In Proceedings of POPL-92

Recognizing Substrings of LR(k) Languages in Linear Time

Joseph Bates and Alon Lavie
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213

Abstract

LR parsing techniques have long been studied as efficient and powerful methods for processing context free languages. A linear time algorithm for recognizing languages representable by LR(k) grammars has long been known. Recognizing substrings of a context-free language is at least as hard as recognizing full strings of the language, as the latter problem easily reduces to the former. In this paper we present a linear time algorithm for recognizing substrings of LR(k) languages, thus showing that the substring recognition problem for these languages is no harder than the full string recognition problem. An interesting data structure, the Forest Structured Stack, allows the algorithm to track all possible parses of a substring without loosing the efficiency of the original LR parser. We present the algorithm, prove its correctness, analyze its complexity, and mention several applications that have been constructed.

1 Introduction

The problem of recognizing substrings of context-free languages has emerged in many practical applications, in the areas of both formal and natural languages. Given a string x, we wish to know whether there exists some string w, such that x is a substring of w, and w is in the language of a given context-free grammar G. The ability to recognize that a given string is not a substring of any sentence in the language allows the early and local detection of syntax errors, without the need to complete a full parse or compilation. Substring recognizers have been considered in several works on recovery from syntax errors. Richter [Ric85] develops a formal method for reporting syntax errors, without attempting to correct them. His method requires a substring recognizer, although no such recognizer is described in his paper. Cormack [Cor89] describes a method for constructing an LR parser that recognizes all substrings of a context-free grammar G. This is done by a more complicated construction of the LR parsing tables, appropriate for dealing with substrings. Cormack's construction provides a deterministic parser (free of table conflicts) for only the *bounded context* class of grammars, which is a class smaller than LR(1). Rekers and Koorn [RK91] propose a substring parsing algorithm for arbitrary context-free grammars based on Tomita's generalized LR parsing algorithm [Tom86]. Although their algorithm has some similarities to the one proposed here, it is not linear, and it's correctness and complexity are not addressed in their paper. Substring recognizers appear to be also useful in the context of parallel and incremental parsing [AD83] [Cel78].

The substring recognition problem can easily be shown to be at least as hard as the full-string recognition problem, as the latter problem is easily reducible to the first in constant time and space. Also, since the set of all substrings of a context-free language is itself a context-free language, the general problem of recognizing substrings is not harder than that of recognizing full-

strings. However, the set of all substrings of an LR(k) language is not necessarily itself an LR(k) language, therefore a linear time bound for recognizing substrings of LR(k) languages is not trivial.

In this paper we show that the substring recognition problem for LR(k) grammars is not any harder than the full-string recognition problem. We present an algorithm for the LR(k) substring recognition problem that runs in *linear* time, which is similar to that of the original LR parsing algorithm [AU72]. While previous substring parsing algorithms such as Cormack's [Cor89] modified the LR parsing tables to accommodate for substring recognition, our algorithm modifies the parsing algorithm itself, while leaving the original LR parsing tables intact. We introduce a data structure, the *Forest Structured Stack* (FSS), that keeps track of all possible parses of the substring, while preserving the efficiency of the original LR parsing algorithm. The SLR, canonical LR(1) and LALR parser variants differ only in the algorithms that produce the parsing tables from the grammar, and share a common LR parsing algorithm that is controlled by these tables. Since our substring algorithm replaces this run-time parsing algorithm while using the parsing tables "as is", it is equally applicable to all of the above LR variants. The parsing algorithm for canonical LR(k) grammars ($k \geq 2$) differs slightly from the other variants, in order to account for the extended lookahead into the input. Thus, a slightly different version of our substring algorithm handles canonical LR(k) grammars [1]

Section 2 describes the FSS data structure and summarizes the substring recognition algorithm for LR(1) grammars. In section 3 we sketch the correctness of the algorithm. Section 4 analyzes the time complexity of our algorithm. An amortized analysis is used to prove that the algorithm does indeed run in linear time. Finally, some applications of the algorithm and our conclusions are presented in section 5. Throughout this paper we touch only on the key points of our work. In particular, only sketches of proofs are presented. An extended paper including the complete proof details will be published elsewhere, and is available from the authors in the form of a technical report [BL91].

[1] Due to space limitations, the substring algorithm for canonical LR(k) grammars is not described in this paper. For a full description of the algorithm see [BL91].

