Using an LALR compiler compiler to generate incremental parsers

J.M. Larchevêque

Altaïr
BP 105, 78153 Le Chesnay Cedex, France

1 Introduction

This paper describes an incremental parsing algorithm and its implementation using optimized LALR tables generated by the Syntax compiler compiler [Boullier 84] [Boullier and Jourdan 87]. The parser described is destined to be part of a Language-Based Editor (henceforth LBE), i.e. an editor which invokes an incremental compiler in an essentially transparent way as a program is being edited. This integration of a compiler into an editor creates a whole spectrum of interesting capabilities:

- immediate syntactic check,
- immediate semantic check,
- advanced debugging,
- static program analysis.

A given LBE does not need to provide all of these features, and as a matter of fact the LBE's described in the literature fall into two categories termed respectively synthetic and analytic editors in [Degano et al. 88]. A synthetic editor supports top-down program writing, in which the user handles an Abstract Syntax Tree rather than a text. This obviates the need for incremental parsing, and synthetic LBE's have been used to focus research on incremental semantic analysis or advanced debugging functions. Synthetic LBE's notably include *DICE* [Fritzson 83] and the *Cornell Program Synthetizer* [Reps et al. 83]. Research around the *CPS* has tended to focus on incremental attribute

evaluation, while *DICE* aims at providing sophisticated development and debugging capabilities, in particular a static analysis database comparable to Interlisp's Masterscope [Teitelman 75]. On the other hand, an *analytic* LBE supports anything from *structured editing*, i.e. editing based on syntactically meaningful substring substitutions (as in the scheme proposed in [Degano et al. 88]), to unrestricted editing of the kind supported by an ordinary text editor. Now, the literature on analytic LBE's focuses on the problem of incremental *parsing* and ignores backend design. So it goes without saying that the ideal LBE would combine the natural interaction supported by an unrestricted analytic LBE with the sophisticated capabilities which —judging from publications on the subject— only synthetic LBE's are currently offering.
The type of incremental parser which will be described here is expected to speed up semantic analysis under certain conditions, and is thus proposed as a contribution to the design of a full-fledged analytic LBE. An LBE using this type of parser is currently being developed for the O_2 language [Lecluse and Richard 89]. O_2 is a database programming language and as such has to meet conflicting demands, with the programming aspect calling for a high level of interactivity and dynamicity, and the database aspect calling for fast execution and tight type-checking. Incremental compilation seemed the right solution to this dilemma, and its study is likely to gain more and more importance with the development of persistent languages and object-oriented DBMS's.

2 Main characteristics of the parsing algorithm

The task of incremental parsing (henceforth IP) consists in parsing a string $w' = x_0y'_0x_1y'_1\ldots x_ny'_n$ using information (generally some form of parse tree) recorded about the syntax of $w = x_0y_0x_1y_1\ldots x_ny_n$ (where any of the x_i, y_i or y'_i can be empty). For the sake of simplicity we will provisionally define this task as the parsing of the string $w' = xy'z$ given syntactic data for $w = xyz$.
The algorithm described here, like most IP algorithms described in the literature, uses LR tables. In addition, unlike its predecessors, it takes into account the consequences of the parsing policy on the cost of attribute-based semantic analysis. In this perspective, an IP algorithm will be optimal if it leads to minimal attribute recomputation, considering both parse-time attribute evaluation and the propagation of attribute values at the

end of a parse. From the point of view of the parser, this optimality criterion amounts to minimality in the number of reductions performed when incrementally analyzing a string of tokens. To see this, first consider that this property, which from now on will be referred to as *reductive minimality*, has two consequences: *(i)* maximal subtree reuse, meaning that subtrees representing unchanged substrings and valid in the new context are not dismantled and subsequently rebuilt by superfluous reductions, and *(ii)* maximal context reuse, meaning that nonterminals are preserved along the longest possible paths from the root[1]. Property *(i)* minimizes parse-time attribute computation, while property *(ii)* minimizes propagation of inherited attribute values. So, it appears that *(i)* is likely to be more important than *(ii)* from the point of view of semantic analysis, while *(ii)* is likely to save considerable parse time and is indeed the exclusive goal of most IP algorithms. The IP algorithm described in this paper is first given in a "pure" form (section 3), which requires the domain of the `goto` function to be known. In this form, the algorithm is shown to be reductively minimal. Then, an implementation using the optimized parse tables generated by Syntax is described and evaluated with respect to reductive minimality (section 4).
An issue which will not be treated here is incremental lexical analysis and its relation to IP, so that the lexical specification is assumed to be sufficiently simple for any character-substring substitution to be easily mapped to a token-substring substitution.

