State University of New York at New Paltz
Department of Mathematics and Computer Science

Technical Report # 98-105
July 5, 1998

Generating Efficient Substring Parsers for BRC Grammars

Martin Ruckert

ruckert@mcs.newpaltz.edu

`http://www.mcs.newpaltz.edu`

75 South Manheim Blvd, Suite 6 • New Paltz, NY 12561

Abstract

The concept of Bounded Context Parseable grammars is a less restricted variation of Bounded Context grammars a class of grammars that is especially suited for substring parsing. Robust parsing is introduced as a special form of parsing for BCP grammars. The differences between robust parsing and LR(k) parsing are explored. A method of generating robust parsers is presented and complemented by performance measurements for the resulting parser. Included are some examples to illustrate the results of robust parsing.

1 Introduction

The method of robust parsing, presented in this paper, was developed as part of ongoing research in automatic error diagnosis and correction. Most parsers will detect an error as soon as the current input string is no longer a prefix of any correct sentence. At this point various error recovery schemes can be used: panic mode, forward repair[13, 11, 22], combinations of both[24], local repair [20, 29, 33, 21], and global least cost repair[1]. A more detailed discussion of error recovery methods can be found in [12] and [27].

Recent approaches to error diagnosis and recovery use combinations of simple local repairs with regional minimum distance correction and elaborate global recovery schemes which have little to do with the primitive panic mode (e.g. [4, 5]).

The theoretically best methods use $O(n^3)$ parsing algorithms [10, 7] and are considered too slow for practical use [37].

None of these methods produces an error diagnosis good enough to attempt true error correction.

Using theoretical investigation of parsing and error correction, Wetherell[36] concludes: "But there is simply no hope of building efficient (i.e. linear time bounded) correctors which can handle any reasonably large class of errors perfectly for practical programming languages. But there are ways around this problem."

Richter[26] proposed non-correcting syntax error recovery which requires, however, a parser that can recognize arbitrary substrings of the language. His idea was made a practical alternative through work of Rekers [25] and Bates[3].

Wetherell proposes a heuristic approach to language understanding and error diagnosis. While the author is not aware of any research using heuristic methods to understand errors in computer programs, heuristic parsing is used frequently for natural languages. Statistical information, as in [15], or more often semantic constraints, as in [18, 39, 14], are used successfully to improve quality and speed[23] of parsing.

Traditional LL or LR parsers offer, after an error has been detected, only the current parse state and the still unprocessed stream of input tokens as a basis for error diagnosis.

Input tokens are certainly not the right level for knowledge based error diagnosis. Instead, we would like to extract as much structure from the token stream as the presence of errors permits.

The parse states, contained in the parse stack, can be interpreted as an encoding of the program structure up to the point of error. Usually, however, a single parse state corresponds to several different productions of the grammar, which are equivalent only in the absence of

errors. Further, a parser can advance using empty productions and defaults, and can reach finally a parse state which is no longer justified by the input. So, the information given by the parse state is often ambiguous or even misleading. Small errors in the input can derail the parser completely.

A parser, suitable to prepare the input for a knowledge based error diagnosis, should be able to combine all input tokens into larger structures as long as the context of the tokens permits this without ambiguities. To allow error correction based on the parser output, small changes (errors) to the input should cause only small changes in the parser output. The process of bounded context parsing defines such a continuous mapping of token strings to parse trees. Weak bounded context parsing, defined below, is a less restrictive version of bounded context parsing that is comparable in power to LR(k) parsing while maintaining the continuity of bounded context parsing. Finally, the parser should run efficiently, in linear time, and a parser-generator should facilitate its production. The robust parser described here is such a parser.

This paper presents, after giving some definitions, a comparison of LR(k) parsing, Bounded Context (BC) parsing, and parsing with Bounded Context Parseable (BCP) grammars. Then, robust parsing is introduced as a form of BCP parsing. Next, the implementation and performance of parser and parser-generator is described. Two examples are given to explain the results of parsing incorrect input. The last section deals with the use and handling of ambiguous grammars. Here and throughout this paper, the programming language C[2, 16] is used to investigate the more practical aspects of robust parsing.

2 LR(k), BC, and BCP parsing

2.1 Definitions

2.1.1 Symbols and Strings

In describing a grammar, the `typewriter` font is used for terminal symbols.

Nonterminal symbols are written using <*italics*> enclosed in pointed brackets.

Uppercase italics are used for symbols, e.g. S, T, U, L, and R; lowercase italics are used for strings of symbols, e.g. l, r, p, q, u, v, w, s, and t. The symbol λ is used to denote the empty string.

2.1.2 Grammar, Deduction, Phrase, and Handle

Grammars are described using production rules of the form $<S> \rightarrow s$. In the usual way, these rules are used to derive a string from another string by repeatedly replacing an occurrence of the left hand side of a rule by the right hand side. A string that can be derived from a single symbol is called a "phrase".

A "handle" of a phrase p is a substring s of p and a production $<S> \rightarrow s$ such that the replacement of $<S>$ by s is one step in the derivation of p. If the production is clear from the context, we speak of the substring s as a handle.

2.1.3 LR(k), BC, BCP, and BRC parsing

Parsing is the reconstruction of the derivation from a given phrase. All the parsing algorithms considered here use bottom-up parsing by handle pruning: the derivation is reconstructed by iteratively finding a handle and replacing it by the left hand side of the handles production. The definitions given now are informal, but sufficient for the present discussion.

A grammar is called a BC grammar—parseable with Bounded Context—, if, given any phrase, it is possible to identify **all** handles of the phrase looking at m symbols to the left and n symbols to the right of the handle (for some finite m and n)[8].

A grammar is called a LR(k) grammar—parseable from Left to Right with k symbols look-ahead— if, given a phrase derived from a special start symbol, it is possible to identify the **leftmost** handle of the phrase looking at all the symbols to the left of the handle and at k symbols to the right of the handle (for some finite k)[17].

Definition: A grammar is called a BCP grammar—Weak parseable with Bounded Context—, if, given any phrase, it is possible to identify **at least one** handle of the phrase looking at m symbols to the left and n symbols to the right of the handle (for some finite m and n)[38].

2.2 Comparison

It is immediately clear from the definition, that the BC grammars are a subset of both LR(k) grammars and BCP grammars. BC grammars never gained much importance, since shortly after Floyd presented them in 1964, Knuth, in 1965, published his landmark paper "On the Translation of Languages from Left to Right" where he introduced LR(k) grammars and showed how very efficient parsers can be constructed for this larger class of grammars. BCP(j, k) Grammars were studied extensively in the 1970's[38][31, 32, 34].

