The Language clafer

BNF-converter

October 1, 2015

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 clafer

Literals

PosInteger literals are recognized by the regular expression ⟨*digit*⟩+

PosDouble literals are recognized by the regular expression ⟨*digit*⟩+‘.’⟨*digit*⟩+ ‘e’‘−’?⟨*digit*⟩+

PosReal literals are recognized by the regular expression ⟨*digit*⟩+‘.’⟨*digit*⟩+

PosString literals are recognized by the regular expression ‘”’(⟨*anychar*⟩ − [“”\”] | ‘\’[“”\nt”]) ∗ ‘”’

PosIdent literals are recognized by the regular expression ⟨*letter*⟩(⟨*letter*⟩ | ⟨*digit*⟩ | ‘_’ | ‘”)∗

PosLineComment literals are recognized by the regular expression {“//”}(⟨*anychar*⟩− ‘ ’)∗

PosBlockComment literals are recognized by the regular expression {“/*”}(⟨*anychar*⟩− ‘*’ | ‘*’ + (⟨*anychar*⟩ − [“*/”])) ∗ ‘*’ + ‘/’

PosAlloy literals are recognized by the regular expression {“[alloy|”}(⟨*anychar*⟩− ‘|’ | ‘|’ + (⟨*anychar*⟩ − ‘]’)) ∗ {“|]”}

PosChoco literals are recognized by the regular expression {“[choco|”}(⟨*anychar*⟩− ‘|’ | ‘|’ + (⟨*anychar*⟩ − ‘]’)) ∗ {“|]”}

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 clafer are the following:

```
abstract  all   assert
disj      else  enum
if        in    lone
max       min   mux
no        not   one
opt       or    product
some      sum   then
xor
```

The symbols used in clafer are the following:

```
=    [     ]
<<   >>    {
}    `     :
->   ->>   :=
?    +     *
..   |     <=>
=>   ||    &&
!    <     >
<=   >=    !=
-    /     %
#    <:    :>
++   ,     --
**   &     .
;    \     (
)
```

Comments

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

The syntactic structure of clafer

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*⟩ ::= ⟨*ListDeclaration*⟩

⟨*Declaration*⟩ ::= `enum` ⟨*PosIdent*⟩ = ⟨*ListEnumId*⟩
| ⟨*Element*⟩

⟨*Clafer*⟩ ::= ⟨*Abstract*⟩ ⟨*GCard*⟩ ⟨*PosIdent*⟩ ⟨*Super*⟩ ⟨*Reference*⟩ ⟨*Card*⟩ ⟨*Init*⟩ ⟨*Elements*⟩

⟨*Constraint*⟩ ::= `[` ⟨*ListExp*⟩ `]`

⟨*Assertion*⟩ ::= `assert [` ⟨*ListExp*⟩ `]`

⟨*Goal*⟩ ::= << ⟨*ListExp*⟩ >>

⟨*Abstract*⟩ ::= ϵ
| `abstract`

⟨*Elements*⟩ ::= ϵ
| { ⟨*ListElement*⟩ }

⟨*Element*⟩ ::= ⟨*Clafer*⟩
| ‘ ⟨*Name*⟩ ⟨*Card*⟩ ⟨*Elements*⟩
| ⟨*Constraint*⟩
| ⟨*Goal*⟩
| ⟨*Assertion*⟩

⟨*Super*⟩ ::= ϵ
| `:` ⟨*Exp18*⟩

⟨*Reference*⟩ ::= ϵ
| −> ⟨*Exp15*⟩
| −>> ⟨*Exp15*⟩

⟨*Init*⟩ ::= ϵ
| ⟨*InitHow*⟩ ⟨*Exp*⟩

⟨*InitHow*⟩ ::= =
| `:=`

⟨*GCard*⟩ ::= ϵ
| `xor`
| `or`
| `mux`
| `opt`
| ⟨*NCard*⟩

⟨*Card*⟩ ::= ϵ
| `?`
| `+`
| `*`
| ⟨*PosInteger*⟩
| ⟨*NCard*⟩

⟨*NCard*⟩ ::= ⟨*PosInteger*⟩ `..` ⟨*ExInteger*⟩

⟨*ExInteger*⟩ ::= `*`
| ⟨*PosInteger*⟩

⟨*Name*⟩ ::= ⟨*ListModId*⟩

⟨*Exp*⟩ ::= `all disj` ⟨*Decl*⟩ | ⟨*Exp*⟩
| `all` ⟨*Decl*⟩ | ⟨*Exp*⟩
| ⟨*Quant*⟩ `disj` ⟨*Decl*⟩ | ⟨*Exp*⟩
| ⟨*Quant*⟩ ⟨*Decl*⟩ | ⟨*Exp*⟩
| ⟨*Exp1*⟩