3 Description of the algorithm

The following description provisionally supposes that the grammar contains no ϵ-production, and that both of `w` (the string already parsed) and `w'` (the string to parse) are acceptable inputs. As already mentioned, we also make the temporary assumption that `w` and `w'` differ by a single substring, so that we have `w` = `xyz` and `w'` = `xy'z`, where any of `x`, `y`, `y'`, or `z` can be empty.
All of these restrictions will subsequently be lifted.

[1]Even if a particular IP algorithm does not concretely use parse trees, this description is sufficiently general for most IP algorithms to be evaluated with respect to subtree reuse and context reuse (for a survey of IP algorithms, see section 5).

3.1 Main data structures

The structure associated to w is a parse tree with special properties. First of all, a suitable indexing system (e.g. using a synthetized attribute) makes it possible to map a character offset to the ancestors of the terminal symbol at that offset. Second, following the terminology of [Ghezzi and Mandrioli 79], the parse tree is *threaded*, meaning that any node contains enough information to retrieve the contents of the parse stack at the time this node was created. More specifically, each node contains the state the automaton went into after the node was created (by a shift or a reduction) and points to the node containing the previous state. For the purpose of threading, we have a bottom-of-stack node which does not correspond to any symbol, but contains the initial state.

3.2 Illustrative example and notation

An example of a threaded tree is the subtree under L_s on figure 1(c), which represents a parse of the string "aabbc" given the grammar of figure 1(a). Threading links are represented by dotted lines[2]. A node is represented by a sequence of the form n: [X, s_i], where n is a unique identifier for the node, X is a symbol, and s_i a parse state.

In the sequel, we will call TOS the node containing the current state, and the fields of a node will be called respectively symbol, state, and link. With this notation, TOS.state denotes the current state. In addition, lhd will denote the lookahead token.

3.3 Initial parse

An initial parse consists in parsing w' = xy'z when all of x, y, and z are empty. The threaded tree given for the empty string w = xyz is shown on the left of figure 1(c). It consists of the node ROOT, with a *gap* as its sole descendant. A gap is a placeholder for a node, it is defined by its position in the tree (and therefore in the threading). In the figure, a gap is represented by a T-shaped branch labeled with the symbol expected at the position (in curly brackets).

In an initial parse, the definitions of the shift and reduce operations are as follow:

shift Advance the reading head. Create a new node n, with n.symbol the token shifted and n.state the new state after the shift, i.e. goto(TOS.state, n.symbol).

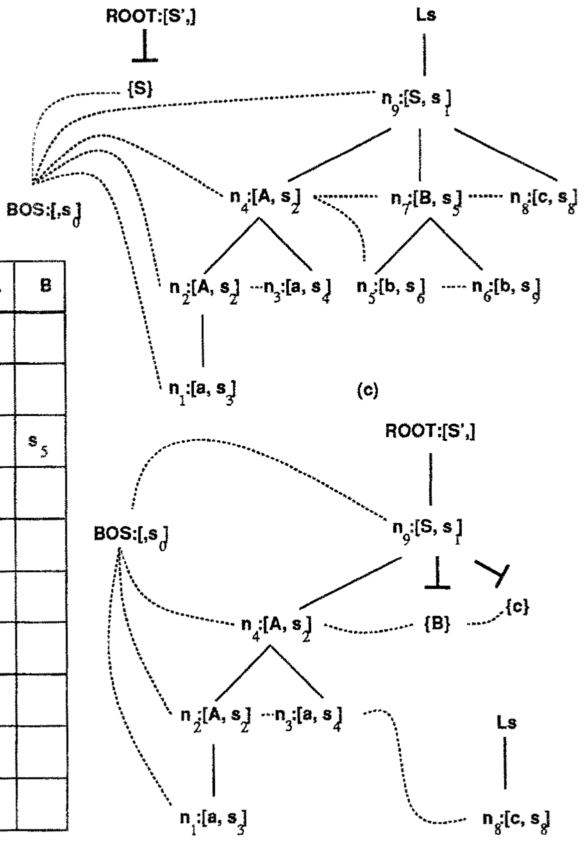
[2]Threading links are used to simplify the description, but they are redundant and could be inferred from the tree structure.

