ELSEVIER

Information Processing Letters 48 (1993) 67–72

Information Processing Letters

An incremental LL(1) parsing algorithm

Wuu Yang *

Department of Computer and Information Science, National Chiao-Tung University, 1001 Ta-Hsueh Road, Hsinchu, Taiwan, ROC

Communicated by D. Gries
Received 21 April 1993
Revised 12 July 1993 and 10 August 1993

Abstract

Given a parse tree for a sentence xzy and a string $\bar{z}$, an incremental parser builds the parse tree for the sentence $x\bar{z}y$ by reusing as much of the parse tree for xzy as possible. The incremental LL(1) parsing algorithm in this paper makes use of a break-point table to identify reusable subtrees of the original parse tree in building the new parse tree. The break-point table may be computed from the grammar.

Key words: Design of algorithms; Incremental parsing; Language-based editor; Language processors; Parsing

1. Introduction

Due to the improvement in hardware and software, language-based editors are becoming increasingly feasible and increasingly capable. These editors check the syntax and semantics of a program and provide immediate feedback to the user when the program is entered into the system. An important component of these editors is an incremental parser that builds a new parse tree for a modified program.

Given a parse tree for a sentence xzy and a string $\bar{z}$, an incremental parser builds the parse tree for the sentence $x\bar{z}y$ by reusing as much of the parse tree for xzy as possible. A "cut" operation breaks the parse tree into a sequence of trees. After replacing z with $\bar{z}$, the resulting sequence of the trees are pasted together.

Fig. 1(a) depicts the parse tree for the expression "$id + id + id$" (the grammar is shown in Fig. 4). Suppose that the first "+" sign is replaced by "$*$", resulting in the expression "$id * id + id$". The tree is cut at the right relatives of the leftmost id node and of the leftmost "+" node (the right relatives of a symbol represent the parse stack immediately after that symbol is matched; we define the term formally in the next section). At this point, there are four trees, as shown in Fig. 1(b). The first tree is the main parse tree for the prefix "id ...". The second and the third trees are generated when the "+" sign is parsed. The last tree is for the suffix "... $id + id$". Since the first "+" sign is replaced by "$*$", the second and the third trees are discarded. A tree consist-

* This work was supported in part by National Science Council, Taiwan, R.O.C. under grant NSC 82-0113-E-009-265-T.

SSDI 0020-0190(93)E0151-9

ing of a single "*" node is inserted between the first and the fourth trees. The resulting three trees have to be pasted together.

The main parse tree can be reused without change. The second tree, which consists of a single token "*", is parsed with the conventional parsing steps, given the parse stack with Y and X on top. At this point, only two trees remain, as shown in Fig. 1(c). The nonterminal on top of the parse stack, T_1 (the subscripts are used to distinguish different occurrences of a symbol), does not match the root of the second tree, E. In order to paste the two trees together, T, E, and the next input token *id* are used to index into a break-point table to find the break point, which is T_2. This means that subtree T_2 of the second tree may be reused. The second tree is cut at T_2 and its right relative, X_2. Tree T_2 is then pasted to the main tree at node T_1. Similarly, tree X_2 is pasted to the main tree at X_1. The resulting parse tree is shown in Fig. 1(d). Note that trees T_2 and X_2 have been reused.

We present an incremental LL(1) parsing algorithm. The crux of the algorithm lies in the use of

