Polynomial Time and Space Shift-Reduce Parsing of Arbitrary Context-free Grammars.*

Yves Schabes
Dept. of Computer & Information Science
University of Pennsylvania
Philadelphia, PA 19104-6389, USA
e-mail: schabes@linc.cis.upenn.edu

Abstract

We introduce an algorithm for designing a predictive left to right shift-reduce non-deterministic push-down machine corresponding to an arbitrary unrestricted context-free grammar and an algorithm for efficiently driving this machine in pseudo-parallel. The performance of the resulting parser is formally proven to be superior to Earley's parser (1970).

The technique employed consists in constructing before run-time a parsing table that encodes a non-deterministic machine in the which the predictive behavior has been compiled out. At run time, the machine is driven in pseudo-parallel with the help of a chart.

The recognizer behaves in the worst case in $O(|G|^2 n^3)$-time and $O(|G|n^2)$-space. However in practice it is always superior to Earley's parser since the prediction steps have been compiled before run-time.

Finally, we explain how other more efficient variants of the basic parser can be obtained by determinizing portions of the basic non-deterministic push-down machine while still using the same pseudo-parallel driver.

1 Introduction

Predictive bottom-up parsers (Earley, 1968; Earley, 1970; Graham et al., 1980) are often used for natural language processing because of their superior average performance compared to purely bottom-up parsers such as CKY-style parsers (Kasami, 1965; Younger, 1967). Their practical superiority is mainly obtained because of the top-down filtering accomplished by the predictive component of the parser. Compiling out as much as possible this predictive component before run-time will result in a more efficient parser so long as the worst case behavior is not deteriorated.

Approaches in this direction have been investigated (Earley, 1968; Lang, 1974; Tomita, 1985; Tomita, 1987), however none of them is satisfying, either because the worst case complexity is deteriorated (worse than Earley's parser) or because the technique is not general. Furthermore, none of these approaches have been formally proven to have a behavior superior to well known parsers such as Earley's parser.

Earley himself ([1968] pages 69-89) proposed to precompile the state sets generated by his algorithm to make it as efficient as LR(k) parsers (Knuth, 1965) when used on LR(k) grammars by precomputing all possible states sets that the parser could create. However, some context-free grammars, including most likely most natural language grammars, cannot be compiled using his technique and the problem of knowing if a grammar can be compiled with this technique is undecidable (Earley [1968], page 99).

Lang (1974) proposed a technique for evaluating in pseudo-parallel non-deterministic push down automata. Although this technique achieves a worst case complexity of $O(n^3)$-time with respect to the length of input, it requires that at most two symbols are popped from the stack in a single move. When the technique is used for shift-reduce parsing, this constraint requires that the context-free grammar is in Chomsky normal form (CNF). As far as the grammar size is concerned, an exponential worst case behavior is reached when used with the characteristic LR(0)

*We are extremely indebted to Fernando Pereira and Stuart Shieber for providing valuable technical comments during discussions about earlier versions of this algorithm. We are also grateful to Aravind Joshi for his support of this research. We also thank Robert Frank. All remaining errors are the author's responsibility alone. This research was partially funded by ARO grant DAAL03-89-C0031PRI and DARPA grant N00014-90-J-1863.

machine.[1]

Tomita (1985; 1987) proposed to extend LR(0) parsers to non-deterministic context-free grammars by explicitly using a graph structured stack which represents the pseudo-parallel evaluation of the moves of a non-deterministic LR(0) push-down automaton. Tomita's encoding of the non-deterministic push-down automaton suffers from an exponential time and space worst case complexity with respect to the input length and also with respect to the grammar size (Johnson [1989] and also page 72 in Tomita [1985]). Although Tomita reports experimental data that seem to show that the parser behaves in practice better than Earley's parser (which is proven to take in the worst case $O(|G|^2n^3)$-time), the duplication of the same experiments shows no conclusive outcome. Modifications to Tomita's algorithm have been proposed in order to alleviate the exponential complexity with respect to the input length (Kipps, 1989) but, according to Kipps, the modified algorithm does not lead to a practical parser. Furthermore, the algorithm is doomed to behave in the worst case in exponential time with respect to the grammar size for some ambiguous grammars and inputs (Johnson, 1989).[2] So far, there is no formal proof showing that the Tomita's parser can be superior for some grammars and inputs to Earley's parser, and its worst case complexity seems to contradict the experimental data.

