An SPL compiler in Rust

Ties Klappe
s1030293

June 4, 2022

Contents

Chapter 1

Introduction

In this report I introduce the compiler for a programming language called SPL, implemented in Rust. This first chapter gives the motivation for choosing Rust as the implementation language, as well as some general information about SPL itself. Chapter 2 covers the frontend of the compiler, i.e. lexing, parsing and pretty printing. Chapter 3 is concerned with semantic analyses and type checking. In Chapter 4, I discuss code generation. In Chapter 5 I propose and describe an extension to the project. Finally, Chapter 6 provides a conclusion and reflection on the project as a whole.

1.1 Language Choice

The programming language that was chosen to create the compiler is Rust. Although the teachers advised to use the pure functional programming language Haskell, I was not comfortable enough writing Haskell code for a relatively complex application such as a compiler. Rust was chosen because I have used Rust prior to this course. Besides, it provides some nice (functional) features such as pattern matching and closures, which allows for writing expressive code. It also produces very fast programs, similar to C/C++, while suffering less from issues common to the latter two, such as memory unsafety.

1.2 SPL

Simple Programming Language (SPL) is a programming language with a simple C-like syntax. It is a strict language, meaning function arguments must be fully evaluated before a function is called. It is also first-order, as functions may not take other functions as arguments. SPL has a polymorphic type system, where a given function may be called with different parameter types. There are several overloaded operators that work on some or all types, and a number of built-in polymorphic functions. For an overview of the available operators, see Section 2.2.1 and Appendix A.

At the top level, SPL programs consist of a number of variable and function declarations. Users can specify the type of variables explicitly, or declare them as `var`, in which case the type will be inferred. Supported data types are **Int**, **Bool**, **Char**, [**t**] (arrays of type **t**) and **(t1, t2)** (tuples containing values of type **t1** and **t2**).

For function declarations, the function type signature can also be specified or left out. Function signatures consist of zero or more parameter types, as well as a return type. The return

type can also be `Void`, indicating the lack of a return value. Function bodies must contain one or more statements. Available statements are standard `if` (with an optional `else` branch), `while` loops, variable declarations, variable assignments, function calls and returns.

1.2.1 Examples

An example in SPL that checks whether a character is a vowel is given in Listing 1.1. An example in SPL that computes the sum of the first n natural numbers is given in Listing 1.2.

```
1  isVowel(c) :: Char → Bool
2  {
3      if ( c=='a' || c=='e' || c=='i' || c=='o' || c=='u' ) {
4          return True;
5      }
6
7      return False;
8  }
9
10 main() :: → Void
11 {
12     // True
13     print(isVowel('a'));
14
15     // False
16     print(isVowel('b'));
17 }
```

Listing 1.1: Vowels example

```
1  // Compute the sum of the first n natural numbers.
2  sum(n) :: Int → Int
3  {
4      var total = 0;
5
6      while(n > 0) {
7          total = total + n;
8
9          n = n - 1;
10     }
11
12     return total;
13 }
14
15 main() :: → Void
16 {
17     // 15
18     print(sum(5));
19
20     // 5050
21     print(sum(100));
22 }
```

Listing 1.2: Natural numbers example

Chapter 2

Lexing & Parsing

The compiler frontend consists of a scanner and a parser. We have chosen the separate scanner/parser approach because it simplifies the parser: instead of having to parse the raw input text, input is first converted into a sequence of abstract tokens. Moreover, this allows us to perform several processing steps such as filtering whitespace, skipping comments and performing character and string escapes before parsing. That way the parser can focus on determining the structure of the input, without having to worry about irrelevant issues like layout.

A side effect of the separate lexer and parser is that both are easy to unit test, and as a result there is a sizeable set of unit tests for both components, which are run using continuous integration in the Git repository.

In this chapter, we describe the scanner, modifications to the provided grammar, the parser and the challenges encountered while developing the frontend of the compiler.

2.1 Scanner

The scanner (sometimes also called lexer or tokenizer) splits up an input text into a list of abstract tokens which represent one or more characters of concrete input. The scanner filters the input by removing whitespace characters, single-line comments and multi-line comments. If a multi-line comment was not closed, an appropriate Error token is passed on to the parser. For each token type, there exists an associated regular expression. For example, an identifier is described by the regex `[[:alpha:]](_|[[:alnum:]])*`. This approach allows for a very generic and straightforward implementation of the scanner: most tokens are generated by generic regex matching, and only for literal tokens we have to do some 'manual' work by extracting the literal value and placing it into a literal token. For `Char` and `String` literals this includes removing possible escape sequences. The following characters have to be escaped within both literals: \, `\n`, `\r`, `\t`. For a Char literal, a single quote must be escaped. For a String literal, a double quote must be escaped.

The scanner implements the `Iterator`[1] trait. This allows the parser to *collect* the tokens into a vector of tokens, and ultimately into a `Tokens` object. The Tokens type implements all traits that are required by the parser combinator library, which will be introduced in the paragraph 2.4.

[1] `https://doc.rust-lang.org/std/iter/trait.Iterator.html`

2.2 Grammar

Before we continue with the parser, it makes sense to look at the grammar and the modifications to it. The grammar on which the parser is based can be found in Appendix A. There were several issues with the provided grammar that needed to be fixed; these issues were primarily related to left recursion and operator precedence. The compiler uses a top-down parsing approach, which means the parser cannot deal with rules like the following (*left recursion*):

```
Field = [Field ('.' 'hd' | '.' 'tl' | '.' 'fst' | '.' 'snd')]
```

When parsing sentences of this form, the parser would keep recursively deriving the right-hand `Field`, never finishing. For example, given a token `.hd`, the parser would not be able to determine, in general, whether more `Field`s follow, and so it has to decide to parse `Field` again.

Moreover, consider the following (simplified) expression rule:

```
Exp = Exp '+' Exp
    | Exp '*' Exp
    | num
```

This rule, besides being left-recursive and ambiguous, also does not consider operator precedence. A sentence like `1 + 2 * 3` could be parsed as `(1 + 2) * 3` or `1 + (2 * 3)`, with the former clearly being wrong assuming the usual order of operations. To solve the aforementioned issues, we have modified the provided grammar in several places, which are described in the next paragraph.

