A parsing algorithm for non-deterministic context-sensitive languages

Henk Harkema
Computer Science Department, Twente University,
P.O. Box 217, 7500 AE Enschede, The Netherlands
e-mail: harkema@cs.utwente.nl

Masaru Tomita
Center for Machine Translation, Carnegie Mellon
University, Pittsburgh, PA 15213-3890, USA
e-mail: mt@nl.cs.cmu.edu

ABSTRACT

In this paper a shift-reduce algorithm for the recognition of sentences of non-deterministic languages defined by context-sensitive grammars is introduced. The algorithm is a composite of Tomita’s efficient parsing method for non-deterministic languages and a procedure for processing deterministic context-sensitive languages, proposed by Walters. The algorithm that is presented in this paper employs two (graph-structured) stacks, one of which is used for representing the context of the several active processes (an extension of the traditional input buffer). The other stack is the normal parsing stack. The two stacks are linked by means of a set of pointers. When manipulating context-sensitive languages in an LR-manner, there is no need for a separate goto table anymore. In case of a reduce action, all the symbols of the left-hand side of the rule under consideration are pushed back into the input buffer. The state which appears on top of the stack when having popped off the symbols of the right-hand side, and the first symbol in the input buffer determine the next action.

1 INTRODUCTION

1.1 Context-sensitive grammars and languages

Before we jump into the basic operations of the algorithm, we have to define what exactly a context-sensitive grammar and ditto language is. According to Walters [12], a context-sensitive grammar (CS grammar) is a quadruple \( \{ V_T, V_N, S, P \} \), where \( V_T \) is a finite set of terminal symbols, \( V_N \) is a finite set of non-terminal symbols, \( S \in V_N \) is the distinguished or start symbol, and \( P \) a finite set of rules. The intersection of the sets \( V_T \) and \( V_N \) is the empty set, the union of the sets \( V_T \) and \( V_N \) is called the vocabulary \( V \). The \( p \)-th rule is denoted \( Y_{p1}, Y_{p2}, \ldots, Y_{pmp} \rightarrow X_{p1}, X_{p2}, \ldots, X_{pm} \), where \( 1 \leq m_p \leq n_p \), \( Y_{p1}, X_{p1} \in V \) and \( Y_{p1}, \ldots, Y_{pmp} \notin V_T^* \) (the set of all nonnull strings over \( V_T \)). If \( m_p \geq 2 \), rule \( p \) is a context-sensitive rule; if \( m_p = 1 \), rule \( p \) is a context-free rule. \( Y_{p1}, \ldots, Y_{pmp} \) is the subject or left-hand side of rule \( p \) and \( X_{p1}, \ldots, X_{pm} \) is its right-hand side.

The set of all sentences generable by a CS grammar \( G \) forms the context-sensitive language defined by \( G \). A context-free grammar is a CS grammar all of whose rules are context-free or are of the form \( A \rightarrow \lambda \) (the null string) for \( A \in V_N \). If a context-free grammar contains no rules which have the null string as right-hand side, it is clearly also a CS grammar.

1.2 Tomita’s parsing algorithm

The Tomita parsing algorithm adapts Knuth’s well-known parsing algorithm for LR(k) grammars to non-LR grammars, including ambiguous grammars ([8], [9], [10], [11]). If a grammar is ambiguous then at some point in the analysis of an ambiguous sentence two different parsing actions must be possible that lead to the two distinct analyses of the sentence. The parsing table
of such a grammar will therefore contain multiple entries. Knuth's algorithm is not capable of dealing with this kind of non-determinism. Tomita's algorithm is an extension of Knuth's method, that can handle parsing tables with multiple entries.

When a parsing process in Tomita's algorithm encounters a multiple action, the stack is split, thus creating a new process for each entry. Because the first part of the stack of the new process is exactly the same as the original stack, it is not necessary to copy the whole stack. When a stack is split, the stack is therefore represented as a tree, the bottom of the stack corresponding to the root of the tree.

Whenever two or more processes have the same state number on top of their stacks, they will behave in exactly the same manner until the tops will be popped by a reduce action. It is obvious that these processes should be combined by unifying their top nodes.

