The Language Core

BNF-converter

March 30, 2017

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 Core

Identifiers

Identifiers ⟨*Ident*⟩ are unquoted strings beginning with a letter, followed by any combination of letters, digits, and the characters `_ '`, reserved words excluded.

Literals

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 Core are the following:

There are no reserved words in Core.

The symbols used in Core are the following:

`%module`	`%data`	`=`
`{`	`}`	`%newtype`
`;`	`%rec`	`::`
`@`	`\`	`->`
`%let`	`%in`	`%case`
`%of`	`%coerce`	`%note`
`%external`	`(`	`)`
`%_`	`%forall`	`.`
`*`	`#`	`?`

Comments

There are no single-line comments in the grammar.
There are no multiple-line comments in the grammar.

The syntactic structure of Core

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

⟨*Module*⟩ ::= `%module` ⟨*Ident*⟩ ⟨*ListTdef*⟩ ⟨*ListVdefg*⟩

⟨*Tdef*⟩ ::= `%data` ⟨*QualIdent*⟩ ⟨*ListTbind*⟩ `=` `{` ⟨*ListCdef*⟩ `}`
  | `%newtype` ⟨*QualIdent*⟩ ⟨*ListTbind*⟩ ⟨*MaybeTy*⟩

⟨*ListTdef*⟩ ::= ϵ
  | ⟨*Tdef*⟩ `;` ⟨*ListTdef*⟩

⟨*MaybeTy*⟩ ::= `=` ⟨*Ty*⟩
  | ϵ

⟨*Cdef*⟩ ::= ⟨*QualIdent*⟩ ⟨*ListATbind*⟩ ⟨*ListTyt*⟩

⟨*ListTyt*⟩ ::= ϵ
  | ⟨*Tyt*⟩ ⟨*ListTyt*⟩

⟨*Tyt*⟩ ::= ⟨*Ty2*⟩

⟨*ListCdef*⟩ ::= ⟨*Cdef*⟩
  | ⟨*Cdef*⟩ `;` ⟨*ListCdef*⟩

⟨*Vdefg*⟩ ::= `%rec` `{` ⟨*ListVdef*⟩ `}`
  | ⟨*Vdef*⟩

```
⟨ListVdefg⟩  ::=  ε
             |    ⟨Vdefg⟩ ; ⟨ListVdefg⟩

⟨Vdef⟩  ::=  ⟨QualIdent⟩ :: ⟨Ty⟩ = ⟨Exp⟩
        |    ⟨Ident⟩ :: ⟨Ty⟩ = ⟨Exp⟩

⟨ListVdef⟩  ::=  ⟨Vdef⟩
            |    ⟨Vdef⟩ ; ⟨ListVdef⟩

⟨Exp2⟩  ::=  ⟨Ident⟩
        |    ⟨QualIdent⟩
        |    ⟨Lit⟩
        |    ( ⟨Exp⟩ )

⟨Exp1⟩  ::=  ⟨Exp1⟩ ⟨Exp2⟩
        |    ⟨Exp1⟩ @ ⟨Ty2⟩
        |    ⟨Exp2⟩

⟨Exp⟩  ::=  \ ⟨ListBind⟩ -> ⟨Exp⟩
       |    %let ⟨Vdefg⟩ %in ⟨Exp⟩
       |    %case ⟨Exp2⟩ %of ⟨Vbind⟩ { ⟨ListAlt⟩ }
       |    %coerce ⟨Ty2⟩ ⟨Exp⟩
       |    %note ⟨String⟩ ⟨Exp⟩
       |    %external ⟨String⟩ ⟨Ty⟩
       |    ⟨Exp1⟩

⟨Bind⟩  ::=  ⟨Vbind⟩
        |    @ ⟨Tbind⟩

⟨ListBind⟩  ::=  ⟨Bind⟩
            |    ⟨Bind⟩ ⟨ListBind⟩

⟨Alt⟩  ::=  ⟨QualIdent⟩ ⟨ListATbind⟩ ⟨ListVbind⟩ -> ⟨Exp⟩
       |    ⟨Lit⟩ -> ⟨Exp⟩
       |    %_ -> ⟨Exp⟩

⟨ListAlt⟩  ::=  ⟨Alt⟩
           |    ⟨Alt⟩ ; ⟨ListAlt⟩

⟨Vbind⟩  ::=  ( ⟨Ident⟩ :: ⟨Ty⟩ )

⟨ListVbind⟩  ::=  ε
             |    ⟨Vbind⟩ ⟨ListVbind⟩

⟨Tbind⟩  ::=  ( ⟨Ident⟩ :: ⟨Kind1⟩ )
         |    ⟨Ident⟩

⟨ListTbind⟩  ::=  ε
             |    ⟨Tbind⟩ ⟨ListTbind⟩
```

```
⟨ATbind⟩  ::=  @ ⟨Tbind⟩

⟨ListATbind⟩  ::=  ϵ
               |    ⟨ATbind⟩ ⟨ListATbind⟩

⟨Ty2⟩  ::=  ⟨Ident⟩
        |   ⟨QualIdent⟩
        |   ( ⟨Ty⟩ )

⟨Ty1⟩  ::=  ⟨Ty1⟩ ⟨Ty2⟩
        |   ⟨Ty2⟩

⟨Ty⟩  ::=  ⟨Ty1⟩ −> ⟨Ty⟩
       |   %forall ⟨ListTbind⟩ . ⟨Ty⟩
       |   ⟨Ty1⟩

⟨Kind1⟩  ::=  *
          |   #
          |   ?
          |   ( ⟨Kind⟩ )

⟨Kind⟩  ::=  ⟨Kind1⟩ −> ⟨Kind⟩
         |   ⟨Kind1⟩

⟨Lit⟩  ::=  ( ⟨Integer⟩ :: ⟨Ty2⟩ )
        |   ( ⟨Double⟩ :: ⟨Ty2⟩ )
        |   ( ⟨Char⟩ :: ⟨Ty2⟩ )
        |   ( ⟨String⟩ :: ⟨Ty2⟩ )

⟨QualIdent⟩  ::=  ⟨Ident⟩ . ⟨Ident⟩
```