A very interesting subclass of BCP grammars is the class of BRC (Bounded Right Context) grammars[8]—basically the intersection of LR(k) and BCP(j, k). This class of grammars is large enough to generate all deterministic languages but does still allow for fast linear parsing algorithms[9]. Further interesting results on the hierarchy of grammatical classes can be found in [31].

The following example will illustrate some more practical aspects of parsing programming languages.

2.2.1 Example 1

This example uses a simplified subset of a grammar describing the C programming language (a more complete BRC grammar for C can be found in the appendix). Further, we turn our interest to the problem of parsing program fragments not just complete programs. By definition, LR(k) parsing operates only on phrases derived from a special start symbol (complete programs); it can be adapted to handle fragments of programs by removing this restriction from the definition. Note, however, that this change poses quite some difficulties for the usual LR(k) parser-generators[3, 30]. BCP parsers are not affected by this.

In C, the syntax of expression used to define variables is a restricted version of the general expression syntax. Only preceding symbols, like `int` or the use of operators, like `+`, that are not valid in a variable declaration disambiguate the situation.

<*expr*> → <*id*>
<*expr list*> → <*expr*>
<*expr list*> → <*expr list*>,<*expr*>

<*var*> → <*id*>
<*var list*> → <*var*>
<*var list*> → <*var list*>,<*var*>

<*function*> → <*id*>(<*expr list*>);
<*declaration*> → `int`<*var list*>;
<*prog*> → <*function*>
<*prog*> → <*declaration*>

Given this limited grammar, and looking at a comma separated list of identifiers
. . . <*id*>,<*id*>, . . .
it is not possible to decide whether this list is a list of variables or expressions. Only in the context of a complete function call or variable declaration, is it possible to identify the handles.

If the fragment is `int` <*id*>,<*id*>,<*id*>;, then `int` <*id*> is enough to decide that <*id*> must be reduced to <*var*> and then <*var*>,<*id*> can be reduced to <*var*>,<*var*> , and so on. Thus, the whole list can be parsed considering one or two symbols to the left of the handle. One can see how the information provided by the symbol `int` can travel over the list of identifiers without encoding it in the state of an LR(k) parser.

Consider a different program fragment: <*id*>,<*id*>,<*id*>; . Here, an LR(k) parser is lost since the leftmost handle can not be identified. A BCP parser can tell from the sequence <*id*>; that the last <*id*> must be a <*var*> because otherwise a) should be between the <*id*> and the ;. Again, the information propagates over the list reducing <*id*>,<*var*> to <*var*>,<*var*> .

Adding the two productions:

<*expr*> → *<*expr*>
<*var*> → *<*var*>

no longer allows a BCP parser to unwind the list from the left because an unlimited number of stars may be between the `int` and the <*id*> .

Finally, if the productions

<*expr*> → (<*expr*>)
<*var*> → (<*var*>)

are added, the grammar is no longer a BCP grammar. An identifier can be buried in an unbounded amount of parenthesis.

If a BCP grammar is desired in this or a similar situation, there are still some options left without resorting to encoding parse states in the grammar as described earlier.

- Pragmatic approach: a finite limit, large enough for all practical programs, can be set on the size of the expressions. This makes the original grammar immediately a BCP grammar.

- The grammar can be rewritten to identify symbol strings common to both kinds of expressions. Reducing such symbol strings to <*var or expr*> allows to postpone the decision whether the substring is a variable declaration or an expression. As soon as the immediate context clarifies the situation, one of the additional rules <*expr*> → <*var or expr*> and <*var*> → <*var or expr*> can be used.
- The best technique, with respect to error correction, uses an ambiguous grammar. For example:

  <*var*> → <*id*>
  <*expr*> → <*id*>
  <*var or expr*> → <*id*>
  <*var*> → (<*var*>)
  <*expr*> → (<*expr*>)
  <*var or expr*> → (<*var or expr*>)
  <*var*> → <*var or expr*>
  <*expr*> → <*var or expr*>

  Having all these rules available ensures that there is always at least one recognizable handle but renders the grammar highly ambiguous. Using directives, as described below, the parser can be instructed to perform always the most specific reduction justified by the context. If necessary, the decision is postponed. The different possible parse trees all have the same structure but the semantic actions connected with the reductions need to be postponed as well.

Experience shows that for practical programming languages, there are always ways to overcome the shortcomings of BCP grammars.

3 BCP Parsing

3.1 Contexts

To construct a BCP parser, for each production <S> → s of the grammar a set of contexts is needed, where a context is a pair of strings (l, r). The sets of contexts must satisfy the following two conditions:

3.1.1 Correctness:

Given a production <S> → s, one of its contexts (l, r), and a phrase p such that lsr is a substring of p, then the substring s with the production <S> → s is a handle of p.

3.1.2 Completeness:

Given a phrase p, then there is a production <S> → s and a pair (l, r) in its context set such that lsr is a substring of p.

The fact that the grammar is a BCP grammar ensures that such sets of contexts exist and vice versa. The context sets are, however, not uniquely determined.

3.2 The BCP Parsing Algorithm

Given a correct and complete set of contexts, BCP parsing proceeds by selecting a production $<S> \rightarrow s$ and a context (l, r) for it such that lsr is a substring of the given phrase and replacing this substring by $l<S>\ r$.

Completeness ensures the existence of a production and a context. Correctness ensures that the substitution is exactly the pruning of a handle.

The algorithm is nondeterministic and any implementation will convert this into an deterministic algorithm by using a specific search algorithm to find the substring, the context, and the production.

3.3 Correct Contexts

Given a production $<S> \rightarrow s$ and a context (l, r), correctness can be tested by considering all strings p, q, t, v, and w with $ptq = vlsrw$, where s and t cover at least one common symbol, t is the right hand side of a production $<T> \rightarrow t$, and $p<T>q$ is a phrase of the language. If these strings can be found, the context (l, r) does not ensure that s is a handle and, hence, it is not correct. A complete and detailed, list of all the strings and relations that must be tested can be found in [8].

Large correct context sets are easily obtained by generating finite strings l and r and testing them for correctness.

3.4 Left-to-Right Collection

Starting with a correct, but otherwise arbitrary, collection of contexts makes it difficult to prove completeness. The situation changes, if we consider collections of contexts gathered in some systematic way. One such way of systematic collection is Left-to-Right Collection.

To describe the algorithm, the notion of prefix and suffix of a symbol is useful.