S -> A a c	(1)
S -> A B c	(2)
A -> a	(3)
A -> A a	(4)
B -> b b	(5)

(a)

	$	a	b	c	S	A	B
s_0		sh. 3			s_1	s_2	
s_1	halt						
s_2		sh. 4	sh. 6				s_5
s_3	A -> a						
s_4	A -> A a			sh. 7			
s_5				sh. 8			
s_6			sh. 9				
s_7	S -> A a c						
s_8	S -> A B c						
s_9	B -> b b						

(b)



(d)

Figure 1: Grammar, initial parse, and beginning of incremental parse

Make `n.link` point to the current value of `TOS` and give the new value `n` to `TOS`. In other words, the new node is pushed on the parse stack, which is materialized by the threading.

reduce by `A` $\rightarrow$ α Create a new node `n`, with `n.symbol` = `A` and `n.state` = `goto(n'.state, A)`, where `n'` is $|\ \alpha\ |$ nodes down the stack. Make `n` point to its descendants (whose threading is unchanged). Node `n` becomes the new `TOS` and is linked to `n'`. If `A` is the axiom `S`, it becomes the descendant of `ROOT` (in other words, it fills the gap).

The symbol L_s which appears in figure 1(c) represents the upper portion of the parse stack, as explained in section 3.7.

3.4 Pruning rules

Though the gap mechanism may seem gratuitous at this stage, it plays a crucial role in the general case. In fact, gap forming is the dual notion of context reuse. Maximal context reuse will be achieved if all paths from the root are preserved except for the path connecting the root to the smallest subtree spanning the whole of `y'`, which is cut at the root of this subtree. Let $T_{y'}$ be this subtree, this means that a node can be replaced by a gap if a node of $T_{y'}$ is to be created at its position. This will occur either because *(i)* this node represents a token of the replaced string y, *(ii)* it is to be retained in the parse tree for `w'`, but under a different parent node, or *(iii)* it has more than one gap among its children, so that it is to be located at or below the root of $T_{y'}$.

These 3 conditions give rise to 3 gap-forming rules: *(i)* at initialization time, the terminals of y are pruned, *(ii)* when a node of the tree is involved in a shift or a reduce, it has to be pruned, *(iii)* when all the children of a node are gaps, the children are eliminated and the node pruned (replaced by a gap). Rule *(iii)*, called dangling-node rule, is equivalent to condition *(iii)* on the assumption that, when more than one gap form under a given node, then the algorithm will not stop until all the children of this node are pruned. It is used rather than condition *(iii)* to simplify the description of the algorithm's actions.

This of course, will become clearer when the algorithm is described in detail. In addition, it will be shown in section 3.9 that rule *(ii)* conforms to condition *(ii)*.

3.5 Initial node invalidation

In figure 1(c), a parse tree is given for the string `"aabbc"` and the incremental parse consists in parsing `"aac"`, the string obtained by deleting the substring `"bb"`. The task at hand can thus be represented by the following equations:

```
x = "aa"
y = "bb"
y' = ∅
z = "c"
```

We start the incremental parse by invalidating the nodes representing tokens of the replaced substring y, i.e. nodes n_5 and n_6 in figure 1(c). In figure 1(d), the dangling-node rule (among others) has been applied, so n_7: [B, s_5] is replaced by a gap.

3.6 Selection of an initial parse configuration

A parse configuration consists of a parse stack and the substring `rest(w)`, i.e. the portion of the input lying to the right of and including the lookahead. In the present context, a configuration can be defined relative to a given threading of the parse tree and a given input string, i.e. as a pair `[TOS, lhd]`.
The optimal choice for the initial lookahead is `first`$(y'z)$. In our example, `lhd` = `"c"`. The optimal selection for an initial `TOS` amounts to selecting the root of the largest tree whose frontier immediately precedes `first`$(y'z)$ and whose associated state (the state of its root) has a non-error action on `first`$(y'z)$. In the example, there are two candidate nodes, n_4 and n_3. Node n_4 dominates the largest subtree, but `action(`s_2`, "c")` = `error`. So the initial top of stack is n_3. (We assume the input to be correct.)
It is important to notice that, if we use LALR tables, it is not always enough to check that `action(`n_i`.state, first(y'z))` is not error; it is also necessary to check that `[`n_i`.state, first(y'z)]` leads to a configuration in which `first(y'z)` is shifted, for some reductions may be allowed to take place on an error token. This can be done by "exploratory parsing", i.e. by running the automaton until a `shift` or `error` action is encountered, but without modifying the tree.

