The Language LAMA

BNF-converter

August 22, 2012

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 LAMA

Literals

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

Identifier literals are recognized by the regular expression $(\langle letter \rangle \mid$ '_'$)(\langle letter \rangle \mid \langle digit \rangle \mid$ '_'$)*$

StateId literals are recognized by the regular expression $(\langle letter \rangle \mid$ '_'$)(\langle letter \rangle \mid \langle digit \rangle \mid$ '_'$) *$ '''

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 LAMA are the following:

`and`	`assertion`	`automaton`
`bool`	`constants`	`default`
`definition`	`div`	`edge`
`enum`	`false`	`initial`
`input`	`int`	`invariant`
`ite`	`let`	`local`
`location`	`match`	`mod`
`node`	`nodes`	`not`
`or`	`project`	`real`
`returns`	`sint`	`state`
`tel`	`transition`	`true`
`typedef`	`uint`	`use`
`xor`		

The symbols used in LAMA are the following:

`;`	`=`	`{`
`}`	`,`	`^`
`(`	`#`	`)`
`[`	`]`	`-`
`/`	`:`	`.`
`_`	`=>`	`<`
`>`	`<=`	`>=`
`+`	`*`	

Comments

Single-line comments begin with −−.
There are no multiple-line comments in the grammar.

The syntactic structure of LAMA

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*⟩	::=	⟨*TypeDefs*⟩ ⟨*ConstantDefs*⟩ ⟨*Inputs*⟩ ⟨*Declarations*⟩ ⟨*Flow*⟩ ⟨*Initial*⟩ ⟨*Assertion*⟩ ⟨*Invariant*⟩
⟨*TypeDefs*⟩	::=	ϵ
	|	`typedef` ⟨*ListTypeDef*⟩

```
⟨ListTypeDef⟩  ::=  ⟨TypeDef⟩ ;
                 |  ⟨TypeDef⟩ ; ⟨ListTypeDef⟩

⟨TypeDef⟩  ::=  enum ⟨Identifier⟩ = { ⟨ListEnumConstr⟩ }

⟨EnumConstr⟩  ::=  ⟨Identifier⟩

⟨ListEnumConstr⟩  ::=  ⟨EnumConstr⟩
                    |  ⟨EnumConstr⟩ , ⟨ListEnumConstr⟩

⟨Type⟩  ::=  ⟨BaseType⟩
          |  ⟨Identifier⟩
          |  ⟨BaseType⟩ ^ ⟨Natural⟩
          |  ( # ⟨ListType⟩ )

⟨ListType⟩  ::=  ⟨Type⟩
              |  ⟨Type⟩ ⟨ListType⟩

⟨BaseType⟩  ::=  bool
              |  int
              |  real
              |  sint [ ⟨Natural⟩ ]
              |  uint [ ⟨Natural⟩ ]

⟨ConstantDefs⟩  ::=  ϵ
                  |  constants ⟨ListConstantDef⟩

⟨ListConstantDef⟩  ::=  ⟨ConstantDef⟩ ;
                     |  ⟨ConstantDef⟩ ; ⟨ListConstantDef⟩

⟨ConstantDef⟩  ::=  ⟨Identifier⟩ = ⟨Constant⟩

⟨Natural⟩  ::=  ⟨Integer⟩

⟨IntegerConst⟩  ::=  ⟨Integer⟩
                  |  ( − ⟨Integer⟩ )

⟨Constant⟩  ::=  ⟨BoolV⟩
              |  ⟨IntegerConst⟩
              |  ⟨IntegerConst⟩ / ⟨IntegerConst⟩
              |  sint [ ⟨Natural⟩ ] ( ⟨IntegerConst⟩ )
              |  uint [ ⟨Natural⟩ ] ( ⟨Natural⟩ )

⟨BoolV⟩  ::=  true
           |  false

⟨Inputs⟩  ::=  ϵ
            |  input ⟨VarDecls⟩
```

```
⟨Initial⟩ ::= ϵ
          |   initial ⟨ListStateInit⟩ ;

⟨Assertion⟩ ::= ϵ
            |   assertion ⟨Expr⟩ ;

⟨Invariant⟩ ::= ϵ
            |   invariant ⟨Expr⟩ ;

⟨ListStateInit⟩ ::= ⟨StateInit⟩
                |   ⟨StateInit⟩ , ⟨ListStateInit⟩

⟨StateInit⟩ ::= ⟨Identifier⟩ = ⟨ConstExpr⟩

⟨ConstExpr⟩ ::= ⟨Expr⟩

⟨TypedVar⟩ ::= ⟨Identifier⟩ : ⟨Type⟩

⟨ListTypedVar⟩ ::= ⟨TypedVar⟩
               |   ⟨TypedVar⟩ , ⟨ListTypedVar⟩

⟨MaybeTypedVars⟩ ::= ϵ