3.4.1 Full Prefix and Suffix

Given a symbol S and a production $<T> \rightarrow pSq$, p is a full prefix of S and q is a full suffix of S.

Further, if v is a full prefix of $<T>$ then vp is a full prefix of S and if w is a full suffix of $<T>$ then qw is a full suffix of S.

3.4.2 Reduced Prefix and Suffix

If st is a full prefix of S then t is a reduced prefix of S and if st is a full suffix of S then s is a reduced suffix of S.

It follows immediately from the definition that a reduced prefix or reduced suffix does not contain handles, provided the grammar is unambiguous.

3.4.3 Prefix and Suffix

If st can be derived from a full prefix of S then t is a prefix of S and if st can be derived from a full suffix of S then s is a suffix of S.

Left-to-Right context collection aims at a parser that will successively identify and prune the leftmost handle. Hence, this parser will do exactly the same reductions as an LR(k) parser. To obtain this parser the only contexts (l, r) that need to be considered for a production $<S> \rightarrow s$ are those where l is a reduced prefix of $<S>$ and r is a suffix of $<S>$.

The algorithm to collect a correct and complete set of contexts operates on two families of context sets, a test family **T** and a final family **F**. The sets in each family are indexed by the productions of the grammar and contain the contexts that belong to its index.

3.4.4 Algorithm

1. Initialize **F** to contain the empty set for all productions of the grammar. Initialize **T** to contain the set $\{(\lambda, \lambda)\}$ for all productions of the grammar.

2. Select a production and one of its contexts contained in **T**:

   Delete the context from the set in **T**.

   If the context is correct add it to the corresponding set in **F**.

   If the context is not correct find a set of refinements of the context (see below) and add these contexts to the appropriate set in **T**.

3. Repeat the previous step until all sets in **T** are empty.

At any time, **F** is correct and the union of **F** and **T** is complete. Therefore, **F** will contain a complete and correct collection of contexts for the grammar as soon as all sets in **T** are empty.

3.4.5 Refining a Context

A context (l, r) for the production $<S> \rightarrow s$ can be refined by refining either l or r.

To refine l, find all symbols $T_1, \ldots, T_n$ such that $T_i l$ is a reduced prefix of $<S>$. This gives the set of refinements $\{(T_i l, r) | i = 1, \ldots, n\}$.

And similar, to refine r, find all symbols $T_1, \ldots, T_m$ such that rT_i is a suffix of $<S>$. This gives the set of refinements $\{(l, rT_i) | i = 1, \ldots, m\}$.

The trick here is to use only **reduced** prefixes to refine the left context. This is motivated by the attempt to identify the leftmost handle. Using arbitrary prefixes instead would prevent the algorithm from terminating even on the most simple languages.

3.4.6 Discussion

The algorithm has two obvious problems: it is nondeterministic, since there are two choices to refine a context, and secondly there is no guarantee that it will terminate, even if the grammar is a BCP grammar.

For practical purposes however, we are interested in contexts (l, r) where l and r are short strings. Hence, it is possible to explore the different possibilities of refining and find the best choice. If, after a while, some productions in $\mathbf{T}$ are left which require still longer context strings, it is best to inspect these cases individually and resolve the problems by changing the grammar.

The programming languages we have today seem to be designed with left to right parsing in mind. (This not only makes them suitable for LR(k) parsing but also programmers read their programs left to right.) Therefore, it is often more promising to extend the left instead of the right context; for most programming languages one symbol look-ahead is sufficient. In addition, when parsing from left to right, symbols in the left context usually summarize whole subtrees and provide more information than the terminal symbols one finds in the right context.

Guided by these heuristics, a good approach is to extend the empty contexts first by one symbol to the left, then add one look-ahead symbol to the right, and then continue to add more symbols to the left context. If the algorithm does not terminate after three or four iterations, one should inspect the productions and contexts still in the test set. It might be that for these productions further look-ahead is needed. Usually, however, it turns out that the grammar is either ambiguous or can be changed slightly to reduce the amount of left context needed. If the algorithm terminates, a correct and complete collection of contexts has been found.

The class of grammars for which a parser can be constructed using Left-to-Right collection is a proper subset of LR(k) grammars. If it would not be for the purpose of robust parsing, discussed next, this form of BCP parsing would not offer any advantages over LR(k) parsing.

4 Robust Parsing

All parsing algorithms compute correct parse-trees if applied to correct symbol strings. Applied to an incorrect symbol string, most algorithms have the "correct prefix property": an error can be detected as soon as the symbol string seen so far, is no longer the prefix of any correct symbol string. Beyond this, usually very little can be said about the effect of applying the algorithm to an incorrect symbol string. Small errors in the symbol string can derail the parser completely. From a robust parser, one expects that its performance degrades gracefully as the number of errors in the symbol string increases.

This expectation can be formulated as a continuity property: Any step in the derivation (node in the parse-tree) depends only on a fixed, finite number of symbols to the left and right of the position where the reduction takes place. Further formalization leads directly to bounded context parsing. A continuous mapping from symbol strings to parse-trees will limit the effect of errors: small changes in the symbol string will result in small changes in the parse-tree. Here, the extra effort which might be needed to convert a LR(k) grammar to a BCP grammar starts to pay off.

Starting with a correct and complete collection of contexts two operations can help to make the resulting BCP parser more robust: context expansion and context extension.

4.1 Context Extension

Consider the productions:

<*prototype*> → <*header*> ;
<*function*> → <*header*> <*compound stmt*>

which are part of a grammar for the C programming language.

The pair (λ, λ) turns out to be enough context for the first production. In the presence of errors, however, it might be wise to check for some context in any case. The following C program gives an example:

```
int square(int x);
{ return x*x;
}
```

The first line can be parsed into <*header*>;, which could be further reduced to <*prototype*> , which in turn can not be followed by a compound statement. The error is the extra semicolon. The reduction to <*prototype*> should not take place. An extra test which could prevent this reduction is achieved by using the set of extended contexts (λ, R), where the R stands for all symbols that could possibly follow after a (global) function prototype.

In general, given a production <S> → s and a context (l, r) of it,

- one can replace this context by the set of contexts $\{(L_i l, r) | i = 1, \ldots, n\}$, where $L_1, \ldots, L_n$ are all the symbols such that $L_i l$ is a prefix of <S> (left extension).
- one can replace this context by the set of contexts $\{(l, r R_i) | i = 1, \ldots, m\}$, where $R_1, \ldots, R_m$ are all the symbols such that $r R_i$ is a suffix of <S> (right extension).