2.2.1 Modifications

- `FArgs = [ FArgs ',' ] id` has been rewritten to `FArgs = id [ ',' FArgs ]` in order to remove left-recursion.
- `Field = [ Field ( '.' 'hd' | '.' 'tl' | '.' 'fst' | '.' 'snd' ) ]` has been rewritten to `Field = ( '.' 'hd' | '.' 'tl' | '.' 'fst' | '.' 'snd' ) [ Field ]` We have done this in order to remove left-recursion and to make field access consistent in the grammar, i.e. optional at the location where it is used instead of optional in its definition.
- Following the previous point, we have made `Field` optional in assignment statements: `id [ Field ] '=' Exp ';'` Field access is also optional in expression atoms, as discussed in the next point.
- The `Exp` nonterminal has been rewritten to remove left-recursion and to encode operator precedence in the grammar. Each precedence level has a separate rule in the grammar. The precedence levels are listed below, lowest to highest. These were partly inspired by Haskell's operator precedence.
  1. `||`
  2. `&&`
  3. `==`, `!=`, `<`, `<=`, `>`, `>=`
  4. `:`
  5. `+`, `-` (binary)

6. `*`, `/`, `%`
7. `-` (unary), `!`

Operator fixity is also encoded in the grammar: infix operators always occur between nonterminals, while prefix operators always precede nonterminals. Operators of the same precedence level can be chained. However, associativity is not taken care of in the grammar. Rather, chained operators are handled during parsing, manually enforcing the desired associativity.

At the highest precedence level are expression atoms:

- variables with optional field access;
- integer, character, boolean and string literals;
- bracketed expressions;
- function calls;
- the empty list literal;
- tuples.

- `int = [ '-' ] digit+` rewritten to `int = digit+`
  Negative numbers are consistently handled by the unary minus operator, obviating the need for negative-number literals.

2.2.2 Dangling Else

A common problem with grammars and parsers of programming languages is the "dangling else" problem, as illustrated in Listing 2.1. Since omitting braces is allowed for if statements where the true/false cases consist of single statements, ambiguity is introduced (does the `else` belong to the outer or inner `if`?). Some languages, like C, solve this by always associating the else branch with the closest if. Other languages, like Rust or Swift, choose to always enforce braces. We have opted for the second solution in the grammar, since it tends to make code more readable, and can even prevent difficult-to-diagnose bugs.[2]

```
1 if (condition1)
2   if (condition2)
3     statement1;
4   else
5     statement2;
```

Listing 2.1: Dangling else

[2] `https://dwheeler.com/essays/apple-goto-fail.html`

2.2.3 Lexer Hack

Some programming languages may have (syntactical) constructs which could have several different meanings. Two examples in C, taken from the lecture slides, are the following:

```
1 void fun()
2 {
3   T (x);
4 }
```

Listing 2.2: Function call or cast?

```
1 void fun()
2 {
3   T * x;
4 }
```

Listing 2.3: Multiplication or pointer declaration?

Here, the scanner cannot know how to tokenise the given program without additional contextual information. Modern compilers usually solve this issue by feeding this contextual information from a later analysis or parsing stage back into the scanner, allowing it to disambiguate the input.

Luckily, SPL does not contain such constructs, which simplifies the scanner and maintains a clean scanner/parser separation.

2.3 Abstract Syntax Tree

The abstract syntax tree is a data structure to represent the syntactic structure of programs in an abstract way, without concerning itself with issues like bracketed expressions and layout.

For the definition of the abstract syntax tree, we have taken advantage of Rust's type system. Rust supports algebraic data types: product types in the form of `struct`s and tuples, and sum types in the form of `enum`s. Using these constructs, combined with Rust's `Vec<T>` (growable list) and `Option<T>` (optional, cf. `Maybe` in Haskell), the abstract syntax tree structure is built based on the grammar. For convenience and in order to flatten the structure, the AST slightly deviates from the grammar in several places:

- `FArgs` is directly included in the `FunDecl` type, as it maps one-to-one to a list of identifiers;
- `FTypes` and `RetType` have been "folded" into the `FunType` structure;
- `BasicType` has been folded into the `Type` enumeration;
- The `Exp` operator precedence ladder has been flattened into a single `Expr` type;
- Optionality of variable and function types is represented using `Option<T>`.

2.3.1 Expressions

An interesting challenge encountered when defining the expression type is that types in Rust generally must have a size known at compile time. This shows up when defining a recursive data type for expressions. For instance, we would like to represent an expression containing an addition as follows:

```
1 enum Expr {
2     Add(Expr, Expr),
3     ...
4 }
```

However, this results in a recursive type of infinite size, which Rust cannot deal with. The way to solve this is to "box" the contained expressions:

```
enum Expr {
    Add(Box<Expr>, Box<Expr>),
    ...
}
```

In Rust, `Box<T>` is a pointer to heap-allocated data. By doing this, the resulting type *does* have a known size, as the contained expressions are now simply pointers. Although this does incur some (minimal) performance overhead, in practice the difference is negligible. In addition, Rust supports automatic dereferencing[3] in many places, which makes working with these boxed values effectively the same as working with (references to) the values themselves. This also avoids the need for automatic boxing and unboxing. The parser is the only place where we have to call `Box::new` quite a few times, but this is not really bothering.

2.4 Parser

For parsing SPL files, we employ a top-down approach. This is achieved using the `nom`[4] parser combinator library for Rust. Parsers created using `nom` are LL(k), meaning they scan left-to-right and perform leftmost derivation of input sentences, using a lookahead of k tokens. Parser combinator libraries in general allow building up larger parsers by first writing parsers for specific constructs, and then combining these parsers using combinators. `nom` was chosen because it is the state-of-the-art parser combinator library for Rust, with over 2 million downloads per month. It also strikes a nice balance between usability and efficiency: writing a top-down (e.g. recursive descent) parser by hand would likely result in a faster parser overall, but `nom` is much more user-friendly and flexible in case the grammar changes.

A benefit offered by parser combinators is that the resulting parsers often match their grammar quite closely. Since Rust is more verbose than a language like Haskell, the parser code does not resemble the grammar syntactically, but we have attempted to keep the structure similar to the grammar. Most nonterminals in the grammar have a parser function that can (attempt to) parse the given construct. Higher-level constructs then use combinator functions provided by `nom`, which take one or more parsers and return a new parser function with the desired behavior. For example, there is the `opt` combinator, mapping to the `rule = [ token ]` construct in (e)BNF grammars, which takes a parser `p` and returns a parser that applies `p` zero or one times. A more complete overview of the supported combinators can be found in `nom`'s documentation.[5]

The parser operates on a list of tokens produced by the scanner. It walks the list of tokens from left-to-right while consuming tokens and producing nodes and leaves of the abstract syntax tree data type. The AST is described in more detail in Section 2.3.

Because each parser is a function, we have unit tests for most parser functions. Moreover, we generate a set of integration tests for the parser (and scanner) from the provided set of example SPL programs, which test whether those examples parse correctly and completely.

