First Class Grammars for Language Oriented Programming

Tony Clark
Centre for Model Driven Software Engineering
Thames Valley University, St Mary's Road, Ealing, UK, W5 5RF
tony.clark@tvu.ac.uk

January 22, 2009

Abstract

Programming languages that support Language Oriented Programming (LOP) allow developers to extend the host language with new programming features. The extensions include new concrete syntax constructs that are both replacements for and interleaved with the host language. This paper describes the general features that are required in order to support the LOP approach in a language. The paper shows how many of the features are implemented in the XMF language system, describes the shortcomings of the implementation and outlines an extension to Java that would address these shortcomings.

1 Introduction

Language oriented programming (LOP) [3] is a technique used to support language driven software engineering and domain specific languages [4]. In his influential presentation, Guy Steele [5] identified the need for developers to grow languages as part of the development process.

There are many different technologies currently used to support LOP. These include macros [6], pre-processors and parsing libraries. LOP involves defining new languages that can be categorized as *external* and *internal* [14]. An external language is used in separate program units and is not integrated with a host language. An internal language provides constructs that are embedded within a host language and possibly interleaved with host language constructs.

Ultimately, technology supporting LOP must provide ways of integrating new language features with a host language [2]. This paper describes the general features of such technology and shows how they are implemented in the language system XMF. The implementation takes the form of a novel parsing engine that can be embedded in other language systems. The paper concludes by analysing the shortcomings of the XMF mechanisms and proposing how these can be addressed by implementing the features in Java.

2 Language Features for DSLs

Many languages provide some form of interface to relational databases and SQL. Often this is achieved using strings to represent SQL; however, if the language is embedded as a DSL then values from the surrounding program can be used and tool support can check type information. For example the following produces a collection of employee names for those less than a given age:

```
Vector<String> youngEmployees(int ageLim) {
  @select age,name from Employees
    where age < ageLim {
    produce name;
  }
}
```

Consider processing text that represents customer sales transactions. Often this information is held in a text file and must be dissected before it can be processed. A convenient way to process this information is via a DSL, in this case specified externally to the main application, that specifies the format of the data:

```
reader CallReader
  map(SVCL,ServiceCall)
    4-18:CustomerName
    19-23:CustomerId
    ...
  end
  map(USGE,Usage)
    ...
  end
end
```

Finally, frameworks often require additional information to be supplied together with the source or binaries for an application. A typical example of this is J2EE which requires information to be added to Java classes describing how they are to be used. Rather than mark up the existing language and supply the information via XML files, it is more convenient to modify the externally visible language constructs. For example, the following is a Java class definition with methods using a language designed to support enterprise beans:

```
@enterprisebean Calculator isStateless {
  remotelocal double calculate(...) {
    ...
  }
  remote String getServerInfo() {
    ...
  }
}
```

What kind of technology is required to support the definition of these features? It is possible to support language extension via pre-processing. However, it is difficult to embed the features properly since the pre-processor does not have access to the language analysis and manipulation features of the host language tools. It is much more effective to embed the language extension mechanisms inside the host language.

Each of the examples given above requires the syntax of the host language to be extended in some way. In some cases, e.g. the bean example, extensions will be fairly trivial wrappers around existing constructs. In some cases the extensions will be completely external languages as in the text dissection example. However, the SQL example shows that ultimately, syntax must be interleaved; Java statements and expressions can occur within select-statements that occur within Java expressions and statements etc.

Ideally, extensible language technology will allow new language constructs to be defined as modular units in terms of their syntax and semantics. Such units should be portable between different users of the technology so that DSL developers can easily distribute the language definition without having to also distribute the supporting technology.

New language features must integrate with the host language in terms of analysis and transformation. For example the SQL example above makes reference to the argument supplied to the method. The host system should be able to check new constructs for unbound identifiers and type inconsistencies. Furthermore, new language constructs should be able to transform themselves into existing constructs or compile themselves directly in order to implement the feature. All of these activities imply that the abstract syntax of the language has a standard interface that can be implemented and extended by DSL constructs.