With these stack splitting and combination techniques, the stack becomes an directed acyclic graph called a graph-structured stack. Each path through the graph from the joint node representing the start state, to whichever end node delineates an ordinary LR parse stack.

1.3 Walters' parsing algorithm

Walters also takes Knuth's algorithm as a base and adds to it to handle context-sensitive grammars [12]. The CS(k) processor is basically the same as an LR(k) processor, a one-way input tape, a stack to keep track of the parsing process, an LR-table and a control mechanism that uses the table to guide the process. The LR-table normally consists of an action part and a goto part. The goto part is used to conclude what state to push on top of the stack after a reduce action. The action part contains the actions (shift, reduce or accept) to be performed on the stack. In a reduce action a number of states is popped off the stack. The number of states that is popped off is equal to the length of the right-hand side of the rule used in the reduction. After a reduce action, the entry of the goto part of the table corresponding to the new topmost state on the stack and the (non-terminal) symbol of the left-hand side of the rule gives the new state.

The way a CS(k) processor handles a reduce action differs slightly, but decisively, from the LR(k) approach. In addition to popping a number of states from the stack, the left-hand symbols of the rule that is used in the reduction are pushed on the input tape. This makes the goto part of the action table superfluous, because now the next action can simply be determined by looking up the appropriate entry in the action part, according to the new state on top of the stack and the first symbol on the input tape (which is the first symbol of the left-hand side of the used rule). Since this rule can be context-sensitive, this symbol is possibly a terminal symbol.

Using Walters' method we can picture the parsing process as a constant exchange of symbols between the stack and the input buffer. Hence, the input buffer acts as a second stack.

2 The basic operations

The configuration of the recognizer comprises two stacks and a set of so-called pointer elements. One of the stacks is Tomita's graph-structured stack, the other is Walters' extended input buffer, which likewise is a graph-structured stack. Stack \( \Gamma_r \), formerly Tomita's stack, is the one the symbols of the input buffer are pushed onto. In all the following figures \( \Gamma_r \) is depicted on the left. The input buffer, which is now a stack in its own right, is called \( \Gamma_r \). When a reduce action is performed, the symbols of the left-hand side of the rule used in the reduction are added to this stack. In the figures \( \Gamma_r \) is on the right. In order to link each process represented in \( \Gamma_r \) with its own set of contexts in \( \Gamma_r \), a set of pointer elements \( \Pi \) is created. Each element of \( \Pi \), called pointer element or context marker, points to one process on the left and to one or more contexts on the right. The pointer elements are depicted between the two stacks, as asterisks within circles.

Besides \( \Pi \), there are three more sets playing an important role in the algorithm: \( A \), \( R \) and \( S \). Set \( A \) is the set of active triples. An active triple is a triple for which the control mechanism has to decide what the next action is that should be performed on the process the triple is representing. If the action is shift, a triple containing information about the action is added to \( S \). The shift procedure uses these triples in \( S \) to actually execute the shift actions. In case of a reduce action a triple is added to \( R \). This set is used by the reduce procedure.

Using these sets enables the algorithm to develop the two stacks and pointer set by merely adding vertices, edges and elements to them.
There are no removals of vertices or edges, nor deletions of pointer elements. As a result we do not really pop away items from the stack; we leave the items and mark them inactive by removing triples from \( A \) as in Tomita's algorithm [11].

### 2.1 Initial configuration

At the very beginning the input stack \( \Gamma_r \) has a vertex for each symbol of the input string. Furthermore a vertex labeled \( \# \) is created in \( \Gamma_r \). This vertex demarcates the end of the input string. Stack \( \Gamma_r \) simply consists of one single vertex, an initial state vertex labeled 0. The two stacks are linked by a pointer element of set \( \Pi \). Figure 1 gives the initial framework for the recognizer if the input string is \( e c f \).

![Figure 1: initial situation of the stacks and the pointer set](image)

The first recognizer process is started up by adding triple \((v, \sigma, l)\) to the set of active triples \( A \). The control mechanism takes over and determines what action is next.

### 2.2 Shift

A shift action is characterized by a triple \((\sigma, s, l)\), where \( \sigma \) is a pointer element, \( s \) is the state to go to after the shift action and \( l \) is the vertex in \( \Gamma_r \) to be shifted. Figure 2 and 3 which portray parts of the stacks and the pointer set before and after a shift action, will illustrate the operation.