[3] `https://doc.rust-lang.org/book/ch15-02-deref.html#implicit-deref-coercions-with-functions-and-methods`
[4] `https://lib.rs/crates/nom`
[5] `https://github.com/Geal/nom/blob/main/doc/choosing_a_combinator.md`

2.4.1 Example Parser Function

To give an idea of how parsers in Rust/Nom tend to look, Listing 2.4 shows the implementation of a parser for SPL function declarations. The type signature indicates that the function takes a list of tokens and returns nom's `IResult` type. IResult is defined as an enum with two variants: `Ok((InputType, OutputType))` and `Err(ErrorType)`. In this case, the input type is `Tokens` and the output type is `FunDecl`, i.e. the function returns the rest of the tokens and the parsed function declaration. The function itself consists of the following elements:

1. `map(parser, f)` takes a parser and a function and returns a new parser applying the function to the result of the given parser. Here, the function is defined on lines 17-22, and it constructs a `FunDecl` AST node. `|params| expression` is Rust's syntax for closures/inline functions.

2. `tuple((parser1, parser2, ...))` takes a tuple of parsers and returns a parser that succeeds if it can apply all parsers in order. This parses the following elements:

   - `identifier_parser`, parsing a single identifier;
   - `delimited(first, second, third)`, which applies the first, second and third parser in order and only returns the result of the second: a list of parameters surrounded by parentheses;
   - `separated_list0(sep, parser)` parses zero or more of the second parser, separated by the first: a list of identifiers separated by commas;
   - `fun_decl_type_parser` parses a function type signature;
   - `many1(statement_parser)` parses one or more statements (zero statements produces an error).

```
1  fn fun_decl_parser(tokens: Tokens) -> IResult<Tokens, FunDecl> {
2      map(
3          tuple((
4              identifier_parser,
5              delimited(
6                  opening_paren_parser,
7                  separated_list0(comma_parser, identifier_parser),
8                  closing_paren_parser,
9              ),
10             fun_decl_type_parser,
11             delimited(
12                 opening_brace_parser,
13                 many1(statement_parser),
14                 closing_brace_parser,
15             ),
16         )),
17         |(name, params, fun_type, statements)| FunDecl {
18             name,
19             params,
20             fun_type,
21             statements,
22         },
23     )(tokens)
24 }
```

Listing 2.4: Function declaration parser

2.4.2 Chain Rules

As mentioned previously, expressions allow operators of the same precedence level to be chained, e.g. `1 + 2 + 3` is legal syntax. For left-associative operators, `nom` supports the `fold_many0` combinator. This operator applies a parser `p` zero or more times, performing a left fold in the process, in a similar fashion to the `pChainl` combinator introduced in the lectures. In this way, we can build up a left-associate tree of operator applications.

Unfortunately, there is no right chain combinator. This means that to parse the list constructor operator (`:`), the implementation consists of a parser that collects the chain to a vector and then walks the vector right-to-left to build up the abstract syntax tree, instead of a simple right fold.

2.5 Pretty Printer

The pretty printer was implemented using the `pretty_trait`[6] library. This library provides functionalities to print arbitrary types by joining constructs, adding indentations and newline symbols and grouping related content together. The `to_pretty` function was implemented for every type of the AST. Pretty printing an entire AST can be done by simple converting the top level AST object (*Program*) to a `Pretty` object and printing it.

When printing expressions, some parts may need to be parenthesized. This is the case when a child expression node in the AST has a lower precedence than its parent: the child expression must be parenthesized in this case. For example, figure 2.1 displays an AST where the addition operator occurs below the multiplication operator. Since addition has lower precedence than multiplication (see section 2.2.1), the addition expression must be parenthesized when we pretty print this AST: $(1 + 2) * 3$. Figure 2.2 displays an AST where no child expressions have lower precedence, hence no parenthesis are needed when we pretty print this AST: $1 + 2 * 3$.

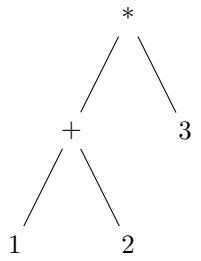

Figure 2.1: Parentheses required

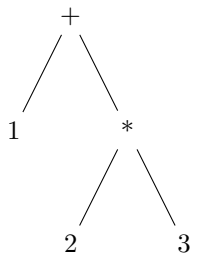

Figure 2.2: No parentheses required

The pretty printer does not keep comments, newlines and meaningless white spaces. Tabs have a size of four white spaces. A single line has a maximum length of forty white spaces.

Finally, several tests are written to verify that the AST of the original input file and the AST of the pretty printed file are identical.

[6] `https://lib.rs/crates/pretty-trait`

2.6 Errors

To make errors more readable, localized errors are added to the parser. Here, the goal is to be concise in what went wrong and help the programmer. A parser combinator can return two types of errors, a normal `Error` indicating that a specific combinator did not succeed but another might, or a `Failure` which indicates that the combinator was in fact the right one, but some syntax error occurred and the parsing has to be aborted.

To provide the parser with some additional information about errors that may occur, three customary combinators were written. The first one, `expected_token`, returns a failure when its inner combinator fails, and provides which token was expected. For example, when a variable declaration is not terminated by a ; and we were already certain that we could only be parsing a variable declaration, we can throw a failure and state that we expected the token ;. The second combinator, `expected_context`, is similar to the first one but provides some context in order to make a chain of errors nicely readable. In the example of the missing ; from above, we could have been in the function declaration parser, and thus it would be nice to add some information about the context in which a failure was returned. Finally, the `require` combinator turns any sort of error from its inner parser into a failure.

```
1 main() {
2     // Forgot the ';'
3     var x = 1
4
5     print(x);
6 }
```

Listing 2.5: Syntax error

A small program with a syntax error is given in listing 2.5. The format function for errors will combine the raw input, source location of tokens and error data to create an error message for the end user. Compiling the above example gives the error message from listing 2.6.

```
INFO  [spl_compiler] Lexing...
INFO  [spl_compiler] Parsing...
Error:

 at 5:5:
    print(x);
    ^
expected ';'

at 3:5, in var declaration:
    var x = 1
    ^

at 1:1, in function declaration:
main() {
^
```

Listing 2.6: Syntax error message

Chapter 3

Analyses & Typing

The middle end of the compiler implements binding time analysis, return path analysis and polymorphic type inference. In this chapter the implementation details of these functionalities are described. Several pieces of Rust code are shown to substantiate the statements made for each analysis, which are copied directly from the source code (available on Gitlab).

