Language Modules

Tony Clark*

May 14, 2010

A language consists of syntax and semantics. The concrete syntax can be defined using a grammar and actions in grammar rules can provide a language with a semantics. For example, a grammar for a simple integer expression language can evaluate the expressions as the parse progresses. However, fixing the semantics of a language too early means that the language cannot easily be viewed as a reusable *module*.

Why would we want reusable language modules? A key driver is that language engineering has become an increasingly important part of system engineering. Modern programming languages provide facilities for extensibility and *domain specific languages* are developed as part of system architectures; therefore there are increasing opportunities for taking parts of one language and using them as the basis for another language.

A paradigmatic example of a language module is an expression language. Almost all data-related languages need to represent simple expressions, for example languages involving any of the following: initialization of record fields; state-machine guards; procedure arguments; rule actions; event construction; widget positioning. A language system that allows expression sub-languages to be defined and reused in different contexts will increase the quality of the resulting products.

However, direct reuse of a language module is not always realistic. The expected meaning of a module will depend on the context of the language in which the reuse takes place. For example, an integer expression language may fail when division by zero takes place or it may degrade gracefully through the use of exceptions. For example, an integer expression language should be reusable in a context where the underlying data type is changed to rational numbers or floating point numbers.

Therefore, a programming system that supports language modules must provide the user with the ability to define language families or *language factories*. A language factory is a flexible module that can be transformed into members of a set of fixed language modules. A language factory might leave parts of the syntax undefined or fix the sytax and leave the semantics flexible.

This document introduces language modules using some simple examples in XPL and shows how XPL can support language factories. A language module in XPL has the following basic structure and can be used to parse a string written in the language using the parse operation of a grammar:

```
module = {
  syntax(semantics) = {
    // grammar rules
  };
  semantics = {
    // semantic rules
  }
}

parseL(L,s) = L.syntax(L.semantics).parse(s)
```

A language module consists of a two sub-components: a syntax description and a semantics description. The syntax description is a function that accepts a record of semantics and returns a grammar that uses the semantics as part of the grammar actions. The semantics descriptions is a record of functions. The following is an example of a simple integer arithmetic laguage module:

* http://www.eis.mdx.ac.uk/staffpages/tonyclark/

```
arithInfixSyntax(semantics) = {
  arith     -> whitespace a=atom tail^(a);
  atom      -> i=int | '(' a=arith ')' { a };
  whitespace -> (32 | 10 | 9 | 13)*;
  int       -> n=(['0','9'])+ { semantics.int(n) };
  tail(l)   -> o=op r=arith {
    semantics.binExp(l,o,r)
  };
  tail(l)   -> { l };
  op        -> whitespace ('/' | '*');
}

arithSemantics = {
  binExp(left,op,right) =
    case op {
      '*' -> left * right
      '/' -> left / right
    };
  int(cs) = asInt(cs)
}

arithInfixL = {
  syntax = arithInfixSyntax;
  semantics = arithSemantics}

parseL(arithInfixL,'10␣/␣2')
=> 5
```

Since the syntax and semantics has been defined independently, we can construct other languages based on the semantics of simple expressions:

```
arithPostfixSyntax(semantics) = {
  postfix   -> a=atom tail^(a);
  tail(l)   -> r=atom o=op n={
    semantics.binExp(l,o,r)
  } tail^(n);
  tail(l)   -> { l };
  atom      -> int | '(' a=postfix ')' { a };
  whitespace -> (32 | 10 | 9 | 13)*;
  int       -> whitespace n=(['0','9'])+ {
    semantics.int(n)
  };
  op        -> whitespace ('/' | '*')
}

arithPostfixL = {
  syntax = arithPostfixSyntax;
  semantics = arithSemantics}

parseL(arithPostfixL,'10␣2␣/')
=> 5
```

Notice that the reuse has been achieved by making a grammar the return value (and therefore a first-class value) of a function. The grammar controls the use of a package of semantic operators, however the grammar has no knowledge, other than the interface, of the implementation of the semantic package. This allows us to vary the semantics while leaving the syntax the same. For example, suppose that we change the semantic domain of the languages defined above to be rational numbers represented as records:

```
rationalSemantics = {
  binExp(l,op,r) =
    case op {
     '*' -> { num=l.num * r.num;
              den=lft.den * r.den }
     '/' -> rationalSemantics.binExp(l,'*',
             { num=r.den; den=r.num })
    };
  int(n) = {
    num=asInt(n);
    den=1
  }
}
```

The new semantics can be combined with the two previous syntaxes (since the interface is compatible with them) to form two new languages:

```
rationalInfixL = {
  syntax=arithInfixSyntax;
  semantics=rationalSemantics
}

rationalPostfixL = {
  syntax=arithPostfixSyntax;
  semantics=rationalSemantics
}

parseL(rationalInfixL,'10␣/␣2')
=> {num=10;den=2}

parseL(rationalPostfixL,'10␣2␣/')
=> {num=10;den=2}
```

Things get even more interesting when we define language module transformations. Suppose that we want to keep the history of execution (perhaps as

input to a debugger). The execution history of expressions can be represented as a tree. The leaves of the tree are the atomic data values and the nodes of the tree are labelled with the operators that are used to combine the values produced by sub-trees. It does not matter what the underlying data values are: they could be integers or rational numbers (or execution histories). A suitable language module transformation is:

```
history(L) = {
  syntax = L.syntax;
  semantics = {
    binExp(l,op,r) = {
      calc = op;
      children = [l,r];
      value = L.semantics.binExp(l.value,op,r.value)
    };
    int(cs) = {
      calc = 'int';
      children = [];
      value = L.semantics.int(cs)
    }
  }
}
```

Any language L that implements the binary expression interface can be transformed to a language over expression histories:

```
arithInfixHistL = history(arithInfixL)
arithPostfixHistL = history(arithPostfixL)
rationalInfixHistL = history(rationalInfixL)
rationalPostfixHistL = history(rationalPostfixL)

parseL(arithInfixHistL,'10␣/␣2')
=> {calc=/;
    children=[
      {calc=int;children=[];value=10},
      {calc=int;children=[];value=2}];
    value=5}

parseL(rationalPostfixHistL,'10␣␣2␣/')
=> {calc=/;
    children=[
      {calc=int;children=[];value={num=10;den=1}},
      {calc=int;children=[];value={num=2;den=1}}];
    value={num=10;den=2}}
```

The language that is produced by the language module transformation also conforms to the required expression interface. Therefore, the transformation can be applied repeatedly. Of course, this does not make much sense in this particular example, however it is an important feature, showing that module transformations are composable:

```
parseL(history(arithInfixHistL),'10␣/␣2')
=> {calc=/;
    children=[
      { calc=int;
        children=[];
        value={calc=int;children=[];value=10}},
      { calc=int;
        children=[];
        value={calc=int;children=[];value=2}}];
    value={
      calc=/;
      children=[
        {calc=int;children=[];value=10},
        {calc=int;children=[];value=2}];
      value=5}}
```

This article has introduced language modules that separate syntax definition and semantics. It has shown that, providing both syntax and semantics components are first-class values in a meta-language, then the language module approach allows language reuse including language module transformations.