Left and right extensions preserve correctness and completeness of a collection of contexts. Although they do not change the behavior of the parser if applied to a correct symbol string, they will make the parser more reluctant to reduce if applied to incorrect sequences. In practice, it is useful to extend all contexts to include at least one symbol to the left and one to the right. The additional tests can be implemented without a performance penalty.

4.2 Context Expansion

Consider the production:

<*stmt*> → <*expr*>;

The production requires a non empty context because its right hand side is also part of the for-statement:

`for(`<*expr*>`;` <*expr*>`;` <*expr*>`)` <*stmt*>

Using Left-to-Right collection, the following contexts, having a single symbol as left context, are obtained: (<*stmt list*> ,λ), (<*decl list*> ,λ), (`:` ,λ), (`{` ,λ), (`else` ,λ), (`do` ,λ), (`)` ,λ). This is enough to parse correct C programs. However in the presence of errors, "`}`" also could be a possible left context, as in

```
while (...)  {
...}
<expr>;
```

Should the reduction of the while statement fail, due to a syntax error in the condition or the compound statement, it is no longer possible to reduce the symbols preceding the expression to a *<stmt list>* . Consequently, the reduction of the following expression to a statement will fail as well. Adding the correct context $(\texttt{}, \lambda)$ will allow the parser to continue correctly.

Adding contexts does not affect the completeness of a collection of contexts. Hence, given a production $<S> \rightarrow s$, a prefix l, and a suffix r of $<S>$ one can add the context (l, r) provided the context is correct. Adding a context will make the parser more eager to reduce and enables valid reductions that might be useful in the presence of errors.

In practice, Left-to-Right collection is used to determine the minimum left and right contexts needed. In particular, only reduced prefixes are considered as left contexts. Once the minimal context sets are determined, context expansion will add for a context of the form (l, r) all correct contexts of the form (t, r), where t is a string of the same length as l such that from l one can derive pt for some p. The process is repeated until no new contexts can be added. If a different context collection schema is used, right contexts can be expanded in a similar manner.

4.3 Examples

4.3.1 Example 2

Consider the fragment:

```
func(1*5+6; 2*3, 4/2);
```

containing a ";" instead of a "," after the "6". The authors robust parser will derive the following list of symbols:

<postfix> `(`*<expr>*`;`*<expr>*`);`

The relevant productions of the adapted C grammar are:

<postfix> → *<postfix>* `(`*<assign list>*`)`

<assign list> → *<assign>*
<assign list> → *<assign list>*`,`*<assign list>*

<expr> → *<assign list>*

<for stmt> → `for (`*<expr>*`;`*<expr>*`;`*<expr>*`)` *<stmt>*

Note the rules for *<assign list>* (see section 4.4). Further, no distinction between the comma as a separator in a parameter list and the comma operator is made. This distinction is needed for semantic processing, but hampers syntactic error correction. (Whether the dual use of the comma was a wise decision in the design of the C language is more than

questionable.) This example focuses on the dual use of the ; in the for-statement and to produce statements from expressions.

The string `1*5+6` between "`(`" and "`;`" must be an expression. The only place where this situation occurs, however, is in a for-statement. While the reduction of `1*5+6` to <*assign list*> is still correct, the further reduction to <*expr*> is not. This last reduction could be prevented by further extending the left context for the reduction by one symbol. The additional left symbol could only be a `for`, not a <*postfix*>, and testing it would prevent the reduction in the given situation.

The following string, "`2*3,4/2`", was parsed correctly to <*assign list*> and then again incorrectly to <*expr*> . Like before, the problem is caused by the for-statement, where an expression occurs between a semicolon and a closing parenthesis. An extended left context of three symbols would be needed to prevent this reduction.

The effects of the error are strictly contained within the fragment shown. Everything before or after this fragment can be parsed as usual. Reductions that would contain the fragment as a subtree can not be performed. The partly completed parse is a good starting point for an error analysis. Two technical changes (undoing two reductions) and one change of the source (changing the "`;`" into a "`,`") are needed to correct the error. Other possible corrections, such as inserting a "`)`" before the semicolon, can be ruled out easily since only a very limited (3 symbols instead of 9), bounded look-ahead reveals the extra closing parenthesis.

4.3.2 Example 3

Consider the fragment:

```
func 1*5+6, 2*3, 4/2);
```

which is reduced by the parser to

<*id*>`(`<*num*>`*`<*cast*>`+`<*mult*>`,`<*assign list*>`)`;

The identifier `func` is followed immediately by the numeral `1`. A robust parser will reduce none of these symbols, since for example, an <*expr*> can not be directly preceded by an identifier or followed by a numeral. The next numerals, `5` and `6`, can be reduced. The operators surrounding them make it safe to reduce to a <*cast*> or even a <*mult*> independent of a possible operator that might be between the numeral and the identifier and that could have either higher or lower precedence. The parser then gets on track again and parses the rest of the argument list into the single symbol <*assign list*> .

Example 3 illustrates the very difficult problem of inserting or removing parentheses—a non-local correction. A LR(k) parser would need unbounded look-ahead to deal with this situation correctly. It will detect the error after reading the `1` and can correct it in many ways: e.g. insertion of a semicolon, insertion of an operator, or insertion of the opening parenthesis. Most likely it will reduce `func` to <*expr*> and then insert a semicolon. Parsing can then continue until the closing parenthesis is found. Only here it becomes clear that inserting the opening parenthesis is the best solution.

A robust parser, on the other hand, can continue immediately with parsing. The question concerning the best correction is postponed until the parsing is finished and maximum information is available.

4.4 Ambiguous Grammars

The grammar, given in example 2, contains the ambiguous productions

$$<\textit{assign list}> \rightarrow <\textit{assign}>$$
$$<\textit{assign list}> \rightarrow <\textit{assign list}>\texttt{,}<\textit{assign list}>$$

instead of

$$<\textit{assign list}> \rightarrow <\textit{assign}>$$
$$<\textit{assign list}> \rightarrow <\textit{assign list}>\texttt{,}<\textit{assign}>$$

To see why the ambiguous grammar is better, consider the previous example. After parsing has produced the symbol string

$$<\textit{id}>\ <\textit{num}>\ \texttt{*}<\textit{cast}>\ \texttt{+}<\textit{mult}>\ \texttt{,}\ <\textit{assign}>\ \texttt{,}\ <\textit{assign}>\ \texttt{);}$$

the unambiguous grammar would not allow the two assignments to be combined into a single $<\textit{assign list}>$. It requires a correct parse of the first assignment before it can combine a sequence of assignments into a list. Only the first assignment of the list must be reduced to an $<\textit{assign list}>$, and doing otherwise would prevent further use of the left recursive production. Using the ambiguous grammar, any assignment which is between commas can be reduced to an $<\textit{assign list}>$ and further reductions can combine them in any order into a list (or better a binary tree) of assignments.

