A Tool for Constructing Syntax-Directed Editors

Yung-Shen Chang and Nai-Wei Lin
Department of Computer Science and Information Engineering
National Chung Cheng University, Chia-Yi, 621, Taiwan, ROC
E-mail: {cysh92, naiwei}@cs.ccu.edu.tw

Abstract

Syntax-directed editors can perform syntax analysis during program editing. This capability allows programmmers to correct syntax errors in a much more efficient way. Because of this, syntax-directed editors have become an integral component in current integrated development environments. This paper describes a tool that is designed to facilitate the construction of syntax-directed editors. This tool consists of two components: a generator for incremental parsers and a simple application program interface that facilitates the integration of incremental parsers and editors. The application program interface is based on a simple editing model that can be applied in most editors.

1. Introduction

Traditional text editors can be used to edit programs and are language independent. Using text editors, a programs is developed by first editing the program and then compiling the program for syntax correctness. Program editing and compiling will be performed repeatedly until the program is syntactically correct.

Syntax-directed editors are language dependent and can perform syntax analysis during program editing. This capability allows programmmers to perform program editing and program compiling at the same time so that programmers can correct syntax errors in a much more efficient way. Because of this, syntax-directed editors have become an integral component in current integrated development environments.

The main technique applied in syntax-directed editors is incremental parsing. Incremental parsing allows the syntax analysis to be incrementally performed only on the modified portions of a program. This ability allows program editing and compiling to be performed at the same time.

This paper describes a tool that is designed to facilitate the construction of syntax-directed editors. This tool consists of two components: a generator for incremental parsers and a simple application program interface that facilitates the integration of incremental parsers and editors. The application program interface is based on a simple editing model that can be applied in most editors.

A parser generator can automatically generate a parser for a language based on the syntactical definition of the language. One of the design goals for our generator of incremental parsers is to base our generator on a widely used open source generator for batch parsers so that most of the work for a compiler can be easily migrated into the work for the corresponding syntax-directed editor. Therefore, our generator of incremental parsers is based on the widely used parser generator Bison [2].

The advance of graphical user interface techniques has promoted various editing libraries and tools. To facilitate the integration of the incremental parser generated by our generator with an arbitrary editor, we have designed a simple editing model to specify the most common interactions between an incremental parser and an editor. We have also implemented a simple application program interface to simplify the integration.

The remainder of this paper is organized as follows. Section 2 reviews the related work on incremental parsing. Section 3 describes the incremental parsing algorithm we used in our implementation. Section 4 presents the simple editing model we proposed. Finally, section 5 gives the conclusions of this paper.

2. Related work

The research on syntax-directed programming environments has been going on for many years. Between late 70's and early 80's, there have already be some research projects on developing such programming environment. Tim Teitelbaum presented a syntax-directed editing environment for FORTRAN called Cornell [5]. Users of Cornell need to learn and follow a set of conventions to edit FORTRAN programs. With these conventions, Cornell only allows users to enter syntax correct programs. Cornell does not provide the ability incremental parsing.

Pierpaolo Degano presented an approach to perform incremental parsing by classifying language constructs into groups in an ad hoc way [1]. If any modification to a program occurs, the parser just have to parse up to the group that the modification belongs to, and it does not have to parse the whole program. Hence, this approach is difficult to extend to support generators for incremental parsing.

A general incremental parsing algorithm should be able to verify syntax correctness of a program by parsing only the modified portions of the program. In order to achieve this, the incremental parsing algorithm should have the ability to reuse the previous parsing results. In general, a parse tree is maintained during an incremental parsing. Therefore, the reuse of the previous parsing results usually means the reuse of the nodes in the parse tree. There have been two main approaches to general incremental parsing. One is the state-matching approach [4] and the other is the sentential-form approach [7]. We now give a more detailed description to these two approaches.

2.1. The state-matching approach

The state-matching approach associates a state with each node of the parse tree and uses the state of a node to determine if we could reuse this node. The state-matching approach is based on Mandrioli's [3] work. Mandrioli presented the idea of using a threaded-tree to implement the incremental parsing. A threaded-tree is a combination of a parse tree and a parse stack.

