UNIVERSITY OF DUBLIN
TRINITY COLLEGE

DEPARTMENT OF COMPUTER SCIENCE

Modular parsers

Simon Dobson

29 September, 1998

Abstract

Traditional parser generators are designed to produce highly optimised parsers for grammars known in their entirety ahead of time. There are some circumstances where it is desirable to construct a parser from grammar fragments, for which the flexibility of combination is more important than the speed of the final parser. We describe a system of parser components, generating parsers for grammar fragments and which may be combined using a small set of parser combinators to produce a final parser.

1. Introduction

Parsing is one of the most common operations in computing, lying at the heart of all systems which must manipulate text-based data encodings. The parser's task is to convert a linear text-based representation into a structured representation suitable for use within algorithms. Typically this results in a "parse trees" reflecting the nested structure of the textual encoding.

Traditional parser generators such as `lex`/`yacc` and `JavaCC`/`jjtree` are designed to produce highly optimised parsers for grammars known in their entirety ahead of time. In the case of a programming language, for example, the parser generator takes a description of the language's grammar (its *concrete syntax*) and produces a tree representing the structure of the program (its *abstract syntax*) for use by other components of the compiler.

There are some circumstances, however, where it is desirable to construct a parser from grammar *fragments* rather than from a complete grammar. Whilst there are complex dependencies between fragments, in terms of the support they require from other fragments in order to function properly, there is also considerable independence which may be exploited to increase modularity, flexibility and re-use. Fragments may be re-used in several different contexts, and the final grammar may be changed by changing its component fragments.

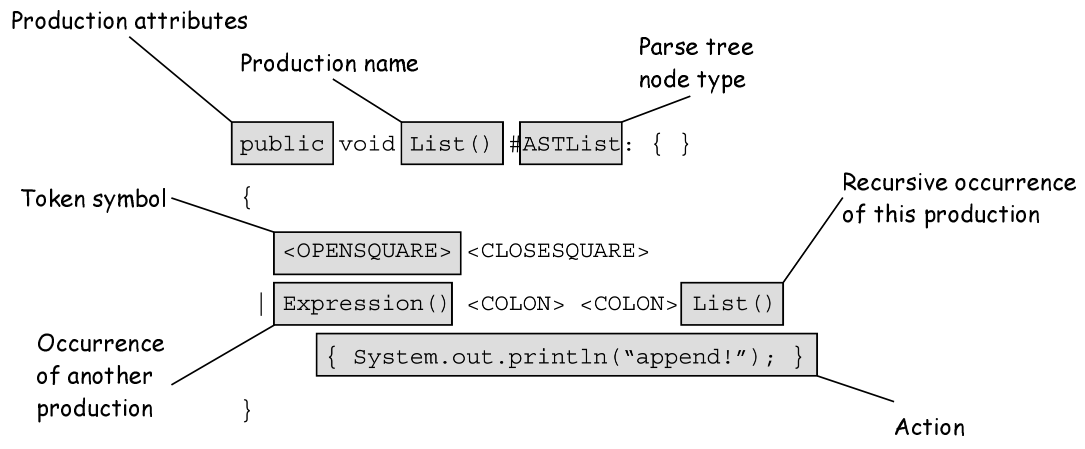
In this paper we present a system for constructing modular, recursive descent parsers. We allow grammar fragments to be compiled into *parser components* representing a parser for that fragment. These components may be combined using *parser combinators* into a "full" parser. The combinators allow new syntactic structures to be integrated into the structure of a basic grammar, extending the definitions of its elements. Our presentation is very pragmatic, focussing on the structure of parsers and the way the combinators affect these structures: we leave a more abstract treatment for possible future work.

Section two presents a brief overview of parsers and parsing. Section three outlines the structure of parser components. Section four describes the parser combinators and their effects in terms of the parser structure. Section five presents two examples of "fragmentary" grammars. We conclude with a discussion of the impact of modular parsing on programming language design.

2. Parsers and parsing

A parser converts text into trees according to some grammar. A grammar is composed of two distinct but related elements, variously called *terminals* and *non-terminals* or (more commonly) *tokens* and *productions*. We shall use the latter terms here.