Using the ambiguous grammar creates a problem: Given the string
$\ldots,<\textit{assign list}>\texttt{,}<\textit{assign list}>\texttt{,}<\textit{assign list}>\texttt{,}\ \ldots$
no finite context can help to decide whether to apply the reduction to the first or second couple of $<\textit{assign list}>$'s.

While some uses of ambiguous grammars, e.g. to describe the if-else statement, can be avoided in principle, the examples above illustrate why ambiguous grammars are essential for robust parsing. An ambiguous grammar, however, is usually not a BCP grammar.

To cope with this situation the parser-generator provides two directives:

- A production can be marked as "ambiguous" indicating that the generation of contexts stops after finding all correct contexts of one symbol to the left and one symbol to the right. No (futile) attempt is made to find a complete set of contexts for this production.
- A production can be marked as "unambiguous" indicating that all prefixes and suffixes of length 1 should be used as valid contexts even if they are not correct.

For example, the rule $<\textit{if stmt}> \rightarrow$ `if` $<\textit{stmt}>$ is marked as ambiguous and the rule $<\textit{if stmt}> \rightarrow$ `if` $<\textit{stmt}>$ `else` $<\textit{stmt}>$ is marked as unambiguous. Doing so achieves the desired effect of associating an `else` with the most recent `if`.

Strangely enough, the rule $<\textit{assign list}> \rightarrow <\textit{assign list}>\texttt{,}<\textit{assign list}>$ is marked both as ambiguous and as unambiguous. It prevents the generator to search—without success—for a complete set of contexts and allows the use of this production at all places where an $<\textit{assign list}>$ possibly might occur.

5 Implementation and Efficiency

A parser, built using Left-to-Right collection, can parse a correct phrase from left to right by reducing repeatedly the leftmost handle. It can be implemented using a stack to hold the symbols to the left of the handle and the handle itself in exactly the same way as a LR(k) parser. As usual, the contents of the stack can be encoded into the state of a finite automaton to avoid the search for a possible reduction. The resulting parser will then be as efficient as any LR(k) parser.

The situation changes, if context expansion and context extension is used to obtain a robust parser and the input contains errors.

1. It is not always possible to detect the leftmost handle.

2. After a reduction is made, further reductions might be possible to the left of the present reduction. This makes it necessary to back-up the parser and reconsider the symbols on the parse stack.

A reduction replaces a substring s of the input string by a new symbol S. Parsing left to right, we know that left of the new symbol S there were no recognizable handles. This might have changed only for handles that need the new symbol S either directly as part of the right side of the production or indirectly as part of the right context. Hence, if the maximum length of the right contexts (the look-ahead) is k, the parser needs to go back k symbols after a reduction. Therefore, the parser still operates in linear time.

In practice, the look-ahead k is often 1. This means that only the top symbol of the parse stack needs to be pushed back into the input stream. The push back mechanism for the input stream can be implemented efficiently with a second symbol stack. Moreover, for many reductions the back-up step is not necessary at all. Assume the reduction replaces a string Tt by the new symbol S, then a back-up is only necessary if there exists a production that has the context (l, S), but not (l, T). Due to context expansion, a robust parser will not encounter this situation very often. For example, our parser for the C programming language needs the back-up step only after about 4% of the reductions. Obviously, in languages like C, that are designed with a one-token-look-ahead parser in mind, the symbol S, summarizing the whole sequence Tt, usually does not offer substantially more information than the first symbol T of Tt. Thus, the overall costs of the backup mechanism are negligible.

5.1 Parser Generator

The parser-generator is written entirely in Prolog, which proved to be a good decision. The core of the algorithm can be described nicely as sets of symbols, satisfying certain relations. The description is directly executable. Each step, as described below, is performed using a generate-test-assert-fail loop. Candidates are produced by a generator predicate and those passing a test will be asserted, defining a new relation. There is often no sharp boundary between generator and test as variables are instantiated as late as possible. The "fail" forces backtracking to the generator until all possibilities are exhausted. This schema is suited to the problem at hand since backtracking is generally very efficient and allows all dynamic memory to be reclaimed.

The input grammar is written in BNF notation. To describe semantics, each production rule can be followed by a double colon and the name of a C function. For example:

```
add => mult.
add => add, plus, mult :: addition.
add => add, minus, mult :: subtraction.
```

First, this is translated into internal form, where strings of symbols are represented as Prolog lists:

```
rule(add,[mult],[],[]).
rule(add,[add,plus,mult],[],[]).
rule(add,[add,plus],[mult],[]).
rule(add,[add],[plus,mult],[]).
rule(add,[add],[plus],[mult]).
. . .
```

Each production rule is stored in multiple forms to expose explicitly the different decompositions of the right hand side into substrings. This makes unification a fast way to find all productions that start with a given string, end with a given string, or contain a given string.

Next, the following auxiliary predicates are computed:

1. `symbol(`S`)` S is a symbol.
2. `terminal(`S`)` S is a terminal symbol.
3. `nonterminal(`S`)` S is a nonterminal symbol.
4. `right(`S`, `T`)` from S one can derive sT for some s.
5. `left(`S`, `T`)` from S one can derive Ts for some s.
6. `same(`S`, `T`)` from S one can derive T.
7. `post(`S`, `T`)` T is a suffix of S.
8. `pre(`S`, `T`)` S is a reduced prefix of T.
9. `prepre(`S_1`, `S_2`, `T`)` S_1S_2 is a reduced prefix of T.

A predicate to generate or test reduced prefixes longer than two symbols is given as a recursive relation using the production rules directly.

For left-to-right collection, only a predicate to test for correctness needs to be added. This test is a straight forward translation of the relations given in [8] into Prolog.

The contexts considered contain a maximum of one symbol as right context and three symbols as left context.

Sets of symbols are represented as Prolog lists of atoms and sets of strings are represented as (unions of) direct sums of sets of symbols. Most sets of strings encountered are direct sums; unions are rarely needed.