3.7 Automaton operation

In the initial configuration, the stack consists of `BOS` and (in general) nodes of the parse tree. In the process of reparsing, we will have to push nodes which do not yet have a position in the parse tree, so the upper items of the stack will often not belong to the parse tree, but to a "stack list", which we will call L_s, as in the example of the initial compilation.

In what follows, it is understood that, whenever a gap forms, we check for dangling nodes.

3.7.1 Shift

A symbol shifted is pushed to the end of L_s. Two cases need to be distinguished.

1. If the lookahead is a token of `y'`, a new node is created to the end of L_s and duly threaded.
2. If `y'` has already been consumed, a *node* from the parse tree has to be replaced by a gap and pushed to the end of L_s. Now, for subtree reuse to be maximal the `shift` operation is extended to the shifting of nonterminals and we will try to shift the highest node we can with the conditions that *(i)* it covers a substring starting at the lookahead and *(ii)* its root is a valid transition from the current state. The second condition can be restated as follows: A node `n` can be shifted in state `s` only if `goto(s, n.symbol)` is defined. (But see section 4.5 on implementation problems.)

The situation of the example belongs to the second case (with `y'` trivially consumed). The only node to fulfill both conditions is node n_8, which is replaced by a gap and pushed on L_s (figure 1(d)).

3.7.2 Reduce by $A \rightarrow \alpha$

With respect to the initial parse (section 3.3), a provision has to be added:

> Let `k` be $\mid \alpha \mid$. If `k` is greater than the length of L_s, then nodes from the parse tree are replaced by gaps and tacked to the beginning of L_s.

In the example at hand (figure 2), nodes n_2 and n_3 are involved in a reduction by $S \rightarrow$ `Aac`, so they are replaced by gaps, which makes n_4, and —indirectly— n_9, dangling nodes, so that we end up with one dangling node, under `ROOT`.

3.8 Graft

When, at the outcome of a shift or reduce move, *(i)* `TOS` is the only item in L_s, *(ii)* there is a single gap in the parse tree, *(iii)* `y'` has been consumed, and *(iv)* `TOS.symbol` matches the symbol of the node which was pruned to create the gap, then `TOS` can be grafted at the gap, parsing stops and attribute values have to propagate from the nodes which were grafted during the parse.

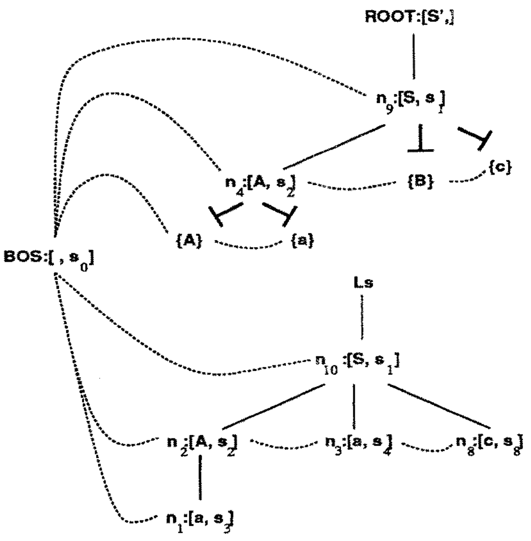


Figure 2: Situation after reducing by `A` $\rightarrow$ α

Condition *(iv)* might seem suboptimal in view of the existence of different productions with the same lhs and with rhs's of the same lengths, like productions (1) and (2) in figure 1(a). For example it might seem too strong in the case where `"a"` is substituted for `"bb"` in `"abbc"`. But, as a grafting position is used as a starting point for attribute *propagation*, semantic rules above this point cannot be allowed to change; therefore, if the symbol of any child of a nonterminal node changes, this nonterminal node must be recreated by a reduction before attribute propagation can take place. So if `w'` = `"aac"` and `w` = `"abbc"` the S-reduction must necessarily be redone.

In figure 2, we notice that all 4 conditions are met to graft the new node n_{10} at the only gap of the tree. In particular, `TOS.symbol` is `S`, which matches the gap's symbol (not represented).

3.9 Property recapitulation

3.9.1 Soundness

A soundness proof would run along the following lines: The IP algorithm which has been presented starts in a valid configuration, has the same configuration transitions as a standard LR algorithm and, furthermore, a correct input is guaranteed to lead to a graft, at worst following an S-reduction.