2 The Algorithm

The substring recognition algorithm we describe in this section is denoted by *SSR*. It is a variation of the conventional LR parsing algorithm, denoted by *LRP*.

2.1 The Forest Structured Stack

The Forest Structured Stack (FSS) is a graph, consisting of a set of trees, representing a possibly infinite set of stacks of *LRP*. The nodes of the graph are labeled by states of the LR machine. The edges that connect the state nodes are labeled by grammar symbols. Each path from a root to a leaf corresponds to the top portion of an *LRP* stack, in which the node at the root of the path represents the state at the top of the stack.

The algorithm simulates the behavior of *LRP* on all the stacks represented in the FSS, adding nodes in correspondence with actions that push items on the stack (shifts), and removing nodes in correspondence with stack reductions. The tree representation avoids the duplication of stacks which have an identical top part but which differ in content deeper down.

2.2 An Informal Description of the Algorithm

The idea behind *SSR* is to effectively simulate the behavior of *LRP* on all possible strings of which the input is a suffix. When parsing a string w, of which our input string $x = x_1 x_2 \cdots x_n$ is a suffix, *LRP* is in some state (at the top of the stack) upon shifting x_1, the first symbol of x. We are interested in all such states and thus we initialize *SSR* by building a FSS with a distinct single node tree for each state that can be the result of shifting x_1 according to the pre-compiled action table. Since each single node tree represents all stacks with that state at the top, the initial FSS represents the set of all possible stacks after the shifting of x_1.

From here on we continue the parsing of x according to each of the FSS trees. *SSR* performs a series of alternating *Reduce* and *Shift* phases, one pair of phases for each input symbol.

During a *Reduce Phase*, reductions are performed on all trees whose top state indicates that a reduction is to be performed. In LR parsing, reductions remove nodes from the stack. When performed on a tree, they are done on all paths in the tree, starting at the root, to a depth corresponding to the number of symbols on the right-hand side of the rule being reduced.

Reductions are a problem only when they wish to remove nodes deeper than the length of some path in the FSS. This corresponds to a reduction that includes symbols derived from parsing the part of the full string that is prior to x. In our algorithm, we refer to such reductions as *long reductions*, and treat them in a manner somewhat similar to our initialization.

A reduction normally removes the right-hand side of the rule being reduced, and then shifts the non-terminal symbol A of the left-hand side of the rule. The new state at the top of the stack is determined from the *goto table*, and depends on A and on the state revealed at the top of the stack by the reduction. With long reductions, since only a partial stack exists, this state is not known. Our algorithm determines all such possible states by a lookup in the *long reduction goto table*. This supplemental table specifies for each possible reduction from a state at the top of the stack, the set of states that may be reached as a result of the shifting of the left-hand side non-terminal of the rule being reduced. The table is easily constructed from the parsing tables prior to run-time. Each of the determined goto states corresponds to at least one full string, the parsing of which would have resulted in that state being at the stack top at this point in the parsing process. It is sufficient at this point to add these states to the FSS as single node trees. Long reductions are performed at most once per state in a Reduce Phase, since a second long reduction from the same top state would produce the same new trees, and thus would be redundant [2].

When the action defined by the table on the root node of a tree is *error*, the entire tree is discarded. These are trees that correspond to prefix strings of x that cannot be completed to strings in the language. A Reduce Phase terminates when the action indicated by the table, on each of the tree root nodes, is to shift the next input symbol. All the shift operations are done in the consequent *Shift Phase* of the algorithm.

Upon reaching the end of the input x, if the FSS is not empty, we can safely assume that there exists a prefix string y such that the parsing of the string yx by the LR parser would not have caused a parsing error by this point. Properties of LRP guarantee the existence of a suffix z, such that $w = yxz$ is accepted. Thus x is confirmed to be a valid substring.

To increase the efficiency of the algorithm, two operations, $SUBSUME$ and $CONTRACT$, are performed on the FSS structure at appropriate times. When a single node tree is added to the FSS, and the state of the node is identical to that of some other tree root node in the FSS, the larger tree may be deleted from the FSS, since the single node tree represents all stacks of LRP that have that particular state at the top of the stack. This set of stacks necessarily includes all stacks that were represented by the larger tree rooted at a node of the same state. The $SUBSUME$ operation detects such conditions and deletes the larger tree. Long reductions frequently create single node trees that subsume other trees in the FSS.