![Figure 2: situation of the stacks before the shift operation](image)

Pointer element \( \sigma \) points to the process whose stack begins with the state and symbol vertices labeled 3 and D respectively and to two contexts starting with the symbols A, b, D and c, C, F. At this moment we are not interested in the remainders of the stacks nor in the rest of the pointer elements. We focus on the implementation of the shift action.

We are looking at the active triple \((v, \sigma, l)\). The vertices \( v \) and \( l \) and the pointer element \( \sigma \) give the control mechanism all the information needed to look up the next action in the action table. Suppose the entry in the action table for state 3 and symbol c, which is the first symbol of the current context, is 'shift 5'. The triple \((\sigma, 5, l)\) is handed over to the shift procedure. This procedure, not surprisingly called the shifter, will create two new vertices in \( \Gamma_s \), one labeled with the symbol to be shifted and one labeled with the new state. Furthermore a new pointer element is added to \( \Pi \). This element connects the new state vertex with all the successors of vertex \( l \). In this case there is only one successor, the vertex labeled C. Note that the original contexts are left unchanged; nothing is removed, only added. Figure 3 illustrates the operation of the shifter.

![Figure 3: situation of the stacks after the shift operation](image)

Triple \((q, \tau, z)\) is added to \( A \), the set of active triples. If \( l \) has more than one successor, more than one new active triple will be added to \( A \).

Shifting is one of the two ways of exchanging symbols between the two stacks. At a shift action only one symbol at a time is moved unchanged from \( \Gamma_r \) to \( \Gamma_s \).

### 2.3 Reduce

A reduce action is the other occasion on which symbols are moved from one stack to another. The direction now is from \( \Gamma_s \) to \( \Gamma_r \). The number of symbols that is exchanged depends on the production rule that is used. The number of vertices stripped from \( \Gamma_s \) equals two times the length of the right-hand side of the rule (the vertices labeled with the symbols and the matching state vertices). The symbols of the left-hand side of the rule are pushed onto \( \Gamma_r \).

Consider the configuration of the stacks as depicted in Figure 4. The triple \((v, \sigma, l)\) is processed by the control mechanism which takes a look at the action table and decides that, say, a reduce action using rule 4 should be executed. Hence
the control mechanism renders a triple \((\sigma, 4, 1)\) to the reducer. Assume for the moment that rule 4 is \(A \ b \rightarrow C \ D\). The reducer will look for a path of twice the length of the right-hand side of rule 4, starting in \(v\) and whose symbol vertices match the symbols of the right-hand side of rule 4. If there is already a pointer element pointing to the last vertex of this path, the reducer simply adds a new context to that pointer element by creating vertices in \(V\), for the symbols \(C\) and \(D\) of the rule’s left-hand side and connecting them with the pointer element. If there is no such pointer, the procedure will create one and then attaches the symbols to its right. In both cases, the vertex labeled with \(Y_{pm}\), the last symbol of the left-hand side, will be connected to \(l\). Performing ‘reduce 4’ in the previous figure, results in the situation presented in Figure 5.

In this example vertex \(w\) has no pointer element pointing to it, so a pointer element \(\tau\) is created. After the reduction, triple \((w, \tau, q)\) is added to the set of active triples. Because in general there can be only one vertex \(w\), only one new triple is added to \(A\) in case of a reduction.

3 A NECESSARY IMPROVEMENT

The implementation of the basic operations shift and reduce as sketched in the last sections is straightforward. When tested with some grammars the reduce action turned out not to be very efficient.

3.1 The problem

The main drawback of the first version of the algorithm is that it doesn’t provide any rules for merging nodes in the stack which represent the context of a process. As a consequence, a stack configuration as pictured in Figure 6 can occur. The process depicted in this figure has two contexts, both starting with the symbol \(A\). Because the algorithm uses one symbol of the context for lookahead, the same action will be performed twice on this process, one time for each context. This is, of course, useless.

