INCREMENTAL LL(1) PARSING IN LANGUAGE-BASED EDITORS

John J. Shilling
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280
shilling@cc.gatech.edu

ABSTRACT

This paper introduces an efficient incremental LL(1) parsing algorithm for use in language-based editors that use the structure recognition approach. It features very fine grained analysis and a unique approach to parse control and error recovery. It also presents incomplete LL(1) grammars as a way of dealing with the complexity of full language grammars and as a mechanism for providing structured editor support for task languages that are only partially structured. The semantics of incomplete grammars are presented and it is shown how incomplete LL(1) grammars can be transformed into complete LL(1) grammars. The algorithms presented have been implemented in the fred language-based editor

INTRODUCTION

This paper introduces an efficient incremental LL(1) parsing algorithm for use in language-based editors that use the structure recognition approach. It is motivated by a style of interaction that parses the user input at intervals of very small granularity. A second motivation for the algorithm is the problem of changes internal to the editing buffer. Because incremental analysis can occur after each keystroke, an unrestricted parser will attempt to include too much into its focus before a change is complete causing the editor to detect erroneous states that will become irrelevant as the user completes the change. The parsing algorithms presented in this paper use the user focus as a guide in restricting parsing. The algorithm presented has been implemented in the fred language-based editor [Shi83, Shi85].

Incomplete LL(1) grammars are introduced as a way of dealing with the complexity of full language grammars and as a mechanism for providing structured editor support for task languages that are only partially structured. Incomplete grammars were introduced by Orailoglu [Ora83] for the fred editor [Shi85, Shi86] as a method of dealing with the complexity of full language grammars. Incomplete grammars allow incremental refinement of language grammars and also allow grammars to be defined for languages that are not LL(1). Defining an incomplete grammar for a non-LL(1) language allows the editor to give structured support for the LL(1) subset of the language rather than disallowing the language completely. Another useful application of incomplete grammars is in providing structured support for tasks whose languages are only partially structured. An example of this is a grammar that facilitates structured support for editing LaTeX documents. A LaTeX documents contains structured elements but much of the document can be treated as unstructured text.

This paper introduces incomplete LL(1) grammars and characterizes their parsing semantics. It then shows how the grammars can be translated into conventional LL(1) grammars, eliminating the need for specialized parsing algorithms.

INCREMENTAL LL(1) PARSING

The goal of incremental parsing is to re-establish a correct structuralization of the user's editing buffer after changes have been made. The approach taken must differ from straightforward *once-only* top-down parsing because a once-only parser never needs to reverse decisions after they are made. In incremental parsing decisions are unmade and sections of the parse tree are deleted, transformed, and grafted into new locations. At the same time, the amount of parsing actually done must be limited if the algorithms are going to provide real-time response to a user. The algorithms must first establish the scope of modifications and efficiently restructure the parse tree within this scope.

The parsing method described in this paper is more fine grained than previous methods. The goal is to restructure the editing buffer after each text-modifying keystroke of a user. The challenge is that it is often not possible to achieve a complete, correct structuralization because the user is in the process of making a change that is not yet complete. On the other hand, the user

```
while (TRUE)
    <user change>
    <retokenization>
    <preparation of Parse Tree (Sweep)>
    <incremental parse>
    <semantic update>
```

Figure 1: Change-Update Loop

should be notified at the earliest possible moment if an error is made. The solution to this conflict is to implement what is called *follow-the-cursor* parsing with *soft templates*. As a user makes changes the method will parse only up to (and including) the token that contains the cursor. This keeps it from trying to parse past the cursor when a user has not yet completed a change. Unsatisfied elements of a production are indicated to the user as soft templates. Soft templates visually show the user what is missing in the parse tree. They are *templates* in that they should a valid production at the point they appear but they are *soft* because they do not constrain the user in any way. Further text is brought into consideration through cursor movement. The incremental LL(1) parsing algorithms presented here are a generalization of the table driven LL(1) parsing algorithms presented by Lewis, Rosenkrantz, and Sterns [PLRS76] and use *Select*, *Nullable* and *Follows* tables.

THE CHANGE-UPDATE LOOP

As a user changes a program the editor executes the loop illustrated in figure 1 to achieve a correct restructuralization. The localized region of change must be retokenized, the tree prepared, and the new tree state incrementally parsed. The data structures of the non-incremental algorithm are extended to facilitate incremental parsing. The parsing queue is modified to handle both tokens and non-terminals so that subtrees from the parse tree do not always have to be broken down into tokens as they are moved to the parse queue. This means that the parsing tables must be expanded to take account of non-terminals. We now assume that both the *Select* table and the *Follows* table cross reference non-terminals with both tokens and non-terminals.