As explained, the previous attempts to compile the predictive component are not general and achieve a worst case complexity (with respect to the grammar size and the input length) worse than standard parsers.

The methodology we follow in order to compile the predictive component of Earley's parser is to define a predictive bottom-up pushdown machine equivalent to the given grammar which we drive in pseudo-parallel. Following Johnson's (1989) argument, any parsing algorithm based on the LR(0) characteristic machine is doomed to behave in exponential time with respect to the grammar size for some ambiguous grammars and inputs. This is a result of the fact that the number of states of an LR(0) characteristic machine can be exponential and that there are some grammars and inputs for which an exponential number of states must be reached (See Johnson [1989] for examples of such grammars and inputs). One must therefore design a different pushdown machine which can be driven efficiently in pseudo-parallel.

We construct a non-deterministic predictive push-down machine given an arbitrary context-free grammar whose number of states is proportional to the size of the grammar. Then at run time, we efficiently drive this machine in pseudo-parallel. Even if all the states of the machine are reached for some grammars and inputs, a polynomial complexity will still be obtained since the number of states is bounded by the grammar size. We therefore introduce a shift-reduce driver for this machine in which all of the predictive component has been compiled in the finite state control of the machine. The technique makes no requirement on the form of the context-free grammar and it behaves in the worst case as well as Earley's parser (Earley, 1970). The push-down machine is built before run-time and it is encoded as parsing tables in the which the predictive behavior has been compiled out.

In the worst case, the recognizer behaves in the same $O(|G|^2n^3)$-time and $O(|G|n^2)$-space as Earley's parser. However in practice it is always superior to Earley's parser since the prediction steps have been eliminated before run-time. We show that the items produced in the chart correspond to equivalence classes on the items produced for the same input by Earley's parser. This mapping formally shows its practical superior behavior.[3]

Finally, we explain how other more efficient variants of the basic parser can be obtained by determinizing portions of the basic non-deterministic push-down machine while still using the same pseudo-parallel driver.

2 The Parser

The parser we propose handles any context-free grammar; the grammar can be ambiguous and need not be in any normal form. The parser is a predictive shift-reduce bottom-up parser that uses compiled top down prediction information in the form of tables. Before run-time, a non-deterministic push down automaton (NPDA) is constructed from a given context-free grammar. The parsing tables encode the finite state control and the moves of the NPDA. At run-time, the NPDA is then driven in pseudo-parallel with the help of a chart. We show the construction of a basic machine which will be driven non-deterministically.

In the following, the input string is $w = a_1 \cdots a_n$ and the context-free grammar being considered is $G = (\Sigma, NT, P, S)$, where Σ is the set of terminal

[1] The same argument for the exponential grammar size complexity of Tomita's parser (Johnson, 1989) holds for Lang's technique.

[2] This problem is particularly acute for natural language processing since in this context the input length is typically small (10-20 words) and the grammar size very large (hundreds or thousands of rules and symbols).

[3] The characteristic LR(0) machine is the result of determinizing the machine we introduce. Since this procedure introduce exponentially more states, the LR(0) machine can be exponentially large.

symbols, NT the set of non-terminal symbols, P a set of production rules, S the start symbol. We will need to refer to the subsequence of the input string $w = a_1 \cdots a_N$ from position i to j, $w_{]i,j]}$, which we define as follows:

$$w_{]i,j]} = \begin{cases} a_{i+1} \cdots a_j & , \text{if } i < j \\ \varepsilon & , \text{if } i \geq j \end{cases}$$

We explain the data-structures used by the parser, the moves of the parser, and how the parsing tables are constructed for the basic NPDA. Then, we study the formal characteristics of the parser.