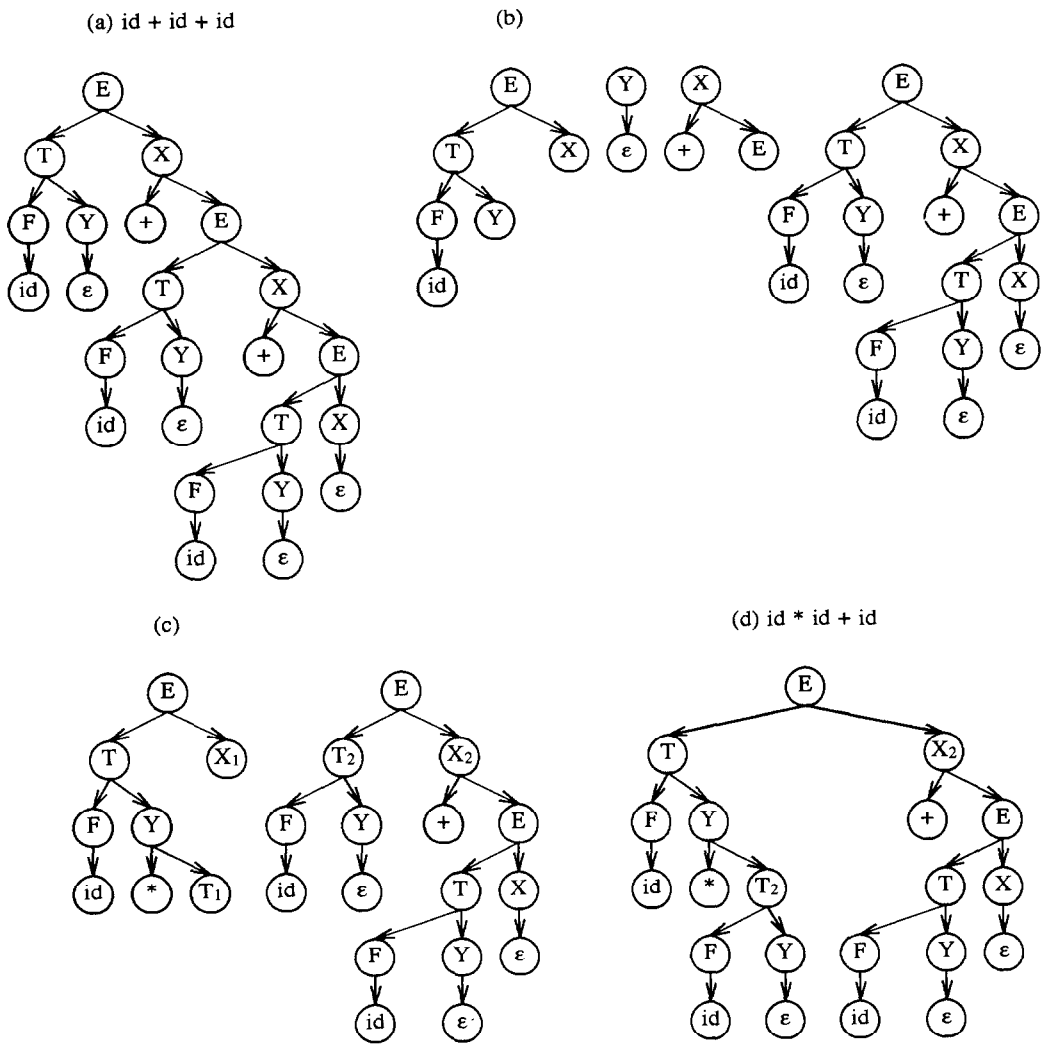

Fig. 1. An example parse tree for the incremental parser. Subscripts distinguish different occurrences of a symbol.

the break-point table to identify reusable subtrees. The next section presents the incremental LL(1) parsing algorithm. The third section discusses related work.

2. An incremental LL(1) parsing algorithm

We consider only LL(1) grammars. We first define some terminology. A *symbol* denotes either a terminal or a nonterminal. N denotes either a node of the parse tree or the symbol at that node. We use α or β to denote a (possibly empty) string of symbols. A *cut* of a tree at a node N splits the tree into two trees: One is the subtree rooted at N; the other is the original tree with N's descendants removed. Note that cutting a tree at a leaf node still results in two trees; one of the trees consists of a single leaf node. The *right relatives* of a node N consist of the siblings of N that are to the right of N and the right relatives of the parent of N. If N is a terminal node, N's right relatives, from left to right, represent the contents of the parse stack immediately after the terminal at node N is matched during parsing.

