Assignment 1: Lexical analyses

Compiler Construction
Radboud University

1 Background

Simple Programming Language (SPL) is a strict first order programming language similar to C but with a polymorphic type system and some built-in overloaded functions.

This first assignment of the project consists of the first phase of the compile: lexical analyses. The input for the compiler is a plain-text file containing a single SPL program. The output is the pretty-printed abstract syntax tree shown as a concrete syntax.

2 Learning Outcome

Create a lexer, parser and optionally a pretty printer for a given language specification. If the specification is incomplete, make and document the well-founded design decisions about the changes.

3 Instructions

3.1 Parser

Implement a parser for SPL's concrete syntax. The given grammar may contain ambiguities or other unwanted properties and needs changing. Think carefully about the design of the abstract syntax tree, since this is the foundation of your compiler. A proper design will make your life easier in the upcoming assignments.

You are free to use support libraries (e.g. parser combinators) provided by the language in which you write your compiler as long as you are able to explain the underlying mechanisms in the report and the oral exam.

3.2 Printer (optional)

Implement a pretty printer for the SPL abstract syntax tree. Printing a parsed program should produce an equivalent program that is accepted by the parser again. Differences between the original file and the produced file typically concern layout, comments, and parentheses.

4 Deliverables

Presentation A third of the groups presents this assignment. Which groups have to present this assignment is announced on Brightspace.

Property	Insufficient	Sufficient	Good	Optional
Design	No motivated design decisions.	Motivated design decisions.		
Errors	No error handling.	Barely any error handling. Bail out on the first error.	Line and column based error handling. Acumulate errors.	Reparations and/or suggestions.
Lexer	No motivated design decision.	Argumentation on why a separate lexer or a single lexer-parser was chosen.		
Parser	Ad hoc unfounded expression parser with errors or the grammar does not match the parser.	Embedded fixity and binding strength in the grammar or use a different expression parsing technique. Proper treatment of non-associative infix operators.	Well founded design decision about expression parsing. Addressed edge cases such as dangling else, lexer hack.	Nested comments, a pretty printer.
General	Non-working code.	Working code with a good I/O interface.	Well structured modular code.	Use of CI and issue tracker, other useful extensions regarding this phase.

Table 1: Assignment 1 rubric

In the presentation of this phase you tell the other students how your scanner, parser and printer are constructed and demonstrate your parser and pretty printer. We all know the language, so you should just mention things specific to your parser, like implementation language used, changes to the grammar, and other interesting points.

Report sections Furthermore, finish the up to and including the second chapter of the report[1]. The provided skeleton contains example questions that you can answer but make sure the report is self contained.

5 Discussion

During the course you make your own implementation of a complete SPL compiler accompanied by a report that will be tried on the oral exam. Each phase on its own is not graded but only as part of the total project. Comments are provided on the report sections that were handed for guidance and affirmation.

Criteria for this phase can be found in Table 1.

6 Simple Programming Language

6.1 Grammar

Comments on a single line start with `//` and end with the first newline character. Comments on multiple lines are delimited by `/*` and `*/`. A grammar can be found in Table 2.

6.2 Semantics

The syntax of SPL is very similar to C and the keywords need no introduction. SPL is strict, i.e. function arguments are always evaluated. Since there are side effects, the evaluation order of

