Department of Information and Computing Sciences
Utrecht University

INFOB3TC – Assignment P1

Andres Löh

Deadline: Tuesday, 1 December 2009

The goal of this assignment is to write a parser (including a few so-called "semantic functions") for files in the *Portable Game Notation* (PGN) for chess games. See for instance Wikipedia at

```
http://en.wikipedia.org/wiki/Portable_Game_Notation
```

for an informal explanation of the format.

The PGN format is used to document the progress of one or more chess games in a file in a standardized format. Programs understanding the format can read and analyze the games, and for instance allow an interested user to replay the game.

The PGN format has two variants: the 'import' and 'export' format. The 'export' format is more strict and it should be understood by all readers, while the 'import' format is more flexible, so that it is easier to produce by hand. The specification of the formats is given at

```
http://www.very-best.de/pgn-spec.htm
```

and – compared at least to typical syntax specifications of programming languages – is rather vague in places.

We will therefore not strictly adhere to the PGN specification in this assignment, but implement our own variant, which should get us somewhere in between the import and export format, with the result that many PGN files available for download on the internet should be parseable by our implementation.

Note that it is not necessary to know chess in order to solve this assignment. No detailed knowledge about the rules is assumed, but if you really don't know anything about the game, you should probably read

```
http://en.wikipedia.org/wiki/Chess
```

to get familiar with the way the game board looks and the names of the pieces.

Parser combinators

For this task, you are supposed to use the parser combinators as discussed in the lectures. These are contained in a Haskell package called `uu-tc` which is available for download from the course Wiki.

There are two versions of the parser combinator library in that package. You can get the one which is as described in the lecture notes by saying `import ParseLib` or alternatively `import ParseLib.Simple`. In the lectures, I use a variant of that library that keeps the parser implementation abstract. This variant is available by saying `import ParseLib.Abstract`.

You can choose which variant you want to use, but I recommend `ParseLib.Abstract`.

General remarks

Here are a few remarks:

- Make sure your program compiles (with an installed `uu-tc` package). Verify that `ghc --make -O Chess.hs` works prior to submission. If it does not, your solution will not be graded.
- Include *useful* comments in your code. Do not paraphrase the code, but describe the structure of your program, special cases, preconditions etc.
- Try to write readable and idiomatic Haskell. Style influences the grade! The use of existing higher-order functions such as `map`, `foldr`, `filter`, `zip` – just to name a few – is explicitly encouraged. The use of all existing libraries is allowed (as long as the program still compiles with the above invocation).
- Copying solutions from the internet is not allowed. Teamwork in teams with a maximum of 2 members is allowed, but each team member has to submit individually and state the cooperation in a README file or at the top of the source file.
- Textual answers to tasks can either be included as comments in a source file, or be submitted as text or PDF files. Word documents are not accepted!

Moves

We will build the abstract syntax and the parser bottom-up. We will start with a single move, then extend everything to multiple complete games including metadata.

The concrete syntax of moves in so-called *Standard Algebraic Notation* (SAN) is given by the following grammar:

$$
\begin{array}{lll}
\textit{move} & ::= & \textit{piece}\ \textit{disambiguation}\ \textit{capture}\ \textit{square}\ \textit{promotion}\ \textit{check} \\
 & & \quad \mid \texttt{O-O-O} \mid \texttt{O-O} \\
\textit{piece} & ::= & \texttt{N} \mid \texttt{B} \mid \texttt{R} \mid \texttt{Q} \mid \texttt{K} \mid \varepsilon \\
\textit{disambiguation} & ::= & \textit{file} \mid \textit{rank} \mid \textit{square} \mid \varepsilon
\end{array}
$$

capture	::= `x` | ε
square	::= *file rank*
promotion	::= `=` *piece* | ε
check	::= `+` | `#` | ε
file	::= `a` | `b` | `c` | `d` | `e` | `f` | `g` | `h`
rank	::= `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8`

Terminals are written in typewriter font, nonterminals in italics. A regular move is a sequence indicating a piece to be moved, possible disambiguation between several pieces of the same category by giving as much information about the initial square of the move as necessary, an indication if the move is a capture, the target square of the move, information about a possible pawn promotion, and an annotation if the move results in check or checkmate.

There are two special moves, called *castling*, where the king and one of the rooks move at the same time. Queenside castling is indicated using `O-O-O` (composed of the letter `O`, not the number `0`, and dashes), kingside castling using `O-O`.

The piece that is moved is indicated using a single uppercase letter. Here, `N` stands for *knight*, `B` for *bishop*, `R` for *rook*, `Q` for *queen*, and `K` for *king*. If a *pawn* is moved, no letter is used.