3.9.2 Reductive minimality

The dramatic structural change which occurred in the example spared few reduction steps. The only nonterminal node which remained untouched (apart from ROOT) is n_2:[A,s_2]. However, the scheme conforms to our definition of reductive minimality, as this section attempts to prove (informally).

An optimality proof was sketched in the case of initialization gaps and dangling-node pruning in section 3.4. In addition, it has been made clear that we used the weakest possible condition on initial-state selection, subtree shifting, and grafting. So what remains to be proved is that the pruning of nodes involved in shifts and reductions obeys the second pruning condition (section 3.4), namely:

> A node has to be pruned if it is to be retained in the parse tree for w', but under a different parent node.

As far as reductions are concerned, when a parse-tree node is involved in a reduction, it is bound to get a new parent, and thus satisfies pruning condition *(ii)*. As far as shifts are concerned, since we shift the highest possible node, a node n will be shifted only if its parent n' cannot, meaning that either n' will be replaced by a new node or n will be replaced by the root of $T_{y'}$.

3.10 Adding generality to the scheme

This section analyzes the special issues of ϵ-productions, multiple changes, and incomplete or erroneous programs. It is not essential for an understanding of the algorithm and can be skipped on first reading.

3.10.1 ϵ-productions

An ϵ-reduction creates a terminal node whose state and threading link are undefined, and whose symbol field indicates its nature. During a subsequent parse, initial invalidation and shifts are affected by the existence of such nodes. Indeed, an ϵ-node should be pruned *(i)* if the following terminal is invalidated, and *(ii)* if the reading head moves to it. The idea is that, from the point of view of input indexing, an ϵ-node behaves as if it was at the same position as the following nonterminal, which triggered its creation.

3.10.2 Multiple changes

To cater for the modification of several substrings of w, the main thing to notice is that a nonterminal should not be shifted if its yield overlaps one or several of the replaced strings y_i. Apart from this, it is simply an iterative application of the process described so far. When the reading head comes to the position of `first`$(y_i z_i)$ for some `i`, the tokens of y_i are invalidated, and we start shifting from y_i'.

3.10.3 Incomplete and erroneous programs

To handle an incomplete program, instead of the single tree at `ROOT`, we can use a list of subtrees whose root symbols form the left substring of a right sentential form. In a complete program, the only item in this list (which we will call L_r, for "root list") is the node corresponding to `S'`. So the `halt` action creates an `S'`-node whose only child is `TOS` (an `S`-node). If an item `T` of L_r is pruned, it leaves no gap, and it is then necessary to check whether L_r is still a sentential prefix. If not, rather than dismantling all subtrees to the right of `T`, these are moved to a list L_e, in which subtrees are sorted according to the order of their frontiers in the input string (as in L_s or L_r). More generally, when the only action available is `error`, the largest tree covering the substring starting at the lookahead and all successive maximal subtrees are pruned and moved to L_e. When, during a subsequent parse, there are no more symbols to shift from L_r, the parsing process can continue with the first terminal of L_e as the lookahead, so that subtrees in L_e can have a chance to be pushed on L_s and eventually be involved in a graft.

4 Implementation using Syntax

4.1 Brief introduction to Syntax

The Syntax compiler compiler is an attempt to integrate comprehensive analyzer-generation methods, including a sophisticated error-recovery scheme and automatic abstract-tree construction. The parse tables generated by Syntax are highly optimized, both for space and time efficiency, so that their use induces alterations in the IP algorithm. What is interesting is that several solutions are sometimes available, with varying degrees of simplicity and approximation to reductive minimality.

4.2 Use of Floyd-Evans productions

Actions and transitions are recorded using a restricted form of Floyd-Evans productions. Ignoring a few niceties, these production can be described as a combination of the following instructions:

scan advance the reading head

goto t_state make `t_state` the new state

push push the new state

pop (p,n) reduce by p, popping n items

goto lhs make `goto(TOS.state, lhs)` the new state

error

halt

As expected, the `error` action is represented by the production `[error]` and the `halt` action by `[halt]`. Less obviously, the action **`shift`** s_i is represented by `[scan,` **`goto`** s_i, **`push`**`]`, while a Floyd-Evans representation for the combination of the action `reduce p`, with p equal to $X \rightarrow \alpha$, and the `goto` entry s_i consists of the production `[pop (p,`$|\alpha|$`), goto X]` followed by the production `[goto` s_i, **`push`**`]`. This last type of production will be called *nt-production*, because it is reached on a nonterminal transition.