The next steps are context extension and context expansion. To generate prefix- and suffix-strings, the predicate `pair(`S`,`T`)` is computed such that S is a prefix of T, and hence also T is a suffix of S. This predicate is already quite large; therefore, longer prefixes or

suffixes are not stored, but computed as needed. With the predicate `pair/2`, candidates for extensions and expansions can be generated. Fortunately, for all but a few productions, a single symbol left and right proves to be enough context.

For simplicity, the code used for left-to-right collection is more or less duplicated here and the results of the left-to-right collection phase are used only to put limits on the length of the prefixes generated.

The final step is to generate C code for the parser itself.

5.2 Parser

The parser is written entirely in C. The reader, scanner, preprocessor, the handling of symbol-tables, and primitive routines to manipulate the parse stack are hand-coded.

The generator defines an enumeration type for the token types and produces a large switch-statement containing the main part of the parsing algorithm. It selects a case depending on the element on top of the stack (last symbol of the handle). Each case then starts with a test for the look-ahead symbol, followed by a test for the other symbols of the handle and the left context. If successful, reduction takes place followed by a back-up operation if needed.

The following code fragment shows two cases: parsing of a function header (note the back shift after the reduction) and parsing an addition or subtraction (see the relevant piece of grammar above).

```
while(1)
switch(STACK(0)) {
 . . .
case close_ :
if (in_set(LOOK_AHEAD, set6)) {
  if (STACK(-1)==param_list_ &&
      STACK(-2)==open_ &&
      STACK(-3)==id_ &&
      STACK(-4)==type_spec_ &&
      in_set(STACK(-5), set1)) {
    reduce(header_, 5);
    goto backshift; }
}
if (in_set(LOOK_AHEAD, set28)) {
 . . .

case mult_ :
if (in_set(LOOK_AHEAD, set40)) {
  if (in_set(STACK(-1), set39)) {
    reduce(add_, 1);
    continue;}
  if (STACK(-1)==minus_ &&
      STACK(-2)==add_ &&
```

```
      in_set(STACK(-3), set39)) {
    subtraction(add_, 3);
    continue;}
  if (STACK(-1)==plus_ &&
      STACK(-2)==add_ &&
      in_set(STACK(-3), set39)) {
    addition(add_, 3);
    continue;}
}
goto shift;

 . . .

default:
shift:
  SHIFT;
  continue;
backshift:
  BACKSHIFT;
  continue;
}/*end switch/while*/
```

The variables `set6`, `set28`, ... are constant integer arrays encoding the necessary symbol sets as bit-vectors. Their definitions and initializations (e.g. `char set6[] = {32, 10, 4, 32, 0, 0, 28, 0};`) are produced by the generator.

While it would have been possible to modify the code of a standard table-driven parser, simulating a finite state machine, a more direct approach was used here. As a result, the parser does some unnecessary testing and the performance is not optimal. Yet, the parser is about as fast as a parser generated with lex and yacc. 1000 lines of correct C-source code (5300 tokens) are parsed in 149.35ms (0.028ms/token) on a 33MHz-486PC under Linux. In comparison, the function `yyparse`, generated using lex and yacc, used 130.00ms (0.026ms/token) under the same conditions. On the other hand, experimentation with the parser was greatly simplified since the code generated could be changed by hand without having to rewrite and rerun the generator.

As an application, a parser for the C programming language was produced and used to implement a pretty printer[28].

6 Related Work

It is not yet possible to compare the quality of error correction since the parser is intended to produce the starting point for knowledge based error diagnosis and recovery, which is not yet implemented. Currently the parser does non-corrective error recovery in the spirit of Richters[26] proposal. Some other implementations of substring parsers according to this idea are known:

Lang's method[19], based on Earley's parser[7], produces a collection of all possible parses. The method is powerful but not very efficient. Cormacks substring recognition algorithm[6], while efficient, is limited to bounded context grammars.

Bates and Lavie[3] present an algorithm to parse substrings of LR(k) languages. A clever representation of the multiple possible stacks, corresponding to multiple possible parses of an LR parser for the substring, allows to merge different parse stacks once they become again equivalent and achieves a linear time bound. No performance data is given but the parser was reportedly used to check single lines of COBOL and Pascal programs as part of an editor. Whether it is possible to check entire programs using this approach is unclear.

Rekers and Koorn[25] describe a similar parser based on Tomitas work[35] where multiple parallel parses compute all possible interpretations of a given substring. They give performance measurements for short sentences (100 tokens) using a small grammar (20 rules). Implemented in Lisp on a SUN SPARC station, the parsing time for a substring varies between 0.5s and 1.5s (5 to 15 ms/token). This is considerably slower than our implementation. Further, the algorithm is not linear and it is unclear how the approach would scale for real grammars and real programs.

In general, the overhead of computing all possible interpretations of a substring seems considerable compared to our approach where only the common subset of all possible interpretations is computed.

7 Conclusion

The general idea behind robust parsing is to obtain more sensible results when parsing incorrect programs, without spending too much extra work on correct programs. The definition of robust parsing as a form of BCP parsing is a formalization of this idea. Unlike at the time when the idea of bounded context parsing was explored first, in the 60's, the automatic generation of BCP parsers is feasible today. The programming language C, used here to give examples, is certainly among the more complex and difficult languages in respect to parsing. It still proved to be amenable to the new technique. The efficiency of a robust parser can be as good as that of a LR(k) parser.

Transition from LR(k) parsing to robust parsing is mainly a problem of rewriting the grammar. The extra effort seems worthwhile for applications where the input may contain errors or is only a program fragment. If the input contained errors, the output of the robust parser is a much better starting point for further processing than the information provided by a LR(k) parser.

In the process of studying many examples, during the work on robust parsing, it became more and more obvious that reading a program mainly from left to right and backing up a little bit in case something is unclear, is the natural way for a human being to read a program. Human beings are not good at switching between different states while reading. In contrast, as [17] observes, a computer can use state information easily and this allows different parsing algorithms. Robust parsing, seen from this perspective, is a step towards bringing the way a computer sees a program and the way a programmer sees the program closer together. Hopefully, it will make it easier for the programmer to understand the computers diagnostic messages.

In the long run, the greatest benefits of robust parsing could be its influence on language design. If BCP parsing captures more closely the way human beings read programs, a programming language designed for BCP parsing is a better human-computer interface.

Two follow-up projects are under way using the parser-generator described here: A minimum-distance pretty-printer and a system for heuristic error correction. So far, the experiences are very encouraging.

References

[1] Alfred V. Aho and Thomas G. Peterson. A minimum distance error-correcting parser for context-free languages. *SIAM Journal on Computing*, 1(4):305–312, December 1972.