Set *FIRST* of a symbol N consists of all the terminals a such that $N \rightarrow^* a\beta$, where β is a string of symbols. If $N \rightarrow^* \varepsilon$ (where ε is the null string), then $\varepsilon \in FIRST(N)$. Set *FOLLOW* of a symbol N consist of all the terminals a such that $S \rightarrow^* \alpha Na\beta$ where S is the start symbol of the grammar and α and β are strings of symbols. The computation of sets *FIRST* and *FOLLOW* is described in [1].

Given a parse tree for a sentence xzy and a string $\bar{z}$ that replaces z, the parse tree is cut at the right relatives of the rightmost symbol of x and at the right relatives of the rightmost symbol of z. If x is a null string, the tree is cut at the root. If z is ε, the associated cut operations will not be performed. Cutting yields a sequence of trees. The first tree in the sequence, the *main* parse tree, is for the prefix "x ...". The parse stack corresponding to the main parse tree consists of the right relatives of the rightmost symbol of x.

Now consider all the trees except the main parse tree in the sequence. Since trees that derive null strings are not reused, they are discarded. Then $\bar{z}$ replaces the trees that derive z; each symbol of $\bar{z}$ represents a tree by itself. The trees are grafted onto the main parse tree. The situation is drawn in Fig. 2, where N is the root of the second tree.

To graft a tree to the main parse tree, the root, the leftmost terminal leaf of the tree, and the "attach" point in the main parse tree are examined. Specifically, in Fig. 2, consider grafting the tree rooted at node N to the main parse tree at node M, where M corresponds to the symbol on the top of the parse stack. If nodes M and N contain the same symbol, tree N is simply pasted at node M.

Suppose nodes M and N contain different symbols. Let a be the leftmost terminal leaf of the N tree. If $a \in FIRST(M)$, consider the *stems* from M to a and from N to a, where the stem is defined as follows.

Definition. The *stem* from a symbol M to a terminal a is the sequence of symbols on the path from M to the leftmost a in the parse tree for $M \rightarrow^* a\beta$. If $a \notin FIRST(M)$ then $stem(M, a) = \varepsilon$.

Note that $stem(M, a)$ is unique, since LL(1) grammars are deterministic. If no symbol can appear on both $stem(M, a)$ and $stem(N, a)$, the usual parsing steps are performed, using M as the symbol on the top of the parsing stack and a

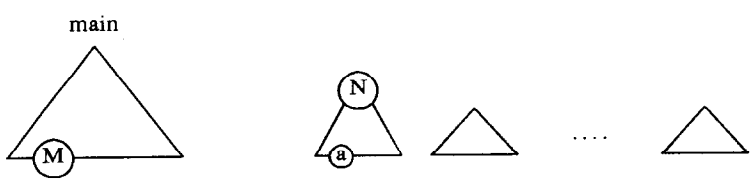

Fig. 2. The incremental parser attempts to graft the small subtrees on the right to the main parse tree on the left.

as the look-ahead symbol. If some symbol appears on both $stem(M, a)$ and $stem(N, a)$, we say that the two stems "overlap". When two stems overlap, they must share a common segment from bottom up, since LL(1) grammars are deterministic. Let the *break point* be (the symbol of) the "highest" node in the overlapped segment. The break points, which may be computed from the grammar, indicate reusable subtrees.

Definition. BP is the break-point table. Suppose that M and N are nonterminals and a is a terminal.

(1) If $a \in FIRST(N) \cap FIRST(M)$, then $BP[M, N, a]$ is the symbol of the highest node on the overlapped segment of $stem(N, a)$ and $stem(M, a)$.

(2) If $a \in FIRST(N)$, $a \notin FIRST(M)$, M may derive ε, and $a \in FOLLOW(M)$, then $BP[M, N, a] = N$.

(3) If $a \in FIRST(N)$, $a \notin FIRST(M)$, and either M may not derive ε or $a \notin FOLLOW(M)$, then $BP[M, N, a] = error$.