The parser uses two moves: shift and reduce. As in standard shift-reduce parsers, shift moves recognize new terminal symbols and reduce moves perform the recognition of an entire context-free rule. However in the parser we propose, shift and reduce moves behave differently on rules whose recognition has just started (i.e. rules that have been predicted) than on rules of which some portion has been recognized. This behavior enables the parser to efficiently perform reduce moves when ambiguity arises.

2.1 Data-Structures and the Moves of the Parser

The parser collects items into a set called the *chart*, $\mathcal{C}$. Each item encodes a well formed substring of the input. The parser proceeds until no more items can be added to the chart $\mathcal{C}$.

An *item* is defined as a triple $\langle s, i, j \rangle$, where s is a state in the control of the NPDA, i and j are indices referring to positions in the input string ($i, j \in [0, n]$).

In an item $\langle s, i, j \rangle$, j corresponds to the current position in the input string and i is a position in the input which will facilitate the reduce move.

A *dotted rule* of a context-free grammar G is defined as a production of G associated with a dot at some position of the right hand side: $A \rightarrow \alpha \bullet \beta$ with $A \rightarrow \alpha\beta \in P$.

We distinguish two kinds of dotted rules. *Kernel* dotted rules, which are of the form $A \rightarrow \alpha \bullet \beta$ with α non empty, and *non-kernel* dotted rules, which have the dot at the left most position in the right hand side ($A \rightarrow \bullet\beta$). As we will see, non-kernel dotted rules correspond to the predictive component of the parser.

We will later see each state s of the NPDA corresponds to a set of dotted rules for the grammar G.

The set of all possible states in the control of the NPDA is written $\mathcal{S}$. Section 2.2 explains how the states are constructed.

The algorithm maintains the following property (which guarantees its soundness)[4]: if an item $\langle s, i, j \rangle$ is in the chart $\mathcal{C}$ then for all dotted rules $A \rightarrow \alpha \bullet \beta \in s$ the following is satisfied:

(i) if $\alpha \in (\Sigma \cup NT)^+$, then $\exists \gamma \in (NT \cup \Sigma)^\star$ such that $S \stackrel{\star}{\Longrightarrow} w_{]0,i]} A\gamma$ and $\alpha \stackrel{\star}{\Longrightarrow} w_{]i,j]}$;

(ii) if α is the empty string, then $\exists \gamma \in (NT \cup \Sigma)^\star$ such that $S \stackrel{\star}{\Longrightarrow} w_{]0,j]} A\gamma$.

The parser uses three tables to determine which move(s) to perform: an action table, *ACTION*, and two goto tables, the kernel goto table, $GOTO_k$, and the non-kernel goto table, $GOTO_{nk}$.

The goto tables are accessed by a state and a non-terminal symbol. They each contain a set of states: $GOTO_k(s, X) = \{r\}, GOTO_{nk}(s, X) = \{r'\}$ with $r, r', s \in \mathcal{S}, X \in NT$. The use of these tables is explained below.

The action table is accessed by a state and a terminal symbol. It contains a set of actions. Given an item, $\langle s, i, j \rangle$, the possible actions are determined by the content of ACTION(s, a_{j+1}) where a_{j+1} is the $j + 1^{th}$ input token. The possible actions contained in ACTION(s, a_{j+1}) are the following:

- KERNEL SHIFT s', ($ksh(s')$ for short), for $s' \in \mathcal{S}$. A new token is recognized in a kernel dotted rule $A \rightarrow \alpha \bullet a\beta$ and a push move is performed. The item $\langle s', i, j+1 \rangle$ is added to the chart, since αa spans in this case $w_{]i,j+1]}$.
- NON-KERNEL SHIFT s', ($nksh(s')$ for short), for $s' \in \mathcal{S}$. A new token is recognized in a non-kernel dotted rule of the form $A \rightarrow \bullet a\beta$. The item $\langle s', j, j+1 \rangle$ is is added to the chart, since a spans in this case $w_{]j,j+1]}$.
- REDUCE $X \rightarrow \beta$, ($red(X \rightarrow \beta)$ for short), for $X \rightarrow \beta \in P$. The context-free rule $X \rightarrow \beta$ has been totally recognized. The rule spans the substring $a_{i+1} \cdots a_j$. For all items in the chart of the form $\langle s', k, i \rangle$, perform the following two steps:
  - for all $r_1 \in GOTO_k(s', X)$, it adds the item $\langle r_1, k, j \rangle$ to the chart. In this case, a dotted rule of the form $A \rightarrow \alpha \bullet X\beta$ is combined with $X \rightarrow \beta\bullet$ to form $A \rightarrow \alpha X \bullet \beta$; since α spans $w_{]k,i]}$ and X spans $w_{]i,j]}$, αX spans $w_{]k,j]}$.
  - for all $r_2 \in GOTO_{nk}(s', X)$, it adds the item $\langle r_2, i, j \rangle$ to the chart. In this case, a dotted rule of the form $A \rightarrow \bullet X\beta$ is combined with $X \rightarrow \beta\bullet$ to form $A \rightarrow X \bullet \beta$; in this case X spans $w_{]i,j]}$.

[4] This property holds for all machines derived from the basic NPDA.

The recognizer follows:

begin (* recognizer *)

Input:

$a_1 \cdots a_n$	(* input string *)
$ACTION$	(* action table *)
$GOTO_k$	(* kernel goto table *)
$GOTO_{nk}$	(* non-kernel goto table *)
$start \in \mathcal{S}$	(* start state *)
$\mathcal{F} \subset \mathcal{S}$	(* set of final states *)

Output:acceptance or rejection of the input string.

Initialization: $\mathcal{C} := \{\langle start, 0, 0\rangle\}$

Perform the following three operations until no more items can be added to the chart $\mathcal{C}$:

(1) KERNEL SHIFT: if $\langle s, i, j\rangle \in \mathcal{C}$ and if $ksh(s') \in \text{ACTION}(s, a_{j+1})$, then $\langle s', i, j+1\rangle$ is added to $\mathcal{C}$.

(2) NON-KERNEL SHIFT: if $\langle s, i, j\rangle \in \mathcal{C}$ and if $nksh(s') \in \text{ACTION}(s, a_{j+1})$, then $\langle s', j, j+1\rangle$ is added to $\mathcal{C}$.

(3) REDUCE: if $\langle s, i, j\rangle \in \mathcal{C}$, then for all $X \rightarrow \beta$ s.t. $red(X \rightarrow \beta) \in \text{ACTION}(s, a_{j+1})$ and for all $\langle s', k, i\rangle \in \mathcal{C}$, perform the following:

- for all $r_1 \in GOTO_k(s', X)$, $\langle r_1, k, j\rangle$ is added to $\mathcal{C}$;
- for all $r_2 \in GOTO_{nk}(s', X)$, $\langle r_2, i, j\rangle$ is added to $\mathcal{C}$.

If $\{\langle s, 0, n\rangle \mid \langle s, 0, n\rangle \in \mathcal{C} \text{ and } s \in \mathcal{F}\} \neq \emptyset$
then return acceptance
otherwise return rejection.

end (* recognizer *)

In the above algorithm, non-determinism arises from multiple entries in ACTION(s, a) and also from the fact that $GOTO_k(s, X)$ and $GOTO_{nk}(s, X)$ contain a set of states.

2.2 Construction of the Parsing Tables

We shall give an LR(0)-like method for constructing the parsing tables corresponding to the basic NPDA. Several other methods (such as LR(k)-like, SLR(k)-like) can also be used for constructing the parsing tables and are described in (Schabes, 1991).