Figure 6: a situation that causes inefficiency

Suppose the action table tells us that ‘shift 5’ is the action to be executed when a process is in the state 3 and symbol \(A\) is the first symbol of the input stack. Then Figure 7 shows us the next step in the trace of the parser. In this figure the first symbol of the second context is shifted onto the shift stack. Because of the first context, one action ‘shift 5’ still remains to be executed on the process with context marker 1. We cannot simply discard this action, saying we already did a ‘shift 5’ action. If we do so and bluntly proceed with processing context marker 2, we loose the rest of the first context (containing the symbols \(b, D\) et cetera).

Figure 7: execution of the first shift action

Figure 8: situation of the stacks after the second shift operation

What if the second ‘shift 5’ is executed straightforward? Take a look at Figure 8. There are no
more active triples left containing context marker 1, in other words: there are no more actions for this process. In the state vertex labeled 3 stack $\Gamma_s$ splits in two identical parts, ending in two different contexts. The obvious thing to do, is to merge the vertices of these identical parts, as shown in Figure 9.

![Figure 9: merging vertices in $\Gamma_s$](image)

The process ending in context marker 2 now has two lookahead symbols and therefore two actions attached to it (unless these actions are the same).

Is this merge-when-shift approach a solution to the problem? Not really. The same action, in this case 'shift 5', is still performed twice. The problem of having two identical vertices in $\Gamma_r$ is more or less shifted onto the shift stack and solved there, when it is too late. And what happens if the double action is a reduce action instead of a shift action?

There shouldn't be two or more contexts of a process starting with the same symbols in the first place. So the solution lies not in the merge-when-shift approach, but in a merge-when-reduce approach.

### 3.2 Merge-when-reduce approach

Let's take a second look at the problem. In a merge-when-reduce approach the situation appearing in Figure 10 should be impossible.

![Figure 10: situation of the stacks that shouldn't occur](image)

How can we prevent this from happening? We have to take back a step in the history of this graph and look at Figure 11.

Assume rule 4 is A b $\rightarrow$ X Y. When executing 'reduce 4' on the process of context marker 2, is the right time for the merge-when-reduce approach to come into action and to prevent the stack from slipping in the prohibited configuration of Figure 10. It is clear what should be done: just merge the identical vertices labeled A. The result of this strategy is shown in Figure 12.

![Figure 11: situation of the stacks before the reduce operation](image)

![Figure 12: merging vertices in $\Gamma_r$](image)

In fact, this way of acting seems to be a very simple solution for the problem of performing the same actions twice. Alas, there are some snakes in the grass.

These difficulties are related to the fact that a context marker can be part of an active or inactive triple. Consider the situation in Figure 13.

![Figure 13: challenging the merge-when-reduce approach](image)

In this case, the action 'shift 5' of pointer element 1 is executed before the reduce action has taken place. There are no more actions left for pointer element 1, so there are no more active triples containing that element. The next thing to do is the reduce action. Execution with the simple merge-when-reduce approach yields a new configuration, displayed in Figure 14.

There we go again! The reduce action adds a triple holding context marker 1 to A. Since the first symbol of its context is still A, the action for this context marker is 'shift 5' afresh. Two
problems arise: we already did a ‘shift 5’ action on context marker 1, and by activating a triple with context marker 1 again, we not only add a new context starting with the symbols A, b and C, but also reactivate the old context with symbols A, C, f and cetera. Note that in Figure 14 the vertex labeled C can be reached from context marker 1 as well as from context marker 3, both of which are part of an active triple. In fact, the processes of which an edge points to vertex C are the same. They look different because they are in different phases (symbol A not yet shifted, symbol A already shifted).

Back to the figure. Since the vertex A is already shifted, it seems we are too late to apply our merge-when-reduce approach. However, if the reducer knows that symbol A is already shifted at the moment of executing the reduce action, it can catch up with the active process. This happens in Figure 15. A triple $(v, \sigma, l)$ is added to the set of active triples, where $v$ is the state vertex in $\Gamma_r$, labeled 3, $\sigma$ pointer element 3 and $l$ the vertex in $\Gamma_r$, labeled b.

Figure 15: catching up with an active process

To find out whether a symbol is shifted or not, a pointer from the corresponding vertex in the reduce stack to the context marker of its shifted counterpart in the shift stack is needed. This pointer is not a part of any stack, nor the pointer set and therefore not shown in the figure. In our example the pointer points from vertex A in $\Gamma_r$ to context marker 3.