3 XMF

XMF is a system that supports language oriented programming. It provides many of the technologies required to support DSL definition, in particular the ability to embed language definitions within the host language and to interleave new language constructs with existing constructs. This section describes the key aspects of XMF that support LOP. The following section describes how they are implemented and therefore how they could be integrated within other language systems. The final section of this paper analyses the features in term of their shortcomings and proposes a mechanism for extending Java for LOP based on an extension of XMF.

Suppose that we have a class `Sig` that defines a simple type signature. Given an instance `s` of `Sig`, it is possible to check that a given table `t` conforms by performing `s.check(t)`. We could require all uses of tables to conform to the following pattern:

```
  let t = Table(3)
  in t.put("x",100);
     t.put("y","Fred");
     t.put("z",true);
     s.check(t);
     ...
```

However, programmers can easily forget or purposfully omit the check. It would be better to have a language construct for table construction that enforces the check:

```
parserImport Table;
...
let t =
  @Table(s)
    x = 100;
    y = "Fred";
    z = true
  end
in ...
```

This new construct is embedded within the surrounding XMF language and can be used wherever an expression is expected. The new construct can be implemented within XMF using syntax classes. Each syntax class defines a grammar for the concrete syntax. The grammar synthesizes an abstract syntax construct that implements a syntax interface. In the case of `Table`, define below, the grammar uses quasi-quotes to synthesize an instance of existing abstract syntax classes defined by XMF:

```
@Class Field
 @Attribute name : String end
 @Attribute value : Exp end
end

context Table
 @Grammar extends XMF.grammar
  Table ::= s = Sig fs = Field* 'end' {
   [| let t = Table(<fs->size.lift()>)
      in <fs->iterate(f e=[|<s>.check(t)|] |
           let nExp = f.name.lift()
               vExp = t.value
           in [| t.put(<nExp>,<vExp>);
                 <e> |]
           end)>
      end |]
  }.
  Sig ::= '(' s = Exp ')' { s }.
  Field ::= n = Name '=' e = Exp ';'
            { Field(n,e) }.
end
```

A grammar can be added to any class in XMF, resulting in a syntax class for a new language construct. The grammar must synthesize a instance of abstract syntax in XMF which guarantees that the host system can manipulate the new language construct using a predefined interface. The grammar for Table returns an instance of existing abstract syntax classes in XMF and therefore does not need to define any of the interface operations. If it returned an instance of a new class then that would need to define operations such as `eval`, `FV` and `compile`.

A grammar consists of a collection of named clauses (`Table`, `Sig` and `Field` in the example). Each clause body is a sequence of parse elements, each of which produces a value. A parse element is: a call (e.g. `Sig` in the clause `Table`); a binding to a local variable (`s =` and `fs =`); a terminal (`'end'`, `'='`); the repetition of an element (`*`) that produces a sequence of all synthesized values, an action which is an expression inside { and }that can reference any local variables in scope. A grammar may also extend an existing grammar in which case all the clauses from the parent are included in the child.

The grammar for `Table` parses a signature followed by a sequence of fields and returns a let-statement that is constructed in terms of the signature and fields. A signature is any valid XMF expression inside parentheses. A field is a name, followed by `=` and then an expression followed by `;`.

The final `Table` action constructs a let-statement in terms of the signature and fields. Working with abstract syntax can be done by creating instances of abstract syntax classes and linking them together. However, this can be a lengthy process and does not provide the reader with an appropriate visual rendering of the translation. Therefore, the `Table` grammar makes use of XMF quasi-quotes which allow *abstract syntax* to be constructed in terms of *concrete syntax*.

Quasi-quotes are `[|` and `|]` that surround any XMF expression or statement and are based on the Template Haskell feature [13]. The result is an instance of appropriate abstract syntax classes. The quotes are usually used when constructing syntax that conforms to some sort of pattern or template. In this case, there will be fixed parts of the template and variable parts of the template. The variable parts can be inserted by surrounding them with `< >` drop quotes. Therefore `[| x + y |]` is a binary addition expression involving variables x and y, whereas `[| <x> + <y> |]` is a binary addition expression that whose left and right operands are the expressions which are the values of x and y.

`Table` also uses XMF sequences and the `iterate` expression. A sequence in XMF `Seq{h|t}` is built from cons-pairs like Lisp, where `Seq{}` is equivalent to nil. An iterate expression is used to accumulate a value whilst looping through the elements of a list: `s->iterate(x sum = 0 | sum + x)` adds up all the elements of the sequence `s`, building up the result in `sum`.