The $CONTRACT$ operation merges two trees, the roots of which are of the same state, returning a single tree as a result. The merging is done recursively down the two trees, to ensure that no immediate sibling nodes in the FSS are labeled by the same state. This in turn guarantees that at all times, the branching degree of every node in the FSS is bounded by the number of states in the parsing table, a property essential for maintaining a linear bound on the running time of the algorithm. Two trees may end up having the same top state as a result of either a shift operation or a reduction. In the shift case, since prior to the shift the trees necessarily had different top states, they may be simply merged at the top node level, and no deeper tree contraction is needed. However, in the case of a reduction, if the result of the reduction is a top state which is the same of that of another existing tree in the FSS, a full $CONTRACT$ operation is performed.

[2] We believe that we can manage without the long reduction goto table, and simply consider all states that are a result of shifting A. This leads to a somewhat simpler implementation, but the proof of correctness appears to be more difficult in this case.

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

Figure 1: A simple grammar for arithmetic expressions

The *RECLAIM* operation is responsible for freeing the dynamically allocated storage for those nodes and trees that are discarded in the course of the algorithm.

2.3 An Example

To further clarify how the algorithm works, we present a simple example. Figure 1 contains a simple arithmetic expression grammar, taken from [ASU86], page 218. Table 1 contains the SLR parsing table for this grammar, as it appears in Figure 4.31 of [ASU86] (page 219). Table 2 shows the long reduction goto table for this parsing table. For each state, the long reduction goto table contains the list of states into which the parser may shift after a reduction from that state[3] . Figure 2 shows the contents of the FSS along the various stages of the execution of the algorithm on the input ``* id)'' .

Let us follow a trace of this execution. The initialization stage of the algorithm results in entering a single node of state 7 into the FSS, since this is the only state that is the result of shifting the first input symbol ``*'' . Thus, after the initialization, the FSS contains the the single node tree shown in Figure 2a. State 7 wishes to shift the next input symbol ``id'', thus the first Reduce Phase is empty, and the shifting of ``id'' occurs in the Shift Phase, resulting in the tree in Figure 2b. The next Reduce Phase includes several reductions. State 5 on input ``)'' indicates a reduction by rule 6. This is a normal reduction, and results in the tree in Figure 2c. State 10 on input ``)'' indicates a reduction by rule 3. This is a long reduction. According to the long reduction goto table, this long reduction results in the single node tree of state 2, as depicted in Figure 2d. State 2 on input ``)'' indicates a reduction by rule 2. This again is a long reduction, and according to the long reduction goto table, it results in two single node trees, of states 1 and 8 respectively, as can be seen in Figure 2e. The Reduce Phase terminates at this point, since neither state 1 nor state 8 indicate a reduction on input ``)''. The following Shift Phase discards the node of state 1, since the parsing table indicates an error for state 1 on input ``)''. State 8 indicates a shift of ``)'' into state 11, resulting in the tree shown in Figure 2f. This completes the Shift Phase. The consequent termination test discovers that we have reached the end of the input. Since the FSS is not empty, the input is a valid substring (of an arithmetic expression in the language of our grammar), and the algorithm terminates. Note that due to the simplicity of the chosen example, no CONTRACT or SUBSUME operations occurred in the execution outlined above.

Top state	Goto states after reduction
0	
1	
2	1 8
3	2 9
4	
5	3 10
6	
7	
8	
9	1
10	2
11	3

Table 2: Long reduction goto table for the parsing table in Table 1

[3] Note that in the general LR(k) case, the action table may indicate reductions by several different rules from a particular state for different lookaheads. Thus, strictly speaking, the long reduction goto table specifies a partial function from top states and lookaheads to sets of states. However, in our simple example, the grammar is LR(0) and a reduction by at most a single rule is possible from each state. We have therefore simplified the table for this example by omitting the lookaheads.

