Handling Multiple Language Contexts by Subtractive Context Switching

M. Löberbauer*
* Johannes Kepler University/Institute for System Software, Linz, Austria
Loeberbauer@ssw.jku.at

***Abstract*—We propose a solution for analyzing programs with multiple language contexts as they occur, for example, in C# 3.0. C# 3.0 has two contexts: the normal C# context and the language integrated query (Linq) context, which are basically the same, but Linq has some additional keywords that are treated as identifiers in the C# context. We demonstrate our solution by using the compiler generator Coco/R, which provides no support for enabling or disabling tokens in a grammar, as tokens are a global property of a programming language. We show how to handle additional tokens on top of the infrastructure provided by Coco/R.**

I. Introduction

This paper shows how to parse C# 3.0 programs [3]. The focus is on the *language integrated query* (*Linq*) part of C#. Linq provides a type-safe and flexible way to query various data sources. We present a solution based on the compiler generator Coco/R [4]. Coco/R takes an attributed grammar for a language and generates a scanner and a top-down parser. The scanner works as a deterministic finite automaton and the parser uses recursive descent. Parsers can be generated for all context-free grammars meeting the LL(1) property [1], i.e. the parser works from left to right, derives the left most symbol, and decides for an alternative with one look-ahead symbol. LL(1) conflicts can be solved as described in [6]. Coco/R generates grammar-dependent code when executed, and embeds it into code templates taken from so-called *frame* files. These files influence the behavior of the scanner (*Scanner.frame*) and the parser (*Parser.frame*) to some degree.

The problem with C# 3.0 arises because additional keywords such as `from`, `where` and `select` can be used in a Linq expression. Outside of a Linq expression, these keywords must be recognized as identifiers. Keywords, or tokens in general, are a global property of a language. Thus C# and Linq are basically two intertwined languages. Whenever a Linq expression gets analyzed, the analysis must switch to a mode where Linq keywords are known, and back as soon as the Linq expression is completed. In a case where two languages are intertwined, the common solution is to write two attributed grammars and switch between them when necessary. In the concrete case, the two grammars are similar. A two-grammar-solution is possible, but would result in code duplication. Therefore, we suggest a lightweight context-based approach. A *language context* consists of productions and a set of known tokens. Productions of one language context do not complicate parsing of another language context, and can therefore co-exist in a grammar. Disjoint tokens on the other hand must be transformed according to the current language context. A *context* in our terminology

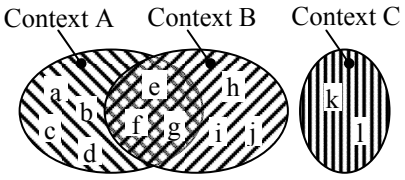

Figure 1. Three contexts, two intersecting

means the set of known tokens in a language context. Fig. [1] shows three contexts, two partial intersecting and one disjoint.

We divide C# into two contexts, the *normal context* (C#) and the *embedded context* (Linq). The Linq context consists of a few simple productions with C# expressions and some additional keywords. No new token classes are needed and white spaces remain unchanged. The only difference is that certain identifiers are treated as additional keywords. Token boundaries remain the same, i.e. there is no need to split a token or merge tokens when switching contexts.

The switch into the Linq context occurs at a single position in the grammar, namely in the expression production. Linq itself is an expression, therefore embedded Linq expressions can occur. Thus our solution must only switch back to the C# context when the outermost Linq expression is finished. To solve the problem of the additional Linq keywords the following solutions are possible:

- Write a context-aware scanner. However, this would void the advantages of using Coco/R for the scanner.
- Write multiple scanners and switch between them. However, Coco/R offers no support for doing so.
- Change tokens after the scanner delivers them. In this case Coco/R can be used to generate the full scanner.

We use the third solution, i.e. we add the context switching to the parser. The outcome of this is a maintainable solution integrated into the grammar specification.

II. Subtractive Context Switching

The normal C# context has less keywords then the embedded Linq context. Our Coco/R generated scanner is not context-aware and always treats identifiers such as `from` or `where` as Linq keywords. The context switch itself takes place in the parser. When the parser is in the normal context it replaces Linq keywords with identifiers.

In other words, it reduces the amount of known keywords in the normal context, i.e. it *subtracts* keywords.

The architecture of our solution can be separated into a collection of tokens that are to be replaced, a function to replace the tokens, a function to check for a context switch for every context, and a function to set up as well as a function to tear down the context switch. For each context, the context-related parts are encapsulated into a decorator (*filter*) around the full scanner. The demonstration is implemented in Java [2], using Coco/R for Java. The solution is constrained to the grammar, i.e. no changes to Coco/R are necessary.