The most mature state-matching approach is developed by Larchevêque [4]. The reuse of nodes in the state-matching approach is called the "node grafting". A node grafting replaces a subtree (the original program segments) on the parse tree with a new subtree (the modified program segments). The state-matching approach splits the original program into three parts *xyz*, where *x* and *z* represent the unmodified parts, *y* represents the part being modified. We will use *y'* to represent the part that replaces *y*. The state-matching approach can be outlined as follows:

1. Set the parser configuration to the state just after parsing the last terminal of *x*.
2. Parse *y'* exhaustively.
3. While parsing *z*, find an appropriate graft point that can achieve a maximal reuse of *z*.
4. Perform node grafting.

Appropriate graft point can be selected from the Common Ancestor (CA) nodes. We can start the searching from the Nearest Common Ancestor (NCA) node. There are many ways to find the NCA node, but in the perspective of the incremental parsing, most of them are time-consuming. For example, a naïve method costs $O(n^2)$, where n is the number of nodes in the parse tree. In order to find the graft point efficiently, Larchevêque presents a lazy computation method which works incrementally. Lazy NCA computation could find not only NCA node efficiently, but also could find the node to be deleted incidentally.

Larchevêque's implementation is based on Syntax® compiler compiler. However, the Syntax is not widely used today and the Larchevêque's implementation is also not publically available today.

2.2. The sentential-form approach

The sentential-form approach is based on Wuu Yang's [9] work, which is based on the sentential-form parsing theory. The sentential-form parsing doesn't split program into several parts, it just classifies the input into terminals and nonterminals. The parser will take different actions for terminal and nonterminals.

The most mature sentential-form approach is developed by Wagner and Graham [7]. Different from the state-matching approach, the sentential-form approach needs to modify the parsing table of the corresponding batch parser. With the assistance of the modified parsing table, the incremental parser could determine if a terminal or a nonterminal could be shifted into the parse stack.

Wagner's algorithm also needs to maintain a complete modification history of the parse tree by using a version control system. The version control system maintains three important versions of parse trees:

- the reference parse tree,
- the previous parse tree, and
- the current parse tree.

The reference version is the last parsed and syntax-correct version. The previous version is the version immediately prior to the start of parsing. The current version is the version that under constructing by the parser. Wager discussed the implementation of such a version system in [6]. Wager's implementation is also not publically available today.

2.3. Comparison of the two approaches

We compare the two approaches based on the following criteria:

- Error recovery: Both approaches need error recovery routine to handle syntax error, Wagner presents an error recovery scheme [8] to handle syntax error during incremental parsing. This scheme can also be used in conjunction with Larchevêque's algorithm.

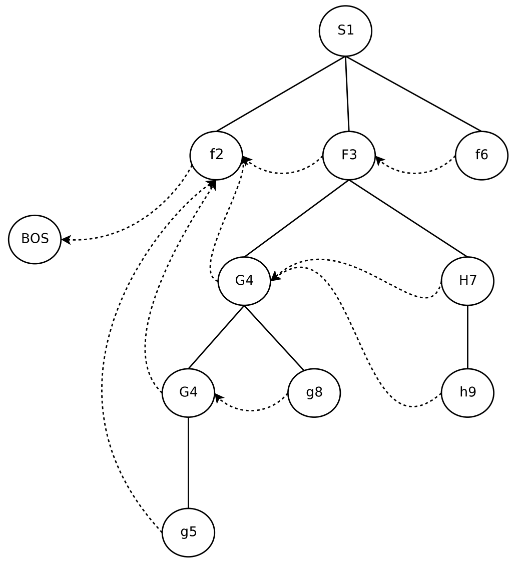


Figure 1. An example threaded tree.

- Overheads during parsing: Because of Wagner's method needs a version control system, it will need more computation power while retrieving nodes from the parse tree in contrast to Larchevêque's method. On the other hand, Wagner's method doesn't have to search NCA node for each reduce like Larchevêque's method, which needs less computation power in comparison to Larchevêque's method.
- Complexity of implementation: Since Wagner's method needs to modify the corresponding batch parsing table, and needs a version control system, the complexity of implemenetation of Wagner's method is higher than that of Larchevêque's method.

We choose Larchevêque's algorithm to implement our incremental parsing driver because of the following reasons:

- it does not have to modify batch parsing table, and
- it does not need a version control system.