	Action						Goto		
State	id	+	*	(	)	$	E	T	F
0	sh5			sh4			1	2	3
1		sh6				acc			
2		r2	sh7		r2	r2			
3		r4	r4		r4	r4			
4	sh5			sh4			8	2	3
5		r6	r6		r6	r6			
6	sh5			sh4				9	3
7	sh5			sh4					10
8		sh6			sh11				
9		r1	sh7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Table 1: SLR parsing table for grammar in Figure 1

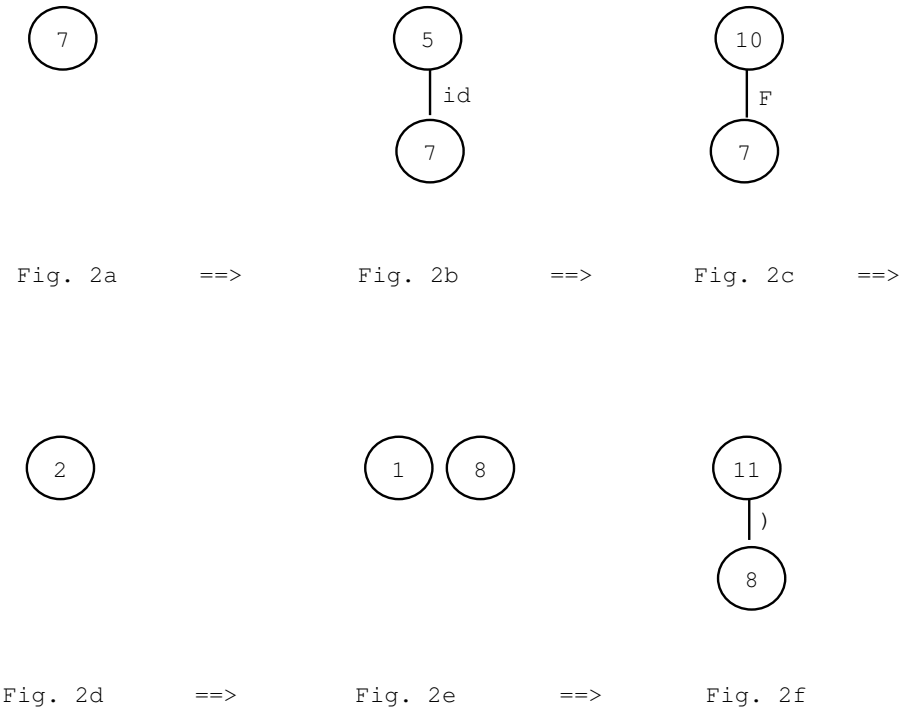

Figure 2: Structure of the FSS throughout the execution of algorithm SSR on the example

3 Correctness

Now we sketch the correctness of *SSR*. The reader is referred to Aho and Ullman [AU72] for a comprehensive proof of correctness of the original LR parsing algorithm *LRP*. In our proof, we rely on the correctness of *LRP*, namely that given an LR grammar G, and an input string x, *LRP accepts* x if and only if $x \in L(G)$. We therefore concentrate on proving the following theorem :

Theorem: Let G be an LR(1) grammar and x be an input string. *SSR accepts* x if and only if there exist strings y, z such that $w = y \cdot x \cdot z$ is *accepted* by *LRP*.

We show that *SSR* simulates the parsing of x by *LRP* for all possible prefix strings y. If upon shifting x_n, the last input symbol of x, *SSR* has not rejected x, there exists at least one such prefix string y, for which *LRP* has not rejected the input $y \cdot x$ after the shifting of x_n. The existence of a suffix string z, for which $w = y \cdot x \cdot z$ is *accepted* by *LRP* is assured by the fact that LR parsers reject inputs as early as possible [AU72]. We now provide a sketch of how the above outline may be formalized.

A *stack configuration* c is a triple (s, x, i), where $s = [st_1, st_2, \cdots, st_k]$ is a stack of states (with st_k at the top), x is the input string of length n, and $0 \leq i \leq n$ is a position within the input string. The set of stack configurations represented at any point of *SSR* includes a configuration for each path from a root node to a leaf in the FSS. The *LRP* stack configurations are those particular configurations that correspond to stacks manipulated by *LRP*. A stack configuration $c = (s', w, j)$ is an *LRP* stack configuration if after some number of steps of *LRP* on input w, s' represents the *LRP* stack and j is the parser's position within the input string. In particular, the stack representation s' of an *LRP* stack configuration c always has the LR machine's start state at the bottom of the stack.