Context marker 3 is part of a still active triple, so we can simply add a new action according to the lookahead symbol b. This method can easily be generalized for situations in which more than one vertex can be merged and more than one matching symbol of the context is shifted.

With this modification, the general behavior of the reducer can be outlined as follows. First the reducer determines the new state of the process, which is given by the label of the new top of stack $w$, after popping off the symbol and state vertices. If there is no context attached to this vertex (i.e. there is no pointer element pointing to $w$), the reducer creates a pointer element and vertices in $\Gamma_r$ labeled with the symbols of the left-hand side of the rule used in the reduction and links them. If there is a context marker, say $\tau$, the reducer starts looking for the longest path in the context of $w$ whose symbols match the symbols of the left-hand side. When a matching symbol in $\Gamma_r$ is already shifted, we can continue the matching process in the context belonging to the context marker of the shifted vertex. An illustration of this can be found in Figure 16 and 17, which will be explained shortly.

In case there is no matching path in $\Gamma_r$, the reducer creates new vertices for the symbols and makes a right pointer of context marker $\tau$ pointing to the first of the new vertices (which is labeled $Y_{nl}$). On the other hand, if a path does exist but it doesn't cover all the symbols of the left-hand side, the reduce procedure creates vertices for the symbols that are left over. If the last vertex of the path is not shifted, an edge will be created between the last vertex of the path and the first of these new vertices, otherwise the context marker belonging to the last vertex of the path will be pointing to the first of the new vertices. In the former case there will be no triple added to A.

The mainspring in Figure 16 is the stack ending in the vertices B and 2, and its context consisting of the symbols A, R and d. The symbols A, R and d are all shifted onto $\Gamma_r$. (The vertices and edges resulting from shifting R and d are not shown in the figure.) The shifted symbols R and d are then reduced to C F. This adds the context C P to context marker 2. In the next action symbol C is shifted on the shift stack (context marker 3). Finally the symbols X, Y and Z are reduced to A C g. That leads to the situation of Figure 17.

The reducer starts looking for a matching path in the context to the right of context marker 1. Only the first symbol, A, matches. The vertex labeled A is shifted, so the reducer continues to
look for the rest of the symbols in the context of context marker 2. Here it finds another matching symbol. Because the vertex labeled with that symbol is also shifted the search continues in the context of context marker 3, without success. The reducer now has found the longest matching path. Since the last vertex of this path, labeled C, is shifted, a vertex labeled g is attached to context marker 3. The edge leaving vertex g ends in the vertex labeled E in \( \Gamma_r \).

4 A BRIEF REMARK ABOUT THE ACTION TABLE

The parsing table for the algorithm can be obtained by using the existing LR parsing table construction methods ([2], [3], [4], [7]). One important modification for our recognizer is that each entry in the table should be a set of actions, rather than a single action [11]. However, since we are not only dealing with non-deterministic languages, but also with their context-sensitiveness, there turns out to be another difficulty to overcome for the table constructor.

Consider a grammar with, among others, the following 4 rules:

1. \( S \rightarrow A \ B \)
2. \( B \rightarrow D \)
3. \( A \ D \rightarrow a \ d \)
4. \( A \ C^n \rightarrow a \ c^m \ (m \geq n) \)

The left-hand side of rule 4 contains \( n \) C's. The first state of the action table contains the items \( S \rightarrow \cdot A \ B \) and \( A \ D \rightarrow \cdot a \ d \), because of the rule 2 (\( S \Rightarrow A \ B \Rightarrow A \ D \Rightarrow a \ d \)). Now, add the left-recursive rule

5. \( B \rightarrow B \ D \)

to the grammar. The first state still contains the two dotted rules mentioned above. However, with the new rule in the grammar we also have to take a look at the infinite derivation \( S \Rightarrow A \ B \Rightarrow A \ B \ D \Rightarrow A \ B \ D \ D \Rightarrow \ldots \) and its possible derivations to \( A \ C^n \). Stopping after \( j \) D's (\( j < n - 1 \)) could mean that the constructor misses the derivation of \( B \ D^j \) to \( C^n \), and thus incorrectly omits item \( A \ C^n \rightarrow \cdot a \ c^m \) from the first state. The answer to the question is given by the constraint \( m_p \leq n_p \) on the lengths of the left-hand and right-hand sides of a context-sensitive rule. This constraint implies that the reducer can stop looking at the infinite derivation as soon as the length of \( B \ D \ldots D \) is
greater than the length of \( C^n \), since a string of 
\( n \) symbols can never be derived from a string of 
length \( n + 1 \).