3. A generator for incremental parsers

In this section, we will describe a generator for incremental parsers. This generator is based on Bison parser generator. We use the original Bison parsing table and extend the Bison parsing driver to incoporate the Larchevêque's incremental parsing algorithm so that it has the capability of incremental parsing.

3.1. Batch threaded-tree parsing

Larchevêque's algorithm uses a threaded-tree to perform incremental parsing. Threaded-trees are a variation of bianry search trees. But nodes of a threaded-tree are not restricted to have a degree of two.

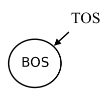
As we mentioned before, all parsing actions are operated on the threaded-trees. A threaded-tree is a combination of a parse stack and a parse tree. Threaded-tree parsing can be viewed as an implementation of traditional LR-parsing with a different data sturcture configuration. Traditional LR-parser is based on a parse stack and does not really construct a parse tree. Threaded-tree parsing explicitly construct a parse tree. Besides, each node in the threaded-tree has an extra field, called the threading field, to maintain the parse stack. Threaded-trees use the linked list of the threading fields to represent the parse stack. The threading field of a node representing a symbol *X* points to the node that representing the symbol just under *X* in parse stack. Figure 1 illustrated an example threaded-tree. The black lines represent the parse tree and the dotted lines represent the parse stack.

There are two special pointers for threaded-trees. The pointer TOS points to the top of the stack and the pointer BOS points to the bottom of the stack. Nodes between these two pointers are the current parse stack configuration. We now describe how to perform the shift and reduce action on the threaded-trees.

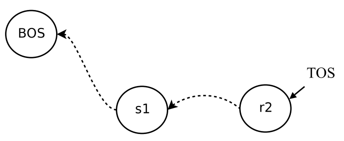
Shift: Assume that we are going to shift a terminal *r*, and the state of the parser after the shift of *r* is *S*. Then the shift procedure goes as follows.

1. Construct a node *N* whose symbol is *r* and state is *S*.
2. Set the threading field of *N* to the node that is pointed by TOS.
3. Point TOS to *N*.

Figure 2 illustrates such shift actions—shift s_1 and r_2.



(a) The empty stack



(b) The stack after shifting s_1, r_2

Figure 2. Shift action on threaded-tree.

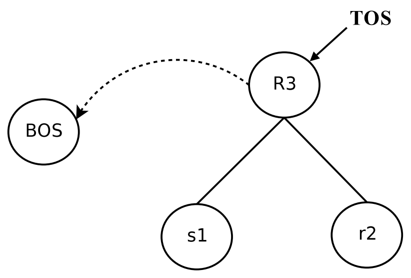


Figure 3. Reduce action on threaded-tree.

Reduce: Assume that we are going to reduce a production $R \rightarrow sr$, and the state of parser after the reduce is S. Then the reduce procedure goes as follows.

1. Construct a node N whose symbol is R and state is S.
2. Establish the connection between R and s, and R and r.
3. Set the threading field of N to the node pointed by the threading field of N's leftmost child (i.e., s).
4. Point TOS to N.

Figure 3 illustrates such a reduce action—reduce R_3.

The description above is about how to perform batch parsing on threaded-trees. Next, we're going to describe how to perform incremental parsing on threaded-trees.

3.2. Incremental threaded-tree parsing

As we mentioned in section 2, Larchevêque's algorithm consists of three steps and splits the original program into three parts *xyz*. We will use *y'* to represent the part that replaces *y*. The first step reuses the parsing results from the beginning to the end of the *x*. The second step parses *y'* exhaustively. Finally, the third step tries to find graft point candidate while parsing *z*.

In order to reuse the beginning to the end of the *x* directly, we have to reset the parse stack to the configuration that is just after shifting the last terminal of *x*. Becuase the threading fields of the threaded-tree keeps the previous parse stack information, we can reset the parse stack by traversing the threading fields of the threaded-tree. Assume that the last terminal of *x* is *t*, and *t*'s threading field is t_p. Then the parse stack resetting procedure goes as follows:

1. Find t_p via *t*'s threading field.
2. While t_p is not null,
3. push *t* onto the parse stack,
4. reassign t_p to *t*, search toward to the threaded-tree's root.