3.1 Binding time analysis

The binding time analysis is largely integrated in the type inference process, on which paragraph 3.3 elaborates. This paragraph explains all the rules on scoping and identifiers that are enforced by the compiler.

3.1.1 Declaration order

A problem arises when a programmer calls a function f that has not yet been defined. The AST constructed from the tokens in the previous phase simply reads the source code from top to bottom, and thus the AST nodes are not sorted on usage thereafter.

One of the possible solutions to this problem is to construct the call graph from the AST, performing topological sort and resorting the AST afterwards. In such a call graph, functions are presented as nodes and function calls as edges. As global variables are not allowed to make function calls (to be explained in section 3.1.2), we can effectively move all global variables to the front of the declarations list of the AST and sort them among themselves (they may be defined in terms of each other). The Reorder procedure below provides an abstract overview of the complete AST reordering.

Reorder(T)

1. Construct call graph G_v of global variables of T
2. Run topological sort on G_v
3. Insert global variable declarations from G_v in reverse at the front of T'
4. Construct call graph G_f of functions of T
5. Run topological sort on G_f
6. Insert function declarations from G_f in reverse at the back of T'
7. Return T'

We have chosen not to rearrange local variables, as we don't know whether a local variable may invoke a side effect and reordering them could possibly change the semantics of the program. For example, the program from listing 3.1 will give an error on line 7, because the variable y is undefined. If we would have swapped line 7 and 8, the program is no longer semantically equivalent because it would print 2 1 rather than 1 2. For global variables we do not have to worry about this, as they can be considered a kind of constant (which will be explained next).

```
1  foo(x) :: Int → Int {
2      print(x);
3      return x;
4  }
5
6  main() {
7      var x = foo(1) + y;
8      var y = foo(2);
9
10     return;
11 }
```

Listing 3.1: Semantics preserving example

3.1.2 Global variable restriction

A global variable is restricted from using function calls in the expressions representing its value. Allowing this would impose a difficulty when defining semantics for programs such as the one given in listing 3.2. Here, we don't know the value for x, because its expression is defined by a function f which is defined in terms of x. Using global variables in functions is in general not a problem, as can be seen in listing 3.3. Another solution would have been to analyse all function calls made by global variables, determine whether that function is defined in terms of that same global variable and give a compilation error if that is the case.

```
1 var x = f();
2 f() {
3     return x;
4 }
```

Listing 3.2: Global variable restriction

```
1 var x = 1;
2 f() {
3     return x;
4 }
```

Listing 3.3: Global variable usage

3.1.3 Context and scopes

The context that is used throughout type checking, is composed of three type environments $\{functions, global\_vars, local\_vars\}$. A type environment is implemented as a hashmap, that maps an identifier to a type scheme. The environments that compose the context have fairly self-explanatory names: the *functions* environment contains all function identifiers and types, whereas the *global_vars* and *local_vars* environments contain the identifiers of global resp. local variables at a given moment in the algorithm.

The compiler enforces the following rules:

1. Function and variable names may overlap. For example, a program that has a function named f and a global variable also named f is fine

As SPL only has first order functions, it is nice feature to allow the same name for a function and a variable.

2. Variables may be shadowed, but a warning is produced

   If a global variable x exists, and a local variable or function parameter is also identified by x, the last defined x will be used. Listing 3.4 shows a small example: if x occurs in the remaining body of f, the x declared at line 4 is the variable being referred to. Two warnings will be given: the first one when the function parameter at line 3 hides the global variable x, and the second one when local variable x at line 4 hides both the function parameter and global variable x.

3. Variable names must be unique in a scope

   As variables may be declared in a different order than in which they are used, allowing redefinition of variables in the same scope would make it very hard to determine which variable an identifier is referring to. For that reason, all variable names inside the same scope must be unique. A small example is given in listing 3.5: trying to compile this code would fail with an error that function argument x was previously declared.

```
1 var x = 10;
2
3 f(x) {
4     var x = 1;
5     ...
6 }
```

Listing 3.4: Local variable shadowing

```
1 var x = 10;
2
3 // This is not allowed.
4 f(x, x) {
5     ...
6 }
```

Listing 3.5: Function parameter duplicate

3.2 Return path analysis

Return path analysis guarantees that a function which is supposed to return something actually does so. This prevents executing code in which the result of a function call is undefined.

Return path analysis is performed immediately before we run the type checker on a function declaration. In the implementation, a trait named `ReturnPathAnalysis` is used and implemented for the AST nodes Statement and FunDecl. This trait provides just a single function: `check_returns`, which returns information about the return type of the function. This information is one of four variants:

1. `NoneExplicit`: the function returns void on all branches.
2. `NoneImplicit`: the function has no return statements, implicitly returning void. A `return void` will be inserted at the end of the statements of the AST node.
3. `All`: the function did return something on all branches. We are not interested in the type, as this will be inferred directly after this analysis by the type checker.
4. `Incomplete`: the function did not return something on all branches. This is the case when for example the true branch of an if-statement did return a value, and the false branch did not (like in the example above), an no other returns occur inside the function body.

The implementation of check_returns for a FunDecl AST node is made by a fairly simple traversal of the statements inside the function body, the most interesting part is given in listing 3.6. The implementation for a vector of statements simply loops over all statements, until a complete return (All or NoneExplicit) is encountered after which it immediately returns. Otherwise the function returns NoneImplicit or Incomplete, depending on the results of the recursive calls.

```
fn check_returns(&self) → Returning {
        match self {
            Statement::If(i) => match (i.if_true.check_returns(), i.if_false.check_returns()) {
                (Returning::NoneImplicit, Returning::NoneImplicit) => Returning::NoneImplicit,
                (Returning::NoneImplicit, _) => Returning::Incomplete,
                (_, Returning::NoneImplicit) => Returning::Incomplete,
                (Returning::Incomplete, _) => Returning::Incomplete,
                (_, Returning::Incomplete) => Returning::Incomplete,
                (Returning::NoneExplicit, Returning::NoneExplicit) => Returning::NoneExplicit,
                (Returning::All, Returning::All) => Returning::All,

                // Here we have a type conflict, this will become a proper error from the
                // type inferencer in the next analysis.
                (Returning::All, Returning::NoneExplicit) => Returning::All,
                (Returning::NoneExplicit, Returning::All) => Returning::All,
            },

            Statement::While(_w) => Returning::NoneImplicit,
            Statement::Assign(_a) => Returning::NoneImplicit,
            Statement::FunCall(_f) => Returning::NoneImplicit,
            Statement::Return(e) => match e {
                Some(_) => Returning::All,
                None => Returning::NoneExplicit,
            },
        }
    }
```

Listing 3.6: Return path analysis

3.3 Type checking

This section elaborates on the implementation of type checking/inference.

