also `Type variable instantiation` above). So within `foo`, arg1 and the return type should have the same type. But we made sure that this does NOT have to be the same type as the return type of `bar`. Because we made the sensible choice that the `a` in the return type of `bar` should be a different type variable.

3.4.2 Empty list types

When we where assigning types to expressions we started by making a function that assigned a type to literals. We hit a problem when we had to type the empty list literal. What type do we assign to this expression? We have not figured that out yet.

4 Appendix

4.1 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
    | Char
    | '('Expr ',' Expr')'
    | '[]'

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

Bibliography

Najd, Shayan, and Simon Peyton Jones. 2016. "Trees That Grow." *CoRR* abs/1610.04799. http://arxiv.org/abs/1610.04799.