Once a new syntax class has been introduced to XMF, the language construct can be used in a file providing that the file header includes a `parserImport` declaration for the name-space including the syntax class. Each file is processed with respect to a collection of imported name-spaces, and `parserImport` extends the current list.

Once a syntax class has been imported, concrete instances of the construct can be encountered anywhere the host language permits an at-escape (so-called because the default character that denotes the start of a language escape is @). When an at-escape is encountered, the XMF parser resolves the following reference to produce a grammar via the syntax class and then makes a context switch to use the new grammar. When the new grammar has completed a parse, the XMF parser returns to the original grammar and continues.

4 Implementation

The previous section has described technology for LOP that provides a modular approach to the definition of embedded and external DSLs. The technology is implemented in the XMF system, however the key features are not restricted to that system and could be implemented in any object-oriented based language system.

This section describes the implementation in terms of 3 phases: section 4.1 defines a simple representation for grammars and shows how a grammar is translated into a normal form and then into machine instructions. Section 4.2 describes how LL(1) prediction tables are constructed ready to drive the parseing engine. Section 4.3 defines the parsing engine and how it handles syntax classes.

4.1 Grammar Compilation

An XMF grammar is written in a language called XBNF that extends BNF with: grammar inheritance that allows one grammar to extend another; clause arguments that allow data to be passed between grammar rules; clause actions and local variables that allow data to be synthesized by a parse; cut that provides control over choice points.

The body of each grammar clause is a data value of type `Parse` as defined below:

```
Parse ::=
  Action(Fun)
| And(Parse,Parse)
| At
| Bind(Str)
| Call(Str,[Str])
| EOF
| Float
| Integer
| Name
| Or(Parse,Parse)
| Star(Parse)
| String
| Term(Str)
```

An action consists of a function that is invoked with respect to the current parse context (and can therefore access local variables). The `At` parse element is used as an escape that defines where new language features can be embedded in the host language. Each parse element produces a value that can be bound to a local name using `Bind`. A clause is called and supplied with the values of local variables using `Call`. The elements `EOF`, `Float`, `Integer`, `Name`, `String` and `Term` are used to test the next input token. Alternative parses are expressed using `Or` and sequenced parses are expressed using `And`. The `Star` parse element is used to greedily consume input tokens and to synthesize sequences of results.

A grammar is a collection of clauses that define non-terminals, and a name that designates a starting non-terminal:

```
Grammar == ([Clause],Str)
Clause == (Str,[Str],Parse)
```

The following simple examples will be used to show how compilation works:

```
@Grammar
  S ::= x=Int xs=(',' Int)* '.' {f(x,xs)}.
end
```

which is represented as an instance of `Grammar` as follows (nested `And` trees are shown as a single `And` with a sequence argument):

```
([(S,[],And([Int,
             Bind(x),
             Star(And(Terminal(","),Int)),
             Bind(xs), Terminal("."),
             Action(f)]))],S)
```

Compilation of a grammar is defined as a seqeunce of transformations that produces a compiled grammar containing compiled clauses. The first transformation is to remove uses of the `Star` parse element that greedily consumes tokens and builds lists of synthesized values. Each `Star` element is replaced by a call to a new recursive clause that continually calls itself and builds a list of values from the results or alternatively produces the single value `Nil` (using an action with the same name). The transformation is performed by the function `nf`. The result of performing `nf` on the example grammar is shown as follows:

```
([(S,[],And([Int,Bind(x),
             Call(c,[]),
             Bind(xs),Terminal(","),
             Action(f)])),
  (c,[],Or(Nil,
           And([Term(","),Int,Bind(h),
                Call(c,[]),Bind(t),
                Cons(h,t)])))],
 S)
```

Each clause body is translated into a clause normal form that removes disjuctions by distributing `And` elements over `Or` elements to produce clause bodies that are disjunctive sequences. After translation with `cnf`, the example grammar clause bodies are:

```
[[Int,Bind(x),Call(c,[]),Bind(xs),Action(f)]]
[[Nil],
 [Term(","),Int,Bind(h),
  Call(c,[]),Bind(t),
  Cons(h,t)]]
```