3.3.1 AST modifications and decorations

The AST that was produced in the previous phase of the compiler contained different data types for function and variable types. In order to simplify the type checking analysis, these types were merged into a single datatype given in listing 3.7. The parser is modified accordingly, to prevent `Void` to be a valid type for a variable or function parameter.

```
pub enum Type {
    Int,
    Bool,
    Char,
    Void,
    Function(Vec<Type>, Box<Type>),
    Tuple(Box<Type>, Box<Type>),
    List(Box<Type>),
    Var(Id),
}
```

Listing 3.7: SPL types

Types for functions and variables make use of Rust's *Option* type (similar to *Maybe* in Haskell). Whenever a type is inferred, the AST is updated with *Some(inferred type)* (if no type was specified by the programmer in the code), or the unification of the programmer-specified type and the inferred type applied on the specified type. The latter case does not use normal unification, but a special unification function that checks whether the programmer-specified type is not *more general* than the inferred type. This is required, because in the case that the programmer specified for example :: $a \to a$ as function type and the type checker inferred :: $Int \to Int$, the programmer specified type would be substituted without any complaint. This now results in a compilation error.

The AST nodes are not decorated with source information. Although this is a nice-to-have in order to give errors including the location in the source code, we had to prioritize working on other parts of the compiler due to time constraints.

3.3.2 Hindley-Milner: Polymorphic type inference

The compiler implements polymorphic type inference, closely following both the lecture slides and the implementation by Martin Gradmüller [1] (W-algorithm). The core of the implementation is made up by two traits: `TypeInstance` and `TypeInference`. The `TypeInstance` trait provides the two functions `ftv` and `apply`, as given in listing 3.8. This trait is implemented for `Type`, `TypeScheme` and `TypeEnv`, allowing to determine the free type variables and to apply substitutions to these datatypes. The `TypeInference` trait provides only a single function `infer` and is implemented for most AST nodes, as given in listing 3.9.

```
1 trait TypeInstance {
2     // Determines the free type variables of a type.
3     fn ftv(&self) → HashSet<TypeVar>;
4
5     // Apply a substitution.
6     fn apply(&self, subst: &Subst) → Self;
7 }
```

Listing 3.8: TypeInstance trait

```
1 trait TypeInference {
2     fn infer(
3         &mut self,
4         context: &Context,
5         expected: &Type,
6         generator: &mut VarGenerator,
7     ) → Result<Subst, String>;
8 }
```

Listing 3.9: TypeInference trait

The `infer` function implements the presented function M from the lecture slides. The implementations make a clear distinction between inferring the type of statements (for which we normally don't use the expected type) and expressions (for which we unify the inferred and expected type).

Running the complete type inference process is done by creating an empty context in which the builtin functions `print`, `isEmpty` and field operators are loaded, reordering the AST and then calling infer on the (mutable) root node of the AST. We then get an empty result (for success) in which the passed AST is completely typed, or an error.

Variable restriction

As presented in the lectures, a program such as the one in listing 3.10 would successfully pass the type checker if we blindly follow the generalization rule for let expressions. Since SPL has only the empty list as a polymorph value, we can use *variable restriction* (not generalizing variables) so that the type checker does not accept this problematic program.

```
1 problematic() {
2     var x = [];     // Context: Aa. x: [a], Values: x = []
3     x = 3 : x;      // Context: Aa. x: [a], Values: x = [3]
4     x = True : x;   // Context: Aa. x: [a], Values: x = [True, 3]
5 }
```

Listing 3.10: Polymorph values

Void restriction

Void can be seen as a special type. It is only allowed to be used in return types of functions. This introduces some problems, for example, an expression such as *print(1) == print(2)* effectively compares two voids and should thus not be allowed. Similarly, assigning void to a variable by for example the expression *var x = print(1);* should not be allowed. These examples have one commonality, namely that they use *function calls in expressions* to create a void expression. To disallow such problematic expressions, using function calls that return void are disallowed in expressions by the implemented compiler. As this rule does not apply for function calls in statements, writing *print(1);* in a function body is still perfectly fine for the compiler.

Typing variables with fields

All field operators (`hd, tl, fst` and `snd`) are treated as builtin unary functions. This means that whenever we want to type an expression such as *x.hd.fst.tl*, we simply convert this expression to an equivalent sequence of function calls whose type we want to infer. In the case of the above example, the we return the type of the expression *tl(fst(hd(x)))*. Applying standard unification and substitution gives the type $[([a], b]) \rightarrow [a]$ of the composed functions, and thus after applying it on x the expression has type $[a]$ (a possibly substituted for its type in x).

3.3.3 Mutual recursion

As explained in section 3.1.1, the declaration order of functions and variables does not have to be the same as the order in which they are used. A new problem arises when functions are defined in terms of each other: mutual recursion. A small example is given in listing 3.11: function foo calls function bar and vice versa. When we apply the type inference algorithm as normal, the function identifier bar will be unknown when we try to type foo, and an error is returned.

```
1 foo() {
2     return bar();
3 }
4 bar() {
5     var res = foo();
6     return res + 1;
7 }
```

Listing 3.11: Mutual recursion example

A solution to this problem, as was presented during the lectures on typing, is to type these functions together. Moreover, we can determine which groups of functions have to be typed

together by performing strongly connected component analysis on the call graph. The presented Tarjan algorithm[2] does exactly that, and also performs topological sorting as a byproduct which solves the problems described in section 3.1.1.

The Petgraph[1] crate is used to construct the call graph, and also provides an implementation of Tarjans algorithm. When we have to type the function declarations, we iterate over the components returned by the algorithm. The implementation follows the slides on mutual recursion. First, a new context is created into which all functions of the component are given a fresh type variable. Then, we run the type inference algorithm on each function declaration of the component, while continuously applying and composing the resulting substitutions. Finally, all functions from the component are generalized.

[1] `https://docs.rs/crate/petgraph/latest`

Chapter 4

Code Generation

The compiler's backend is made up by the code generator. A code generator transforms the intermediate code representation to another language. In the case of the SPL compiler the desired target language is the instruction set of the Simple Stack Machine (SSM), hence the compiler transforms the Abstract Syntax Tree (AST) which was created and decorated in the previous phases of the compiler to executable SSM instructions. This includes instructions for the builtin functions *print* and *isEmpty*. The code generator must also deal with overloading: the builtin function *print* and (in)equality operator are both overloaded. Calling these functions with arguments that have different types results in different instructions.

4.1 Data representation

This section describes how data is stored inside the memory.