After we complete above instructions, we've reconstructed the parse stack. Next step is to parse *y'* exhaustively, this step is the same as the batch threaded-tree parsing, so we won't discuss this step in more detail. After finishing the parse of *y'*, we need to parse the last part *z* of program, and find graft point candidate to reuse the previous parse tree as much as possible.

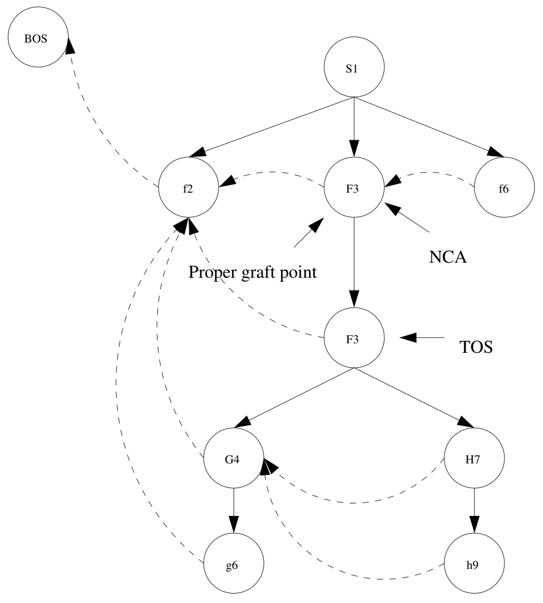


Figure 4. The operation of node grafting

A graft point candidate *C* must satisfy three requirements to be a final graft node. They are:

- the terminal predecessor of C must be the same as the predecessor of TOS,
- the terminal successor of C must be the same as the lookahead, and
- the symbol of C must be the same as the symbol of TOS.

If a candidate of the graft point satisfies these three requirements, we can use this node to replace the node pointed by TOS, which completes the incremental parsing algorithm. Figure 4 illustrates the operation of node grafting.

We select graft point candidate from CAs. The CAs are the nodes that dominate the parse tree of *y* and *y'*. We use Larchevêque's lazy NCA computation algorithm to find NCAs. Lazy NCA computation also works in incremental way. Lazy NCA computation algorithm don't computes the NCA from scratch each time. Lazy NCA algorithm computes partial CA from the information that it has right now. Figure 5 illustrates the operation of lazy NCA computation algorithm.

Lazy NCA algorithm uses flags to store reusing information, each node in the parse tree has three different states, namely "REUSED", "DISPOSABLE" and "UNMARK". REUSED state means this node has been reused, this node will be part of the new parse tree. DISPOSABLE state

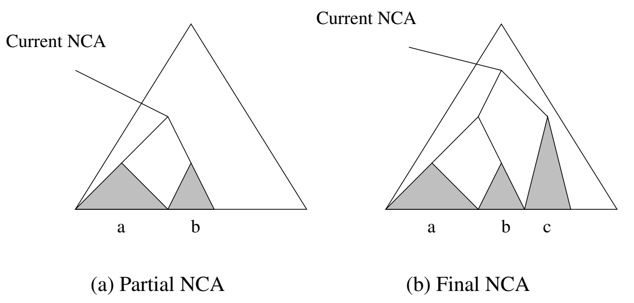


Figure 5. The operation of lazy NCA computation

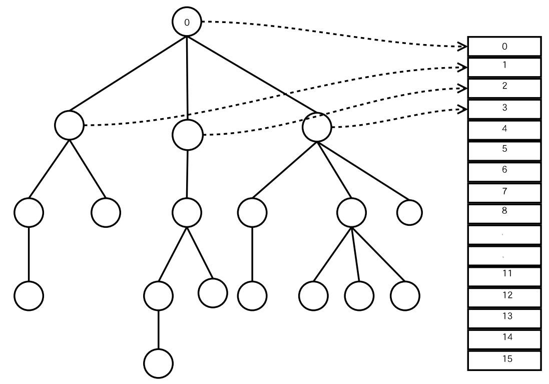


Figure 6. The data structure of threaded-trees

means this node won't appear in the new parse tree, we could delete this node directly. UNMARK represents this node have not been checked by the lazy NCA computation routine. We can know that the lazy NCA computation algorithm not only could find NCA node, but also could find disposable node which can be recycled for next reuse by the design of the DIPOSABLE flag.