In the context-free case this problem does not 
occur, because then you simply look at the sym-
bol just after the dot, since the left-hand side of 
every rule consists of one non-terminal symbol.

5 CONCLUSIONS AND FUTURE RESEARCH

The combination of Tomita's algorithm for non-
LR grammars and Walters' algorithm for context-
sensitive grammars results in a shift-reduce algo-
rithm for parsing sentences of non-deterministic 
languages defined by context-sensitive grammars.
At this stage only a version that recognizes sen-
tences is implemented (in Modula-2 and Lisp). 
This version is to be extended to a full parser.
Walters proposed a notation to represent a parse 
for context-sensitive grammars [12].

An important feature of the algorithm, its 
complexity, is the subject of ongoing research.
The literature on complexity is often limited to 
context-free grammars and algorithms ([5], [6]).
The problem whether an input string is in the lan-
guage generated by an acyclic context-sensitive 
grahm is probably polynomial for fixed gram-
mars [1].

Another point of interest is a formal proof of 
correctness of the algorithm. This proof should 
be based on the proofs of correctness for Tomita's 
and Walters' algorithms.

6 FORMAL SPECIFICATION OF
THE ALGORITHM

6.1 Pre-defined functions and 
global variables

This section presents the pre-defined functions 
and global variables essential to understanding 
the algorithm.

\text{ACTION}(s, a) \quad \text{looks up the action table and}
\text{return one or more actions} 
\text{s is a state number and a} 
\text{is a terminal or non-terminal}
\text{symbol.}

\text{SYMBOL}(l) \quad \text{takes a vertex in } \Gamma_s \text{ or } \Gamma_r 
as its argument and returns a 
symbol labeled with vertex \( l \).
\text{STATE}(v) \quad \text{takes a vertex in } \Gamma_s \text{ as its argu-
ment and returns a state} 
\text{number labeled with vertex} 
v.
\text{LEFTP}(\sigma) \quad \text{takes a pointer element as}
\text{its argument and returns the} 
\text{vertex in } \Gamma_s \text{ the pointer}
\text{element is pointing to.}
\text{SUC}(w) \quad \text{takes a vertex } w \text{ as its argu-
ment and returns the set}
\text{of all vertices for which there}
\text{exists an edge from } w \text{ to each}
of those vertices.

\( a_1, \ldots, a_n \) \quad \text{input string of length } n.
\text{G} \quad \text{grammar.}
\Gamma_s \quad \text{graph-structured stack in which the}
\text{shift actions take place.}
\Gamma_r \quad \text{graph-structured stack in which the}
\text{reduce actions take place.}
\Pi \quad \text{set of context markers or pointer ele-
ments, linking the two stacks } \Gamma_s \text{ and}
\Gamma_r.
\text{r} \quad \text{boolean variable indicating if the input}
\text{string is recognized, if } r \text{ is TRUE}
\text{the input string is accepted, else it is}
\text{rejected.}
\text{A} \quad \text{set of active triples to be processed,}
each triple is of the form \((v, \sigma, l)\), where \( v \) \text{ is a vertex in} 
\Gamma_s \text{, } \sigma \text{ a}
\text{pointer element in } \Pi \text{ and } l \text{ a ver-
tex in } \Gamma_r \text{; } A \text{ is initialized in Parse,}
\text{Actor removes elements from } A \text{,}
\text{Recognizer and Shifter both}
\text{add triples to the set.}
\text{S} \quad \text{set of triples } (\sigma, s, l) \text{ to direct the}
\text{Shifter; } \sigma \text{ is a pointer element,} 
s \text{is the state to go to after the shift}
\text{action and } l \text{ is the vertex in } \Gamma_r \text{ whose}
\text{symbol is to be shifted.}
\text{R} \quad \text{set of triples of the form } (\sigma, r, l) \text{ to}
\text{direct the Reducer; } \sigma \text{ is a pointer}
\text{element, } r \text{ is the number of the rule to}
\text{be used in the reduction and } l \text{ is the}
\text{vertex in } \Gamma_r \text{ the new context should}
\text{be attached to.}
\text{Y}_{pi} \quad i^{th} \text{symbol of the left-hand side of the}
\text{p}^{th} \text{production.}
\text{X}_{pi} \quad i^{th} \text{symbol of the right-hand side of the}
\text{p}^{th} \text{production.}
$|p_r|$ length of the right-hand side of production $p$.