4.1.1 Global variables & LocationEnv

All global variables are stored on the heap, which happens at the start of a program. Only the result of an evaluated expression is stored, not the expression itself. Whenever we need a global variable in an expression, we need to know where on the heap such a variable was stored. To conveniently store this information, we need to know at which address the heap starts. The scratch register $R5$ is used for this purpose, at the start of the program we store the value of the heappointer in this register. Whenever we store a (global) variable on the heap, we simply increase a local offset and store identifiers together with that offset in an environment. Whenever we need the value of a variable, we simply load the value stored at the address stored in $R5$ plus the associated offset.

The environment mentioned above does not merely store location information for global variables, but also for locals and parameters. This means that we have to account for aliasing here: a key is not merely an identifier, but a pair of an identifier and a `Scope`. A Scope is a sum type with alternatives `Global`, `LocalArgs` and `Local`. Whenever we need to retrieve the location of a variable we match the scopes in reverse order of the above. The first match is returned, by which aliasing is preserved in the correct way. The location is also described by a sum type, as data can be stored either on the heap or the stack. Both variants have an associated integer value, which in the case of the heap alternative represents the offset relative to the start of the heap ($R5$) and in the case of the stack alternative represents the offset relative to the

markpointer. After generating instructions for a function, all `Local` and `LocalArgs` elements are removed from the environment.

4.1.2 Lists

Lists are stored on the heap. This is convenient, since we generally do not know the size of a list at compile time. A list is stored as a singly linked list. This is done by first loading all values on the stack, from left to right. The empty list is represented by the value 0. Then we store the values in pairs on the heap, by using the instruction `stmh 2` (this basically happens for every occurrence of the cons operator). As the `stmh` instruction pushes the heap address of the last value on the stack, the value pointed to can be considered the tail of the list.

An example for the list $1 : 2 : 3 : []$ is given in figure 4.1. The top value on the stack points towards the tail of the first node of the list, which contains the address of the next node: 0x07D3. Similarly, at the address of the next node, the address to its successor is stored: 0x07D1. Finally, we encounter a nullpointer at that location, indicating the empty list []. To access the value of a node instead of the pointer towards the next node, we simply use the offset -1. Note that even though all nodes are aligned directly next to each other in this example, this is no requirement.

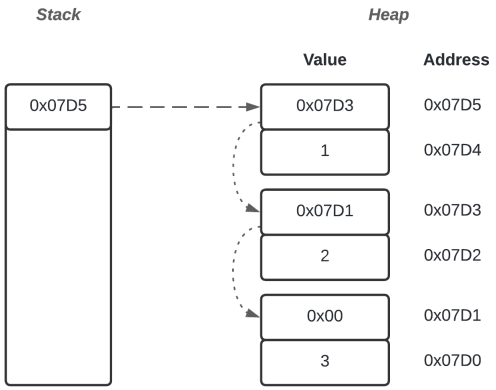

Figure 4.1: SSM memory layout of the list $(1 : 2 : 3 : [])$

4.1.3 Tuples

Like lists, tuples are stored on the heap. Similarly, a pointer that points to a tuple has the address of the second component of the tuple as value. When accessing the components of a tuple with fields, we simply use the `Lda` instruction with offset -1 (`fst`) or offset 0 (`snd`).

4.2 Calling semantics

The compiler implements a Java-like calling semantics. The basic values `Int`, `Bool` and `Char` are passed by value. This means that rather than passing the address of the location at which such a value is stored, the value itself is passed. A consequence of this is that a function which makes assignments to a basic value parameter only updates the value of the parameter, not the variable that was passed to the function. A small example that demonstrates these semantics is given in listing 4.1. The function `foo` takes an integer as argument, assigns the value 10 to it and then prints it. The `main` function defines a local variable x, initializes it to 5 and calls the function foo. Then the result is printed. The result of this program is: 10 5. If the original variable x was passed by reference rather than by value, we would have expected 10 10 to be printed.

```
1  foo(x) :: Int -> Void {
2      x = 10;
3      print(x);
4  }
5
6  main() :: -> Void {
7      var x = 5;
8      foo(x);
9
10     print(x);
11 }
```

Listing 4.1: Basic values are passed by value

The `List` and `Tuple` types are treated as Java objects: they are passed by reference. This means that assignments by *field access* can change the value of the original variable which was passed to the function. A small example that demonstrates these semantics is given in listing 4.2. The function `head_to_one` takes a list of integers as argument, and replaces the first value of the list by 1. The `main` function creates a singleton list with value 1, prints the head of that list, calls the function `head_to_one` and finally prints the head of that list again. The result of this program is: 0 1. If the original variable *list* was passed by value rather than by reference, we would have expected 0 0 to be printed.

```
1  head_to_one(list) :: [Int] -> Void {
2      list.hd = 1;
3  }
4
5  main() {
6      var list = 0 : [];
7      print(list.hd);
8
9      head_to_one(list);
10     print(list.hd);
11 }
```

Listing 4.2: Lists and tuples are passed by reference

4.3 Overloading & Polymorphism

Besides polymorphic functions, we also need to deal with overloaded functions. An overloaded function, like a polymorphic one, may take arguments of various types. However, opposed to a polymorphic one, different instructions will be generated depending on the types of the arguments. This requires several implementations and overloading resolving.

4.3.1 Disallow overloading

In the implemented compiler, user defined overloaded functions are not allowed. A "*Function previously declared*" error will be generated during semantic analysis. However, SPL comes with a builtin overloaded function: `print`. We have to take into account that polymorphic functions which call an overloaded function will also become overloaded, because we would not be able to resolve overloading at compile time otherwise. A small example is given in listing 4.3. Here, the polymorphic function `id_print` implicitly becomes an overloaded function, because it makes a call to `print` using the (generic type) argument a. The implemented compiler forbids this behavior, and generates the error "*Function id_print makes an illegal call to overloaded function 'print' which would make id_print overloaded.*" during code generation.

```
1  id_print(a) {
2      print(a);
3
4      return a;
5  }
6
7  main() {
8      var i = id_print(1);
9      var j = id_print(True);
10
11     return 1;
12 }
```

Listing 4.3: Overloading restriction

4.3.2 Print and Equality

Even though user defined overloading and calling overloaded functions from polymorphic ones are disallowed, we still have to implement overloading for the builtin function print and equality (==) operator. The `print` function implementation has support for all types, this is very convenient when verifying program results in SSM.