In itself, this representation does not cause any major difficulty in the implementation of the IP algorithm; but we are now going to consider problems set by various optimizations.

4.3 Composite actions

The use of Floyd-Evans productions decomposes the `shift` and `reduce` operations into elementary instructions. This fact is used by Syntax to define new operations, which we will call *s-reduce* and *nt-reduce*. They are defined in order to use the fact that, due to non-canonical table construction and reduction-context extension, it is common to find a particular reduction in all entries of an `action` table, as in the tables for s_3, s_7, s_8 and s_9 in figure 1(b). When such a reduction is found, it is combined with the previous action in order to eliminate an `action` table, and avoid pushing a state which is to be immediately popped. For example, the shift found for `action(`s_4`, c)` will be combined with the reduction by `S` $\rightarrow$ `Aac`, so that the shift entry is replaced by the s-reduce [scan, goto S, pop (1,|Aac| − 1)]) and the tables for s_7 disappear. An nt-reduce, similarly, is the combination of 2 reductions, which results in the occurrence of a pop instruction in the nt-production. More exactly, given 2 reductions represented by the 4 productions

1. [pop (p,m), goto X]
2. [goto s_i, push]
3. [pop (q,n), goto Y]
4. [goto s_j, push],

their combination into an nt-reduce is represented by the 2 productions

1. [pop (p,m), goto X]
2. [pop (q,n−1), goto s_j, push].

The incremental parser has to decompose these actions into their shift and reduce components, in order to create all the necessary nodes. However, what cannot be retrieved is the state associated to the node created by the first component, so that some nodes have to be created with an undefined state. On the other hand, we will see in the next section that there is no point in trying to check for a possible graft after the first component of a composite action.

Now, the existence of nodes with undefined states has a potential consequence on initial-state selection, which relies on knowledge of the state associated to a node. But, if we consider that the next operation after the creation of a "stateless" node n is necessarily a reduction, it appears that if n is a valid initial node, then so is its parent, so that we will examine n only if it is *not* a valid TOS. Consequently, the fact that the state is undefined induces no indetermination in the selection of an initial state. However, the next section explains that this fact has an indirect impact on grafting.

4.4 Grafting and lhs availability

The symbol-matching criterion for grafting cannot be applied on account of an optimization called nonterminal equivalence. Indeed, Syntax eliminates single productions when no semantic actions attach to them. This elimination will often create a situation in which several nonterminals are compatible in the sense that A is compatible with B iff, for any state s_i, either *(i)* goto(s_i, A) = goto(s_i, B), *(ii)* goto(s_i, A) is undefined, or *(iii)* `goto(`s_i`, B)` is undefined. When compatibility is an equivalence relation, an equivalence class is formed, thus saving `goto`-table entries (i.e. nt-productions). This has the consequence that, in the reduction descriptors generated by Syntax, the lhs's are not symbols, but equivalence classes, so that it is generally impossible to retrieve the symbol associated to a nonterminal node.

We therefore use a matching condition which does not involve nonterminal symbols. More precisely, when the parse tree has a single gap and `TOS` is the only member of L_s, `TOS` can be grafted at the gap if the gap has a right brother n_i and `action(TOS.state, lhd)` is a shift leading to the state of the lookahead node. In the case where the gap has a right brother, this condition is logically equivalent to the requirement that `TOS.symbol` should match the symbol of the node that was pruned. No such equivalence can be given in the case where the gap is a rightmost child of a node n_j and `action(TOS.state, lhd)` is a reduction involving all siblings of the gap and leading to n_j`.state`, for n_j`.state` cannot be uniquely inferred from `TOS.symbol`. So, this approach is suboptimal, but does not require the presence of a `symbol` field in nonterminal nodes.

Another suboptimality factor is composite actions, which record only states reached after a reduction, i.e. states not uniquely determined by the value taken by `TOS.symbol` before the action, and so have to be wholly redone before a graft can be allowed.

4.5 Construction-time `goto` evaluation

When, whatever the current state, the transition on a given nonterminal `X` leads to a particular state s_i, productions of the form `[pop (p,n), goto X]` become `[pop (p,n), goto @FE]`, where `@FE` is the address of the production `[goto` s_i`, push]`. Similarly when all nonterminal transitions from a given state s_i lead to a particular state s_j, then, rather than pushing s_i on the parse stack, it is more expedient to push the address of `[goto` s_j`, push]`. From the point of view of the IP algorithm, this has 2 important aspects:

1. the value pushed on the parse stack and the new state may differ (when a production address is pushed),
2. the tables do not give the definition domain of the `goto` function, for they specify only those transitions which cannot be statically computed.