A token is an atomic piece of a grammar, usually representing a short text string. Typical tokens might include reserved words, identifiers, numbers *etc*. A single token is usually able to represent a large set of strings, so for example all identifiers might be represented by a token `<IDENTIFIER>` which carries the actual string (the token *image*) along with it. A process called *tokenisation* or *lexical analysis* converts the parser's input text representation into a sequence of tokens. Tokens are usually specified using regular expressions over strings: the tokeniser repeatedly scans the head of the text stream looking for a matching token, and uses the matched portion as the token's image.



```
public void List() #ASTList: { }
{
    <OPENSQUARE> <CLOSESQUARE>
  | Expression() <COLON> <COLON> List()
      { System.out.println("append!"); }
}
```

Figure 1: An example production in `JavaCC/jjtree` syntax

A production is a combination of tokens and other productions, again usually using regular expressions. A production might, for example, define a list as either an empty list or an element prepended to a list: the explicit syntactic parts will be identified as tokens, with the more structured parts identified by productions (in this case, a recursive use of the list production). The parse tree is constructed by associating a node of the tree with some or all productions: when a production is recognised, it generates the associated node with any nodes generated as part of its recognition as children. (This approach is a little too simple for some parsers, which add explicitly-coded actions to be taken during recognition.) Figure 1 shows the elements of a production described using JavaCC syntax.

The most common approach to parser generation recognises the so-called LALR(1) grammars. A true LALR(1) grammar has the property that the parser can always proceed based on the next token in the token stream, and the choice made determines the acceptable structure from there on. This means that ambiguities involving more than one choice stemming from a common token must be resolved early, either by re-organising the grammar (which can make it difficult to generate the desired parse tree) or by relaxing the one-token requirement and looking further ahead in the token stream. The parser never back-tracks once a decision has been made, so any unexpected tokens must be syntax errors. LALR(1) parsers can be table-driven and so be made very compact and efficient.

By contrast, *recursive descent* parsers perform arbitrary back-tracking: there may be many possible alternatives for a production, and each is tried in sequence until one succeeds or all fail. This allows easier handling of ambiguous grammars (as long as the ambiguity can be resolved eventually), but complicates error reporting as it is hard to differentiate between a genuine syntax error and an alternative (but still legal) syntax which will be recognised by another production. The parsing algorithm follows the recursive structure of the grammar, which is conceptually simpler and lends itself to analysis but may be less efficient than the table-driven approach.

3. Parser components

In overview, a parser component is a self-contained recursive-descent parser generated from a grammar fragment, which defines some tokens and productions in terms of themselves and some other, externally-defined tokens and productions. A component with no imported elements is what is traditionally meant by "a parser"; components with imports must have them resolved by combination with other parser components before the parser can be used.

In this section we describe the run-time structure of parser components and parser description files as used and generated by our parser generator, using them to illustrate the underlying ideas. This also gives us a concrete syntax with which to present parser combinators in the next section.

3.1 Representing tokens and productions

We represent tokens and productions using objects, with the structure of the objects reflecting the parsing structure. We use objects throughout the representation, for both production and token regular expressions. This makes the system a little inefficient but clearer to explain.

Regular expressions are represented using a tree of objects (figure 2). The leaves of the tree are objects representing literals such as tokens; internal nodes represent the various combinators for building the regular expressions such as sequencing, alternatives, repetitions *etc*. A token in a production is tested against the head of the token stream, and matches if the head token is an instance of the token being asked for.

The matching algorithm is very simple. The algorithm maintains a token stream of tokens derived from the input text by tokenisation, and a stack of abstract syntax tree nodes produced during matching. To match a production, it first marks the current positions of the token stream and node stack. It then evaluates the regular expression for the production. If the expression is matched, it consumes the tokens from the token stream; if it fails, it rolls the token stream and node stack back to the start.