TOKENIZATION

We will regard the tokenization phase as a black box process that produces a series of tokens from the localized region of change. It is assumed that incremental tokenization produces a queue of tokens and two markers in the parse tree denoted the Lexical Left Boundary and the Lexical Right Boundary. These markers point out the region along the frontier of the parse tree (inclusive) that has become invalid as a result of the new tokenization.

TREE PREPARATION - SWEEP

The next step in the change-update loop is the tree preparation process called Sweep. This is the process that breaks down the affected region of the parse tree and prepares the tree for the parsing algorithm. Two nodes of the parse tree have special meaning in this process. They are called the *Common Ancestor* and the *Royal Node* and are defined as follows:

- The **Common Ancestor** is the lowest node in the parse tree that is an ancestor of both the Lexical Left Boundary and the Lexical Right Boundary.
- The **Royal Node** is the highest node in the parse tree such that the Lexical Left Boundary is the first token of the production[1]. If there is no such node then the Royal Node is the Lexical Left Boundary.

Two basic ideas drive the tree preparation. The first is that the region of the tree defined by Lexical Left Boundary, Lexical Right Boundary and Common Ancestor is invalidated because the tokens along its frontier have been recalculated. The second is that the subtree of the parse tree rooted at Royal Node is suspect because it was instantiated on the basis of a token that has been altered.

Figure 2 shows the Sweep algorithm. It begins by identifying the Common Ancestor and the Royal Node and then cleans the region modified by the lexical tokenization. This is a wedge in the parse tree that is bounded by the path from the Lexical Left Boundary to the Common Ancestor to the Lexical Right Boundary. All nodes on the interior of the modified region are deleted except the direct sons of the nodes along the boundary.

The algorithm must now decide what to do about the Royal Node. We distinguish two cases in dealing with the Royal Node based on the relationship between the Royal Node and the Common Ancestor. If the Royal Node is a descendent of the Common Ancestor then there is no conflict because there are no tokens in the subtree rooted at Royal Node. If Royal Node is the same as, or an ancestor of the Common Ancestor then the subtree rooted at the leftmost son of Common Ancestor is clipped. This will in general leave parts of the parse tree intact that may not be valid with the new tokenization.

Before exiting, the the Sweep algorithm pushes the current parse pointer back to the left in the parse tree

[1] We will ignore non-significant nodes such as error nodes and (usually) white space in this presentation

Sweep(LexLeftBound, LexRightBound):

```
CommonAncestor = CommonAncestor(LexLeftBound, LexRightBound);
RoyalNode = RoyalNode(LexLeftBound);

CleanRegion(LexLeftBound, LexRightBound, CommonAncestor);

if (RoyalNode in subtree of CommonAncestor)
    DeleteSubtree(RoyalNode);
else
    DeleteSubtree(LeftmostSon(CommonAncestor));
endif

BackUp(Parse Position);
```

Figure 2: Sweep

as far as it can until it hits a token. The first non-terminal to the right of that token becomes the location of the current parsing position.

INCREMENTAL PARSING

We now enter the actual incremental parsing algorithm. The idea of the algorithm is similar to straightforward LL(1) parsing with several major differences. The incremental algorithm must decide how to handle the situations when it advances to a satisfied token element but has a non-empty parsing queue and conversely when it empties the parsing queue but has unsatisfied productions in the parse tree. The second situation is handled in follow-the-cursor parsing by essentially doing nothing. We do not want to remove any further tokens from the parse tree so the algorithm simply leaves unsatisfied productions in the tree and displays them to the user as soft templates. In the first situation the algorithm needs to open up space in the parse tree to accommodate the elements of the parsing queue. This is done by invoking a conflict resolution algorithm described below. Following the description of the conflict resolution algorithm we will present two algorithms that together accomplish the incremental parsing desired. The first is the inner parsing algorithm that does most of the work and the second is the outer parsing algorithm that provides high level control.

CONFLICT RESOLUTION

In our parsing algorithm we will need to resolve a conflict if the element at the front of the parse queue cannot be parsed at the current parse position. The conflict can exist because there is already a token at Parse Position as described above or it can exist simply because the Queue Element does not fit into the terminal or non-terminal symbol at the Parse Position. The general algorithm would have grafted such an element as an error. That is not satisfactory here for two reasons. The first is that there are now non-terminal rooted subtrees on the Parse Queue as well as tokens. A subtree may not be parsable at this point but the tokens along its frontier may be. The second reason is that the algorithm does not have the guarantee that the subtree rooted at Parse Position is properly prepared to be parsed because it may not have deleted the entire subtree rooted at Royal Node in the Sweep algorithm.