The consequence of the first aspect is that, instead of recording one state value in a node, we have to record the value pushed (to handle subsequent reductions in the current

parse) and the real state (to be able to reparse from this node).
The consequence of the second aspect is that nonterminal shifting cannot be decided on the basis of a call to the `goto` function, since the tables do not implement the theoretical `goto` function. Solutions to this problem are studied in the next sections.

4.5.1 The preprocessor solution

It is possible to fool an optimizing parser generator into building the domain of the `goto` function by defining a dummy terminal for each nonterminal and adding a production in which those dummy terminals appear for each production whose rhs contains non-terminals. Then, the action table can be looked up to check the domain of `goto`. This, however, doubles the size of parse tables and has the disadvantage of altering the user's grammar in the listings produced by the generator. (Note, however, that the existence of nt-reduces and s-reduces does not complicate the task, for only `scan` instructions are relevant.)

4.5.2 The predecessor-state solution

This solution is due to [Jalili and Gallier 82]. It consists in recording in each terminal node the state of the preceding stack item (*predecessor state*). Then, when the lookahead is a terminal node from the parse tree and the action is [scan, goto s_i, push], the largest tree incident at the lookahead can be shifted if *(i)* the predecessor state of the lookahead is equal to the current state, and *(ii)* the terminal following this tree is not a new terminal. This is not reductively minimal because *(i)* different states may have a transition on a given nonterminal, and *(ii)* a new terminal might be in the successor set of the reduction which created the subtree root. However, it is extremely interesting from a pragmatic point of view.

5 Related works

Most incremental parsers use a bottom-up strategy, sometimes mixing in top-down operations [Ghezzi and Mandrioli 79] [Jalili and Gallier 82]. A top-down parsing algorithm for LL(1) grammars is described in [Shilling 86].

A popular strategy consists in starting the parsing with `last(x)` as the lookahead and halting the process when some matching condition between parsing configurations is satisfied. In [Celentano 78], the condition is configuration equality, which is equivalent to the definition of optimal context reuse given in 3.4 (at the point of grafting, the parse configuration is identical to a configuration found during the parsing of `w`). In [Yeh and Kastens 88] reparsing stops when the yield of the node corresponding to `TOS` overlaps substring `z`, the state is equal to what it was in the previous parse at the same position in `z` and the node corresponding to the node preceding `TOS` covers a substring of `x`. The data structure used is a *stack list*, which is a threaded structure containing only terminal nodes.

[Ghezzi and Mandrioli 79] use an `LR∧RL` algorithm in which a doubly-threaded tree is created (with both `LR` and `RL` threadings). The `RL` threading is created and maintained in a top-down fashion. Reparsing is limited to `last(x)y'first(z)`, and the new nodes are connected to the parse tree using the threading.

While all the approaches just mentioned aim exclusively at context reuse, [Jalili and Gallier 82] attempt to reuse (i.e. shift) subtrees. When a subtree cannot be reused, it is replaced by a list of smaller subtrees after deletion of some nonterminal nodes. Initially, the whole parse tree is split into a list of subtrees by deleting all nodes on a path leading from the root to `last(x)`. A subtree `T` can be reused iff the lookahead is in `z`, and the current state equals the state of the node preceding the lookahead on the previous parse stack. When considering multiple changes, the additional condition is set that the yield of `T` does not overlap with a y'_i.

[Ghezzi and Mandrioli 80] seek both *(i)* context reuse, by looking for a reduction modifying the stack in such a way that the lhs of this reduction is guaranteed to span the whole of `y'`, and *(ii)* subtree reuse, using a criterion amounting in effect to that of [Jalili and Gallier 82].

An original approach is found in [Degano et al. 88], where the parse matrix is split into submatrices corresponding to sublanguages, whose root symbols are selected nonterminals called *categories*. The algorithm used is called IJSR (for Incremental Jump-Shift-Reduce), for —on certain lookaheads— the action will be a jump to another submatrix. This algorithm requires that an editing action should appear as a category substitution, so that reparsing starts with a jump into the submatrix of the new category and ends with the reduction creating this category. So it aims at context reuse and requires structured editing.