(4) If $a \notin FIRST(N)$ then $BP[M, N, a]$ is a *don't_care* entry.

Let $X = BP[M, N, a]$, which is not an *error* nor a *don't_care* entry. Tree N is cut at X and X's right relatives, resulting in a sequence of trees. Tree N and all the trees that derive the

```
Algorithm: inc-parse(main, z̄)
    /* main is the parse tree for the string xzy. z is replaced by z̄. */
    cut the main tree at the right relatives of the last symbols of x and z
    remove all trees that derive the null strings
    replace all trees that derive a symbol of z with z̄, each symbol of z̄ being a tree by itself.
    let S be the stack of the remaining trees, excluding the main tree (the leftmost tree is on the top)
    the parsing stack consists of the right relatives of the last symbol of x in the main tree
    repeat
        let M be the symbol on the top of the parsing stack
        pop a tree from the S stack, the root of which is denoted by N
        let a be the leftmost leaf terminal of the N tree
        if M = N then pop M from the parsing stack
        else if M is a terminal node then
            if M = a then
                pop M from the parsing stack
                cut the N tree at the right relatives of a
                remove the N tree and all trees that derive the null strings
                push the remaining trees onto the S stack (the leftmost tree is on the top)
            else error
        else /* M is a nonterminal node */
            if N is a nonterminal symbol and BP[M, N, a] = error then error
            else if N is a nonterminal symbol and N ≠ BP[M, N, a] then
                let X be BP[M, N, a]
                cut the N tree at X and the right relatives of X
                remove the N tree and all trees that derive the null strings
                push the remaining trees onto the S stack (the leftmost tree is on the top)
            else if P[M, a] = Y1Y2...Yk then
                pop M from the parsing stack
                push Yk, ..., Y2, Y1 onto the parsing stack
            else error
    until M is the end-of-file marker
```

Fig. 3. The incremental parsing algorithm.

null strings are then discarded. The remaining trees represent the input that must be parsed. Note that the root of the leftmost tree in the remaining trees is X. Call this leftmost tree the X tree. Suppose $a \in FIRST(M)$. Since $X \in stem(M, a)$, tree X will be reused during the parsing of a with M as the symbol on the top of the stack. On the other hand, when $a \notin FIRST(M)$, M must derive ε given a as the look-ahead terminal symbol. In either case, the usual parsing steps are performed.

Fig. 3 is a recast of the incremental parsing algorithm. In Fig. 3, P is the LL(1) parsing table. In addition to the parse stack, a stack S is maintained of the trees that need to be grafted to the main parse tree. Stack S represents the input to the incremental parser.

Fig. 4 shows an example grammar and table BP. The unspecified entries in the table are *don't_care* entries. The table is used to parse the example in Fig. 1 incrementally.

The incremental parser contains the parse tree and table BP in addition to the traditional LL(1) parsing table. Table BP contains m^2n entries, where m is the number of nonterminals and n is the number of terminals in the grammar. We conjecture that most entries in the BP table are either *error* or *don't_care* entries. For instance, 63 out of the 75 entries of table BP in Fig. 4 are such entries.

Consider the parse trees for x, $\bar{z}$, and y. The parse tree for x can be reused directly. Therefore, we save all the efforts of parsing x. The parse trees for $\bar{z}$ are actually a sequence of terminals, which is exactly what a traditional parser needs to process. The cut operations at a node's right relatives can be implemented efficiently with a threaded-tree structure similar to that used in [4,5]. At most one set of cut operations is performed for each terminal in the trees for y. However, each set of cut operations identifies a subtree that can be reused. As long as the subtree is large enough, the efforts spent in cutting can be offset by the efforts saved by reusing the subtree. Grammars for practical programming languages are designed with modularization in mind. This implies that syntactic structures are likely to limit the propagation of changes. For instance, changes within a statement will not affect the integrity of the following statements, though the relationship between the statements might be changed. Therefore, we conjecture that reused subtrees are quite substantial, and most parts of the trees for y can be reused.