                 |   ⟨ListTypedVar⟩

⟨Node⟩ ::= node ⟨Identifier⟩ ( ⟨MaybeTypedVars⟩ )
           returns ( ⟨ListTypedVars⟩ ) let
             ⟨Declarations⟩
             ⟨Flow⟩
             ⟨ControlStructure⟩
             ⟨Initial⟩
             ⟨Assertion⟩
           tel

⟨ListNode⟩ ::= ⟨Node⟩
           |   ⟨Node⟩ ⟨ListNode⟩

⟨Declarations⟩ ::= ⟨NodeDecls⟩ ⟨LocalDecls⟩ ⟨StateDecls⟩

⟨VarDecls⟩ ::= ⟨TypedVar⟩ ;
           |   ⟨TypedVar⟩ ; ⟨VarDecls⟩

⟨NodeDecls⟩ ::= ϵ
            |   nodes ⟨ListNode⟩

⟨LocalDecls⟩ ::= ϵ
             |   local ⟨VarDecls⟩
```

```
⟨StateDecls⟩ ::= ϵ
              |  state ⟨VarDecls⟩

⟨Flow⟩ ::= ⟨LocalDefinitions⟩ ⟨Transitions⟩

⟨LocalDefinitions⟩ ::= ϵ
                    |  definition ⟨ListInstantDefinition⟩

⟨Transitions⟩ ::= ϵ
               |  transition ⟨ListTransition⟩

⟨ListInstantDefinition⟩ ::= ⟨InstantDefinition⟩ ;
                         |  ⟨InstantDefinition⟩ ; ⟨ListInstantDefinition⟩

⟨ListTransition⟩ ::= ⟨Transition⟩ ;
                  |  ⟨Transition⟩ ; ⟨ListTransition⟩

⟨InstantDefinition⟩ ::= ⟨Identifier⟩ = ⟨Expr⟩
                     |  ⟨Identifier⟩ = ( use ⟨Identifier⟩ ⟨ListExpr⟩ )

⟨Transition⟩ ::= ⟨StateId⟩ = ⟨Expr⟩

⟨ControlStructure⟩ ::= ⟨ListAutomaton⟩

⟨Automaton⟩ ::= automaton let
                  ⟨ListLocation⟩
                  ⟨InitialLocation⟩
                  ⟨ListEdge⟩
                  ⟨Defaults⟩
                tel

⟨Location⟩ ::= location ⟨Identifier⟩ let ⟨Flow⟩ tel

⟨InitialLocation⟩ ::= initial ⟨Identifier⟩ ;

⟨Edge⟩ ::= edge ( ⟨Identifier⟩ , ⟨Identifier⟩ ) : ⟨Expr⟩ ;

⟨ListLocation⟩ ::= ⟨Location⟩
                |  ⟨Location⟩ ⟨ListLocation⟩

⟨ListEdge⟩ ::= ⟨Edge⟩
            |  ⟨Edge⟩ ⟨ListEdge⟩

⟨ListAutomaton⟩ ::= ϵ
                 |  ⟨Automaton⟩ ⟨ListAutomaton⟩
```

```
⟨Defaults⟩  ::=  ϵ
             |   default ⟨ListDefault⟩ ;

⟨ListDefault⟩  ::=  ⟨Default⟩
                |   ⟨Default⟩ , ⟨ListDefault⟩

⟨Default⟩  ::=  ⟨Identifier⟩ = ⟨Expr⟩

⟨Atom⟩  ::=  ⟨Constant⟩
         |   ⟨Identifier⟩

⟨Expr⟩  ::=  ⟨Atom⟩
         |   ( ⟨UnOp⟩ ⟨Expr⟩ )
         |   ( ⟨BinOp⟩ ⟨Expr⟩ ⟨Expr⟩ )
         |   ( ⟨TernOp⟩ ⟨Expr⟩ ⟨Expr⟩ ⟨Expr⟩ )
         |   ( # ⟨ListExpr⟩ )
         |   ( project ⟨Identifier⟩ ⟨Natural⟩ )
         |   ( match ⟨Expr⟩ { ⟨ListPattern⟩ } )

⟨ListExpr⟩  ::=  ϵ
             |   ⟨Expr⟩ ⟨ListExpr⟩

⟨ListPattern⟩  ::=  ⟨Pattern⟩
                |   ⟨Pattern⟩ , ⟨ListPattern⟩

⟨Pattern⟩  ::=  ⟨PatHead⟩ . ⟨Expr⟩

⟨PatHead⟩  ::=  ⟨EnumConstr⟩
            |   _

⟨List2Id⟩  ::=  ⟨Identifier⟩ ⟨Identifier⟩
            |   ⟨Identifier⟩ ⟨List2Id⟩

⟨UnOp⟩  ::=  not
```

$$
\begin{array}{lcl}
\langle \mathit{BinOp} \rangle & ::= & \texttt{or} \\
 & | & \texttt{and} \\
 & | & \texttt{xor} \\
 & | & \texttt{=>} \\
 & | & \texttt{=} \\
 & | & \texttt{<} \\
 & | & \texttt{>} \\
 & | & \texttt{<=} \\
 & | & \texttt{>=} \\
 & | & \texttt{+} \\
 & | & \texttt{-} \\
 & | & \texttt{*} \\
 & | & \texttt{/} \\
 & | & \texttt{div} \\
 & | & \texttt{mod}
\end{array}
$$

$$
\langle \mathit{TernOp} \rangle \;::=\; \texttt{ite}
$$