Stella Reference Manual

Antonio Kantek (ak2765)
CVN student
COMS W4115, Spring 2008

1. Introduction

Stella is a Domain Specific Language for Machine Learning (ML) model implementation. The process of discovering patterns in data is a semiautomatic (empirical) process. Classifiers for data analysis should be easily created and tested. Data analysis can be understood as the process of extracting a set of patterns (knowledge) from raw data (information).

Generic languages like Java and C++ demands too much effort for ML model implementation. In the other hand, specific query languages for ML provided by RDBMS vendors like Microsoft SQL Server are too weak. A SDL is the perfect solution for real world machine learning.

2. Lexical Structure

The sequence of input is composed by comments, tokens, operators and literals (all in ASCII characters). All terminal tokens are in blue and all non-terminal are in black.

2.1 Keywords

The following character sequences represent reserved keywords. They cannot be used as identifiers:

```
attribute   boolean     classifier  continue    dataset     date
dim         define      double      false       for         function instance
if          int         is          new         NaN
new         NOMINAL     null        NUMERIC     self        string      true
void
```

2.2 Operators

Besides the keywords described above, Stella provides the following special operators:

Return operator: `^`
Assign operator: `:=`

Code block delimiters: `{` and `}`
Matrix/Array delimiters: `[` and `]`
Open/Close parenthesis: `(` and `)`

Logical operators: NOT: `~` , AND: `&&` and OR: `||`

Relational operators: EQUAL: `==` and NOT_EQUAL: `!=` ,

Relational operators: GREATER THAN: > , GREATER EQUALS THAN: >= , LESS THAN: < and LESS EQUALS: <=

2.3 White spaces

Horizontal tab, line terminator, carrier return and space character define a white space.

2.4 Comment

The language supports the C++ end-of-line comment. Any valid character after `//` is considered part of the comment.

Example of comment:

```
// This is a comment.
```

2.5 Identifier

An identifier is defined by an unlimited sequence of underscore, letters or digits where the first one is a letter or underscore.

2.6 String Literal

A string is represented by an unlimited sequence of characters under single quotes. In a string, the character " ' " must be preceded by a " \ ".

Examples of strings:

'Hello World' 'John\'s Pizzas'

2.7 Integer Literal

Integers are expressed in base 10 (decimal). Integers are represented by **`int`** keyword.

2.8 Floating Point Literal

A floating point literal can start with an integer (the whole-number) followed by a fractional part or an exponent part (the exponent part is represented by letter e or E). The integer is optional. A floating point is represented by **`double`** keyword.

Example of floating point literals:

1.	0.5	0.5e10	0.5e+10	0.5e-10	
3e10	3e+10	3e-10	.3	.3e10	.5e-10	.5e+10

3. Syntax Notation

Stella is a strong typed language and contains three types of storage data (objects): basic types (boolean, string, int and doubles), classifiers and dataset related types (attribute, instance and dataset itself). It is possible to construct arrays of all types.

All programs written in Stella are composed by statements.

```
program: (statement SEMI!)* EOF!;
```

3.1 Statement

A statement can be a reference declaration, a code execution, a function definition or a classifier definition.

```
statement: (refDeclaration | codeExecution | functionDef | classifierDef );
```

3.2 Reference Declaration

A reference for any kind of object (basic type, dataset type or classifier) follows the rule bellow:

```
refDeclaration: type identifier (assignValue)? ;
```

A type can be one of the objects described in the introduction or an array of them. The length of the array (dimension) is specified by `dim` keyword.

```
type:
  (int | double | boolean | instance | attribute | classifier | dataset ) (array)?
;
```

```
array: [ (dim : INTEGER)? ] ( [ (dim: INTEGER)? ] )*
```

An object can be initialized (assigned to) an expression. Expression are fully explored on the next session.

```
assignValue: := expression;
```

3.3 Expressions

Expressions represent the most extensive rules. They can produce constant values, logical operations, relational operations and so on.

```
expression: andCondition (OR andCondition)? ;

andCondition: condition (AND condition)? ;

condition: ( (operand (rightOperator) ) | (NOT condition));

rightOperator: (is (NOT)? identifier ) | (compare operand);
```

```
compare: ( == | != | > | < | >= | <= );

operand: factor ( (+|-) factor )?

term: value ((*|/) value)?

value: ( constant | identifier | expression);

constant:
( null |stringValue | intValue | doubleValue | booleanValue | arrayDefinition);

stringValue: ' ([a..zA..Z_] ([a..zA..Z_0..9] | whitespace )* '

intValue: INTEGER

doubleValue: DOUBLE

arrayDefinition: [ INTEGER ] ( [ INTEGER ] )*

booleanValue: true | false
```

3.4 Functions

Global functions are one of the most intuitive way to implement mathematical procedures. Stella defines a function pretty much in the same way as C does (the only difference is the function keyword in the beginning).

```
functionDef: function type IDENTIFIER ( argumentList ) { codeExecution }

argumentList: type IDENTIFIER (COMMA type IDENTIFIER)*
```

3.5 Code Execution

A code execution is composed by statements like method invocation, return statement, reference declaration and assignment, conditional block, conditional block and loop block.

```
codeExecution: (methodInvocation | return |  refDeclaration |
conditionalBlock | loopBlock )

methodInvocation: reference (DOT reference)* DOT ( argumentValues )

reference: IDENTIFIER ( [ INTEGER ] )*

argumenValues: ( methodInvocation | expression )

return: ^ (methodInvocation | expression)

conditionalBlock: if ( expression ) codeExecution
                        (if else (expression) codeExecution)*
                        (else codeExecution)

codeBlock: { codeExecution }

loopBlock: for ( IDENTIFIER [ INTEGER ] )  codeExecution
```

3.5 Classifier Declaration

A classifier is composed by fields and methods. All methods starting by # are class methods (static method in Java).

```
classiferDef: classifier IDENTIFIER { (field | method)+ }

field: declareRef;

method: (#)? type IDENTIFIER ( argumentList ) { codeExecution };
```