6 Concluding remarks

The generality of Floyd-Evans elementary instructions and the thoroughness of Syntax's optimizations seem a good test for the algorithm's adaptability. Problems likely to occur with most parse-table generators are *(i)* absence of an exhaustive goto table, and *(ii)* construction-time computation of terminal and nonterminal transitions. In Syntax, static computation of terminal transitions takes the form of composite actions. In addition, a further source of difficulty was the use of equivalence classes for nonterminals. The definition of the optimal algorithm has provided a framework in which to analyze these difficulties and solve them methodically. An area of further investigation could be the definition of an optimized parse-table generator allowing reductive minimality.

References

[Aho et al. 86] A. Aho, R. Sethi, J. Ullman, *Compilers : principles, techniques and tools*, Addison-Wesley, 1986.

[Boullier 84] P. Boullier, *Contribution à la contruction automatique d'analyseurs lexicographiques et syntaxiques*, Doctor's Dissertation, 1984.

[Boullier and Jourdan 87] P. Boullier and M. Jourdan, "A new error repair and recovery scheme for lexical and syntactic analysis", *Science of Computer Programming 9 (1987) 271-286*, North-Holland.

[Celentano 78] A. Celentano, "Incremental LR Parsers", *Acta Informatica 10, pp. 307-321*, 1978.

[Degano et al. 88] P. Degano, S. Mannucci, B. Mojana, "Efficient Incremental LR Parsing for Syntax-Directed Editors", *ACM Transactions on Programming Languages and Systems, Vol. 10, No. 3, July 1988, pp. 345-373*, 1988.

[Donzeau-Gouge et al. 75] V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang, J.J. Levy, "A structure-oriented program editor: a first step towards computer-assisted programming", *INRIA report R 114, Le Chesnay, France,* 1975.

[Ford and Sawamiphakdi 84] R. Ford, D. Sawamiphakdi, "A Greedy Concurrent Approach to Incremental code generation", ACM SIGPLAN conference, 1984.

[Fritzson 83] P. Fritzson, "A systematic approach to advanced debugging through incremental compilation", *Proceedings of the Symposium on high-level debugging, Asilomar, California, March 1983.*

[Fritzson 84] P. Fritzson, *Towards a distributed programming environment based on incremental compilation*, Linköping Studies in Science and Technology. Dissertations. No. 109. 1984.

[Ghezzi and Mandrioli 79] C. Ghezzi and D. Mandrioli, "Incremental Parsing", *ACM Transactions on Programming Languages and Systems, July 1979, Vol. 1, pp. 58-70*, 1979.

[Ghezzi and Mandrioli 80] C. Ghezzi and D. Mandrioli, "Augmenting Parsers to Support Incrementality", *Journal of the Association for Computing Machinery, Vol. 27, No 4, July 1980, pp. 564-579*, 1980.

[Jalili and Gallier 82] "Building Friendly Parsers", *Proceedings of the 9th Annual ACM Symposium on Principles Of Programming Languages, Albuquerque, N.M., Jan. 25-27, 1982, pp. 196-206*, 1982.

[Lecluse and Richard 89] C. Lécluse, Ph. Richard, "The O_2 Database Programming Language", *Proceedings of the 15th VLDB Conference, Amsterdam, The Netherlands, August 1989.*

[Medina-Mora and Feiler 81] R. Medina-Mora, P. Feiler, "An Incremental Programming Environment", *IEEE Transactions on Software Engineering, vol. SE-7, No. 5, September 1981.*

[Reps 83] T. Reps, *Generating language-based environments*, (ACM doctoral dissertation award; 1983) Thesis (PhD)–Cornell University, 1983. MIT Press

[Reps et al. 83] T. Reps, T. Teitelbaum, A. Demers, "Incremental Context-Dependent Analysis for Language-Based Editors", *ACM Transactions on Programming Languages and Systems, Vol. 5, No. 3, July 1983, Pages 449-477*, 1983.

[Shilling 86] J. Shilling, "Incremental LL(1) Parsing in Language-Based Editors", *IBM technical report RC 12772, 38 pages*, 1986.

[Teitelbaum and Reps 81] T. Teitelbaum and T. Reps, "The Cornell Program Synthesizer: a syntax-directed programming environment", *Communications of the ACM, vol 24, number 9, September 1981.*

[Teitelman 75] W. Teitelman, *Interlisp Reference Manual*

[Yeh and Kastens 88] D. Yeh and U. Kastens, "Automatic Construction of Incremental LR(1) - Parsers", *SIGPLAN Notices, Vol. 23, No. 3, March 1988.*