In most cases, the rules of the game and/or the situation of the game board imply that giving the target square of the move uniquely determines the piece that is moved. If that is not the case, some information of the initial square of the move can be provided, by either giving the *file*, or the *rank*, or the complete *square*. The file is the 'column' in which a piece is located, indicated from left to right by a letter from `a` to `h`. The rank is the 'line' on which a piece is located, indicated from bottom to top by a number from `1` to `8`.

If a move captures an opponent's piece, this is indicated using an `x` between the possible disambiguation information and the target square.

If a pawn succeeds to move all across the board to the base rank of the opponent, it can be 'promoted' to another piece. This is indicated by the symbol `=` and the letter for the appropriate piece.

If a move results in check or checkmate, this is indicated by appending a `+` or a `#` symbol, respectively, to the move description.

No whitespace is allowed anywhere in a move!

1. Define Haskell datatypes to describe the abstract syntax of a move. Call the datatype for a move `Move` as indicated in the skeleton. Note that you will probably have to declare many datatypes, but you may also decide to represent certain syntactic categories using type synonyms if you prefer. Hint: it may be helpful to use `deriving (Show, Eq)` in your datatype declarations.

2 (medium)**.** Define a parser

```
parseMove :: Parser Char Move
```

that can parse a single SAN move. Again, this implies that you have to define parsers for all the other types you have introduced, too.

3. Define a partial function

```
run :: Parser a b -> [a] -> b
```

that applies the parser to the given input. Of all the results the parser returns, we are interested in the *first* result that is a *complete* parse, i.e., where the remaining list of input symbols is empty. If such a result exists, it is returned. Otherwise, `run` should produce an error using a call to the function `error`.

4. Define a printer

```
printMove :: Move -> String
```

that turns a move back into SAN notation. The idea is that for any value `m` of type `Move` we have that

```
(run parseMove . printMove) m == m
```

i.e., that printing the move and then parsing it again succeeds and results in the same abstract representation of the move. Similarly, for valid SAN strings `s` we should have that

```
(printMove . run parseMove) s == s
```

5. Test your parser for moves on a couple of examples, by parsing and printing example moves:

```
*Main> (printMove . run parseMove) "e4"
"e4"
*Main> (printMove . run parseMove) "Nf3xg5+"
"Nf3xg5+"
*Main> (printMove . run parseMove) "Nf3xp5+"
"*** Exception: parse error
*Main> (printMove . run parseMove) "d7xe8=Q"
"d7xe8=Q"
*Main> (printMove . run parseMove) "Qa8=N"
"Qa8=N"
```

To those knowledgable of chess, the last example demonstrates that not only legal chess moves are accepted by our grammar (and hence, by our parser): for instance, promotions are only allowed for pawns, and in the example, a queen is promoted. Promotions are also only allowed if a black pawn moves to rank 1 or a white pawn moves to rank 8.

6. Write a function

```
promotionCheck :: Move -> Bool
```

that verifies that a move does not contain an illegal promotion. In other words, for a non-promotion move the function should always return `True`, but if the move is a promotion, it should check that the target rank is either `1` or `8`, and that the piece that is moved is a pawn. Here are a few examples:

```
*Main> promotionCheck (run parseMove "d7xe8=Q")
True
*Main> promotionCheck (run parseMove "Qa8=N")
False
*Main> promotionCheck (run parseMove "O-O-O")
True
```

Games

The full concrete syntax of a PGN-file is as follows (closely following the specification):

pgn ::= *game pgn* | ε
game ::= *tags movetext*
tags ::= *tag tags* | ε
tag ::= `[` *name value* `]`
name ::= *symbol*
value ::= *string*
movetext ::= *elements termination*
elements ::= *element elements* | *variation elements* | ε
element ::= *number*? *move nag*?
variation ::= `(` *elements* `)`
termination ::= `1-0` | `0-1` | `1/2-1/2` | `*`
number ::= *natural dots*
dots ::= `.` *dots* | ε

Different nonterminals in the above list may be separated by arbitrary amounts of whitespace.

The lexical syntax is given by the following grammar:

symbol ::= (*letter* | *digit*) (*letter* | *digit* | `_` | `+` | `#` | `=` | `:` | `-` | `/`)*
string ::= `"` (*quoted* | *char*)* `"`
nag ::= `$` *digit*+
quoted ::= `\` *char* | `\\` | `\"`
letter ::= any letter (lower- or uppercase)
digit ::= any digit
char ::= any printable symbol except `\` or `"`

No additional whitespace is allowed in the lexical syntax.

Here is an informal explanation of the syntax: Every PGN file consists of a sequence of games. Every game consists of two sections: tags and moves. The tags are a list of name-value pairs, each surrounded by square brackets. The list of moves and thus the whole game is always ended by a game termination, which can be one of `1-0` (White has won), `0-1` (Black has won), `1/2-1/2` (a draw), or `*` (the game has been abandoned, is incomplete, or the result is unknown).