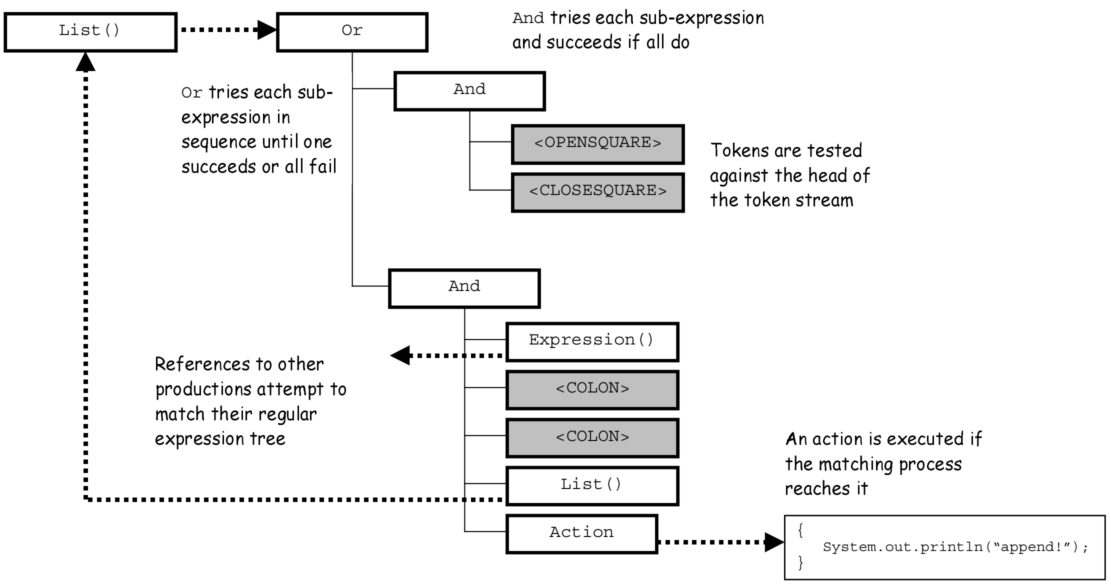

List()
Or
And tries each sub-expression and succeeds if all do
Or tries each sub-expression in sequence until one succeeds or all fail
And
<OPENSQUARE>
<CLOSESQUARE>
Tokens are tested against the head of the token stream
And
Expression()
References to other productions attempt to match their regular expression tree
<COLON>
<COLON>
List()
An action is executed if the matching process reaches it
Action
{ System.out.println("append!"); }

Figure 2: The object structure of regular expressions

Action elements act as wrappers around a piece of code. If the matching algorithm encounters an action, it executes it. It is important to realise that a production may fail after an action has been executed, causing a roll-back, so any code must be written so that roll-backs do not cause problems.

Each production may have an associated abstract syntax node type. If a production succeeds, this type is instanciated and any nodes pushed onto the stack during matching are popped off the stack and added as children of the new node. The new node itself is then pushed onto the stack. For more complex nodes, an action may be used to construct a node manually or manipulate the number and structure of the nodes on the stack.

3.2 Imports

The major difference between a standard parser and a modular one is the existence of imports representing tokens and productions which will be defined by another parser component. Using the list example, the `List()` production uses several tokens and another production. If we consider this production to be a grammar fragment, some or all of the tokens and production may be imported. The list fragment may then be used to define a syntax for lists over *any* sort of expression using any concrete syntax for the tokens. Figure 3 illustrates this principle: the `<COLON>` token and the `Expression()` production are imported, and the fragment defines list construction over *any* expression.

Within a parser component an imported element is represented by a "placeholder" token (or production) which wraps-up another element. The parser combinators described in the next section "fill out" the placeholder with the token when the import is resolved.

```
import <COLON>, Expression();

TOKEN:
{
   < OPENSQUARE:  "[" >
 | < CLOSESQUARE: "]" >
}

public void List() : { }
{
   <OPENSQUARE> <CLOSESQUARE>
 | Expression() <COLON> <COLON> List()
}
```

Figure 3: Lists as a grammar fragment

3.3 Exports

The dual of importing some productions is making tokens and productions available for use by others. Exporting subsumes two distinct functions: making a production available to be imported by other parser components, and allowing components to extend the definition of a production.

Importing is controlled by visibility modifiers, similar to those used in Java to control method accesses[3]. A `public` production is available for import by other parser components; `protected` and `private` productions are not available. (These modifiers are also propagated into the definitions of the variables in the parser component.) This allows a parser component to present a slightly abstract interface to other components by hiding some productions.

```
public export priority void Expression() : { }
{
   List()
}

private void List() : { }
   { … }
```

Figure 4: Visibility modifiers

Extending productions is discussed in more detail below, but it is often not desirable to allow components to extend arbitrary productions. An extra `export` visibility modifier marks a production which can be extended. A further `priority` modifier indicates a production which takes precedence when combined, an idea described in the next section.

Figure 4 illustrates the visibility modifiers. The `Expression()` production is declared to be public (available for import by other components), exported (available to be extended by other components), and having priority (see later). The `List()` production is not available to other components at all.

3.4 The parser component description

The final piece of a parser component is a description of the structure of the component to be used by the combinators. The parser description maintains a list of all the tokens and productions defined within the component, their visibility properties, and any imported elements which must be resolved.

