Advanced Programming

Assignment 2: Subs Parser

Per Steffen Czolbe, Konrad Gnoinski

Handed in: September 26, 2017

Contents

Description of the Assignment

This assignment is about implementing a Parser for a conservative subset of Mozilla's JavaScript implementation, including list comprehensions, which we will call SubScript. The Parser reads a file written in the SubScript syntax and parses it to a abstract syntax tree.

Solution

This section gives a brief overview about the structure of the solution and how to execute the code.

Implementation

This solution of the assignment is implemented in Haskell.

Source-code Organisation

The code is organised in multiple files:

- `src/Subs.hs`: the file containing the main function of the programm: Parsing a file.

- `src/SubsPArser.hs`: the module exporting the parser functions.
- `src/Parser/Impl.hs`: a file containing the implementation of the Parser.
- `src/Parser/SubsAst.hs`: a file containing the abstarct syntax tree.
- `src/Parser/Test/SimpleTests.hs`: a file containing static tests of the interpreter functionality.
- `src/Parser/Test/quickCheckTests.hs`: a file containing tests with expressions randomised by quickcheck.

Test Cases

The `src/Parser/Test` directory contains multiple tests. The test framework used is `HSpec`. For property based testing, `QuickCheck` is used.

How to Execute the Program

The program can be compiled using the `src/Subs.hs` file as the primary component. The compiled program can be executed by passing a file containing the SubScript code as an argument:

```
$ gci Subs.hs
$ ./Subs <filename>
```

Alternatively, the `SubsParser` module can be loaded into the `ghci`. This way the function `parseString` can be called with a string representing the SubScript programm. The interpreted expression is returned.

The tests can be run similar to the mentioned ways, just using the files or modules from the `src/Parser/Test` directory instead.

Table 1: Comparison of Parser Combinator Libraries for Haskell

ReadP	Parsec
No error messages	Precise error messages
List based, backtracking	No backtracking by default

Implementation

A description of the implementation.

Assumptions

We assume the parser should be implemented according to the principles and code constructs shown in the lecture. This might not be the most runtime efficient or source-code compact solution, but helps greatly to understand the basics of parser combinators.

Choice of Parsing Combinator Library

There are two parser combinator libraries to choose from: ReadP and Parsec. Table 1 gives a short overview of the benefits of each of them.

Based on this selection, we chose Parsec to implement this solution. The precise error messages have proven to be invaluable during the implementation and testing of the parser.

Use of Backtracking

Backtracking parsers pose the risk of space leaks [Leijen, 2001]. Due to this, parsec does not offer backtracking by default. Instead, the user has to account for possible areas of backtracking explicitly and mark then with parsecs `try`-statement.

To avoid backtracking as much as possible, the grammar used was left-factored. This removes common factors from the left of expressions, and thus reduces the amount of backtracking required.

However, there are still some instances that mandate backtracking due to the similarity of keywords and identities. Both are strings which are not marked otherwise (such as 'quotationmarks' on strings in javascript), and have the potential to be quite similar. For example, `false` can get parsed as an ident, reading in the entire symbol. Only after it has been read entirely, the symbol can be evaluated. The evaluation fails, as `false` is an invalid ident. But to parse it as a constant later on, the parser needs to backtrack to a state before it started parsing the first letter of `false`. In this case, a `try` statement is necessary.

This solution uses `try` statements on parsing the ident, and all keywords that can appear in the same instances as ident (`true, false, undefined`).

Grammar

This is the grammar implemented in the parser. It was obtained by performing these steps on the original grammar:

- Establishing precedence of operators
- Establishing associativity of operators
- Elimination of left recursion
- Left factoring

$$
\begin{aligned}
Expr &::= Expr1\ Expr' \\
\\
Expr' &::= \text{`,`}\ Expr \\
&\quad | \ \epsilon \\
\\
Expr1 &::= Ident\ \text{`}=\text{`}\ Expr1 \\
&\quad | \ Expr2 \\
\\
Expr2 &::= Expr3\ Expr2' \\
\\
Expr2' &::= \text{`}===\text{`}\ Expr3\ Expr2' \\
&\quad | \ \text{`}<\text{`}\ Expr3\ Expr2' \\
&\quad | \ \epsilon \\
\\
Expr3 &::= Expr4\ Expr3opt \\
\\
Expr3opt &::= \text{`}+\text{`}\ Expr4\ Expr3opt \\
&\quad | \ \text{`}-\text{`}\ Expr4\ Expr3opt \\
&\quad | \ \epsilon \\
\\
Expr4 &::= ExprX\ Expr4opt \\
\\
Expr4opt &::= \text{`}*\text{`}\ Expr5\ Expr4opt \\
&\quad | \ \text{`}\%\text{`}\ Expr5\ Expr4opt \\
&\quad | \ \epsilon \\
\\
Expr5 &::= Number \\
&\quad | \ String \\
&\quad | \ \text{`}true\text{`} \\
&\quad | \ \text{`}false\text{`} \\
&\quad | \ \text{`}undefined\text{`} \\
&\quad | \ ExprIdent \\
&\quad | \ ExprBracet \\
&\quad | \ \text{`(`}\ Expr\ \text{`)`}
\end{aligned}
$$

$$
\begin{aligned}
ExprIdent &::= Ident\ ExprIdent' \\
ExprIdent' &::= \text{`(`} Exprs \text{`)`} \\
&\mid \epsilon \\
ExprBracet &::= \text{`[`} ExprBracet' \\
ExprBracet' &::= Exprs \text{`]`} \\
&\mid ArrayFor \text{`]`} \\
Exprs &::= Expr1\ CommaExprs \\
&\mid \epsilon \\
CommaExprs &::= \text{`,`} Expr1\ CommaExprs \\
&\mid \epsilon \\
ArrayFor &::= \text{`}for\text{` `(`}Ident \text{`}of\text{`} Expr1 \text{`)`} ArrayCompr \\
ArrayIf &::= \text{`}if\text{` `(`}Expr1 \text{`)`} \\
ArrayCompr &::= Expr1 \\
&\mid ArrayFor \\
&\mid ArrayIf \\
Number &::= \text{` - `} Digits \\
&\mid Digits \\
Digits &::= \\
Ident &::= \\
String &::=
\end{aligned}
$$

Eliminating Whitespace

Whitespaces can appear between any two tokens. Valid whitespaces are spaces, `\n` `\t` and comments.

This parser follows the strategy of reading all whitespace at the start of the file, and then reading it after each token. This is more runtime-efficient than doing it the other way around, as matching symbols is much quicker when the parser does not have to account for leading whitespaces in each case.

Support of SubScript

This parser supports the full functionallity of SubScript.

Assessment

This section contains the assessment of the presented solution.

Correctness of Solution

Based on the extensive tests, including property based tests performed with QuickCheck, we assume the solution is likely to be correct.
Additionally, the solutions passed the Online TA test.

Summary of Code Quality

Based on the extensive tests, we do believe that the functionality of this parser matches the functionality specified in the assignment.

The quality of the code could be improved by using more functions from the haskell standard library or the parsec library. However, we selected this more ”low level” implementation, as it is closer to the code examples from the lecture and more comprehensible.