$|p_l|$ length of the left-hand side of production $p$.

In the following sections the formal description of the algorithm is given, based on Tomita’s description [11].

6.2 Parse

The procedure Parse forms the top level of the algorithm. In Parse the sets $R$, $S$ and $A$ are initialized. One vertex is created in $\Gamma_s$, labeled 0 and representing the start state. The input string is the initial context. Vertices labeled with these terminal symbols appear in in $\Gamma_r$, along with edges connecting them. The initial state and context are connected by the first pointer element in $\Pi$. A call to the procedure Recognize starts the actual recognition process. In the end the variable $r$ contains the result.

PARSE $(G, a_1, \ldots, a_n)$

- $\Gamma_s, \Gamma_r, \Pi \leftarrow \emptyset$
- $R, S \leftarrow \emptyset$
- $a_{n+1} \leftarrow ‘\#’$
- $r \leftarrow FALSE$
- create in $\Gamma_s$ one vertex $v$, labeled 0
- create $n+1$ vertices in $\Gamma_r$, $w_0, \ldots, w_n$, labeled $a_1, \ldots, a_{n+1}$ respectively
- create $n$ edges in $\Gamma_r$ from $w_i$ to $w_{i+1}$, $0 \leq i < n$
- create one vertex $\sigma$ in $\Pi$, labeled $*$
- create two pointers from $\sigma$ in $\Pi$ to $v$ in $\Gamma_s$, and from $\sigma$ in $\Pi$ to $w_0$ in $\Gamma_r$
- $\mathcal{A} \leftarrow (v, \sigma, w_0)$
- RECOGNIZER
- RETURN $r$

6.3 Recognizer

The procedure Recognize manipulates the sets the sets $R$, $S$ and $A$ and thus decides in which order the actions will be executed. An important implementation detail is that before a reduce action can take place, all shift actions must have been performed. This synchronization is necessary because the reducer handles reduce actions depending on whether a vertex is shifted or not, as explained in an earlier section. When a vertex is scheduled to be shifted (i.e., element of $S$) but not yet shifted the reducer will assume it will be not shifted, causing it to make wrong moves.

RECOGNIZER

- REPEAT
  - WHILE $A \neq \emptyset$ DO
    - DO ACTOR
  - WHILE $S \neq \emptyset$ DO
    - DO SHIFTER
  - WHILE $R \neq \emptyset$ DO
    - DO REDUCER
      UNTIL $R, A, S = \emptyset$

6.4 Actor

Each time the actor is called it removes one element from $A$ and decides what action should be done according to the action table. Because of multiple entries the ACTION function may return more than one action.

ACTOR

- remove one element $(v, \sigma, l)$ from $A$
- FOR ALL $\alpha \in \text{ACTION}(\text{STATE}(v), \text{SYMBOL}(l))$ DO
  - IF $\alpha = ‘\text{accept}’$ THEN
    - $r \leftarrow TRUE$
  - IF $\alpha = ‘\text{shift s}’$ THEN
    - add $(\sigma, s, l)$ to $S$
  - IF $\alpha = ‘\text{reduce p}’$ THEN
    - add $(\sigma, p, l)$ to $R$
REDUCER