There are substantial similarities between the parser description structure and the symbol table in an object file – indeed, the parser combinators perform substantially the same function as a link loader for a compiler.

3.5 The final structure of a parser component

The parser description files for parser components use a large sub-set of the syntax used by `JavaCC/jjtree`. Running the parser component generator across such a file produces a class representing the parser. (At present these classes are in Java.) This class may then be instanciated as required to produce a parser for the grammar fragment.

4. Parser combination

Defining parser components allows us to represent parts of grammars in a modular fashion. What remains is a way of combining the fragments together into a fully-defined grammar.

The parser component system provides a single basic way to compose parsers: a base parser component may have a fragment parser component appended to it. This append operation it actually defined in terms of some additional, more fundamental combinators on parsers, defining the behaviour of imported elements, additional elements and merging. We shall describe these fundamental operations in isolation first, and then use them to define the append operation.

4.1 Importing

Importing elements involves locating the actual definition of the named element and re-directing any imported references to this definition. This applies equally to productions and tokens.

As explained above, a parser component represents an imported element as a placeholder. Resolving the import involves filling-out this placeholder with the resolved definition (figure 5) by locating the named element in the parser description table.

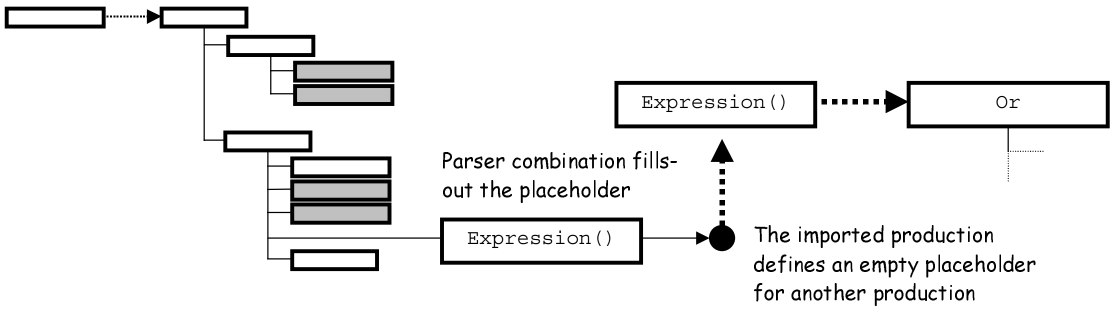
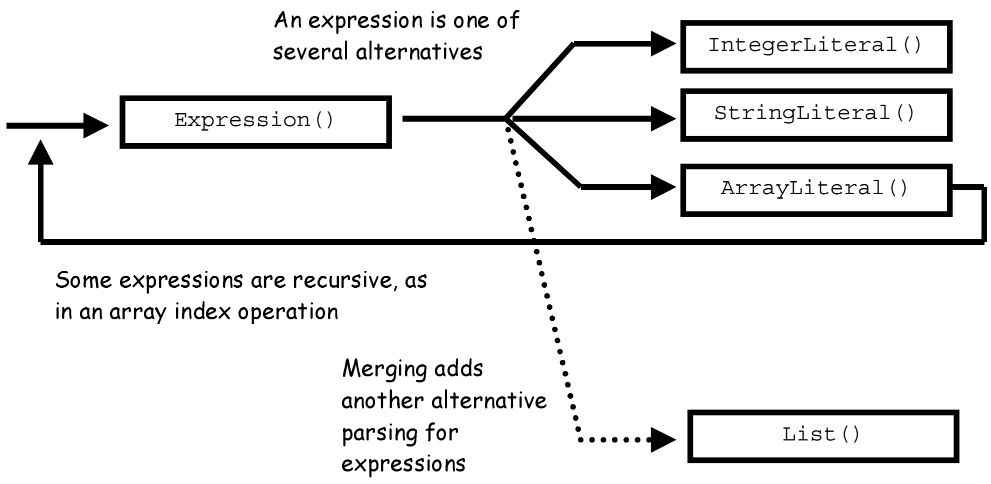

Expression()
Or
Parser combination fills-out the placeholder
Expression()
The imported production defines an empty placeholder for another production

