The Language LAMA

BNF-converter

March 12, 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 ⟨*Int*⟩ are nonempty sequences of digits.

Identifier literals are recognized by the regular expression ⟨*letter*⟩(⟨*letter*⟩ | ⟨*digit*⟩ | '_')∗

StateId literals are recognized by the regular expression ⟨*letter*⟩(⟨*letter*⟩ | ⟨*digit*⟩ | '_') ∗ '"

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   constr
definition   div         edge
enum         false       initial
int          invariant   ite
let          local       location
mod          node        nodes
not          or          output
project      real        record
returns      select      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.

⟨*File*⟩ ::= ⟨*TypeDefs*⟩ ⟨*ConstantDefs*⟩ ⟨*Node*⟩ ⟨*Assertion*⟩ ⟨*Initial*⟩ ⟨*Invariant*⟩

⟨*TypeDefs*⟩ ::= ϵ
| `typedef` ⟨*ListTypeDef*⟩

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

⟨*TypeDef*⟩ ::= ⟨*EnumT*⟩
| ⟨*RecordT*⟩

⟨*EnumConstr*⟩ ::= ⟨*Identifier*⟩

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

⟨*EnumT*⟩ ::= `enum` ⟨*Identifier*⟩ = { ⟨*ListEnumConstr*⟩ }

⟨*RecordField*⟩ ::= ⟨*Identifier*⟩ `:` ⟨*Type*⟩

⟨*ListRecordField*⟩ ::= ⟨*RecordField*⟩
| ⟨*RecordField*⟩ `,` ⟨*ListRecordField*⟩

⟨*RecordT*⟩ ::= `record` ⟨*Identifier*⟩ = { ⟨*ListRecordField*⟩ }

⟨*Type*⟩ ::= ⟨*BaseType*⟩
| ⟨*Identifier*⟩
| ⟨*BaseType*⟩ `^` ⟨*Natural*⟩

⟨*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`

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

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

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

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

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

⟨*ConstExpr*⟩ ::= ⟨*Expr*⟩

⟨*ListIdentifier*⟩ ::= ⟨*Identifier*⟩
| ⟨*Identifier*⟩ `,` ⟨*ListIdentifier*⟩

⟨*TypedVars*⟩ ::= ⟨*ListIdentifier*⟩ `:` ⟨*Type*⟩

⟨*ListTypedVars*⟩ ::= ⟨*TypedVars*⟩
| ⟨*TypedVars*⟩ `;` ⟨*ListTypedVars*⟩

⟨*MaybeTypedVars*⟩ ::= ϵ
| ⟨*ListTypedVars*⟩

⟨*Node*⟩ ::= `node` ⟨*Identifier*⟩ `(` ⟨*MaybeTypedVars*⟩ `)` `returns` `(` ⟨*ListTypedVars*⟩ `)` `;`
`let`
⟨*NodeDeclarations*⟩ ⟨*StateDeclarations*⟩ ⟨*LocalDeclarations*⟩
⟨*Flow*⟩ ⟨*ControlStructure*⟩ ⟨*Initial*⟩
`tel`

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

⟨*VarDecls*⟩ ::= ⟨*TypedVars*⟩ `;`
| ⟨*TypedVars*⟩ `;` ⟨*VarDecls*⟩

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

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

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

⟨*Flow*⟩ ::= ⟨*LocalDefinitions*⟩ ⟨*Outputs*⟩ ⟨*Transitions*⟩

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

⟨*Outputs*⟩ ::= ϵ
| `output` ⟨*ListInstantDefinition*⟩

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

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

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

⟨*InstantDefinition*⟩ ::= ⟨*Identifier*⟩ = ⟨*Expr*⟩
| ⟨*Pattern*⟩ = ⟨*NodeUsage*⟩

⟨*Transition*⟩ ::= ⟨*StateId*⟩ = ⟨*Expr*⟩

⟨*Pattern*⟩ ::= ⟨*List2Id*⟩

⟨*List2Id*⟩ ::= ⟨*Identifier*⟩ `,` ⟨*Identifier*⟩
| ⟨*Identifier*⟩ `,` ⟨*List2Id*⟩

⟨*NodeUsage*⟩ ::= `(` `use` ⟨*Identifier*⟩ ⟨*ListExpr*⟩ `)`

⟨*ControlStructure*⟩ ::= ⟨*ListAutomaton*⟩

⟨*Automaton*⟩ ::= `automaton` `let` ⟨*ListLocation*⟩ ⟨*InitialLocation*⟩ ⟨*ListEdge*⟩ `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*⟩

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

⟨*Expr*⟩ ::= ⟨*Atom*⟩
| `(` ⟨*UnOp*⟩ ⟨*Expr*⟩ `)`
| `(` ⟨*BinOp*⟩ ⟨*Expr*⟩ ⟨*Expr*⟩ `)`
| `(` ⟨*TernOp*⟩ ⟨*Expr*⟩ ⟨*Expr*⟩ ⟨*Expr*⟩ `)`
| `(` `constr` ⟨*Identifier*⟩ ⟨*ListExpr*⟩ `)`
| `(` `project` ⟨*Identifier*⟩ ⟨*Natural*⟩ `)`
| `(` `select` ⟨*Identifier*⟩ ⟨*Identifier*⟩ `)`

⟨*ListExpr*⟩ ::= ⟨*Expr*⟩
| ⟨*Expr*⟩ ⟨*ListExpr*⟩

⟨*UnOp*⟩ ::= `not`

⟨*BinOp*⟩ ::= `or`
| `and`
| `xor`
| =>
| =
| <
| >
| <=
| >=
| +
| −
| `*`
| `/`
| `div`
| `mod`

⟨*TernOp*⟩ ::= `ite`