To formally prove that *SSR* simulates the parsing of the input string x by *LRP* for all possible prefix strings y, we define a meaning function M, mapping general stack configurations to their corresponding *LRP* stack configurations. A stack configuration $c = (s, x, i)$ of the FSS is mapped by M to the (possibly infinite) set of all *LRP* stack configurations with s as the top portion of the stack. Formally, let S^* denote the set of all state stacks, and *LRC* denote the set of all *LRP* stack configurations. Then :

$$M((s, x, i)) =$$
$$= \{(r \cdot s, y \cdot x, |y| + i) \in LRC \mid r \in S^* \ \& \ y \in \Sigma^*\}$$

where $r \cdot s$ denotes the concatenation of the state stacks. We extend the domain of M to the sets of configurations in the natural way, namely $M(\{c_i\}) = \bigcup_i M(c_i)$.

In the following analysis we assume that all states of the *LRP* parsing table are reachable from the start state. If in fact this property does not hold, we may easily (in constant time and space) modify the table to include only such reachable states, and use our modified table instead of the original one.

Lemma 1: Let $c_1 = (s, x, i)$ and $c_2 = (r \cdot s, x, i)$ be two stack configurations. Then $M(c_2) \subseteq M(c_1)$.
Proof : Straightforward from the definitions of *LRP* stack configurations and the configuration mapping function M.

Both the *SUBSUME* and *CONTRACT* operations of *SSR* remove paths from the FSS when there exist other paths in the FSS that are suffixes of the paths being removed. Lemma 1 implies that the removal of such paths from the FSS does not alter the set of *LRP* configurations denoted.

To formalize the effect of the parsing operations of algorithm *SSR* on the FSS, we define the function *next*, from stack configurations to sets of stack configurations. For a given configuration $c = ([st_1, st_2, \cdots, st_l], x, j)$, $next(c)$ is the set of configurations c' that are the result of a *single* *SSR* parsing step from c. Thus, as in the *SSR* algorithm, $next(c)$ is defined according to the action $ACT(st_l, x_{j+1})$ indicated in the LR action table. In the case of a shift or a normal reduction, $next(c)$ is a set containing the single resulting new configuration. In case of a long reduction, $next(c)$ is the set of all stack configurations consisting of single state stacks, the states of which one can reach after shifting the left-hand side nonterminal of the rule being reduced, as determined

by the long reduction goto table. If the action is *accept* [4] or the end of string is reached, we define $next(c) = \{c\}$, and if it is *reject* (a parse error), then $next(c) = \phi$.

To formalize the effect of the Reduce and Shift phases, we define the extension of *next* to sets of stack configurations in the following way. Let $C = C_1 \cup C_2$ be a set of stack configurations such that C_1 contains exactly the stack configurations of C whose top state indicates that the next action is a reduction, and C_2 is the rest of C. If $C_1 \neq \phi$ then $next(C) = \{c' \in next(c) \mid c \in C_1\} \cup C_2$. If $C_1 = \phi$ then $next(C) = \{c' \in next(c) \mid c \in C_2\}$. Thus, reductions have precedence over other actions.

Based on this extended definition of *next* we define for every $n \geq 0$ the function $next^n$, which is the result of n successive applications of *next*. Note that a Reduce phase corresponds to some finite number of applications of *next* and that a Shift phase corresponds to a single application of *next*.

Lemma 2: The Simulation Lemma :
Let C be a set of stack configurations. Then : $M(next(C)) = next(M(C))$
Proof : We prove this by case analysis on the parsing actions that occur on each $c \in C$. The cases of *Shift, Accept, Reject* and *normal* reductions are straightforward, as *next* is identical to the equivalent action of *LRP*. *Long* reductions are more subtle, and in this case the result follows from the definitions of M and *next*.

Lemma 3: The Generalized Simulation Lemma : We generalize Lemma 2 to any finite number of applications of *next*. Let C be a set of stack configurations. For every $n \geq 0$: $M(next^n(C)) = next^n(M(C))$
Proof : By a straightforward induction on n using Lemma 2.

Lemma 4: When parsing an input string $x = x_1 x_2 \cdots x_n$, let C_1 be the set of initial stack configurations of the FSS, and let C_i denote the set of stack configurations represented by the FSS after the *ith* Shift Phase. The following two properties are maintained for each of the C_i $(1 \leq i \leq n)$:

1. *Soundness* : if $c \in C_i$ then $M(c) \neq \phi$

2. *Completeness* : for all *LRP* stack configurations $c' = (r \cdot s, yx, |y| + i)$, such that the last operation of the parser is a shift of x_i, there exists a $c = (s, x, i) \in C_i$ such that $c' \in M(c)$.