Whenever we encounter a call to print or (in)equality during code generation, we look at the argument type to determine what to generate. For the basic value types we can simply use the builtin instruction `trap`. Integers and booleans are printed using option 0 (integers) whereas characters are printed using option 1 (unicode). Tuples and lists are printed as $(1, a)$ and $[1, 2, 3]$, and we need to do a bit more work here. The procedure for printing a tuple with component types t_1, t_2 is described below. We make use of scratch register $R6$ to temporarily store the argument value, because we have to create a new stack frame in the procedure. A similar procedure is implemented for lists.

The `(in)equality` function implementation also has support for all types. The idea is similar to printing. For tuples we first generate the instructions to compare the first component. If the result of those instructions resulted in false, we omit comparing the second component, otherwise we return the result of comparing the second components. Lists are compared element-wise.

$PrintTuple(t1, t2)$

1. Store the argument a in scratch register $R6$
2. Create a new stack frame with space for 1 local
3. Load the value in $R6$ and store on the stack in local 1
4. Print '(' as unicode character
5. Load local 1 followed by `Lda(-1)`
6. Recursively call the print function generating SSM instructions with type $t1$
6. Print ',' as unicode character
7. Load local 1 followed by `Lda(0)`
8. Recursively call the print function generating SSM instructions with type $t2$
9. Print ')' as unicode character
10. Pop the stack frame

4.3.3 Polymorphishm

Polymorphic functions, such as $reverse :: [t] \rightarrow [t]$ put a special requirement on the code generator. Generating code for such functions must work for every type of argument. Due to the chosen data presentation, every possible value of all types occupies exactly one memory cell (in the case of basic values the value, in the case of lists and tuples a pointer to their location on the heap). This means that we don't have to deal with different instructions depending on the argument type.

4.4 Implementation details

In order to generate SSM instructions from the decorated Abstract Syntax Tree, the `SsmInstructions` trait was created and implemented for every AST node. This trait defines merely the `SsmInstructions` function whose type signature is given in listing 4.4. As arguments, it takes a location environment (which was previously described in section 4.1.1), an offset to the heap which contains the amount of cells we already occupied for storing data, and a prefix generator which is used to generate unique prefixes for labels. In order to convert the AST to a list of SSM instructions, we only need to call `instructions` on the root node of the AST. The result may be an error in the case of a failure during code generation (e.g. due to overloading, as described in section 4.3.1.

```
1 pub trait SsmInstructions {
2     fn instructions(
3         &self,
4         env: &mut LocationEnv,
5         heap_offset: &mut i32,
6         prefix_gen: &mut LabelPrefixGenerator,
7     ) -> Result<Vec<SsmInstruction>, String>;
8 }
```

Listing 4.4: SsmInstructions trait

Chapter 5

Extension

For my extension, I chose to compile SPL to C. The C programming language has many similaries to SPL, especially syntax-wise, but also some differences. Most notably, C does not have the builtin-types `List` and `Tuple` and no (real) support for polymorphic functions.

5.1 Composite type representation

As mentioned, C does not have the builtin datatypes `List` and `Tuple`. However, C does have support for structs so that the programmer can create composite datatypes. This section describes how composite types from SPL are represented and used in C.

5.1.1 Lists

Lists are, like SSM, implemented as a singly linked list. A list is represented as a sequence of nodes. The `node` struct, given in listing 5.1, has two members: `data` contains the value of the current node and is of type intptr_t. This guarantees that the size of the integer is the same as the size of a pointer, and thus allows safe casting. `tail` contains a pointer to the next node in the list. Finally, `NULL` represents the empty list.

```
1 struct node {
2     intptr_t data;
3     struct node* tail;
4 };
```

Listing 5.1: List type

5.1.2 Tuples

Similar to the List type we have implemented a struct for tuples, given in listing 5.2. Its two members represent the first and second components of the tuple. Again the intptr_t is used.

```
1 struct tuple {
2     intptr_t fst;
3     intptr_t snd;
4 };
```

Listing 5.2: Tuple type

5.1.3 Casting

In order to prevent warnings from the C-compiler, e.g. when using an intptr_t as a character, we have to do some casting. In fact, for nested composite types this is required. If the first component of a tuple represents another tuple whose components we want to access, we need to cast the intptr_t to a tuple pointer.

Consider the tuple `nested = (1, (3, 4));`. When we want to retrieve the 4 and store it in a variable, we could write the following SPL code: `var s = nested.snd.snd;`. Compiling this code to C would result in the following code `int s = (int) to_tuple_ptr(nested->snd)->snd;`.

5.2 Overloaded functions

Similar to code generation for SSM, as described in section 4.3.2, we need to deal with the overloaded functions `print` and `==`. Similar as for SSM, we thought it would be nice to implement these functions for all types. Instead of generating specialized instructions for each call, the C code contains two builtin functions `print` and `equals` which take as additional argument a flattened list of types. Based on that argument, the function recursively applies the desired operation.

5.2.1 Printing

Printing is implemented by the function `struct node* print(intptr_t value, struct node* types)`. The value argument is the value to be printed, and the types argument is a flattened list of the type of the value to be printed. Similar to SSM, lists are printed as `[1,2,3]` and tuples as `(1, 2)`.

5.2.2 Equality

Equality is implemented by the function `bool equals_(intptr_t v1, intptr_t v2, struct node** types)`. The v1 and v2 arguments are the two values to be compared with each other. The types argument is the list of types to be compared for equality. This is a pointer type that is being advanced by recursive calls to `print_` so that we know which types still have to be compared. Once all the types are processed or if we see that one of the basic types to be compared is not equal, we return.

5.3 Other differences

Some smaller differences are polymorphic functions and mutual recursion. C has support for both constructs in a way, but this is not promoted as an official language construct.

5.3.1 Polymorphic functions

For polymorphic functions from SPL we know that the contents of a polymorphic value are not touched. This means that we don't really care about the type of the argument passed to a polymorphic function. All type variables of a function type in SPL are simply substited by the intptr_t type in C. Whenever we call such a function we simply cast all arguments to an intptr_t type. If a function returns a value of a type variable type, we analyze the arguments to determine what the type has to be and cast the result. A small example of the polymorphic identity function is given in listing 5.1 (SPL) and listing 5.2 (the C translation).

```
1 id(x) :: a → a {
2     return x;
3 }
4
5 main() {
6     var i = id(1);
7 }
8
```

Figure 5.1: Polymorphism in SPL

```
1  intptr_t id(intptr_t x)
2  {
3      return x;
4  }
5
6  void main()
7  {
8      int i = (int)id((intptr_t)1);
9  }
10
```