Each move is introduced by an optional number, which in turn may be followed by arbitrarily many dots. Next, the move is described in SAN. Finally, an optional so-called *Numeric Annotation Glyph* (NAG) follows, which is a number that indicates certain additional and often subjective information about a move (such as whether it was a good move or a bad move). The PGN format can also describe variations of a game (what-if scenarios). Such a variation is a sequence of move elements surrounded by (literal!) parentheses – please distinguish the literal parentheses `( )` in *variation* from the meta-parentheses ( ) in, for instance, *symbol*. The former are terminal symbols, the latter are used for grouping in the grammar description.

A symbol is introduced by a letter or digit, and then followed by an arbitrary number of letters, digits or specific symbolic characters. Symbols are used to describe the names of tags, but also numbers, SAN moves, and termination markers could be lexically classified as symbols.

A string is surrounded by double quotes `"`. If a double quote occurs in a string, it can be quoted using a backslash `\`. A backslash itself can be quoted using a double backslash `\\`. A backslash followed by any other character is interpreted as that character. Strings are used to describe the values of tags.

A NAG is introduced by a `$` symbol followed by arbitrarily many digits.

7. Define Haskell datatypes (or type synonyms) to describe the abstract syntax of a PGN file. Call the type for a whole PGN file `PGN`.

8 (difficult)**.** Define a parser

```
parsePGN :: Parser Char PGN
```

that can parse a complete PGN file. Note that you have a choice here. You can directly define the parser on strings, or you can write a lexical analyzer (scanner, lexer) that first transforms the input into a stream of tokens. While the latter is slightly more elegant (you don't have to care about whitespace in the second phase), it also is significantly more work.

9. Define a function

```
readPGN :: FilePath -> IO PGN
```

that uses `readFile` to read a file of the given name and then parses it. There is a small example file `Mate.pgn` on the Wiki that you should be able to parse using this function. There are many PGN files on the net, most of which should be readable. Pick one that does not have comments, because comments are not yet handled by the parser (you can add support for comments as a bonus exercise).

10. Define a printer

```
printPGN :: PGN -> String
```

that generates a string representation from an abstract PGN game. Try to make the layout of the generated string readable. For instance, put different tags on different lines. Put pairs of moves on different lines. Since the printer changes the layout, we will not in general have the property that parsing a file and printing it results in the original string again. However, the other direction should hold. For any value g of type `PGN`,

```
(run parsePGN . printPGN) g == g
```

Here's a possible (not the only valid) result of printing `Mate.pgn` – note the changes in layout:

```
[Game "Scholar's Mate"]

1. e4 e5
2. Bc4 Nc6
3. Qh5 Nf6
4. Qxf7#
1-0
[Game "Fool's Mate"]

1. f4 e6
2. g4 Qh4#
0-1
```

11 (medium)**.** Write a function (or several functions) that for a value of type `PGN` answers the following questions:

- how many games were played?
- how often did White win?
- how often did Black win?
- for each game, how many pieces did White and Black have at the end of the game? (Hint: each player starts with 16 pieces, and every capture move reduces the number of opponent's pieces by 1.)

Hint: You can choose whatever form of computation you find easiest. In particular, you can choose to define own datatypes, and whether you want to define one or multiple functions to collect the data. In any case, remember the standard recipe for defining functions on datatypes!

Bonus exercises

12 (bonus)**.** Handle comments in PGN format. Adapt your whitespace parser for this purpose.

13 (bonus, medium)**.** Usually, PGN describes complete games. Sometimes, especially for chess problems, only the end or a specific sequence of games is interesting. Therefore, a different format called FEN exists to quickly describe the situation of the board. An FEN situation for the 'beginning' of the game to be described can be given as a PGN tag. Implement a parser for FEN and extend your PGN parser to handle such tags.

14 (bonus, medium)**.** Write a viewer that can visualize a FEN description of a board, either in ASCII graphics or using a GUI.

15 (bonus, difficult)**.** Write a tracker that can track the positions of all the pieces throughout a description of a game in PGN format. Write a printer that can print the current positions of all pieces in FEN format. A side effect of a tracker is that you can discover illegal moves and report them.

16 (bonus)**.** You now have everything to be able to replay recorded chess games. You can generate a sequence of FEN descriptions from a PGN file, and draw them using your viewer.

17 (bonus, difficult)**.** Extend your viewer with features to navigate through the game tree. In order to do this, do not just generate a sequence of FEN descriptions, but a tree (for variations of the game). You may want to step forward, backward, or sideways (in and out of variations) in the tree. You can even allow to edit the current situation or change the game.