The goal is to parse the elements of the parse queue by disrupting as small a region of the parse tree as possible. There is a conflict here because we want to parse the tokens in the parsing queue but we would like to keep the tokens that are on the tree intact if possible. Our solution to this is to give priority to the parsing of tokens before the cursor. This may mean dislocating tokens on the parse tree. If tokens are displaced, they are grafted to the tree as error nodes rather than moving them to the parsing queue.

We first present some definitions.

- As a generalization of the previous definition we define **Royal Node** is defined to be the highest node in the tree that has Parse Position as the first leaf of its frontier. If no such node exists then Royal Node is defined to be the node at Parse Position.
- **Decision Node** is defined to be the *lowest* node on the path from Parse Position to Royal Node that has the element at the front of the Parse Queue in its first set. If no such node exists then Decision Node is defined to be NULL.
- **List Node** is defined to be a node on the path from the Decision Node to the Royal Node (inclusive)

that is a list structured production. If no such node exists then List Node is defined to be NULL.

- **Nullable Node** is defined to be a node along the path from the Parse Position to the Royal Node that is nullable and has the element at the front of the Parse Queue in its follow set. If no such node exists then Nullable Node is defined to be NULL.

The Royal Node is the highest point in the parse tree where the token at Parse Position (or the token that previously was the first token of Parse Position) caused a decision to be made. The Decision Node, if it exists, is the lowest production along the path from Parse Position to Royal Node that the front of the Parse Queue can belong to. If the Decision Node exists then we can try to find a List Node. List Node is a place in the parse tree where a list production can be found. This makes it a place where we can wedge in a new production without tearing down any existing parse tree. At most one list node can be found because if there were two or more then there would be an ambiguous parse. Finally, Nullable Node is a node that can be nulled while still allowing the element at the front of the Parse Queue to be correctly parsed.

The algorithm for resolving the conflict is presented in figure 3. It first finds the four nodes described above. If List Node exists then the list production is expanded by an additional element using the **GraftNewList** subroutine. In the **StealProduction** subroutine the tokens in the subtree rooted at the node of the first parameter are grafted to the right as error nodes. The (tokenless) subtree rooted at the node is then deleted leaving an open non-terminal that is either nullable or has the element at the front of the parse queue in its first set. The final chance to avoid grafting an error token is if there is a non-terminal subtree at the front of the parse queue. In this case the nonterminal is removed and replaced with its children in the **Reduce** subroutine. This process continues until the algorithm has freed up a non-terminal in the parse tree or has emptied the parse queue.

INNER PARSING ALGORITHM

Figure 4 shows the inner parsing algorithm. This algorithm iterates through its parsing decisions until it runs out of tokens and/or runs out of open parse tree.

If the front of the parse queue and the predicted parse tree element at the current parsing position agree then the queue element is simply grafted onto the tree at the current position. The parse queue is then popped and the parse position advanced. It may be that there is not an exact match but that the queue element is in the select set of Parse Position. In that case the production

OuterParse

```
while (NOT Empty(ParseQueue)) do
    InnerParse(ParsePosition, ParseQueue);

    if ((Satisfied(ParsePosition))
    AND (NOT Empty(ParseQueue))) then
        ResolveConflict(ParsePosition);
    endif
endwhile

ErrorRecovery();
```

Figure 5: Outer Parse for Follow-the-Cursor Parsing

indicated is instantiated (there can be only one by LL(1) restrictions) and the Parse Position is advanced to the first element of the new production.

If neither of the above cases hold then the element at the front of the parse queue does not fit at the current position. The algorithm checks to see if there is a non-terminal subtree at the front of the parse queue that can be reduced. If this is not the case then it checks to see if Parse Position is nullable with Queue Element as a correct follow. If this is the case then the non-terminal at Parse Position is nulled and Parse Position advances. If none of the above cases holds then the conflict resolution algorithm is invoked.

OUTER PARSING ALGORITHM

The outer parsing algorithm provides high level control over the inner parsing algorithm. It resolves conflicts when Parse Position is advanced to a token and Parse Queue is not empty or Parse Queue is empty but Parse Position is a non-satisfied production element. The former case is handled by the conflict resolution algorithm. The latter case is allowed as a legal state in follow-the-cursor parsing because tokens to the right of the cursor are not taken to satisfy the parse position.