[1]The provided skeleton for the report is mandatory and can be found here: `https://gitlab.science.ru.nl/compilerconstruction/material`

```
SPL       = Decl+
Decl      = VarDecl
          | FunDecl
VarDecl   = ('var' | Type) id  '=' FExp ';'
FunDecl   = id '(' [ FArgs ] ')' [ ':' RetType ] '{' VarDecl* Stmt+ '}'
RetType   = Type
          | 'Void'
Type      = BasicType
          | '[' Type ']'
          | id
BasicType = 'Int'
          | 'Bool'
          | 'Char'
FArgs     = [ FArgs ',' ] id [ ':' Type ]
Stmt      = 'if' '(' FExp ')' '{' Stmt* '}' [ 'else' '{' Stmt* '}' ]
          | 'while' '(' FExp ')' '{' Stmt* '}'
          | id [ Field ] '=' FExp ';'
          | FunCall ';'
          | 'return' [ FExp ] ';'
FExp      = Exp [ Field ]
Exp       = id
          | FExp Op2 FExp
          | Op1 FExp
          | int
          | char
          | 'False' | 'True'
          | '(' FExp ')'
          | FunCall
          | '[]'
Field     = '.' 'hd' | '.' 'tl'
FunCall   = id '(' [ ActArgs ] ')'
ActArgs   = FExp [ ',' ActArgs ]
Op2       = '+'  | '-' | '*' | '/'  | '%'
          | '==' | '<' | '>' | '<=' | '>=' | '!='
          | '&&' | '||'
          | ':'
Op1       = '!'  | '-'
int       = [ '-' ] digit+
id        = alpha ( '_' | alphaNum)*
```

Table 2: SPL grammar.

Function	Type	Description
`+`	$\mathbf{Int} \times \mathbf{Int} \to \mathbf{Int}$	Adds the arguments.
`-`	$\mathbf{Int} \times \mathbf{Int} \to \mathbf{Int}$	Subtracts the arguments.
`*`	$\mathbf{Int} \times \mathbf{Int} \to \mathbf{Int}$	Multiplies the arguments.
`/`	$\mathbf{Int} \times \mathbf{Int} \to \mathbf{Int}$	Divides the arguments.
`%`	$\mathbf{Int} \times \mathbf{Int} \to \mathbf{Int}$	Takes the modulus of the arguments.
`-`	$\mathbf{Int} \to \mathbf{Int}$	Negates the number.
`==`	$t \times t \to \mathbf{Bool}$	Checks if the arguments are equal.
`!=`	$t \times t \to \mathbf{Bool}$	Checks if the arguments are not equal.
`<`	$t \times t \to \mathbf{Bool} \mid t \in \{\mathbf{Int}, \mathbf{Char}\}$	Checks the lesser than relation.
`>`	$t \times t \to \mathbf{Bool} \mid t \in \{\mathbf{Int}, \mathbf{Char}\}$	Checks the greater than relation.
`<=`	$t \times t \to \mathbf{Bool} \mid t \in \{\mathbf{Int}, \mathbf{Char}\}$	Checks the lesser or equal relation.
`<=`	$t \times t \to \mathbf{Bool} \mid t \in \{\mathbf{Int}, \mathbf{Char}\}$	Checks the greater or equal relation.
`&&`	$\mathbf{Bool} \times \mathbf{Bool} \to \mathbf{Bool}$	Takes the conjunction of the arguments.
`||`	$\mathbf{Bool} \times \mathbf{Bool} \to \mathbf{Bool}$	Takes the disjunction of the arguments.
`!`	$\mathbf{Bool} \to \mathbf{Bool}$	Negates the truth value.
`hd`	$[t] \to t$	Takes the head of a list.
`tl`	$[t] \to [t]$	Takes the tail of a list.
`:`	$t \times [t] \to [t]$	Constructs a list.
`print`	$t \to \mathbf{Void}$	Prints the argument.
`isEmpty`	$[t] \to \mathbf{Bool}$	Returns whether the list is empty.
`[]`	$[t]$	Constructs the empty list.

Table 3: SPLs built-in functions and operators.

function arguments and the arguments of operators can influence the result of a program. You are free to make any documented and well-motivated choice that fits your implementation. SPL contains some built-in overloaded functions that either work on some types or on all types. Furthermore, SPL contains some polymorphic functions. The types and semantics of the operators and built-in functions can be seen in Table 3.

7 SPL Example Code

Test programs can be found in the materials repository[2]. All participants of the course can submit *Merge Requests* for this repository to add interesting test programs or other material.

[2] `https://gitlab.science.ru.nl/compilerconstruction/material`