⟨*Exp1*⟩ ::= `max` ⟨*Exp2*⟩
| `min` ⟨*Exp2*⟩
| ⟨*Exp1*⟩ <=> ⟨*Exp2*⟩
| ⟨*Exp2*⟩

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

⟨*Exp3*⟩ ::= ⟨*Exp3*⟩ || ⟨*Exp4*⟩
| ⟨*Exp4*⟩

⟨*Exp4*⟩ ::= ⟨*Exp4*⟩ `xor` ⟨*Exp5*⟩
| ⟨*Exp5*⟩

⟨*Exp5*⟩ ::= ⟨*Exp5*⟩ `&&` ⟨*Exp6*⟩
| ⟨*Exp6*⟩

⟨*Exp6*⟩ ::= `!` ⟨*Exp7*⟩
| ⟨*Exp7*⟩

⟨*Exp7*⟩ ::= ⟨*Exp7*⟩ < ⟨*Exp8*⟩
| ⟨*Exp7*⟩ > ⟨*Exp8*⟩
| ⟨*Exp7*⟩ = ⟨*Exp8*⟩
| ⟨*Exp7*⟩ <= ⟨*Exp8*⟩
| ⟨*Exp7*⟩ >= ⟨*Exp8*⟩
| ⟨*Exp7*⟩ `!=` ⟨*Exp8*⟩
| ⟨*Exp7*⟩ `in` ⟨*Exp8*⟩
| ⟨*Exp7*⟩ `not in` ⟨*Exp8*⟩
| ⟨*Exp8*⟩

⟨*Exp8*⟩ ::= ⟨*Quant*⟩ ⟨*Exp12*⟩
| ⟨*Exp9*⟩

⟨*Exp9*⟩ ::= ⟨*Exp9*⟩ + ⟨*Exp10*⟩
| ⟨*Exp9*⟩ − ⟨*Exp10*⟩
| ⟨*Exp10*⟩

⟨*Exp10*⟩ ::= ⟨*Exp10*⟩ `*` ⟨*Exp11*⟩
| ⟨*Exp10*⟩ `/` ⟨*Exp11*⟩
| ⟨*Exp10*⟩ `%` ⟨*Exp11*⟩
| ⟨*Exp11*⟩

⟨*Exp11*⟩ ::= `sum` ⟨*Exp12*⟩
| `product` ⟨*Exp12*⟩
| `#` ⟨*Exp12*⟩
| − ⟨*Exp12*⟩
| ⟨*Exp12*⟩

⟨*Exp12*⟩ ::= `if` ⟨*Exp12*⟩ `then` ⟨*Exp12*⟩ `else` ⟨*Exp13*⟩
| ⟨*Exp13*⟩

⟨*Exp13*⟩ ::= ⟨*Exp13*⟩ `<:` ⟨*Exp14*⟩
| ⟨*Exp14*⟩

⟨*Exp14*⟩ ::= ⟨*Exp14*⟩ `:>` ⟨*Exp15*⟩
| ⟨*Exp15*⟩

⟨*Exp15*⟩ ::= ⟨*Exp15*⟩ ++ ⟨*Exp16*⟩
| ⟨*Exp15*⟩ `,` ⟨*Exp16*⟩
| ⟨*Exp16*⟩

⟨*Exp16*⟩ ::= ⟨*Exp16*⟩ −− ⟨*Exp17*⟩
| ⟨*Exp17*⟩

⟨*Exp17*⟩ ::= ⟨*Exp17*⟩ `**` ⟨*Exp18*⟩
| ⟨*Exp17*⟩ `&` ⟨*Exp18*⟩
| ⟨*Exp18*⟩

⟨*Exp18*⟩ ::= ⟨*Exp18*⟩ `.` ⟨*Exp19*⟩
| ⟨*Exp19*⟩

⟨*Exp19*⟩ ::= ⟨*Name*⟩
| ⟨*PosInteger*⟩
| ⟨*PosDouble*⟩
| ⟨*PosReal*⟩
| ⟨*PosString*⟩
| `(` ⟨*Exp*⟩ `)`

⟨*Decl*⟩ ::= ⟨*ListLocId*⟩ `:` ⟨*Exp15*⟩

⟨*Quant*⟩ ::= `no`
| `not`
| `lone`
| `one`
| `some`

⟨*EnumId*⟩ ::= ⟨*PosIdent*⟩

⟨*ModId*⟩ ::= ⟨*PosIdent*⟩

⟨*LocId*⟩ ::= ⟨*PosIdent*⟩

⟨*ListDeclaration*⟩ ::= ϵ
| ⟨*Declaration*⟩ ⟨*ListDeclaration*⟩

⟨*ListEnumId*⟩ ::= ⟨*EnumId*⟩
| ⟨*EnumId*⟩ | ⟨*ListEnumId*⟩

⟨*ListElement*⟩ ::= ϵ
| ⟨*Element*⟩ ⟨*ListElement*⟩

⟨*ListExp*⟩ ::= ϵ
| ⟨*Exp*⟩ ⟨*ListExp*⟩

⟨*ListLocId*⟩ ::= ⟨*LocId*⟩
| ⟨*LocId*⟩ `;` ⟨*ListLocId*⟩

⟨*ListModId*⟩ ::= ⟨*ModId*⟩
| ⟨*ModId*⟩ \ ⟨*ListModId*⟩