Proof : By induction on i. C_1 has both properties due to the way it is constructed. The induction step is proven by the following arguments. Since the *next* function is a formal modeling of the Reduce and Shift phases of the algorithm (excluding the process of possibly discarding some configurations by SUBSUME and CONTRACT operations), it follows that for some n, $C_i \subseteq next^n(C_{i-1})$ (with the "missing" configurations being those discarded by the SUBSUME and CONTRACT operations) and since SUBSUME and CONTRACT have no effect on the set of configurations represented by M, $M(C_i) = M(next^n(C_{i-1}))$. The *next* function has the property that if $M(c) \neq \phi$ and $next(c) \neq \phi$, then $M(next(c)) \neq \phi$, which extends to $next^n$ and thus guarantees soundness. By Lemma 3 $M(C_i) = M(next^n(C_{i-1})) = next^n(M(C_{i-1}))$, which guarantees completeness.

Corollary: If C_n is the set of stack configurations represented by the FSS after the *nth* Shift Phase, where $n = |x|$, then $C_n \neq \phi$ if and only if there exists an *LRP* configuration $c' = (s', yx, |y|+|x|)$. Note that the existence of such an *LRP* configuration c' implies the existence of a string $w' = yx$, such that w' is not rejected by *LRP* by the time x_n was shifted. The soundness property of Lemma 4 guarantees that if $C_n \neq \phi$, such an *LRP* stack configuration c' exists. The completeness property guarantees that if such a configuration c' exists, $C_n \neq \phi$. Since *LRP* has the property that an input is rejected at the first possible opportunity on a left to right scan of the input string [AU72], this implies that there exists a string z such that $w = w' \cdot z = y \cdot x \cdot z$ is *accepted* by *LRP*, completing the correctness of *SSR*.

[4] Notice that in practice the action will never be *accept*, since the algorithm will have terminated upon reaching the end of the input string. However, we include this case for the sake of completeness.

4 Complexity Analysis

We will now prove that *SSR* runs in linear time for grammars free of epsilon rules. In [BL91] we demonstrate that *SSR* maintains a linear running time even in the presence of such rules.

After the initialization of the FSS, the algorithm enters a loop that consists of a termination test for end of input, examining the next input symbol, a Reduce Phase and a Shift Phase. This loop can be executed up to $n-1$ times, until the end of string is reached. The initialization of the FSS that precedes the loop requires only constant time. It involves scanning a column of the LR action table, and the creation of a constant number of root nodes. The termination check also takes constant time. Since there are only a constant number of root nodes (see Lemma 5 below), each Shift Phase involves only a constant number of shift operations and thus takes constant time. However the time cost of each Reduce Phase is not uniform, and varies from one run through the loop to the next. Each Reduce Phase involves some number of Tree Reductions, which are reductions on all paths of an FSS tree to a constant depth. We will show that each such Tree Reduction is completed in constant time and then use an amortized cost evaluation to obtain a linear bound on the total number of Tree Reductions. Finally, we will argue that the total time cost of all *SUBSUME, CONTRACT* and *RECLAIM* operations also is at most linear in the length of the input.

In the following analysis, S denotes the set of states of the parser, and $|S|$ is the size of this set. We distinguish between root nodes of the FSS and internal nodes.

Lemma 5 : At any time there is at most a single root node of any given state.
Proof : The claim holds after the initialization of the algorithm, and throughout Reduce and Shift phases *SSR* explicitly checks for root nodes of identical state, and when detected, merges the appropriate trees, using *SUBSUME* and *CONTRACT* as necessary.

Lemma 6 : The total number of nodes that become internal in the course of execution of the algorithm on a string x of length n is $O(n)$.

Proof : In the case that the grammar is free of epsilon rules, root nodes become internal only as a result of shift operations. Once a node becomes internal, it never again becomes a root node. Thus, the Lemma is a direct result of the fact that the number of root nodes at the start of any Shift Phase is bounded by $|S|$, and there are at most n Shift Phases. Thus the total number of shift operations is $O(n)$.

Lemma 7 : No node in the FSS ever has more than $|S|$ children.
Proof : *CONTRACT* operations are performed whenever necessary so as to maintain this property.