The next step in compilation is to translate clause bodies into sequences of parse machine instructions. Each Parse constructor has a Parse_I equivalent except for Bind that compiles to SetLoc. The following definition shows just the non-obvious instructions:

```
Parse_I ::=
  ...
| Call_I(Str,[Int])
| Cons_I(Int,Int)
| ...
| SetLoc(Int)
| ...
```

A call is transformed into an instruction that names the clause to be called and includes an argument map. An argument `m` map associates argument positions with local variable offsets in the current context. A map will be used as a function from integers to integers. A cons instruction constructs a cons-pair from local variable offsets for the hed and tail. A local variable is updated using its offset in the current context by `SetLoc`.

The instruction compilation is performed by the compiler defined below (only the non-trivial mappings are shown). It takes a parse element and an argument map as arguments:

```
compile : (Parse,[Str]) -> Parse_I
...
compile(Bind(s),ls) = SetLoc(ls(s))
compile(Call(s,ss)) = Call_I(s,map ls ss)
compile(Cons(h,t),ls) = Cons_I(ls(h),ls(t))
...
```

A compiled grammar consists of compiled clauses as defined below. A compiled clause consists of a name, the number of locals and a disjunctive sequence of instruction sequences:

```
CGrammar == ([CClause],Str)
CClause == (Str,Int,[[Parse_I]])
```

The grammar compiler is defined below (`FV` calculates free variables):

```
compile : Grammar -> CGrammar
compile(g) = (compile(cs),n)
  where (cs,n) = nf(g)
compile : [Clause] -> [CClause]
compile(c:cs) = compile(c):compile(cs)
compile : Clause -> CClause
compile(n,as,p) =
  let pss = cnf(p)
      vs = FV(p) then
      ls = removeDups(as + vs) then
      iss = map(\ps.map(\p.compile(p,ls))ps)pss
  in (n,len(ls),iss)
```

The result of compiling the example grammar is as follows:

```
([(S,2,[[Int_I,SetLoc(1),
         Call_I(c,[]),SetLoc(2),
         Terminal_I("."),
         Action_I(f)]]),
  (c,2,[[Nil_I],
        [Term_I(","),Int_I,SetLoc(1),
         Call_I(c,[]),SetLoc(2),
         Cons_I(1,2)]]))],
 S)
```

4.2 Parsing Tables

The XBNF parse is LL(k) which means that it can cope with grammars that require lookahead in the input tokens. They are not as efficient as LL(1) grammars that do not require any lookahead. A standard way to parse an LL(1) grammar is to use a prediction table in which clause bodies can be indexed by clause names and tokens.

XMF tries to achieve the best of both worlds by compiling XBNF grammars to prediction tables for LL(1) parsing but allows the entries in the table to be ambiguous. In the case that the entry is not ambiguous, the parse is very efficient. Otherwise, the parse maintains a stack of choice points that can be used if the wrong alternative is chosen. This section gives an overview of the approach.

A compiled grammar is used to construct a preduction table in two phases. Firstly, a clause table is constructed containing information about a clause: whether one of the bodies allows a parse to succeed without consuming any input; the set of tokens that start a successful parse; the set of tokens that follow a successful parse of the clause. Secondly, the clause table is used to construct a prediction table.

4.2.1 Clause Tables

A clause table has the following type definition:

```
CTable == Str -> (Bool,[Str],[Str])
isNullable : (CTable,Str) -> Bool
setNullable : (CTable,Str) -> CTable
first : (CTable,Str) -> [Str]
addFirst : (CTable,Str,[Str]) -> CTable
follow : (CTable,Str) -> [Str]
addFollow : (CTable,Str,[Str]) -> CTable
```

It is populated by processing a compiled grammar. Each compiled clause in the grammar is analysed in turn and the table is updated with any new information based on the current state of the clause table.