To construct the LR(0)-like finite state control for the basic non-deterministic push-down automaton that the parser simulates, we define three functions, *closure*, $goto_k$ and $goto_{nk}$.

If s is a state, then $closure(s)$ is the state constructed from s by the two rules:

(i) Initially, every dotted rule in s is added to $closure(s)$;

(ii) If $A \rightarrow \alpha \bullet B\beta$ is in $closure(s)$ and $B \rightarrow \gamma$ is a production, then add the dotted rule $B \rightarrow \bullet\gamma$ to $closure(s)$ (if it is not already there). This rule is applied until no more new dotted rules can be added to $closure(s)$.

If s is a state and if X is a non-terminal or terminal symbol, $goto_k(s, X)$ and $goto_{nk}(s, X)$ are the set of states defined as follows:

$goto_k(s, X) =$
$\{closure(\{A \rightarrow \alpha X \bullet \beta\}) \mid A \rightarrow \alpha \bullet X\beta \in s$ and $\alpha \in (\Sigma \cup NT)^+\}$

$goto_{nk}(s, X) =$
$\{closure(\{A \rightarrow X \bullet \beta\}) \mid A \rightarrow \bullet X\beta \in s\}$

The goto functions we define differ from the one defined for the LR(0) construction in two ways: first we have distinguished transitions on symbols from kernel items and non-kernel items; second, each state in $goto_k(s, X)$ and $goto_{nk}(s, X)$ contains exactly one kernel item whereas for the LR(0) construction they may contain more than one.

We are now ready to compute the set of states $\mathcal{S}$ defining the finite state control of the parser.

The SET OF STATES CONSTRUCTION is constructed as follows:

procedure states(G)
begin
  $\mathcal{S} := \{closure(\{S \rightarrow \bullet\alpha \mid S \rightarrow \alpha \in P\})\}$
  repeat
    for each state s in $\mathcal{S}$
      for each $X \in \Sigma \cup NT$ terminal
        for each $r \in goto_k(s, X) \cup goto_{nk}(s, X)$
          add r to $\mathcal{S}$
  until no more states can be added to $\mathcal{S}$
end

PARSING TABLES. Now we construct the LR(0) parsing tables ACTION, GOTO$_k$ and GOTO$_{nk}$ from the finite state control constructed above. Given a context-free grammar G, we construct $\mathcal{S}$, the set of states for G with the procedure given above. We construct the action table ACTION and the goto tables using the following algorithm.

begin (CONSTRUCTION OF THE PARSING TABLES)

Input: A context-free grammar
$G = (\Sigma, NT, P, S)$.

Output: The parsing tables ACTION, GOTO$_k$ and GOTO$_{nk}$ for G, the start state *start* and the set of final states $\mathcal{F}$.

Step 1. Construct $\mathcal{S} = \{s_0, \cdots, s_m\}$, the set of states for G.

Step 2. The parsing actions for state s_i are determined for all terminal symbols $a \in \Sigma$ as follows:

(i) for all $r \in goto_k(s_i, a)$, add $ksh(r)$ to ACTION(s_i, a);

(ii) for all $r \in goto_{nk}(s_i, a)$, add $nksh(r)$ to to ACTION(s_i, a);

(iii) if $A \rightarrow \alpha\bullet$ is in s_i, then add $red(A \rightarrow \alpha)$ to ACTION(s_i, a) for all terminal symbol a and for the end marker \$.

Step 4. The kernel and non-kernel goto tables for state s_i are determined for all non-terminal symbols X as follows:

(i) $\forall X \in NT,\ GOTO_k(s_i, X) := goto_k(s_i, X)$

(ii) $\forall X \in NT,$
$GOTO_{nk}(s_i, X) := goto_{nk}(s_i, X)$

Step 3. The start state of the parser is
$start := closure(\{S \rightarrow \bullet\alpha |\ S \rightarrow \alpha \in P\})$

Step 4. The set of final states of the parser is
$\mathcal{F} := \{s \in \mathcal{S} | \exists\ S \rightarrow \alpha \in P\ s.t.\ S \rightarrow \alpha\bullet \in s\}$

end (CONSTRUCTION OF THE PARSING TABLES)

Appendix A gives an example of a parsing table.

