XBNF Grammars in XMF

Xactium Ltd.

July 31, 2004

1 Introduction

XMF is an environment for constructing advanced software engineering tools and applications. A key feature of the XMF environment is the ability to define *domain specific languages* whose features faithfully represent key aspects of the application domain.

Computer based languages are used as communication media; either system to system or human to system. In either case, a computer must be able to recognise phrases expressed in the language. When phrases are presented to a computer they must be interpreted in some way. Typically, phrases may be interpreted directly, as they are recognized, or some internal representation for the phrases may be synthesized (abstract syntax) and subsequently passed on to an internal module for processing.

There are a variety of formats by which language may be conveyed to a computer; a common mechanism is by text. A standard way of defining how to recognize and process text-based languages is by supplying a *grammar*. A grammar is a collection of rules that define the legal sequences of text strings in a language. In addition to recognising language phrases, a grammar may include actions that are to be performed as sub-phrases are processed.

In order to use grammars to process language phrases we must supply the grammar to a program called a *parser*. The medium used to express the grammar is usually text. BNF (and te extended version EBNF) is a text-based collection of grammar formation rules. BNF defines how to express a grammar and what the grammar means in terms of processing languages.

This document describes the XMF version of EBNF, called XBNF. XBNF integrates EBNF with XOCL (and any language defined by XBNF). XMF provides an XBNF parser that is supplied with an XBNF grammar and will then parse and synthesize XMF-defined laguages.

Grammars (and therefore languages) in XMF are associated with classes. Typically an XBNF grammar defines how to recognize legal sequences of characters in a language and to syntheisize an instance of the associated class (and populate the instance with instances of the class's attribute types).

2 Parsing and Synthesizing

This section describes the basics of using XBNF to define simple grammars that recognize languages and synthesize XMF data values.

A grammar is an instance of the XMF class `Parser::BNF::Grammar`. A grammar consists of a collection of *clauses* each of which is a rule that defines a non-terminal of the grammar. A non-terminal is a name (by convention a non-terminal name starts with an upper case letter).

A clause has the following form:

```
NAME ::= RULE .
```

where the `RULE` part defines how to recognize a sequence of input characters. To illustrate the essential elements of a grammar definition we will build a simple caculator that recognises arithmetic expressions and executes (synthesizes) the expressions (integers) as the parse proceeds. The grammar is constructed incrementally and defined as the value of the global variable `Calculator`:

```
Root::Calculator :=
  @Grammar
```

Typically, a grammar has a starting non-terminal; this is the clause that is used as the tsarting point of a parse. There is nothing special about the starting non-terminal in the definition. In the case of `Calculator`, the starting non-terminal is `Calc`:

```
Calc ::= Mult '='.
```

The clause for `Calc` defines that to recognize this non-terminal a parse must recognize an `Mult` (defined below) followed by a *terminal* `'='`. A terminal is a sequences of characters in quotes; a terminal successfully matches an input when the input corresponds to exactly the characters, possibly preceded by whitespace. Therefore, a `Calc` is recognized when a `Mult` is recognized followed by a `=` .

A `Mult` is a multiplicative expression, possibly involving the operators `*` and `/`. We use a standard method of defining operator precedence in the grammar, where addition operators bind tighter than multiplicative operators. This is achieved using two different non-terminals:

```
Mult ::= n1 = Add (
  '*' n2 = Mult { n1 * n2 } |
  '/' n2 = Mult { n1 / n2 } |
  { n1 }
).
```

The clause for `Mult` shows a number of typical grammar features. A `Mult` is successfully recognized when a `Add` is recognized followed by an optional `*` or `/` operator.

(c) 2004 Xactium Ltd.

Each non-terminal synthesizes a value when it successfully recognizes some input. When one non-terminal *calls* another (for example `Add` is called by `Mult`) the value syntheized by the called non-terminal may be optionally named in the definition of the calling non-terminal. Once named, the syntheized value may be referenced in the rest of the rule. Naming occurs a number of times in the definition of `Mult` where the names `n1` and `n2` are used to refer to syntheisized numbers.

A clause may include optional parts separated by `|`. In general, if a clause contains an optional component of the form `A | B` where `A` and `B` are arbitrary components, then a parse is successful when either `A` or `B` is successful.

The clause for `Mult` contains three optional components that occur after the initial `Add` is successful (hence the need for parentheses around the optional components to force the `Add` to occur). Once the `Add` is successful then the parse is successful when one of the following occurs:

- a `*` is encountered followed by a `Mult`.
- a `+` is encountered followed by a `Mult`.
- no further input is processed (this option must occur last since it subsumes the previous two options – options are tried in turn).

Clauses may contain *actions*. A action is used to synthesize arbitrary values and can refer to values that have been named previously in the clause. An action is an XOCL expression enclosed in { and }. Just like calls to non-terminals in the body of a clause, an action returns a value that may be named. If the last component performed by the successful execution of a clause is an action then the value produced by the action is the value returned by the clause.

The clause for `Mult` contains a number of actions that are used to synthesize values (in this case evaluate numeric expressions).