Figure 5.2: Polymorphism in C

5.3.2 Mutual recursion

In C, definitions and declarations of functions may occur in different places. A function declaration lets the compiler know the name and type of the function, whereas the definition provides the actual semantics of the function. In C, we can't use variables and functions before they are declared. To allow mutually recursive functions, we can simply declare all functions before giving their definition. The example from listing 5.3 would compile to the piece of C code given in listing 5.4.

```
1  odd (x) {
2      if (x==0) {
3          return (False);
4      }
5      else {
6          return even (x - 1);
7      }
8  }
9  even (x) {
10     if (x==0) {
11         return (True);
12     }
13     else {
14         return (odd (x - 1));
15     }
16 }
17
```

Figure 5.3: Mutrec in SPL

```
1  bool odd(int x);
2  bool even(int x);
3
4  bool odd (int x) {
5      if (x == 0) {
6          return (false);
7      }
8      else {
9          return even (x - 1);
10     }
11 }
12 bool even (int x) {
13     if (x == 0) {
14         return (true);
15     }
16     else {
17         return (odd (x - 1));
18     }
19 }
20
```

Figure 5.4: Mutrec in C

Chapter 6

Conclusion

6.1 Functionality

This report described the implementation details and design choices for an SPL-compiler written in Rust. The implemented compiler has support for the standard SPL programming language and provides some nice additional features. Especially the support for localized parser errors, polymorphic type inference, mutual recursion and printing/equivalence for all types distinguishes this compiler from more basic-SPL compilers. The extension allows for compiling SPL to C. Although this is likely not the most difficult extension, it is nice to be able to compile SPL to (real) machine code using the C-compiler.

6.2 Improvements

Although not prescribed by the language, the functionality of the compiler could still be extended of course. Support for higher order functions, user-defined overloaded functions and garbage collection would be nice in particular. These functionalities could be considered extensions on their own however, and thus not trivial to add in the limited amount of time available.

Another thing that would be nice to have is localized errors from the semantic analysers and code generators. In the current implementation error messages from these components do not refer to the source code from which they are generated. Annotated AST nodes with location information would be required to implement this.

6.3 Implementation

A fairly decent amount of the code is unittested, but likely there will still be some bugs present. The code is very modular. The compiler can be run in four modes:

1. `print`: run the lexer, parser and pretty print the result to `out.spl`
2. `type`: similar to print, but runs the type checker and other semantic analyses before printing
3. `ssm`: compile to SSM and print the instructions to `out.ssm`
4. `c`: compile to C and print the code to `out.c`

Chapter 7

Reflection

7.1 The project

Overall I really enjoyed this project. Personally, I had no experience with compilers before I started this course and I wasn't sure what to expect. The fact that we were allowed to implement the compiler in any programming language is a big pro of this course. Instead of bothering with learning new language constructs, this allowed me to focus mainly on understanding the lecture contents and thinking about implementation possibilities.

From all the different stages of the compiler I learned something new. In the frontend phase, I used parser combinators for the first time. Although the lectures were a bit difficult to understand for me (due to the dominance of Haskell for examples), I learned quite a bit from this. The type inference algorithm can be considered the most complex but most interesting component of the compiler in my opinion. I was struggling quite a bit to get the type checker to work, and encountered many bugs. Usually, I would have forgotten an *apply* onto a context or type. Finally, I would consider the SSM backend a nice application and instruction set to use for introducing a code generator. After understanding all the instructions I was able to implement the code generator fairly quick, even though I knew little about machine instructions prior to this course.

7.2 How did it work out

In general, I am quite satisfied with the compiler I implemented. I am pretty confident that the compiler works like it's supposed to in the majority of the cases (maybe some edge cases that won't work) which is acceptable quality for a student project in my opinion.

That my partner decided to stop after a few weeks was a bit of a setback. I had to sacrifice some of my personal wishes due to time constraints. For example, I wanted to keep the amount of code being unittested as high as possible, but had to settle for fewer tests at the end. I also would have liked to work on an extension a bit more exciting than compiling to C, but I was not confident I could manage to implement this in the little remaining time left.

7.3 Pitfalls

Especially the type inferencer is something to get lost in easily. Most literature, including the lecture slides, are very notation-heavy and difficult to understand. Not until after I managed to implement the algorithm I could fully understand them.

Bibliography

[1] Grabmüller, M. Algorithm w step by step, 2006.

[2] Tarjan, R. Depth-first search and linear graph algorithms. *SIAM journal on computing 1*, 2 (1972), 146–160.

Appendix A

Grammar

```
SPL       = Decl+
Decl      = VarDecl
          | FunDecl
VarDecl   = ('var' | Type) id  '=' Exp ';'
FunDecl   = id '(' [ FArgs ] ')' [ '::' FunType ] '{' VarDecl* Stmt+ '}'
RetType   = Type
          | 'Void'
FunType   = [ FTypes ] '->' RetType
FTypes    = Type [ FTypes ]
Type      = BasicType
          | '(' Type ',' Type ')'
          | '[' Type ']'
          | id
BasicType = 'Int'
          | 'Bool'
          | 'Char'
FArgs     = id [ ',' FArgs ]
Stmt      = 'if' '(' Exp ')' '{' Stmt* '}' [ 'else' '{' Stmt* '}' ]
          | 'while' '(' Exp ')' '{' Stmt* '}'
          | id [ Field ] '=' Exp ';'
          | FunCall ';'
          | 'return' [ Exp ] ';'
Exp       = Disjunc
Disjunc   = Conjunc ( '||' Conjunc )*
Conjun    = Compare ( '&&' Compare )*
Compare   = Concat ( ( '==' | '!=' | '<' | '<=' | '>' | '>=' ) Concat )*
Concat    = Term ( ':' Term )*
Term      = Factor ( ( '+' | '-' ) Factor )*
Factor    = Unary ( ( '*' | '/' | '%' ) Unary )*
Unary     = ( '-' | '!' ) Unary
          | Atom
Atom      = id [ Field ]
          | int | char | string | 'True' | 'False'
          | '(' Exp ')'
```

```
          | FunCall
          | '[]'
          | '(' Exp ',' Exp ')'
Field     = ( '.' 'hd' | '.' 'tl' | '.' 'fst' | '.' 'snd' ) [ Field ]
FunCall   = id '(' [ ActArgs ] ')'
ActArgs   = Exp [ ',' ActArgs ]
int       = digit+
id        = alpha ( '_' | alphaNum)*
```

- `char` is a ', followed by a char escape sequence or any Unicode character besides ' and \, followed by '.
- a char escape sequence is a \ followed by ', \, `n`, `r` or `t`.
- `string` is a ", followed by zero or more string escape sequences or any Unicode character besides " and \, followed by ".
- a string escape sequence is a \ followed by ", \, `n`, `r` or `t`.