In case of a recursive production such as a Linq expression in C#, we must keep track of the recursion. A Linq expression may contain an expression, which is a Linq expression itself. Therefore we must leave the Linq context only if the whole recursion has been left. The Linq context is directly recursive. In general indirect recursive contexts can appear, e.g. the parser starts in context *A*, switches to context *B*, from there to context *C* and again into context *B*. In that case, when leaving context *B* the parser must switch back to context *C*, not *A*. The presented solution handles this with a stack. When entering a context, the current filter is put on a stack, and when leaving a context the last filter gets restored.

A. Example

We show the proposed solution using a simple language (*SimLang*). The grammar is given in Fig. [2] (the notation is EBNF [5]). Quoted words represent keywords; `ident` denotes an identifier consisting of one or more letters. Blanks, tabs, and line feeds are considered as white spaces. Comments start with a # character and reach until the end of the line. The language consists of three productions. The entry point *SimLang* derives to one or more statements. A `Statement` can either be an identifier, or an `EmbeddedContext` structure. `EmbeddedContext` serves as the embedded context. It adds the keywords `begin`, `end`, `if` and `then`, which have to be treated as identifiers in the normal context. The two contexts are shown in Fig. [3]. An example for a valid sentence of this language is given in Fig. [4]. Keywords are set *italic*, identifiers normal.

```
SimLang = Statement { Statement }.
Statement =
  ( ident | EmbeddedContext )
.
EmbeddedContext =
  "begin"
  { Statement
  | "if" ident "then" Statement }
  "end"
```

Figure 2. Example grammar

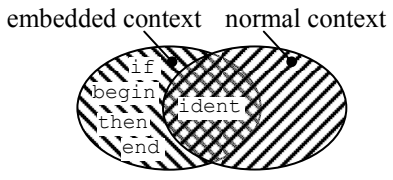

Figure 3. Contexts of SimLang

```
abcd
if end then a
# enter embedded context
begin
  b c
  if a then e
  if f then
  # enter embedded context recursively
  begin
    g
    # leave inner embedded context
  end
  # leave embedded context
end
end
```

Figure 4. Example sentence

B. Filter

To successfully parse the example sentence, we must provide a facility to map tokens of one kind to another as needed in a context. We encapsulate the token switching in a *filter*, which is a context specific decorator around the Coco/R generated scanner. A filter must be generated for every context in a grammar. To provide a common interface every filter extends the class `Filter` (see Fig. [5]). The implemented filters must be initialized before the method `Parser.Parse` is called. We do this in a dedicated method, which must be called before `Parser.Parse` (see Fig [6]). `Init` sets up the filters, chooses the filter to start parsing with and creates a stack to keep track of recursive contexts.

```
abstract class Filter extends Scanner {
  Scanner cocoScanner;
  Filter(Scanner cocoScanner) {
    super(new ByteArrayInputStream(new
        byte[0]));
    this.cocoScanner = cocoScanner;
  }
  void Enter() {
    contextStack.addFirst(current);
    scanner = current = this;
  }
  void Leave() {
    scanner = current = contextStack
        .removeFirst();
    current.Filter(la);
  }
  abstract Token Scan();
  abstract void Check();
  abstract void Filter(Token t);
}
```

Figure 5. Base class of all filter classes

```
Deque<Filter> contextStack;
Filter current, embedded, normal;
int lastKind;
public void Init() {
  contextStack = new LinkedList<Filter>();
  embedded = new EmbeddedFilter(scanner);
  normal = new NormalFilter(scanner);
  scanner = current = normal;
}
```

Figure 6. Initialize context switching

A filter for a context must implement the methods `Scan`, `Check` and `Filter`. `Scan` queries the next token from the Coco/R generated scanner, remembers the token kind and replaces it according to the context. The original token kind must be remembered in case the context is left. `Check` must implement a test for a context switch, e.g. by a syntactic look-ahead or a semantic check. In case a context switch occurs `Check` must set the kind of the look-ahead token to the according token in this context. This can be done by calling `Filter`, which replaces the

kind of the given token with the according kind in the context. The methods `Enter` and `Leave` must be called when a context gets entered respectively left. `Enter` puts the currently active filter on a stack and sets the called filter as the new current filter and scanner. `Leave` takes the last filter from the stack and sets it as the current filter and scanner. Then it calls the method `Filter` from the new current filter to allow it to replace the kind of the look-ahead token accordingly.