Figure 5: Resolving an imported production

4.2 Merging productions

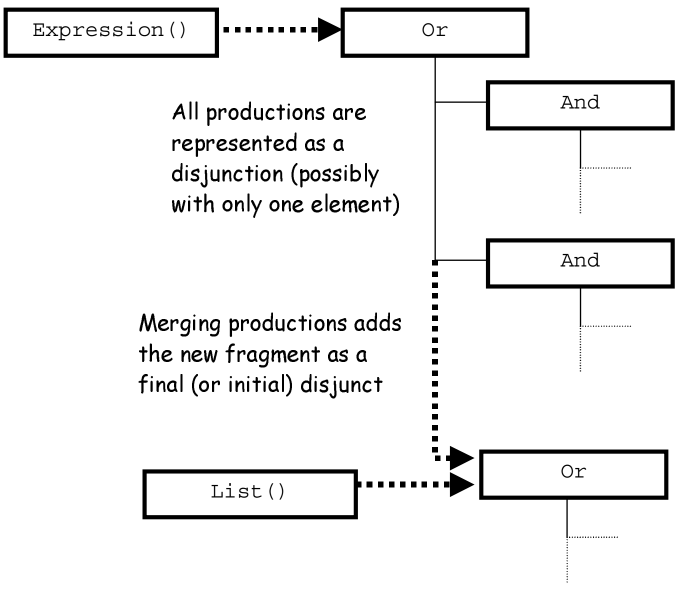
The final part of parser combination involves extending the base components' productions with new alternatives. A list, for example, is generally treated as an expression as well as being composed of other expressions. In the list fragment, we might want to declare that the `List()` production should be accepted as an additional `Expression()`. This may be accomplished by *merging* the `List()` production with the base's `Expression()` production (figure 6).



An expression is one of several alternatives
Expression()
IntegerLiteral()
StringLiteral()
ArrayLiteral()
Some expressions are recursive, as in an array index operation
Merging adds another alternative parsing for expressions
List()

Figure 6: Parsing a merged production

Structurally, merging is a simple process. All production regular expressions are represented as a disjunction – that is to say, the root node of the tree is always an "or". Many productions will have only a single disjunct. To merge a fragment's production to a base's, we simply add the fragment's production regular expression *en masse* to the base's expression (figure 7). Any "self" references in the fragment are re-directed to the overall merged production.



Expression()
Or
All productions are represented as a disjunction (possibly with only one element)
And
And
Merging productions adds the new fragment as a final (or initial) disjunct
List()
Or

Figure 7: Merging productions

In general one wants to preserve the underlying structure of the base grammar. By default, merging adds the fragment's production as a final disjunct. This has the important property that merging is conservative: for all strings parsed by the base, the combined parser will generate exactly the same parse tree, because the base's productions will all be tried before the fragment's productions.

There are circumstances, however, where this is unacceptable, and a fragment wishes to insert its production before those of the base. In the case of these priority productions, merging adds the fragment's production as the initial disjunct of the base's regular expression. This allows a priority production to take precedence over base productions, but also means that the fragment's production can interfere with the way in which the base parses strings, removing the conservative property.

4.3 The `append` operation

We may now show how the basic parser appending operation is defined for two parser components in terms of these basic operations.

Append takes two parse components, the *base* and the *fragment*, changing the base to reflect the additions from the fragment[1]. It walks over the parser description structure of the fragment, applying whichever combinator is appropriate for each element:

- An exported production is merged with the corresponding production in the base if that production is exported too. If it is a priority production it is prepended to the base production's regular expression; otherwise (the default) it is appended. If there is an un-exported production with the same name in the base, the fragment's production masks the base's for future operations but the base production is left unaffected.
- An imported production or token is resolved immediately if there is a corresponding visible element, or added to the import list if not.
- Other productions are added to the productions with the correct visibility.
- An exported token is added to the default token set.

Adding an element may also resolve an import.

There are several error conditions which may occur, mostly relating to symbol table clashes. These throw an exception when detected.

5. Example

To illustrate parser combination, we present a parser for a simple calculator and then add more advanced functions using an additional parser component.

The parser description in figure 8 defines a very simple parser component. The preamble simply wraps-up the parser code into a Java class. Expressions within the

[1] A more functional approach would be to produce a new component. However, the most common arrangement in practice is to append a sequence of components together to produce a final parser, so the default behaviour is more memory-friendly. Parser components are simply classes which may be freely instanciated, so a new class may be created before appending if the more functional behaviour is necessary in a particular case.