At the end of the normal parsing loop an error recovery algorithm is called. The Error Recovery algorithm is the only algorithm that is allowed to parse past the cursor. In follow-the-cursor parsing it is sometimes necessary to invoke the *Steal Production* process that grafts tokens as errors to the right of the current parse position. It is also possible that a token has been inserted which will resolve an error in the syntax of the user buffer if they were included in the parse. The idea of the Error Recovery algorithm is to probe into the error tokens directly past the cursor to see if these tokens can be parsed correctly.

An outline of the error recovery algorithm is presented

ResolveConflict(ParsePosition)

```
while ((NOT Empty(ParseQueue)) AND IsToken(ParsePosition)) do
    RoyalNode = FindRoyal(ParsePosition);
    DecisionNode = FindDecision(ParsePosition, RoyalNode, QueueElement)
    ListNode = FindList(DecisionNode, RoyalNode);
    NullableNode = FindNullable(ParsePosition, RoyalNode, QueueElement);

    if (ListNode != NULL) then
        ParsePosition = GraftNewList(ListNode, ParsePosition);
    elseif (DecisionNode != NULL) then
        ParsePosition = StealProduction(DecisionNode, ParsePosition);
    elseif (NullableNode != NULL) then
        ParsePosition = StealProduction(NullableNode, ParsePosition);
    elseif (IsNonterm(QueueElement)) then
        Reduce(ParseQueue);
    else
        GraftError(ParsePosition);
    endif
endwhile
```

Figure 3: Conflict Resolution Algorithm

InnerParse(ParsePosition, ParseQueue)

```
while ((NOT Empty(ParseQueue)) AND (NOT Satisfied(ParsePosition))) do
    QueueElement = Front(ParseQueue);
    if (QueueElement matches ParsePosition) then
        Graft(QueueElement, ParsePosition);
        Pop(ParseQueue);
        Advance(ParsePosition);
    elseif (Select[ParsePosition, QueueElement] != ERROR) then
        Instantiate(ParsePosition, Select[ParsePosition, QueueElement]);
        Advance(ParsePosition);
    elseif (QueueElement not a terminal) then
        Reduce(ParseQueue);
    elseif (Nullable(ParsePosition) AND (Follows(ParsePosition, QueueElement)) then
        NullProduction(ParsePosition);
        Advance(ParsePosition);
    else
        ResolveConflict(ParsePosition, ParseQueue);
    endif
endwhile
```

Figure 4: Inner Parsing Algorithm

ErrorRecovery

```
< Set Consistent Parse >
while (we have an error token) do
    if (token is parsable) then
        < Parse Token >
        if (Completed Structure)
            < Update Consistent Parse >
        endif
    else
        break;
    endif
endwhile

<Back up to last Consistent Parse>
```

Figure 6: Error Recovery

in 6. The algorithm begins by saving the current parse tree status, called the initial *consistent parse*. Each error token is then considered in turn. If the error token can be parsed correctly then that is done. If parsing the token completes a production in the parse tree then the consistent parse is updated to be the current parse state. The loop terminates when it runs out of error tokens or it encounters an error token that cannot be parsed correctly. It then backs up the state of the parse tree to the last consistent parse and exits.

INCOMPLETE GRAMMARS

Incomplete grammars presented here introduce two new non-terminal classes, *unstructured*[2] and *preferred* non-terminals, into language description grammars. Preferred non-terminals are the left-hand-sides of a special production class called preferred productions. Intuitively, the unstructured non-terminal class allows the language designer to have a production that escapes the structuralization process. A preferred production is a way of finding structure within the lack of structure of the unstructured non-terminal.

A conventional LL(1) grammar can be described as a tuple [PLRS76]

$$G = (S, T, N, P)$$

where

S is the start symbol of G, $S \in N$.

T is a finite set of terminal symbols.

N is a finite set of non-terminal symbols.

P is a set of production rules.

An incomplete LL(1) grammar is described as a tuple

$$G = (S, T, N, U, P, P_U)$$

where S, T, N, and P have their conventional meaning and

U is a distinguished set of non-terminal symbols denoted *unstructured*, $U \in N$.

P_U is a distinguished set of production rules *denoted preferred* productions, $P_U \in P$.