- remove one element \((\sigma, p, l)\) from \(\mathcal{R}\)
- \(v \leftarrow \text{LEFTP}(\sigma) \in \Gamma_s\)
- FOR vertex \(w \in \Gamma_s\) such that there exists a path in \(\Gamma_s\) of length \(2|p_r|\) from \(v\) to \(w\) DO
  - IF there is an element \(\tau\) in \(\Pi\), such that there exists a pointer from \(\tau\) to \(w\) THEN
    - let \(z_1, \ldots, z_j\) be the longest matching path in \(\Gamma_r\), starting from \(\tau\), the symbols of which match the left-hand side of production \(p\)
    - IF \(j = 0\) THEN
      - create vertices \(q_\ell, \ldots, q_{p_m}\) in \(\Gamma_r\), labeled \(Y_{p_1}, \ldots, Y_{p_m}\), respectively
      - create edges from \(q_i\) to \(q_{i+1}\) in \(\Gamma_r\), \(1 \leq i < p_{p_r} - 1\)
      - create an edge from \(q_{p_{p_r}}\) to \(l\) in \(\Gamma_r\)
      - add \((w, \tau, q_1)\) to \(A\)
    - ELSE
      - IF \(j <> |p_r|\) THEN
        - create vertices \(q_{j+1}, \ldots, q_{p_m}\) in \(\Gamma_r\), for the non-matched symbols of the left-hand side, labeled \(Y_{j+1}, \ldots, Y_{p_m}\)
        - create edges from \(q_i\) to \(q_{i+1}\) in \(\Gamma_r\), \(j \leq i < p_{p_r} - 1\)
        - create an edge from \(q_{p_{p_r}}\) to \(l\) in \(\Gamma_r\)
        - IF \(z_j\) is shifted THEN
          - create a pointer from the context marker \(\lambda\) in \(\Pi\), created when shifting vertex \(z_j\), to vertex \(q_1\)
          - add \((s, \lambda, q_{i+1})\) to \(A\), where \(s\) is the vertex in \(\Gamma_s\) \(\lambda\) is pointing to
        - ELSE
          - create an edge from vertex \(z_j\) to vertex \(q_{i+1}\) in \(\Gamma_r\)
      - ELSE
        - IF \(z_j\) is shifted THEN
          - create a pointer from the context marker \(\lambda\) in \(\Pi\), created when shifting vertex \(z_j\), to vertex \(l\)
          - add \((s, \lambda, l)\) to \(A\), where \(s\) is the vertex in \(\Gamma_s\) \(\lambda\) is pointing to
        - ELSE
          - if not already created, create an edge from vertex \(z_j\) to vertex \(l\) in \(\Gamma_r\)
  - ELSE
    - create a pointer element \(\tau\) in \(\Pi\)
    - create a pointer from \(\tau\) to \(w\) in \(\Gamma_s\)
    - create vertices \(q_1, \ldots, q_{p_m}\) in \(\Gamma_r\), labeled \(Y_{p_1}, \ldots, Y_{p_m}\), respectively
    - create edges from \(q_i\) to \(q_{i+1}\) in \(\Gamma_r\), \(1 \leq i < p_{p_r} - 1\)
    - create an edge from \(q_{p_{p_r}}\) to \(l\) in \(\Gamma_r\)
    - add \((w, \tau, q_1)\) to \(A\)
6.5 Reducer

The reducer takes care of the reduce actions. It tries to handle each reduction as efficient as possible by using vertices already existing in $\Gamma_r$. The way the reducer achieves this goal is by making use of so called matching paths. The reducer is shown on a separate page.

6.6 Shifter

The procedure shifter, finally, shifts a vertex from stack $\Gamma_r$ to $\Gamma_r$. Because vertices are only shifted one at a time and merging is virtually impossible, the procedure is really straightforward.

**SHIFTER**

- remove one element $(\sigma, s, l)$ from S
- $v \leftarrow$ LEFTP($\sigma$) $\in \Gamma_s$
- create two vertices $q$ and $w$ in $\Gamma_r$, labeled $s$ and SYMBOL($l$) respectively, and a pointer element $\tau$ in II labeled $+$
- create two edges in $\Gamma_r$ from $w$ to $v$ and from $q$ to $w$
- create a pointer from $\tau$ to $q$
- add a pointer from $l$ to $\tau$
- FOR ALL $z$ in SUC($l$) $\in \Gamma_r$, DO
  - create a pointer from $\tau$ to $z$ in $\Gamma_r$
  - add $(q, \tau, z)$ to A

References