Each compiled clause has a disjunctive collection of bodies consisting of machine instructions. Therefore, a compiled grammar can be thought of as having a collection of clauses of the form:

```
C ::= X Y Z ...
```

Each clause is analysed with respect to three properties: nullable, first sets and follow sets. A clause is nullable when every instruction is nullable:

```
nullable : (Parse_I,CTable) -> Bool
```

The first set of a clause determines the tokens that predict the use of the particular clause body. The first set of each clause is gradually constructed by analysing the instruction prefixes: X, X Y, X Y Z of a clause definition. If every instruction is nullable then the instruction that follows the prefix determines an element for the first set of the clause.
The follow set of a clause is used to calculate those tokens that predict an empty parse for a clause. Consider the following:

```
C ::= X Y Call_I(D,[]) G H I
```

If `G H I` are all nullable then the follow set of `C` is added to the follow set of `D`. If `G H` is nullable then add the first set of `I` to the follow set of `D`.

The clause table for the example grammar is shown below:

```
ctable(S) = (false,[Int],[])
ctable(c) = (true,[","],["."])
```

4.2.2 Prediction Tables

A prediction table maps clause names and tokens to sequences of parse instructions. The parser uses the table when a clause is called. The type of the prediction table is defined below:

```
PTable == (CGrammar,Str,Str) -> [[Parse_I]]
extend:(CGrammar,Str,[Str],[Parse_I],PTable)
       ->PTable
```

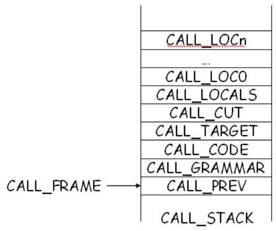
The predict operation is supplied with each clause name n and body is; the operation extends the prediction table by associating the clause and each token with the instructions:

```
predict : (CGrammar,Str,[Parse_I],CTable,Ptable)
            -> Ptable
predict(g,n,is,ct,pt) =
  if every(\i. isNullable(i,ct))is
  then extend(g,n,first(n,ct) + follow(n,ct),pt,is)
  else extend(g,n,first(n,ct),pt,is)
```

4.3 Parsing Engine

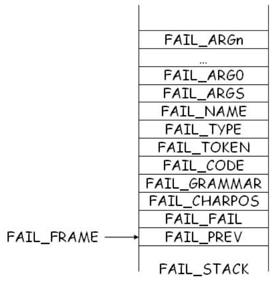
The XMF parser is implemented as a machine that processes a compiled grammar with respect to a tokenizer. The tokenizer is responsible for processing an input stream of characters and transforming them into tokens. Each token has a type which XMF defines to be: INT; FLOAT; STR; NAME; EOF; SPECIAL. When asked for the next token, the tokenizer returns both the type and the string of characters.

The machine uses two stacks: the call stack and the fail stack. The call stack is used to maintain the context of the currently executing clause. It is implemented as an array CALL_STACK with a CALL_FRAME current frame pointer. The elements of a frame are indexed by constant offsets:



The contents of the frame are: the previous call frame (CALL_PREV); the currently executing grammar (CALL_GRAMMAR); the compiled instructions as a list (CALL_CODE); the current value of 'self' (CALL_TARGET); the choices to throw away in the event of cut (CALL_CUT); the number of local variables (CALL_LOCALS); the locals (CALL_LOC0 ...n).

The fail stack is used to record choice points:



A choice point is encountered when a clause is called during a parse and when the prediction table cannot reduce the choice of alternatives down to a single clause body. In that case the alternative bodies, supplied arguments and parsing context are saved on the fail stack.

The current frame (FAIL_FRAME) contains the following information: a pointer to a call frame (FAIL_PREV); a pointer to the previous fail frame (FAIL_FAIL); the character position used to unwind the tokenizer (FAIL_CHARPOS); the grammar that was current when the choice-point was encountered (FAIL_GRAMMAR); a sequence of alternative instruction lists (FAIL_CODE); the token that was current when the choice point was encountered (FAIL_TOKEN and FAIL_TYPE); the number and values of the arguments (FAIL_ARGS, FAIL_ARG0...n).

Figure 1 shows the definition of the parsing engine main loop. The engine is started by supplying some arguments, a target object, the grammar and some instructions (usually `Call_I(NT,[])`). An initial call frame is created on the call stack and the arguments are inserted into the frame. The engine is primed with a new token by calling `nextToken()` which just updates the global variables `TOKEN` and `TYPE` using `TOKENIZER`.

The engine proceeds by entering a loop and processing the next instruction at the head of `CALL_CODE`. If the call stack is ever exhausted then the synthesized result is left in `RESULT`. Otherwise some error was encountered via the `ERROR` flag.