An unstructured non-terminal can occur at any point in the right-hand-side of a production rule. For the purpose of constructing the *select* sets of normal non-terminals (non-terminals that are not unstructured non-terminals) each occurrence of an unstructured non-terminal is treated as a unique, distinguished terminal symbol T_i, $T_i \notin T$. Thus a non-terminal's select set will contain an entry for each terminal symbol in its first set and an entry for any unstructured element that it can be derived from it. This is similar to the way that non-terminals are treated in incremental parsing. For parsing purposes we do not construct the *first* set of an unstructured element but we do construct the *follow* set of an unstructured element in the normal way. We do not construct the first sets for unstructured elements because their first sets vary at parse-time, depending on the shape of the parse tree. Intuitively, the run-time first sets vary because we want the unstructured element to act as a wild card non-terminal and accept any token that is not otherwise accepted at the point that the unstructured element occurs.

Consider, for example, the grammar:

```
A : a
  | C
  ;

B : b
  | C
  ;

C : Unstructured
  ;
```

If we are currently focussed at non-terminal A, we want any token except "a" to lead into production C. If we are focussed at non-terminal B, then we want any token but "b" to be accepted by C. Thus, the meaning of the same unstructured element (and by side-effect,

[2] Orailoglu refers to this non-terminal class as *Unknowns*.

C) will changed at run-time depending on the current parsing context when it is encountered.

A preferred production is a production that can find structure within an unstructured non-terminal. Its first set is calculated as for normal productions rules. Because the preferred production can be followed by the resumption of the unstructured non-terminal then the follow set should be anything that does not cause conflict with the preferred production. Thus if p $\in P_U$, y = left-hand-side(p),

$$\text{Follow(y)} \equiv \text{Can-Legally-Follow(y)}$$

where Can-Legally-Follow is a relation that generates the set of all tokens that can follow a non-terminal without causing a parsing conflict with that non-terminal.

TRANSFORMATIONS

Orailoglu devised specialized algorithms to parse based on incomplete grammars. This section will show how to transform an incomplete grammar into a complete grammar that can be parsed with conventional LL(1) algorithms. The obstacle to the traditional parsing of incomplete grammars has been that the first set of an unstructured element effectively changes at run-time depending on the state of the parse tree where the unstructured element is introduced. It will be shown that the decisions in Orailoglu's implementation which are made at run-time, can be predicted at the time the incomplete grammar is analyzed. This allows the incomplete grammar to be transformed into a complete grammar that recognizes the same language.

A simple example is presented to show the flavor of the material that will follow. Consider the incomplete grammar:

```
A : a c
  | b c
  | U (an unstructured element)
  ;
```

The token set of the grammar is {a, b, c, ERROR}. The intent of the grammar writer is clearly that a leading token of *a* will invoke the first right-hand-side, a leading token of *b* will invoke the second right-hand-side, and any other token will invoke the third right-hand-side because of the unstructured element. Thus the first set of the unstructured element is effectively {c, ERROR} and as a result the first set of non-terminal A is the entire token set.

Now consider the grammar:

```
A : b B          B : b
  | c C            | D
  ;                ;

C : c            D : d
  | D              | U
  ;                ;
```

The token set of this grammar is {b, c, d, ERROR}. The intent of the unstructured element in the grammar varies with the shape of the parse tree. If the current non-terminal is B then any token in the set {c, ERROR} will derive the unstructured element in D but if the non-terminal is C then any token in the set {b, ERROR} will derive the unstructured element. The thing to note is that this can be predicted at the time that the grammar is analyzed.

The above grammar can be transformed into the grammar:

```
A : b B          B : b
  | c C            | D
  ;                ;

C : c            D : d
  | D              | U1
  ;                ;

U1 : Ua ( Ub )*
   ;

Ua : b           Ub : b
   | c                | c
   | ERROR            | d
   ;                  | ERROR
                      ;
```

This grammar has the same token set as the previous grammar. The only difference is that three new productions are introduced to represent the structure of the incomplete element. The first production gives the conceptual structure of the incomplete element. The second production represents tokens that can occur first in the unstructured element and the third production represents what may follow the first element as the body of the unstructured element. Notice that *U1* contains any token that is not otherwise in the first set of D. This causes the grammar to be ambiguous because the token b is in the first set of both alternatives of non-terminal B and the token *c* is in the first set of both alternatives to non-terminal C. The key to the transformation method is to resolve the conflict in each case in favor of the alternative that does *not* derive the unstructured element. With this method of resolving the parsing ambiguity, the transformed grammar recognizes exactly the same language as the untransformed grammar.