In case the grammar is ambiguous, the conflicts during the construction of table BP are resolved in the same way as they are resolved during the construction of the parsing table. The resulting table BP is consistent with the parsing table in that a sentence in the language will have the same parse tree whether the tree is produced by a conventional parser or by the incremental parser.

We conclude this section with a formula to compute the *stems*. Given a nonterminal M and a terminal a such that $a \in FIRST(M)$, $stem(M, a) = \{M\} \cup stem(Q, a)$ where there is a production $M \to \alpha Q \beta$, $\varepsilon \in FIRST(\alpha)$, $a \notin$

(a) the grammar

$E \longrightarrow T\ X$
$X \longrightarrow \varepsilon$
$X \longrightarrow +E$
$T \longrightarrow F\ Y$
$Y \longrightarrow \varepsilon$
$Y \longrightarrow *\ T$
$F \longrightarrow id$

(b) the BP table

terminal = id

	E	T	F
E	E	T	F
X	error	error	error
T	T	T	F
Y	error	error	error
F	F	F	F

terminal = +

	X
E	error
X	X
T	error
Y	X
F	error

terminal = *

	Y
E	error
X	error
T	error
Y	Y
F	error

Fig. 4. An example grammar and the associated BP table.

$FIRST(\alpha)$, and $a \in FIRST(Q)$. (The notation $\{M\} \cup stem(Q, a)$ means to append M to the front of the sequence $stem(Q, a)$.)

3. Related work

The algorithm presented in this paper is motivated by the Magpie environment [8] and the Galaxy system [3]. The incremental LL(1) parsing algorithm in Magpie uses nonterminals, as well as terminals, as look-ahead symbols. It does not use a break-point table to identify reusable subtrees. An earlier version of our incremental parsing algorithm also used nonterminals as look-ahead symbols, but we found that that approach led to an unnecessarily complicated formulation of the algorithm. Galaxy uses recursive descent to reparse the parse tree in which changes have been marked by an incremental scanner. The incremental parser in that system does not use tables to guide parsing and to identify reusable subtrees. Instead, it simply tries different alternatives when a rule does not match the input. Furthermore, the incremental parser in Galaxy can process only a limited class of grammars, due to a limitation in the backtracking mechanism. By contrast, our break-point table can identify reusable subtrees. The incremental parsing in [4,5] discusses incremental LR parsing methods. The systems in [2,6,7] handle incremental changes to programs (mainly) with structure editors. Structure editors avoid the problem of incremental parsing by requesting a user to explicitly identify the modified syntactic structures.

References

[1] A.V. Aho, R. Sethi and J.D. Ullman, *Compilers: Principles, Techniques, and Tools* (Addison-Wesley, Reading, MA, 1986).

[2] L.V. Atkinson, J.J. McGregor and S.D. North, Context sensitive editing as an approach to incremental compilation, *Comput. J.* **24** (1981) 222–229.

[3] J.F. Beetem and A.F. Beetem, Incremental scanning and parsing with Galaxy, *IEEE Trans. Software Engineering* **17** (1991) 641–651.

[4] C. Ghezzi and D. Mandrioli, Incremental parsing, *ACM Trans. Programming Language Systems* **1** (1979) 58–70.

[5] C. Ghezzi and D. Mandrioli, Augmenting parsers to support incrementality, *J. ACM* **27** (1980) 564–579.

[6] R. Medina-Mora and P.H. Feiler, An incremental programming environment, *IEEE Trans. Software Engineering* **7** (1981) 472–482.

[7] S.P. Reiss, An approach to incremental compilation, in: *Proc. SIGPLAN 84 Symp. on Compiler Construction*, Montreal, Canada, 1984; *ACM SIGPLAN Notices* **19** (1984) 144–151.

[8] M.D. Schwartz, N.M. Delisle and V.S. Begwani, Incremental compilation in Magpie, in: *Proc. SIGPLAN 84 Symp. on Compiler Construction*, Montreal, Canada, 1984; *ACM SIGPLAN Notices* **19** (1984) 122–131.