An SPL compiler in Rust

Ties Klappe
s1030293

Koen Bolhuis
s1085616

March 6, 2022

Contents

Chapter 1

Introduction

In this report we introduce our compiler for a programming language called SPL, implemented in Rust. This first chapter gives our 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, we discuss code generation. In Chapter 5 we 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, neither student was comfortable enough writing Haskell code for a relatively complex application such as a compiler. Rust was chosen because both of us had 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.

```
isVowel(c) :: Char -> Bool
{
    if ( c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u' ) {
        return True;
    }

    return False;
}

main() :: -> Void
{
    // True
    print(isVowel('a'));

    // False
    print(isVowel('b'));
}
```

Listing 1.1: Vowels example

```
// Compute the sum of the first n natural numbers.
sum(n) :: Int -> Int
{
    var total = 0;

    while(n > 0) {
        total = total + n;

        n = n - 1;
    }

    return total;
}

main() :: -> Void
{
    // 15
    print(sum(5));

    // 5050
    print(sum(100));
}
```

Listing 1.2: Natural numbers example

Chapter 2

Lexing & Parsing

Our 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 we have a sizeable set of unit tests for both components, which are run using continuous integration in our Git repository.

In this chapter, we describe our scanner, modifications to the provided grammar, our parser and the challenges encountered while developing the frontend of our 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 our modifications to it. The grammar on which our 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. We use 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

- In `FunDecl`, we have removed the requirement for variable declarations to be at the start of the function's body. Instead, `Stmt` now has a `VarDecl` alternative, allowing arbitrary variable declarations throughout the function.

- `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 our grammar, since it tends to make code more readable, and can even prevent difficult-to-diagnose bugs.[2]

```
if (condition1)
  if (condition2)
    statement1;
  else
    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:

```
void fun()
{
  T (x);
}
```

Listing 2.2: Function call or cast?

```
void fun()
{
  T * x;
}
```

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 our 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), our abstract syntax tree structure is built based on the grammar. For convenience and in order to flatten the structure, our AST slightly deviates from the grammar in several places:

- `FArgs` is directly included in our `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:

```
enum Expr {
    Add(Expr, Expr),
    ...
}
```

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.

2.4 Parser

For parsing SPL files, we employ a top-down approach. This is achieved using the `nom`[3] 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.[4]

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 our 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://lib.rs/crates/nom`

[4] `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`, ie. 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 (`:`), our 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`[5] 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: $1 + 2 * 3$.

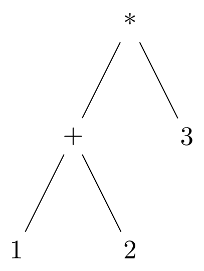

Figure 2.1: Parentheses required

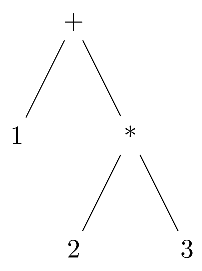

Figure 2.2: No parentheses required

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.

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

Chapter 3

Analyses & Typing

Chapter 4

Code Generation

Chapter 5

Extension

Chapter 6

Conclusion

Appendix A

Grammar

```
SPL       = Decl+
Decl      = VarDecl
          | FunDecl
VarDecl   = ('var' | Type) id  '=' Exp ';'
FunDecl   = id '(' [ FArgs ] ')' [ '::' FunType ] '{' 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* '}'
          | VarDecl
          | 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`.