The Language Grammar

BNF-converter

April 3, 2023

This document was automatically generated by the *BNF-Converter*. It was generated together with the lexer, the parser, and the abstract syntax module, which guarantees that the document matches with the implementation of the language (provided no hand-hacking has taken place).

The lexical structure of Grammar

Literals

Character literals $\langle \mathit{Char} \rangle$ have the form `'c'`, where c is any single character.

Integer literals $\langle \mathit{Int} \rangle$ are nonempty sequences of digits.

UIdent literals are recognized by the regular expression $\langle \mathit{upper} \rangle$(‘_’ | $\langle \mathit{digit} \rangle$ | $\langle \mathit{letter} \rangle$)$*$

LIdent literals are recognized by the regular expression $\langle \mathit{lower} \rangle$(‘_’ | $\langle \mathit{digit} \rangle$ | $\langle \mathit{letter} \rangle$)$*$

Reserved words and symbols

The set of reserved words is the set of terminals appearing in the grammar. Those reserved words that consist of non-letter characters are called symbols, and they are treated in a different way from those that are similar to identifiers. The lexer follows rules familiar from languages like Haskell, C, and Java, including longest match and spacing conventions.

The reserved words used in Grammar are the following:

```
case   data  forall
in     let   of
where
```

The symbols used in Grammar are the following:

`:`	`=`	`(`
`)`	`->`	`.`
`{`	`}`	`+`
`\`	`=>`	`_`
`;`		

Comments

Single-line comments begin with `--`.
Multiple-line comments are enclosed with `{-` and `-}`.

The syntactic structure of Grammar

Non-terminals are enclosed between ⟨ and ⟩. The symbols ::= (production), | (union) and ϵ (empty rule) belong to the BNF notation. All other symbols are terminals.

⟨*Program*⟩ ::= ⟨*ListDef*⟩

⟨*Def*⟩ ::= ⟨*Bind*⟩
  | ⟨*Sig*⟩
  | ⟨*Data*⟩

⟨*Sig*⟩ ::= ⟨*LIdent*⟩ `:` ⟨*Type*⟩

⟨*Bind*⟩ ::= ⟨*LIdent*⟩ ⟨*ListLIdent*⟩ `=` ⟨*Exp*⟩

⟨*Type1*⟩ ::= ⟨*UIdent*⟩
  | ⟨*TVar*⟩
  | ⟨*UIdent*⟩ `(` ⟨*ListType*⟩ `)`
  | `(` ⟨*Type*⟩ `)`

⟨*Type*⟩ ::= ⟨*Type1*⟩ `->` ⟨*Type*⟩
  | `forall` ⟨*TVar*⟩ `.` ⟨*Type*⟩
  | ⟨*Type1*⟩

⟨*TVar*⟩ ::= ⟨*LIdent*⟩

⟨*Data*⟩ ::= `data` ⟨*Type*⟩ `where` `{` ⟨*ListInj*⟩ `}`

⟨*Inj*⟩ ::= ⟨*UIdent*⟩ `:` ⟨*Type*⟩

⟨*Exp4*⟩ ::= `(` ⟨*Exp*⟩ `:` ⟨*Type*⟩ `)`
| `(` ⟨*Exp*⟩ `)`

⟨*Exp3*⟩ ::= ⟨*LIdent*⟩
| ⟨*UIdent*⟩
| ⟨*Lit*⟩
| ⟨*Exp4*⟩

⟨*Exp2*⟩ ::= ⟨*Exp2*⟩ ⟨*Exp3*⟩
| ⟨*Exp3*⟩

⟨*Exp1*⟩ ::= ⟨*Exp1*⟩ + ⟨*Exp2*⟩
| ⟨*Exp2*⟩

⟨*Exp*⟩ ::= `let` ⟨*Bind*⟩ `in` ⟨*Exp*⟩
| \ ⟨*LIdent*⟩ `.` ⟨*Exp*⟩
| `case` ⟨*Exp*⟩ `of` `{` ⟨*ListBranch*⟩ `}`
| ⟨*Exp1*⟩

⟨*Lit*⟩ ::= ⟨*Integer*⟩
| ⟨*Char*⟩

⟨*Branch*⟩ ::= ⟨*Pattern*⟩ `=>` ⟨*Exp*⟩

⟨*Pattern1*⟩ ::= ⟨*LIdent*⟩
| ⟨*Lit*⟩
| `_`
| ⟨*UIdent*⟩
| `(` ⟨*Pattern*⟩ `)`

⟨*Pattern*⟩ ::= ⟨*UIdent*⟩ ⟨*ListPattern1*⟩
| ⟨*Pattern1*⟩

⟨*ListDef*⟩ ::= ϵ
| ⟨*Def*⟩
| ⟨*Def*⟩ `;` ⟨*ListDef*⟩

⟨*ListBranch*⟩ ::= ϵ
| ⟨*Branch*⟩
| ⟨*Branch*⟩ `;` ⟨*ListBranch*⟩

⟨*ListInj*⟩ ::= ϵ
| ⟨*Inj*⟩
| ⟨*Inj*⟩ `;` ⟨*ListInj*⟩

⟨*ListLIdent*⟩ ::= ϵ
| ⟨*LIdent*⟩ ⟨*ListLIdent*⟩

⟨*ListType*⟩ ::= ϵ
| ⟨*Type*⟩ ⟨*ListType*⟩

$$\begin{array}{lll} \langle \mathit{ListTVar} \rangle & ::= & \epsilon \\ & | & \langle \mathit{TVar} \rangle \; \langle \mathit{ListTVar} \rangle \end{array}$$

$$\begin{array}{lll} \langle \mathit{ListPattern1} \rangle & ::= & \langle \mathit{Pattern1} \rangle \\ & | & \langle \mathit{Pattern1} \rangle \; \langle \mathit{ListPattern1} \rangle \end{array}$$