The above example illustrates the spirit of the transformation method on a very simple grammar. The remainder of this section will show that the method can be applied to any incomplete grammar of the form described by Orailoglu [Ora83]

For parse table calculations each unstructured non-terminal is recognized as a separate production but is treated somewhat differently when checking LL(1) grammar restrictions. Although they are technically different elements, unstructured elements must satisfy some restrictions as if they were the same terminal. Two distinct unstructured elements cannot both occur in first set of a production or in the follow set of a production. There are also restrictions to avoid ambiguity. An incomplete element cannot be followed by another incomplete element, and incomplete elements can neither start nor end preferred productions. If a token is both in the first set of a preferred production and the follow set of an unstructured element then the conflict will be resolved in favor of the follow set. No token may appear in the first set of more than one preferred production because this would cause a grammar ambiguity.

An unstructured element may be legally derived at run-time if all of the following conditions apply:

- The current parsing position is a non-terminal that can derive the unstructured element in the grammar
- The current parse queue element is a token that is not in the select set of the current non-terminal.
- The current non-terminal is not nullable with the input token in its follow set[3].

If all of the above conditions apply then the tree is expanded to derive the unstructured element and the algorithm enters unstructured parsing mode. While in unstructured mode the parser accepts any token as part of the incomplete element until it receives a member of the follow set of the incomplete element or a member of the first set of a preferred production. If a member of the follow set is encountered then the incomplete element is closed. If a member of the first set of a preferred production is encountered then the preferred production is instantiated and parsed normally, and unstructured parsing is resumed when it completes.

The transformation approach will be to replace each unstructured element U by a non-terminal $U\prime$ which is the left-hand-side of a production rule of the form

$$U\prime \;:\; U_s\ (\ U_b\ )^*$$
$$;$$

where U_s derives tokens and preferred non-terminals that may start the unstructured element and U_b derives the set of tokens and non-terminals that may be in the body of the unstructured element.

The production rule for U_b is the easier of the two to calculate. The first step is to calculate the follow set of U in the normal manner. This calculation is already performed by the existing algorithms. This tells what *not* to include in the token set derivable from U_b. Let the set of preferred non-terminals be denoted P = p_1, ... , p_n and let

$$F \;=\; T - follow(U) - first(p_1) - \ldots - first(p_n)$$

Then the production rule for U_b is

$$\begin{array}{lll} U_b & : & t_1 \\ & | & \ldots \\ & | & t_k \\ & | & p_1 \\ & | & \ldots \\ & | & p_n \\ & ; & \end{array}$$

where t_1, ..., t_k are the elements of F.

This production correctly parses the internal part of the incomplete element because it derives all the preferred productions and all tokens not in the first set of a preferred production or in its follow set. If there is a conflict between the first set of some p_i and the follow set of U then, as before, the conflict is resolved in favor of the follow set.

The calculation of how unstructured elements can be derived involves not only calculation of the production rule for U_s but also the rules for resolving conflicts that arise in the select tables of the grammar. An unstructured element occurs in the right-hand-side of a production of the form

$$\begin{array}{lll} A & : & w\ U\ x \\ & | & rhs_2 \\ & | & \ldots \\ & | & rhs_n \\ & ; & \end{array}$$

where w and x may each be empty and where $n \neq 1$. Thus the simplest production rule containing an unstructured element is of the form

$$\begin{array}{lll} A & : & U \\ & ; & \end{array}$$

The first step in calculating a production rule for U_s is determining whether w is nullable. Let F be the set of tokens that can occur in the first set of U. If w is not

[3]This slight variation from Orailoglu's implementation is introduced to give a more consistent treatment of unstructured elements.

nullable then set F to the entire token set. Any parsing conflicts with w will be resolved in the parse table construction phase. If w is nullable then F must be calculated so that it does not cause a parsing conflict with w or with any other right-hand-side of the production rule. Thus, the lead-in to U can be

$$F = T - first(w) - first(rhs_2) - \ldots - first(rhs_n).$$

The set F is the select set of U for parsing purposes. This will keep members of the first set of a preferred production that are not in F from interfering with calculation of the select table. The set of tokens that can lead directly to U is then

$$F - first(p_1) - \ldots - first(p_n) = t_1, \ldots, t_j$$

and the production rule U_s is

$$\begin{array}{lll} U_s & : & t_1 \\ & | & \ldots \\ & | & t_j \\ & | & p_1 \\ & | & \ldots \\ & | & p_n \\ & ; & \end{array}$$

where some of the p_i may not be derivable because no member of their first set is a member of F. The is allowable because the first set of U has already been calculated.

Using F as the first set for U_s guarantees that the production $U\prime$ will not cause a parsing conflict with the first sets of the right-hand-sides of the production in which it occurs, but it may still cause a conflict in productions that can derive A. The key to the transformation method is to always resolve the ambiguity against the alternative that derives the unstructured element. The first step of this is to calculate the select table and follow sets in the usual manner, using the designated first sets for the transformed elements. Next comes the grammar validity check.