All of the nodes are initialized as UNMARK at start. The node of y will be set as DISPOSABLE after initialization. The node will be set as REUSED when we reuse a node. When we mark a node, we need to check if all of its siblings are marked as DIPOSABLE or REUSED. If true, we apply same computation to the node's parent recursively until there exists a node whose siblings are all DIPOSABLE and REUSED.

Except the flag for each node, lazy NCA computation algorithm also uses a counter to record how many nodes we marked, we call this counter as "graftCount".

Calculation method of the graftCount is listed below:

$$graftCount = graftCount - number\ of\ (child(N)) + 1$$

We need to recalculate the graftCount when each time we mark a node as DISPOSABLE or REUSED, N is the node that we just marked. $child(N)$ means the list of N's children.

When the graftCount equals one, we need to check if the node last marked is the graft point by the three requirements mentioned above. If true, we can perform node grafting.

We concludes the procedure of lazy NCA computation algorithm as follows, step 1-3 is performed during initialization, step 4-5 is performed during parsing of z.

1. Retrieve the list of terminal node L of y
2. Set each node in the list L as DISPOSABLE, and increase the value of graftCount
3. Apply same computation to each node in the list L recursively
4. Mark nodes as REUSED when reusing nodes, and update the graftCount
5. While parsed to the z, start checking if we can find proper graft point
6. If tree, perform grafting

3.3. The implementation

We have implemented the parsing driver in C++. It consists of four classes.

- Class node: threaded-tree node managament module
- Class tree: threaded-tree managament module
- Class parser: batch threaded-tree parsing module
- Class incparser: incremental threaded-tree parsing module

Class `tree` manages threaded-trees, we implement a threaded-tree as an array of nodes of class `node`. Figure 6 illustrates the data structure of threaded-trees in our implementation. Class `parser` is the batch threaded-tree parsing module, this module only supports threade-tree version of shift and reduce actions. Class `incparser` extends class `parser` to have incremenal parsing ability. The application program interface discussed later will interact directly with class `incparser`.

The incremental parsing driver we implemented is very similar to the traditional LR-parsing driver. Only the behavior of reading input and the response after encountered EOF (End-Of-File) are different. When encountering the EOF input, the traditional LR-parsing driver will stop parsing. On the contrary, the incremental parsing driver will not stop parsing becuase of the EOF input. This ability ensures that the incremental parser could perform parsing repeatedly, which is the basic requirement of incremental parsing. Another difference is the behavior of processing tokens. The

```
void
incparser
(POS begin, POS end, string delta, ERR& err);
```

Figure 8. The prototype of function `incparser()`

traditional LR-parser reads tokens one-by-one, but the incremental parser reads all input at once and dispose the last EOF.

Figure 7 illustrates the operation of the incremenal parser. First, the incremenal parser reads input. It then determines which action to take from the information supplied by the parsing table. The incremental parser will call class `tree` to perform the corresponding parsing action (shift/reduce) after determining which action to take. Last, if we've analyzed the whole *y'*, then we can try to find graft point candidate and complete the parsing.

4. The editing model and API

In order to utilize the incremental parser, we developed an interface which can be called by the editor, and an edit model that can help the editor to gather the essential information that the editor needs to call the incremental parser.

4.1. The API

In order to simplify the complexity of calling incremental parser from the editor, the interface we supported is just a function called `incparser()`. Prototype of `incparser()` is illustrated in figure 8. `incparser()` has four parameters, namely `begin`, `end`, `delta`, `err`. First two parameters is used to store the start and the end point of the modification. Third prarmeter is used to store the user modified program. The last parameter is used to store the parsing result.

4.2. The editing model

All of the modifications can be classified into three kinds, and all of these modification could be represented as combination of start point, end point and a modification.

- insert: add content into the edit buffer between the begin and the end point, distance of the begin and end is determined
- delete: remove content from the edit buffer between the begin and the end point, distance of the begin and end is not determined.
- replace: Both insert and delete occurs between the between and the end point, distance of the begin and the end is not determined.

From the above description, we know that if the start point, end point and the user modified program is under our control, we could handle all kinds of modifications.