If the instruction list in the current frame becomes exhausted then the frame is popped via `CALL_PREV`. Otherwise the head instruction is popped and the machine proceeds by case analysis on the instruction. The rest of this section describes each of the main instructions in turn.

The instruction `At_I` is used to switch language context. The input stream will contain a path that references a class in the current context. The operation `resolveGrammar` is used to read the path from the tokenizer and return the grammar. The machine proceeds by calling the starting non-terminal of the grammar with 0 arguments.

The instruction `Call_I(n,as)` causes the clause named `n` to be called. The `as` component is an argument map that describes how to map locals in the current stack frame to arguments in the call frame to be constructed when calling the clause named `n`.

5 Analysis

5.1 Analysis of Technology

The XMF technology for LOP has been tried and tested on a number of industrial applications (including a large meta-modelling toolkit XMF-Mosaic; it is open source and is available at [1].

The LOP features of XMF have proved to be effective at defining language constructs. They provide good modularity and control over the visibility and use of the constructs via syntax-classes contained in name-spaces. Working with syntax has been greatly facilitated through the use of quasi-quotes. Code templates constructed using quasi-quotes are much easier to work with than the equivalent raw data structures. A weakness of the syntax manipulation mechanism is the lack of support for hygene [12].

XMF abstract syntax defines an interface that deals

```
procedure parse(args,target,grammar,code,context) {
  CALL_FRAME := pushCall(-1,grammar,code,target,-1,len(args));
  for i = 0 to len(args) { CALL_STACK[CALL_FRAME + LOC0 + i] := args[i]; }
  nextToken();
  while CALL_FRAME >= 0 & !ERROR {
    if CALL_STACK[CALL_FRAME + CODE] = Nil;
    then popFrame();
    else
      case popInstr() {
        At_I ->
          CGrammar g = resolveGrammar(TOKENIZER,context);
          call(g,startNT(g),[]);
        Call_I(n,argMap) -> call(CALL_STACK[CALL_FRAME + CALL_GRAMMAR],name,argMap);
        Int_I ->
          if TYPE = INT
          then RESULT := asInt(TOKEN); nextToken();
          else fail();
        Action_I(f) -> RESULT := f();
        SetLoc(i) -> CALL_STACK[CALL_FRAME + LOC0 + i] := RESULT;
        Cons_I(h,t) ->
          RESULT := new Cons(CALL_STACK[CALL_FRAME+LOC0+h],CALL_STACK[CALL_FRAME+LOC0+t]);
        Terminal_I(t) -> if TOKEN = t then nextToken(); else fail();
      }
  }
}
procedure call(g,name,argMap) {
  CClause c = clauseNamed(NT,g);
  int locals = clauseLocals(c);
  int cp = charPos(TOKENIZER);
  Object t = CALL_STACK[CALL_FRAME + TARGET];
  List(List(Parse_I)) code = ptable(g,NT,TYPE);
  int prev = CALL_PREV;
  if isEmpty(code)
  then failCall();
  else {
    CALL_FRAME := pushCall(CALL_FRAME,g,head(code),t,FAIL_FRAME,locals);
    for i = 0 to len(argMap) {
      CALL_STACK[CALL_FRAME + CALL_LOC0 + locals,CALL_STACK[prev + LOC0 + argMap[i]]];
    }
    if len(iss) > 1
    then pushFail(prev,FAIL_FRAME,cp,g,tail(code),t,FAIL_FRAME,TOKEN,TYPE,name,argMap);
  }
}
```

Figure 1: The Parsing Engine

with execution and compilation. XMF is a dynamically typed system and therefore there is no typing aspect to this interface. For more widespread use, this interface would need to be more extensive and to incorporate features such as static typing and support for IDE operations such as refactoring.

A final weakness of the XMF-based technology is that XMF is a non-standard language. Its features grew out of the OCL which is part of UML and as such it contains a number of unusual features. In addition, XMF is a dynamic open meta-circular language (in the sense of Smalltalk); this makes static analysis diffficult. For LOP and DSLs to be effective, applications and their languages should be portable and easily accessible without having to distribute implementation technology.