calculator consist of numbers, the four arithmetic operators (with equal precedence), and brackets.

```
PARSER_BEGIN(Calculator)

import ie.tcd.cs.vanilla.grammar.*;

public class Calculator extends ParserComponent {
   public Calculator() { }

   VS_PARSER
}

PARSER_END(Calculator)

SKIP:
{
   < WHITESPACE:         (" " | "\t" | "\n")+ >
}

TOKEN:
{
   < NUMBER:             ["0"-"9"]["0"-"9"]*
                         ("." ["0"-"9"]+)? >
 | < PLUS:               "+">
 | < MINUS:              "-">
 | < STAR:               "*">
 | < SLASH:              "/">
 | < OPEN:               "(">
 | < CLOSE:              ")">
}

public export void Expression() : { } {
   <NUMBER>
 | Expression() Operator() Expression()
 | <OPEN> Expression() <CLOSE>
}

public void Operator() : { } {
   <PLUS> | <MINUS> | <STAR> | <SLASH>
}
```

Figure 8: The basic calculator syntax

Compiling this grammar as a parser component will result in a file `Calculator.java` implementing the parser. A sample of the code produced is shown in figure 9. Each of the disjuncts in the grammar gives rise to an `addDisjunct()` call. The details of the production are then added to the parser's export table.

Suppose we now wish to add two common scientific functions to the calculator. Rather than alter the `Calculator` parser, we may generate an additional component which defines the new forms of expressions and append it to the basic parser (figure 10). The component imports the `Expression()` production and the bracket tokens, adds additional tokens for the function names, and defines an additional `Expression()` production to be combined with the underlying production.

```
p = (Or) Expression.getProduction();
p.setName("Expression");
p.addDisjunct(NUMBER);
p.addDisjunct(
   (new And()).add(Expression)
              .add(Operator)
              .add(Expression));
p.addDisjunct(
   (new And()).add(OPEN)
              .add(Expression)
              .add(CLOSE));
desc = new Parser.ElementDescription();
desc.elementType = Parser.ElementDescription.PRODUCTION;
desc.visibility = Parser.ElementDescription.PUBLIC;
desc.priority = false;
desc.exported = true;
desc.imported = false;
desc.production = Expression;
exportTable.put("Expression", desc);
```

Each grammar disjunct gives rise to an object to recognise it

Each token and production has an entry in the component's description table

Figure 9: Some of the code generated for the `Expression()` production

It is important to note that the two parser components are not related by inheritance, but are related by the details of their imports and exports. This maximises the potential – admittedly rather negligible in this example! – for re-using a component to extend other suitable parsers

```
PARSER_BEGIN(Functions)

import ie.tcd.cs.vanilla.grammar.*;

public class Functions extends ParserComponent {
   public Functions () { }

   VS_PARSER
}

PARSER_END(Functions)

import <OPEN>, <CLOSE>, Expression();

TOKEN:
{
   < SIN:      "sin" >
 | < LN:       "ln" | "log" >
}


public export void Expression() : { } {
   FunctionName() <OPEN> Expression() <CLOSE>
}

public void FunctionName() : { } {
   <SIN> | <LN>
}
```

Figure 10: Component adding functions to the calculator

6. Conclusions

We have presented a system of parser components which allow a parser for a grammar to be dynamically constructed from separately-constructed fragments. This piecemeal approach to parser generation encourages re-use of fairly abstract grammar fragments.

We have used modular parsers as a key component of the Vanilla modular language system[2], so that fragments of a programming language – their syntax, type system and interpretive semantics – may be combined to form a final language. The ability to combine pieces of syntax makes an important contribution to exploring language constructs within a realistic setting. A particularly interesting consequence is that the “import” statement of a programming language may – in addition to its usual function of bringing external definitions into scope – include additional syntax for use within the module.

It is clear that, whilst there is substantial independence between fragments, there are also some close couplings. In particular, one might regard a parser component as being typed by its imports and exports, leading to the possibility of a type system over grammar fragments. This would reduce the number of errors due to unresolved imports and symbol clashes – although to date this has not been a problem.

Larger combinations of components will suffer from increasing numbers of symbol clashes – and this *has* been a problem already, even for comparatively small grammars. The techniques outlined in [1] for programming “in the huge”, especially the notion of “closed” and “sealed” systems, may become useful if grammars become unwieldy. These operations may be implemented very simply as extra combinators which adjust the visibilities of productions.

7. References

[1] Luca Cardelli, “*Typeful programming*,” Research report 45, Digital SRC (1989).

[2] Simon Dobson, “*A first taste of Vanilla*,” Technical report TCD-CS-1998-20, Department of Computer Science, Trinity College Dublin (1998).

[3] David Flanagan, “Java in a nutshell,” O’Reilly (1997).