If there is a first-first conflict in the grammar then check to see if one of the alternatives derives a transformed unstructured element. If so, resolve the conflict by selecting the other alternative. If there is a first-follow conflict caused by the first set of an unstructured element in the follow set, remove the conflicting token from the first set of the following non-terminal that derives the unstructured element. If there is a first-follow conflict caused by an unstructured element in the first set of a non-terminal, then remove the token from the first set of the non-terminal that derives the unstructured element. The first-first conflicts should be resolved before the first-follow conflicts so that the problem of multiple conflicts does not arise. Note that all of these conflicts do not occur in the parse table construction for a parser that treats incomplete grammars specially because the unstructured elements are treated essentially as distinguished unique tokens in the grammar analysis.

The purpose of the above conflict resolution strategy is to make the decisions when the parse tables are built that the parser would make at run-time in a parser for incomplete grammars. To see that this is true, first consider the production U_s in the case where w in the grammar above is non-nullable. In an unstructured parser the incomplete element will be encountered and instantiated when w completes, i.e., when the parser encounters a legal follow of w. This is exactly what happens in the transformed grammar.

Suppose that w is nullable. Then the unstructured element can be derived directly by A and indirectly by productions that derive A. Assume that the current non-terminal is A. The unstructured element will be directly derived if the current token is not in the first set of w or the first set of any other right-hand-side of A, and if A is not nullable with the current token in the follow set. The same action is taken in the transformed grammar because $U\prime$ does not have any members of the first set of w or the other right-hand-sides in its first set.

Now assume that the current non-terminal is not A but one that can derive A. In the unstructured parser, the unstructured element in A can be derived if the current token is not in the first set of the current non-terminal and if the current non-terminal is not nullable with the token in its follow sets. These are exactly the conditions under which $U\prime$ can be derived in the transformed grammar. Tokens that would not derive the unstructured element above will not do so in the transformed grammar because of the manner in which parsing conflicts are resolved in the select table. The tokens that are left are those that do not cause conflicts and they derive the unstructured element.

The last point to establish is the validity of the grammar model in which the incomplete element was introduced. The model is valid because only one unstructured element needs to be concentrated on at a time. This is true because

- A non-terminal cannot have two separate unstructured elements in its first set.
- An unstructured element cannot have an unstructured element in its follow set.
- A preferred production cannot start or end with an unstructured element.

It has been shown that an incomplete grammar may be transformed into an equivalent complete grammar. Is

there any advantage in doing so? The grammar transformation introduces new productions and thus causes the parsing tables to increase in size. This will in turn cause the run-time parse tree data structured to grow in size. The transformed grammar will introduce approximately one extra parse tree node for each token that is parsed as part of an unstructured element. The transformation process also significantly increases the complexity of the grammar analysis process. The real advantage of the algorithm is that it allows the incomplete grammar to be parsed by a conventional LL(1) parser. This is an advantage because it makes the grammars more easily adapted to other parsers and because it reduces the complexity of the parsing algorithm.

PREVIOUS WORK

Syntax-directed editors such as the Cornell Synthesizer [RT84, TR81] allow phrases to be entered as text below some level in the syntax. Textual input is parsed by a stand alone bottom-up parser that begins with the nonterminal represented by the current placeholder. The parsed text must be able to be grafted onto the parse tree as a complete, correct subtree.

Carlo Ghezzi and Dino Mandrioli have developed a bottom up parsing algorithm with is based on the use of grammars that are both LR and RL [GM79b, GM79a]. The authors also have published an algorithm that is more complex but operates on a more general class of LR grammars [GM80]. The BABEL editor [Hor81] is based on the Ghezzi and Mandrioli symmetric algorithm. Programs are not permitted to be incomplete, and it is not possible to place unexpanded placeholders in the tree. Kirslis [CK84, Kir85] has extended the Ghezzi and Mandrioli LR(0) algorithm to LR(1), has modified the parsing algorithm to handle comments and introduced explicit error handling routines.

An editor dubbed SRE for Syntax Recognizing Editor has been developed at the University of Toronto [BHZ85]. This editor provides flexible error handling by dividing the parser function into two levels. A low-level parser guarantees that the user's program consists of a sequence of syntactically correct lines. A high-level parser guarantees that the syntactically legal lines form a syntactically legal program. Only low-level syntactic correctness is enforced while text is being entered. Syntax errors within lines are pointed our immediately and the user is forced to correct them before proceeding. Syntax errors between lines are only pointed out when the user requests a high-level parse. Morris and Schwartz [MS81] published a LL(1) parsing algorithm that maintains a sequence of syntactically correct parse trees.