5.2 Related Work

[7] provides a good overview of the field of DSLs and describes how LOP is supported by the programming language Converge. There are a number of pre-processing systems that have been proposed for implementing DSLs, for example TXL [8] and MetaBorg [10]. A shortcoming of all pre-processing systems is that the support for embedded DSLs is often weak and not integrated with the development tools for the host language. The mechanisms proposed in this paper addess these directly by requiring that the language extension mechanisms be embedded within and fully integrated with the host language. The ASF_SDF language [9] allows DSLs to be defined in similar ways to the XMF system, however, like XMF, the host language is not mainstream. Fortress [11] aims to be a mainstream language and has included specific features in its language definition to allow escapes of the type described in this paper.

5.3 Proposal for Java

The paper [2] describes how the syntax of Java could be extended with syntax-classes. This paper takes the proposal further by showing how the technology could be implemented. Languages defined in Java would be portable. In order to use new language features it would simply be necessary for a Java compiler (and optionally run-tine system) to have access to the compiled syntax-classes, which would be written in Java.

There are other systems that support LOP within Java. OpenJava (OJ) [15] supports LOP using a meta object protocol (MOP) that supports syntax expansion. Each class may specify a meta-class that is responsible for expanding its definition. The meta-class uses a standard interface to process its instances. However, OJ does not include a mechanism for extending the concrete syntax of the language.

References

[1] The XMF system at www.cetev.com/XMF.

[2] A. Clark, J. Willans, P. Sammut. Beyond Annotations: A Proposal for Extensible Java (XJ). In proc. Eighth IEE Int. Working Conference on Source Code Analysis and Manipulation (SCAM 08), Beijing, China, 2008. pp 229-238.

[3] Language Oriented Programming. Martin Ward. Software - Concepts and Tools, Vol.15, No.4, pp 147-161, 1994.

[4] M. Mernik, J Herring, A. Sloane. When and How to Develop Domain-Specific Languages. ACM Computing Surveys, Vol. 37, No. 4, December 2005, pp. 316–344.

[5] Steele, G. L. 1998. Growing a language. In Addendum To the 1998 Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (Addendum) (Vancou- ver, British Columbia, Canada). J. Haungs, Ed. OOPSLA '98 Addendum. ACM Press, New York.

[6] C. Braband, M. Schwartzbach. Growing Languages with Metamorphic Syntax Macros. In Proceedings of Workshop on Partial Evaluation and Semantics-Based Program Manipulation, PEPM 2002. ACM.

[7] L. Tratt Domain Specic Language Implementation via Compile-Time Meta-Programming. To appear, ACM Transactions on Programming Languages and Systems (TOPLAS), October 2008.

[8] J. Cordy. TXL - a language for programming language tools and applications. In Proc. LDTA 2004, ACM 4th International Workshop on Language Descriptions, Tools and Applications.

[9] M. van den Brand, J. Heering, P. Klint, P. A. Olivier. Compiling language denitions: the asf+sdf compiler. ACM Trans. Program. Lang. Syst. 24, 4, 334-368. 2002.

[10] M. Bravenboer, E. Visser. Concrete syntax for objects. Domain-specic language embedding and assimilation without restrictions. In Proc. OOPSLA'04,

D. C. Schmidt, Ed. ACM SIGPLAN, Vancouver, Canada.

[11] The Fortress Language Specication. Allen et al. Version 1.0 2008. Available from http://research.sun.com/projects/plrg/ Publications/fortress.1.0.pdf

[12] E. Kohlbecker, D. Friedman, M. Felleisen, B. Duba, Hygienic macro expan- sion. In Symposium on Lisp and Functional Programming. ACM, 151-161.

[13] T. Sheard, S. Peyton-Jones. Template metaprogramming for Haskell. In proc. ACM SIGPLAN Haskell Workshop 2002. ACM Press.

[14] Martin Fowler's blog on Domain Specic Languages. http://www.martinfowler.com/ articles/languageWorkbench.html

[15] OpenJava: A Class-based Macro System for Java. Michiaki Tatsubori, Shigeru Chiba, Marc-Olivier Killijian and Kozo Itano LNCS 1826, Reection and Software En- gineering, , Walter Cazzola, Robert J. Stroud, Francesco Tisato (Eds.), Springer-Verlag, pp.117- 133, 2000.