[2] American National Standards Institute, New York, NY. *American National Standard for Information Systems —Programming Language—C, ANSI X3.159-1989*, 1990.

[3] Joseph Bates and Alon Lavie. Recognizing substrings of LR(k) languages in linear time. *ACM Transactions on Programming Languages and Systems*, 16(3):1051–1077, May 1994.

[4] Michael G. Burke and Gerald A. Fisher. A practical method of LR and LL syntactic error diagnosis and recovery. *ACM Transactions on Programming Languages and Systems*, 9(2):164–197, April 1987.

[5] Philippe Charles. An LR(k) error diagnosis and recovery method. In *Proc. of the 2nd. Int. Workshop on Parsing Technologies*, Tanncun, Mexico, February 1991. SIGPARSE.

[6] G.V. Cormack. An LR substring parser for noncorrecting syntax error recovery. *SIGPLAN Notices*, 24(7):161–169, 1989.

[7] J. Early. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(4):94–102, February 1970.

[8] Robert W. Floyd. Bounded context syntactic analysis. *Communications of the ACM*, 7(2):62–67, 1964.

[9] Susan L. Graham. *Precedence Languages and Bounded Right Context Languages*. PhD thesis, Stanford University, 1971.

[10] Susan L. Graham, Michael A. Harrison, and Walter L. Ruzzo. An improved context-free recognizer. *ACM Transactions on Programming Languages and Systems*, 2(3):415–462, July 1980.

[11] Susan L. Graham and Steven P. Rhodes. Practical syntactic error recovery. *Communications of the ACM*, 18(11):639–650, November 1975.

[12] James J. Hornig. What the compiler should tell the user. In *Compiler Construction: An Advanced Course*, number 21 in Lecture Notes in Computer Science, chapter 5, pages 525–548. Springer, Berlin, 1974.

[13] E. T. Irons. An error-correcting parse algorithm. *Communications of the ACM*, 6(11):669–673, November 1963.

[14] Paul S. Jacobs, George R. Kupka, and Lisa F. Rau. Lexico-semantic pattern matching as a companion to parsing in text understanding. In *Proc. of the Speech and Natural Language Workshop*, pages 337–340, Pacific Grove, CA, February 1991. DARPA.

[15] Mark A. Jones and Jason M. Eisner. A probabilistic parser applied to software testing documents. In *Proc. of the 10th National Conference on Artificial Intelligence*, pages 322–328, San Jose, CA, July 1992. AAAI.

[16] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, 1988.

[17] Donald E. Knuth. On the translation of languages from left to right. *Information and Controll*, 8:607–639, 1965.

[18] Esther König. Incremental syntactic and semantic processing. In *Proc. of the 12th International Joint Conference on Artificial Intelligence*, pages 925–930, Sydney, Australia, August 1991.

[19] B. Lang. Parsing incomplete sentences. In *Proc. of the 12th International Conference on Computational Linguistics*, pages 365–371. Association for Computational Linguistics, 1988.

[20] R. P. Leinius. *Error Detection and Recovery for Syntax Directed Compiler Systems*. PhD thesis, Univ. of Wisconsin, Computer Science Dept., Madison, Wisconsin, 1970.

[21] J.-P. Lévy. Automatic correction of syntax-errors in programming languages. *Acta Informatica*, 4:271–292, 1975.

[22] Jon Mauney and Charles N. Fisher. A forward move algorithm for LL and LR parsers. *SIGPLAN Notices*, 17(6):79–87, 1982.

[23] Robert Moore and John Dowding. Efficient bottom-up parsing. In *Proc. of the Speech and Natural Language Workshop*, pages 200–203, Pacific Grove, CA, February 1991. DARPA.

[24] Ajit B. Pai and Richard B. Kieburtz. Global context recovery: A new strategy for syntactic error recovery by table-driven parsers. *ACM Transactions on Programming Languages and Systems*, 2(1):18–41, January 1980.

[25] Jan Rekers and Wilco Koorn. Substring parsing for arbitrary context-free grammars. *SIGPLAN Notices*, 26(5):59–66, May 1991.

[26] Helmut Richter. Noncorrecting syntax error recovery. *ACM Transactions on Programming Languages and Systems*, 7(3):478–489, July 1985.

[27] Johannes Röhrich. Methods for the automatic construction of error correcting parsers. *Acta Informatica*, 13:115–139, 1980.

[28] Martin Ruckert. Conservative pretty printing. *SIGPLAN Notices*, 32(2):45–53, 1997.

[29] Seppo Sippu and Elijas Soisalon-Soininen. A syntax-error-handling technique and its experimental analysis. *ACM Transactions on Programming Languages and Systems*, 5(4):656–679, October 1983.

[30] Gregor Snelting. How to build LR parsers which accept incomplete input. *SIGPLAN Notices*, 25(4):51–58, 1990.

[31] Thomas G. Szymanski. *Generalized Bottom-Up Parsing*. PhD thesis, Cornel University, Ithaca, NY, May 1973.

[32] Thomas G. Szymanski and John H. Williams. Non-canonical extensions of bottom-up parsing techniques. Technical Report 75-226, Cornel University, Ithaca, NY, January 1975.

[33] Kuo-Chung Tai. Syntactic error correction in programming languages. *IEEE Transactions on Software Engineering*, 4(5):414–425, September 1978.

[34] Kuo-Chung Tai. Noncanonical SLR(1) grammars. *ACM Transactions on Programming Languages and Systems*, 1(2):295–320, 1979.

[35] M. Tomita. *Efficient Parsing for Natural Languages*. Kluver Academic Publishers, 1986.

[36] Charles Wetherell. Why automatic error correctors fail. *Computer Languages*, 2:179–186, 1977.

[37] Thomas R. Wilcox, Alan M. Davis, and Michael H. Tindall. The design and implementation of a table driven, interactive diagnostic programming system. *Communications of the ACM*, 19(11):609–616, November 1976.

[38] John H. Williams. Bounded context parsable grammars. Technical Report 72-127, Cornel University, Ithaca, NY, April 1972.

[39] Sheryl Young. Using semantics to correct parser output for atis utterances. In *Proc. of the Speech and Natural Language Workshop*, pages 106–111, Pacific Grove, CA, February 1991. DARPA.

A Appendix: A Robust BCP Grammar for the Programming Language C

The following grammar is a subset of the complete grammar of C. To keep the grammar short, some features, like different assignment operators or structure definitions, are not supported. The focus is on expressions and statements.