We now concentrate on analyzing the time complexity of Reduce phases. A normal reduction on a single path of nodes in the FSS is identical to an *LRP* reduction, and takes constant time. Long reductions are very similar to normal reductions. However, they involve accessing the long reduction goto table in order to determine the possible states that may result from the shifting of the left-hand side non-terminal of the rule being reduced. This table access is done in constant time. New root nodes are created for the resulting states of this process, and each new node added may require a *SUBSUME* operation, if there already exists a root node of the same state. This condition can be detected in constant time by a linear scan of the set of root nodes, and need be done only a constant number of times per long reduction, since at most $|S|$ new root nodes may be added. We account for the time spent on the *SUBSUME* operations separately. Therefore, excluding the time spent on all *SUBSUME* operations, a long reduction on a single path requires only constant time. Thus, any reduction, normal or long, on a single path requires only constant time.

A Reduce Phase reduction in *SSR* operates on a FSS stack tree, and performs the reduction on all paths in the tree that originate at the root node to a depth equivalent to the number of symbols on the right-hand side of the rule being reduced. Since this is a constant depth, and the fan-out degree of FSS tree nodes is also bounded by a constant, each such Tree Reduction involves only a constant number of reductions (one for each path), each taking constant time. Thus in order

to complete the time analysis of Reduce phases, we need only demonstrate that $O(n)$ Tree Reductions are performed in the course of the algorithm.

For the purpose of the analysis, we separate the rules of our grammar into two groups. Grammar rules with a single symbol on the right-hand side are grouped together as *non-generative* rules and their corresponding reductions are referred to as *non-generative* reductions. All other rules will be called *generative rules* and their corresponding reductions *generative reductions*. We will show that the cost of performing a generative reduction can be charged to internal nodes of the FSS that are discarded by the reduction, and that only a constant number of consecutive non-generative reductions may occur between the generative ones. Thus, the non-generative reductions may be charged to the generative ones, and they in turn can be charged to the nodes.

Lemma 8 : In the course of the execution of algorithm SSR, only a constant number of consecutive *non-generative* Tree Reductions may be performed.
Proof : Since long reductions are performed at most once per state in a Reduce Phase, we need only consider the normal reductions. Non-generative reductions do not remove internal nodes from the FSS. By a counting argument it can be seen that after a constant number of such reductions on FSS trees, such a reduction is repeated. If this were to occur the non-generative rules that correspond to this series of reductions would form a cycle, in contrast with the fact that any LR grammar must be non-cyclic.

Lemma 9 : In the course of an execution of algorithm SSR, there are only $O(n)$ *generative* Tree Reductions.
Proof : We recall that at most one long reduction can occur for each state per Reduce Phase. Therefore, at most $O(n)$ such reductions may occur in all Reduce Phases combined. Any other generative reductions are performed on trees with internal nodes. Such a Tree Reduction will remove all internal nodes to a depth corresponding to the number of symbols on the right-hand side of the rule. We therefore account for these reductions by charging a unit of cost to each internal node removed by the Tree Reduction. Since the node is removed from the FSS by the Tree Reduction, it may only be charged once. Also, for each generative Tree Reduction performed, at least one internal node is charged. Thus the total number of internal nodes charged is an upper bound on the total number of generative Tree Reductions. By Lemma 6 there are only $O(n)$ nodes that become internal in the course of the execution of SSR. Thus, only $O(n)$ internal nodes may be charged for Tree Reductions and we obtain an $O(n)$ bound on the total number of generative Tree reductions.

Lemma 10 : The Total number of Tree Reductions is $O(n)$.
Proof : We look at the non generative reductions as groups of consecutive reductions that occur before, between and after the generative reductions. Due to the $O(n)$ bound on the number of generative reductions, we obtain a similar bound on the number of such groups of non generative reductions. We therefore get an $O(n)$ bound on the total number of non generative Tree Reductions. This in turn provides us with a bound of $O(n)$ on the total number of all Tree reductions.

We complete the time complexity analysis of our algorithm by showing that all $CONTRACT$, $RECLAIM$ and $SUBSUME$ operations together require only $O(n)$ time.