There are many ways to find the timing to trigger the incremental parser. For example, we could set a timer, then call the incremental parser periodically, or we could call the incremental parser simply by each key press. But both solutions might have the problem of wasting computing power. Tigger by timer might call the incremental parser even the user doesn't make any change, and trigger by key stroke calls the incremental parser too frequently, since most of the token is composed of several characters. So we introduce the edit model here to help the editor triggering incremental parser properly.

We classified all keys on the keyboard into two kinds, one is *modikey*, the other is *non-modikey*. modikeys represent the keys that will change the edit buffer if pressed, non-modikeys represent the keys that will not change the edit buffer if pressed. The editor is in the *modi-state* after pressing modikeys and is in the *nonmodi-state* after pressing non-modikeys. A state change occurs if the editor is changed from the state modi-state to nonmodi-state, or vice versa. The editor has to monitor the state changes to perform proper actions. Actions should be performed as follows.

- modi-state $\rightarrow$ nonmodi-state: trigger incremental parser,
- nonmodi-state $\rightarrow$ modi-state: record the start and the end points by the current cursor positions.

In modikeys, there are two keys that the editor should handle specially, which is DEL (delete) and BS (backspace). The DEL key will modify the end point of the modification, the BS key might modify the start point of the modification. When the editor receieves a DEL key, the editor need to update the end point one step forward. On the other hand, when the editor receieves a BS key, we need to update the start point depending on the value of the counter. We use a counter to record how many modikeys have been pressed (except DEL/BS). This counter will be increased if modikeys pressed, decreased if BS pressed. If the editor discovers the counter is less than zero, the editor need to update the start point one step backward.

Except handling token level modification, user replaces partial token content in most case. To solve this problem, incremental parser records position of each parse tree node in the edit buffer, and tries to complete the lexeme when needed. When the start or end point of the modification is in the middle of a token, The interface will completes the

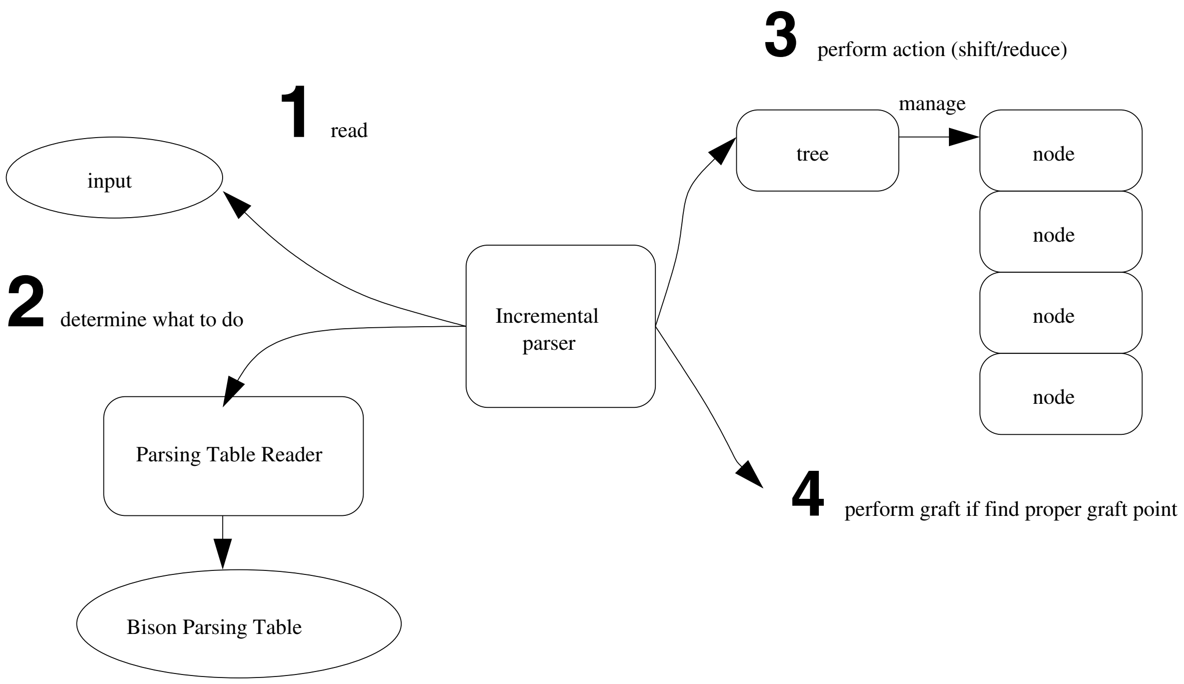


Figure 7. The operation of the incremental parser.

token by the direction which makes the modification area larger. For example, if the end point is in the middle of a token *T*, the interface will advance the end point to the end of *T*, and append the characters between end point to the end of *T* to the user modified program, and vice versa. After completes these actions, the interface could call the incremental parser to perform incremental parsing.