Orailoglu implemented an LL(1) incremental parsing algorithm as part of the the restructuring programmable display editor (RPDE, now called Fred) at the University of Illinois [Ora83, Shi85]. The algorithm maintains a single parse tree but allows multiple errors with unrestricted parsing by invoking a simple context (and history) sensitive error recovery algorithm. The key disadvantage of the algorithm is that it lacks an effective means of limiting parsing and tends to parse forward too far, recovering from errors along the way, when changes are made to the internal structure of a program. Orailoglu [Ora83] provided the original implementation of incomplete grammars.

CONCLUSION

This paper presents an incremental LL(1) parsing algorithm that is suitable for use in language-based editors and that has been implemented in *Fred*, structured, screen-based editor. A keystroke intensive mode of user interaction motivates the follow-the-cursor style of parsing in which parsing is normally halted at the cursor, leaving suspensions in the parse tree that are indicated to the user as soft-templates. Algorithms for tree preparation, incremental parsing, and error recovery are presented. The algorithms implement a style of user interaction that is both efficient and convenient. It is efficient because the editor only needs to perform limited parsing after changes. It is convenient because the user is able to enjoy the benefit of structuralization while retaining complete freedom of program entry.

Incomplete LL(1) grammars are presented as a way of dealing with the complexity of full language grammars and as a mechanism for providing structured editor support for task languages that are only partially structured. Orailoglu devised specialized algorithms for parsing based on incomplete grammars. This work shows how the grammars can be translated into conventional LL(1) grammars, eliminating the need for specialized parsing algorithms.

References

[BHZ85] Frank J. Budinski, Richard C. Holt, and Safwat B. Zaky. Sre - a syntax recognizing editor. *Software-Practice and Experience*, 15(5):489–497, May 1985.

[CK84] Roy H. Campbell and Peter A Kirslis. The saga project: A system for software development. In Peter Henderson, editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, April 1984. (Released as ACM SOFTWARE

ENGINEERING Notes 9(3) and ACM SIGPLAN Notices 19(5).).

[GM79a] C. Ghezzi and D. Mandrioli. Augmenting parsers to support incrementality. *Journal of the ACM*, 27(3):564–579, July 1979.

[GM79b] C. Ghezzi and D. Mandrioli. Incremental parsing. *ACM Transactions on Programming Languages and Systems*, 1(1):58–70, July 1979.

[GM80] C. Ghezzi and D. Mandrioli. Augmenting parsers to support incrementality. *Journal of the ACM*, 27(3), July 1980.

[Hor81] M.R. Horton. *Design of a Multi-Language Editor with Static Error Detection Capabilities.* PhD thesis, University of California, Berkeley, July 1981. ERL Technical Reprot 81/53.

[Kir85] Peter A. Kirslis. *The SAGA Editor: A Language-Oriented Editor Based on Incremental LR(1) Parser.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, December 1985.

[MS81] J. Morris and M. Schwartz. The design of a language-directed editor for block structured languages. *SIGPLAN Notices*, 16(6):28–33, June 1981. Proceedings of ACM SIGPLAN/SIGOA Symposium on Text Manipulation, Portland.

[Ora83] A. Orailoglu. *Software Design Issues in the Implementation of Structured Editors.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1983.

[PLRS76] II P.M. Lewis, J. Rosenkrantz, and R.E. Stearns. *Compiler Design Theory.* Addison-Wesley, 1976.

[RT84] Thomas Reps and Tim Teitelbaum. The synthesizer generator. In Peter Henderson, editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, April 1984. (Released as ACM SOFTWARE ENGINEERING Notes 9(3) and ACM SIGPLAN Notices 19(5).).

[Shi83] John J. Shilling. Improvements to a structured, screen oriented editor. Technical Report Report No. UIUCDCS-R-83-1155, Department of Computer Science, University of Illinois at Urbana-Champaign, December 1983.

[Shi85] John J. Shilling. Fred: A program development tool. In *Proceedings of SOFTFAIR II (San Francisco, California, December 3-5, 1985)*, December 1985.

[Shi86] John J. Shilling. *Automated Reference Librarians for Program Libraries and Their Interaction with Language Based Editors.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, July 1986.

[TR81] T. Teitelbaum and T. Reps. The cornell program synthesizer: A syntax-directed programming environment. *Communications of the ACM*, 24(9), September 1981.