The grammar is written as Prolog code, and is used immediately by the parser generator.

```
:- op(1100,xfx,'=>').
:- op(1050,xfx,'::').


/*rules*/

primary => id.
primary => num.
primary => character.
primary => string.
primary => real.
primary => prim_expr.
prim_expr => open, expr, close.


postfix => primary.
postfix => postfix, open, assign_list, close.
postfix => postfix, open, close.
postfix => postfix, index.
postfix => postfix, dot, id.
postfix => postfix, arrow, id.
postfix => postfix, plusplus.
postfix => postfix, minusminus.


index => open_array, assign, close_array.


assign_list => assign.
assign_list => assign_list, comma, assign_list.
ambiguous(assign_list, [assign_list, comma, assign_list]).
unambiguous(assign_list, [assign_list, comma, assign_list]).
```

```
unary => postfix.
unary => star, unary.
unary => ampersand, unary.
unary => plusplus, unary.
unary => minusminus, unary.
unary => minus, unary.
unary => plus, unary.
unary => tilde, unary.
unary => exclamation, unary.
unary => sizeof, prim_expr.
unary => sizeof, prim_type.


prim_type => open, base_type, close.
prim_type => open, pointer_type, close.


pointer_type => base_type, star.


cast => unary.
cast => prim_type, cast.


mult => cast.
mult => mult, star, cast.
mult => mult, slash, cast.
mult => mult, percent, cast.


add => mult.
add => add, plus, mult.
add => add, minus, mult.


shift => add.
shift => shift, lessless, add.
shift => shift, greatergreater, add.


rel => shift.
rel => rel, less, shift.
rel => rel, lessequal, shift.
rel => rel, greater, shift.
rel => rel, greaterequal, shift.
```

```
equality => rel.
equality => equality, equalequal, rel.
equality => equality, exclamationequal, rel.


bit_and => equality.
bit_and => bit_and, ampersand, equality.


bit_xor => bit_and.
bit_xor => bit_xor, caret, bit_and.


bit_or => bit_xor.
bit_or => bit_or, bar, bit_xor.


and => bit_or.
and => and, ampersandampersand, bit_or.


or => and.
or => or, barbar, and.


/* conditional operator ? : not supported */


assign => or.
assign => unary, equal, assign.
/* other assignment operators not supported */


expr => assign_list.


stmt_list => statement.
stmt_list => stmt_list, comma, stmt_list.
ambiguous(stmt_list, [stmt_list, stmt_list]).
ambiguous(stmt_list, [stmt_list] + [stmt_list]).
unambiguous(stmt_list, [stmt_list, stmt_list]).
```

```
if_stmt => if_part.
ambiguous(if_stmt, [if_part]).
if_stmt => if_part, else_part).
unambiguous(if_stmt,[if_part, else_part]).
if_stmt => if_part, else_if_part.
ambiguous(if_stmt, [if_part, else_if_part]).
ambiguous(if_stmt, [if_part]+[else_if_part]).
if_stmt => if_part, else_if_part, else_part.
unambiguous(if_stmt,[if_part, else_if_part, else_part]).

if_test => if, prim_expr.
if_part => if_test, statement.

else_if_part => else, if_test, statement.

else_if_part => else_if_part, else_if_part.
ambiguous(else_if_part, [else_if_part, else_if_part]).
ambiguous(else_if_part, [else_if_part]+[else_if_part]).
unambiguous(else_if_part, [else_if_part, else_if_part]).

else_part => else, not_if_stmt.


decl_stmt_list => stmt_list.
decl_stmt_list => declaration_list, stmt_list.


compound_stmt => begin, decl_stmt_list, end.

while_stmt => while, prim_expr, statement.
do_while_stmt => do, statement, while, prim_expr, no_semicolon.

switch_stmt => switch, prim_expr, statement.
goto_stmt => goto, num, no_semicolon.
return_stmt => return, expr, no_semicolon.
return_stmt => return, no_semicolon.
break_stmt => break, no_semicolon.
continue_stmt => continue, no_semicolon.

for_controll => expr, semicolon, expr, semicolon, expr.

for_test => open, for_controll, close.
for_stmt => for, for_test, statement.
```

```
label => num, colon.
label => case, expr,optional_colon.


not_if_stmt => compound_stmt.
not_if_stmt => while_stmt.
not_if_stmt => do_while_stmt.
not_if_stmt => switch_stmt.
not_if_stmt => goto_stmt.
not_if_stmt => break_stmt.
not_if_stmt => continue_stmt.
not_if_stmt => return_stmt.
not_if_stmt => for_stmt.
not_if_stmt => expr, no_semicolon.


statement => if_stmt.
statement => not_if_stmt.
statement => label, statement.


storage_class => extern.
storage_class => static.


type_qualifier => const.


base_type_spec => type_id.
base_type_spec => int.
base_type_spec => unsigned, int.
base_type_spec => short, int.
base_type_spec => unsigned, short, int.
base_type_spec => long, int.
base_type_spec => unsigned, long, int.
base_type_spec => void.
base_type_spec => float.
base_type_spec => double.
base_type_spec => long, double.
base_type_spec => char.
base_type_spec => signed, char.
base_type_spec => unsigned, char.

base_type => base_type_spec.
base_type => type_qualifier, base_type_spec.
```

```
declarator => direct_decl.
declarator => star, direct_decl.
declarator => star, type_qualifier, direct_decl.

direct_decl => id.
direct_decl => id, index.

declarator_list => declarator.
declarator_list => declarator_list, comma, declarator_list.
ambiguous(declarator_list, [declarator_list, comma, declarator_list]).
unambiguous(declarator_list, [declarator_list, comma, declarator_list]).

declaration => base_type, declarator_list,semicolon).
declaration => storage_class, base_type, declarator_list,semicolon).

declaration_list => declaration :: local_declarations.
declaration_list => declaration_list, declaration :: local_declarations.

param => base_type.
param => base_type, declarator.

param_list => param.
param_list => param_list, comma, param_list.
ambiguous(declarator_list, [param_list, comma, param_list]).
unambiguous(declarator_list, [param_list, comma, param_list]).


function_header => base_type, id, open, param_list, close
                      :: function_definition.
function_header => storage_class, base_type, id, open, param_list, close
                      :: function_definition.

function => function_header, compound_stmt.

program_element => function.
program_element => declaration :: global_declaration.
program_element => typedef, declaration :: type_definition.

program => program_element.
program => program, program.
ambiguous(program, [program] + [program]).
ambiguous(program, [program, program]).
unambiguous(program, [program, program]).

file => bof, bof, bof, program, eof.
```