The methods `Scan`, `Check` and `Filter` depend on the information how to replace tokens. We keep this information in the filter, in a dictionary of type `Map`. Coco/R stores the token kind as an `int` value, and defines constants for all named tokens. We use these constants to fill the map. The filter for the normal context is shown in Fig. [7]. The normal context in our example is the default and has no entry point, thus the method `Check` is never called and therefore empty.

```
class NormalFilter extends Filter {
  Map<Integer, Integer> filter;
  NormalFilter(Scanner scanner) {
    super(scanner);
    filter = new HashMap<Integer, Integer>();
    filter.put(_begin, _ident);
    filter.put(_end, _ident);
    filter.put(_if, _ident);
    filter.put(_then, _ident);
  }
  Token Scan() {
    Token t = scanner.Scan();
    lastKind = t.kind;
    Filter(t);
    return t;
  }
  void Check() { }
  void Filter(Token t) {
    if (filter.containsKey(lastKind)) {
      t.kind = filter.get(lastKind);
    } else {
      t.kind = lastKind;
    }
  }
}
```

Figure 7. Filter for the normal context

The embedded context in our example uses all keywords, thus no token needs to be replaced. Still we must implement a filter for the embedded context, so `Scan` can save the token kind into `lastKind`, `Filter` can restore the token kind from `lastKind`, and `Check` can setup a context switch (see Fig. [8]). For the embedded context in SimLang `Check` tests for the keyword `begin` in the look-ahead token. In a more complex case a longer look-ahead or even a semantic check might be necessary. If this check indicates a context switch, we must restore the token kind. By restoring the token kind, we allow the parser to decide between the given alternatives.

C. Application of the Semantic Actions

To enable the solution, the semantic actions must be put in the grammar. The fully augmented grammar is given in Fig. [9]. The method to check for a context switch must be called in front of the alternative in `Statement`. The methods `Enter` and `Leave` must be called at the start respectively the end of the entry point to the embedded context

```
class EmbeddedFilter extends Filter {
  public EmbeddedFilter(Scanner cocoScanner) {
    super(cocoScanner);
  }
  public void Check() {
    if (scanner != this && lastKind == _begin)
    {
      // not already in this context,
      // and context switch found
      Filter(la);
    }
  }
  public Token Scan() {
    Token t = cocoScanner.Scan();
    lastKind = t.kind;
    return t;
  }
  public void Filter(Token t) {
    t.kind = lastKind;
  }
}
```

Figure 8. Filter for the embedded context

```
SimLang = Statement { Statement }.
Statement =              (. embedded.Check(); .)
  ( ident | EmbeddedContext )
.
EmbeddedContext =        (. embedded.Enter(); .)
  "begin"
  { Statement
  | "if" ident "then" Statement }
  "end"                  (. embedded.Leave(); .)
.
```

Figure 9. Augmented example grammar

D. Pitfalls

In this section, we discuss pitfalls of an implementation based on the proposed pattern. The following problems may occur: a necessary semantic action may be missing, redundant, or at a wrong position.

Coco/R cannot help with these problems, because conflicts between identifiers and subtracted keywords are unknown at compile time as we manipulate the tokens at run time. Thus it is important not to miss any entry point into the embedded context, and restore the keyword from the embedded context if needed. Otherwise the parser can enter the wrong alternative.

It is crucial that the semantic actions are not part of an alternative, i.e. they must be called all the time. This can be ensured with parenthesis around alternatives, as we did in the SimLang example. An erroneous example is given in Fig. [10]. The semantic action gets only executed if the first alternative is taken. If the check action is placed in such a way, the parser can never enter the alternative to the embedded context. The same error on the enter or leave action causes wrong context information in the parser. If the parser is not aware of the context, the wrong `Scan` method is called and therefore the tokens are not modified as expected.

```
X =
  (. This semantic action takes place in the
     ident branch, not before the alternative.
  .)
  ident
  | "if"
  | "BEGIN"
.
```

Figure 10. Wrongly positioned semantic action

E. Conclusions

The presented solution is elegant for the given problem and environment. We could easily apply the proposed pattern to C# 3.0 including Linq.

REFERENCES

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, & Tools (Second Edition)*. Pearson, Addison-Wesley, 2007.

[2] J. Gosling, B. Joy, G. Steele, G. Bracha, *The Java™ Language Specification (Third Edition).* Addison-Wesley, May 2005.

[3] Microsoft, *C# Language Specification version 3.0, http://download.microsoft.com/download/3/8/8/388e7205-bc10-4226-b2a8-75351c669b09/CSharp Language Specification.doc* - last checked 2008-07-14

[4] H. Mössenböck, *The compiler generator Coco/R. http://ssw.jku.at/coco/* - last checked 2008-07-14

[5] N. Wirth, "*What can we do about the unnecessary diversity of notation for syntactic definitions?,*" *Commun. ACM,* 20 (11):822-823, 1977.

[6] A Wöß, M. Löberbauer, and H. Mössenböck, "*LL(1) conflict resolution in a recursive descent compiler generator.*," *JMLC'03*, 2003.