We use an example program listed in figure 9. We assume the initial cursor position is at (1, 13), the editor is under non-modikeys state and the user modification is:

1. Press BS and DEL for two times
2. key in `int argc, char* argv[]`
3. Press "→" key

At the beginning, since the editor is under the *non-modikeys* state, the start and end position are both N/A. Then the user presses BS and DEL for two times, this makes the editor's changes to *modikeys* state. When the first BS pressed, the start and end position was (1, 13) and (1, 14). But since first two key press are BSs, the start position needs to move TWO characters backward (1, 11). After pressing DEL two times, the end position needs to move TWO characters forward (1, 16). Then the user keys in `int argc, char* argv[]`, this make the counter increase to 21, and cursor position moves to (1, 33). Last, the user presses the "→" key, because the state changes from *modikeys* to *non-modikeys*, the editor should call interface of the incremental parser with the information we gathered right now. After

```
int main (void)
{
        while (1) {
                printf("Hello,␣World\n");
        }
        return EXIT_SUCCESS;
}
```

Figure 9. Edit Model Example

the incremental parser completes its work, the editor could check the variable `err`'s value to determine if the user modification is syntax correct. Example of a modification is illustrated in table 1.

5. Conclusions

Syntax-directed editors can perform syntax analysis during program editing. This capability allows programmmers to correct syntax errors in a much more efficient way. Because of this, syntax-directed editors have become an integral component in current integrated development environments. This paper describes a tool that is designed to facilitate the construction of syntax-directed editors. This tool consists of two components: a generator for incremental parsers and a simple application program interface that facilitates the integration of incremental parsers and editors. The application program interface is based on a simple editing model. In the future, we wish to incoporate the capabil-

Modificaion	Start pos.	End pos.	Cursor pos.	Counter	Inc. parsing
Initial pos.	N/A	N/A	(1, 13)	0	
Press BS and DEL for two times	(1, 11)	(1, 16)	0	0	
Key in `int argc, char* argv[]`	(1, 11)	(1, 16)	(1, 33)	21	
Press “→” key	N/A	N/A	(1, 34)	0	X

Table 1. An example of a program modification

ity of incremental lexing and incremental semantic analysis to our generator.

References

[1] P. Degano, S. Mannucci, and B. Mojana. Efficient incremental LR parsing for syntax-directed editors. *ACM Transactions on Programming Languages and Systems*, 10(3):345–373, 1988.

[2] C. Donnelly and R. Stallman. *Bison: the YACC-compatible Parser Generator*. Free Software Foundation, 1st edition, 1992.

[3] C. Ghezzi and D. Mandrioli. Augmenting parsers to support incrementality. *Journal of ACM*, 27(3):564–579, 1980.

[4] J.-M. Larchevêque. Optimal incremental parsing. *ACM Transactions on Programming Languages and Systems*, 17(1):1–15, 1995.

[5] T. Teitelbaum and T. Reps. The cornell program synthesizer: a syntax-directed programming environment. *Communication of ACM*, 24(9):563–573, 1981.

[6] Tim A. Wagner and Susan L. Graham. Efficient self-versioning documents. *CompCon IEEE Computer Society Press*, pages 62–69, 1997.

[7] T. A. Wagner and S. L. Graham. Efficient and flexible incremental parsing. *ACM Transactions on Programming Languages and Systems*, 20(5):980–1013, 1998.

[8] T. A. Wagner and S. L. Graham. History-Sensitive Error Recovery. *IEEE Transactions on software engineering*, 1999.

[9] W. Yang. Incremental LR Parsing. *International Computer Symposium Conference Proceedings*, 1, 1994.