First we consider the $CONTRACT$ operations. The $CONTRACT$ operation merges two FSS trees that have root nodes of the same state. The contraction itself is done by comparing the states of the children of the first root node with those of the second root node. Lemma 7 guarantees at most $|S|^2$ comparisons. If a child of the first root node has a state identical to that of a child of the second root node, the two subtrees are contracted by a recursive call to $CONTRACT$. All other children (and their appropriate subtrees) are added as children of the first root node, and the second root node is deleted. Thus, the top level $CONTRACT$ operation requires constant time. Note that any recursive call to $CONTRACT$ will necessarily result in the elimination of an internal node. We may thus charge a unit of cost to the node deleted as a result of each recursive call to $CONTRACT$, and since the node is deleted from the FSS by the this operation, it may be

charged only once. Since *CONTRACT* is invoked only after reductions, there are at most $O(n)$ top level calls to *CONTRACT*. Lemma 6 guarantees that at most $O(n)$ internal nodes will be charged, therefore implying at most $O(n)$ recursive calls to *CONTRACT*. This provides us with an $O(n)$ bound on the total number of *CONTRACT* calls and a similar bound on the total time complexity of all *CONTRACT* operations.

Next, we consider the *RECLAIM* operations. These operations delete entire subtrees from the FSS, when these become obsolete. We assume the cost of such an operation is directly proportional to the number of nodes in the subtree, i.e. constant time per node being deleted. The deletion of internal nodes can be charged a unit of cost to the node being deleted, and only $O(n)$ root nodes are reclaimed in all Shift Phases and Reduce Phases combined, since at most one root node is reclaimed per operation.

Finally, we observe that we have already accounted for the *SUBSUME* operations. *SUBSUME* searches for a root node of a state identical to that of a new single node tree created by a long reduction. This requires constant time. If found, the tree is the reclaimed by the *RECLAIM* operation, the time for which we have already accounted for.

This completes the time complexity analysis of our algorithm, under the assumption that the grammar contains no epsilon rules. Our analysis has shown that the total time cost of all operations in an execution of the algorithm on an input string of length n is $O(n)$.

5 Conclusions

We have presented and proved a linear time algorithm for recognizing substrings of LR(k) languages.

The original version of this algorithm was initially developed by the first author in 1980. It did not include the *CONTRACT* operation for merging trees of the FSS. Tree contractions are crucial to retaining a linear bound on the running time of the algorithm. In the process of trying to prove the linear time bound we discovered this deficiency, and the proper modifications were consequently made.

The original algorithm, while in fact not always linear, was used as the basis for a syntax checking modification to the IBM VM/370 editor XEDIT. That modification enabled the IBM editor to check COBOL source code for syntax errors, when users modified lines, screens or files. For instance, when the cursor was moved off a modified line, the editor would beep and display an unobtrusive error message if the line was not a substring of any COBOL program. Though COBOL has a large grammar, this modification had no apparent effect on the speed of XEDIT on machines of the early 1980's. The algorithm was also used to check Pascal programs on an IBM PC editor, and this too had no apparent effect on the speed of the editor. Thus, the original algorithm appeared to be adequately fast in practice.

We have implemented our revised algorithm and have tried it on several test grammars. No precise measurements have been performed to compare the actual running time of our substring algorithm with that of the original LR parser. However, in practice, the revised implementation continues to run as fast as before.

Acknowledgements

Alan Demers provided helpful discussion during development of the original algorithm and proof sketches in 1979. Merrick Furst was helpful in the development of some of the revised ideas described here. We also thank Steve Guattery for his comments.

References

[AD83] R. Agrawal and K.D. Detro. An efficient incremental LR parser for grammars with epsilon productions. *Acta Informatica*, 19:369–376, 1983.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[AU72] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation and Compiling, Vol. I: Parsing*. Prentice-Hall, Englewood Cliffs, N.J., 1972.

[BL91] J. Bates and A. Lavie. Recognizing substrings of LR(k) languages in linear time. *Technical Report CMU-CS-91-188*, 1991.

[Cel78] A. Celentano. Incremental LR parsers. *Acta Informatica*, 10:307–321, 1978.

[Cor89] G. V. Cormack. An LR substring parser for noncorrecting syntax error recovery. In *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation*, pages 161–169, 1989.

[Ric85] H. Richter. Noncorrecting syntax error recovery. *ACM Transactions on Programming Languages and Systems*, 7(3):478–489, July 1985.

[RK91] J. Rekers and W. Koorn. Substring parsing for arbitrary context-free grammars. In *Proceedings of Second International Workshop on Parsing Technologies*, pages 218–224, Cancun, Mexico, 1991.

[Tom86] M. Tomita. *Efficient Parsing for Natural Language*. Kluwer Academic Publishers, Hingham, Ma., 1986.