The clause for `Add` is the same as that for `Mult` except that the clause recognizes and syntheises addition expressions:

```
Add ::= n1 = Int (
  '+' n2 = Add { n1 + n2 } |
  '-' n2 = Add { n1 - n2 } |
  { n1 }
).
```

The complete definition of the calculator grammar is given in appendix A.

A parse is performed by creating a *parsing machine*. A parsing machine is initialized with the grammar that will control the parse and the input channel that will supply the characters for the parse. Given a parsing machine, a parse is performed by running the machine; the name of the starting non-terminal is supplied for each run. The machine will then run and terminate in one of two states:

- the parse is successful and the run returns the value syntheisized by the starting non-terminal.

(c) 2004 Xactium Ltd.

- the parse is unsuccessful.

A parsing machine for the calculator grammar that takes characters from the standard input is created as follows:

```
state := Parser::Machine::State(Calculator,stdin);
```

Run the machine as follows:

```
result := state.run("Calc");
```

Test whether the parse failed:

```
state.failed
```

3 Sequences of Elements

A grammar rule may want to specify that, at a given position in the parse, a sequence of character strings can occur. For example, this occurs when parsing XML elements, where a composite element can have any number of children elements each of which is another element.

XBNF provides the postfix operators `*` and `+` that apply to a clause component `X` and define that the parse may recognize sequences of occurrences of `X`. In the case of `*`, there may be 0 or more occurrences of `X` and in the case of `+` there may be 1 or more occurrences of `X`. If `X` synthesizes a value each time it is used in a parse then the use of `*` and `+` synthesizes sequences of values, each value in the sequence is a value syntheiszed by `X`.

The following gramar shows an example of the use of `*`. XML elements are trees; each node in the tree has a label. The following grammar recognizes XML (without attributes and text elements) trees and synthesizes an XMF sequence representation for the XML tree.

```
Root::XML :=
  @Grammar
    Element ::= SingleElement | CompositeElement.
    SingleElement ::= '<' tag = Name '/>' { Seq{tag} }.
    CompositeElement ::=
      '<' tag = Name '>'
        children = Element*
      '<' Name '/>'
      { Seq{tag | children} }.
  end;
```

(c) 2004 Xactium Ltd.

4 Specializing Grammars

5 Language Definition with At and Imports

6 Type Checking and Predicates

A A Calculator

```
Root::Calculator :=

  @Grammar

    Calc ::= Mult '='.

    Mult ::= n1 = Add (

       '*' n2 = Mult { n1 * n2 } |
      '/' n2 = Mult { n1 / n2 } |
      { n1 }

    ).

    Add ::= n1 = Int (

      '+' n2 = Add { n1 + n2 } |
      '-' n2 = Add { n1 - n2 } |
      { n1 }

    ).

  end;
```

B XBNF Grammar

```
Action ::=
  '{' exp = Exp '}'
  { PreAction(exp) } |
  '?' boolExp = Exp
  { PrePredicate(boolExp) }.

Atom ::=
  Action    |
  Literal   |
```

(c) 2004 Xactium Ltd.

```
  Call      |
  Not       |
  '(' Disjunction ')'.

Binding ::=
  name = Name '=' atom = Sequence {
    And(atom,Bind(name))
  }.

Call ::=
  name = Name { Call(name) }.

Clause ::=
  name = Name '::=' body = Disjunction '.' { Clause(name,body) }.

Conjunction ::=
  elements = Element+ {
    elements->tail->iterate(e conj = elements->head |
      And(conj,e))
  }.

Disjunction ::=
  element = Conjunction [
    '|' rest = Disjunction element = { Or(element,rest) }
  ]
  { element }.

Element ::=
  Optional  |
  Binding   |
  Sequence.

Grammar ::=
  parents = GrammarParents
  imports = GrammarImports
  clauses = Clause* { Grammar(parents,clauses->asSet,"",imports) }.

GrammarImports ::=
  'import' class = Exp classes = (',' Exp)* { Seq{class | classes} } |
  { Seq{} }.

GrammarParents ::=
  'extends' parent = Exp parents = (',' Exp)* {
     parents->asSet->including(parent)
   } |
  { Set{} }.
```

(c) 2004 Xactium Ltd.

```
Literal ::=
  'Char'       { Char() }     |
  'Str'        { Str() }      |
  'Terminal'   { Term() }     |
  'Token'      { Tok() }      |
  'Int'        { Int() }      |
  'Name'       { Name() }     |
  'EOF'        { EOF() }      |
  '@'          { At() }       |
  'ImportAt'   { ImportAt() } |
  terminal = Terminal { Terminal(terminal) }.

Not ::= 'Not' '(' parser = Sequence ')' { Not(parser) }.

Optional ::=
  '[' opt = Disjunction ']'
  { Opt(opt) }.

Path ::= name = Name names = ('::' Name)* { Seq{name | names} }.

TypeCheck ::=
  element = Atom [
    ':' type = Path element = { And(element,TypeCheck(type)) }
  ] { element }.

Sequence ::=
  element = TypeCheck [
    '*' element = { StarCons(element) } |
    '+' element = { PlusCons(element) } ]
  { element }.
```

(c) 2004 Xactium Ltd.