3 Complexity

The recognizer requires in the worst case $O(|G|n^2)$-space and $O(|G|^2n^3)$-time; n is the length of the input string, $|G|$ is the size of the grammar computed as the sum of the lengths of the right hand side of each productions:

$$|G| = \sum_{A \rightarrow \alpha \in P} |\alpha| \quad \text{, where } |\alpha| \text{ is the length of } \alpha.$$

One of the objectives for the design of the non-deterministic machine was to make sure that it was not possible to reach an exponential number of states, a property without which the machine is doomed to have exponential complexity (Johnson, 1989). First we observe that the number of states of the finite state control of the non-deterministic machine that we constructed in Section 2.2 is proportional to the size of the grammar, $|G|$. By construction, each state (except for the start state) contains exactly one kernel dotted rule. Therefore, the number of states is bounded by the maximum number of kernel rules of the form $A \rightarrow \alpha\bullet\beta$ (with α non empty), and is $O(|G|)$. We conclude that the algorithm requires in the worst case $O(|G|n^2)$-space since the maximum number of items $\langle s, i, j\rangle$ in the chart is proportional to $|G|n^2$.

A close look at the moves of the parser reveals that the reduce move is the most complex one since it involves a pair of states $\langle s, i, j\rangle$ and $\langle s', k, j\rangle$. This move can be instantiated at most $O(|G|^2n^3)$-time since $i, j, k \in [0, n]$ and there are in the worst case $O(|G|^2)$ pairs of states involved in this move.[5] The parser therefore behaves in the worst case in $O(|G|^2n^3)$-time.

One should however note that in order to bound the worst case complexity as stated above, arrays similar to the one needed for Earley's parser must be used to implement efficiently the shift and reduce moves.[6]

As for Earley's parser, it can also be shown that the algorithm requires in the worst case $O(|G|^2n^2)$-time for unambiguous context-free grammars and behaves in linear time on a large class of grammars.

4 Retrieving a Parse

The algorithm that we described in Section 2 is a recognizer. However, if we include pointers from an item to the other items (to a pair of items for the reduce moves or to an item for the shift moves) which caused it to be placed in the chart, the recognizer can be modified to record all parse trees of the input string. The representation is similar to a shared forest.

The worst case time complexity of the parser is the same as for the recognizer ($O(|G|^2n^3)$-time) but, as for Earley's parser, the worst case space complexity increases to $O(|G|^2n^3)$ because of the additional bookkeeping.

5 Correctness and Comparison with Earley's Parser

We derive the correctness of the parser by showing how it can be mapped to Earley's parser. In the process, we will also be able to show why this parser can be more efficient than Earley's parser. The detailed proofs are given in (Schabes, 1991).

We are also interested in formally characterizing the differences in performance between the parser we propose and Earley's parser. We show that the parser behaves in the worst scenario as well as Earley's parser by mapping it into Earley's parser. The parser behaves better than Earley's parser because it has eliminated the prediction step which takes in the worst case $O(|G|n)$-time for Earley's parser. Therefore, in the most favorable scenario, the parser we

[5] Kernel shift and non-kernel shift moves require both at most $O(|G|n^2)$-time.

[6] Due to the lack of space, the details of the implementation are not given in this paper but they are given in (Schabes, 1991).

propose will require $O(|G|n)$ less time than Earley's parser.

For a given context-free grammar G and an input string $a_1 \cdots a_n$, let $\mathcal{C}$ be the set of items produced by the parser and $\mathcal{C}_{earley}$ be the set of items produced by Earley's parser. Earley's parser (Earley, 1970) produces items of the form $\langle A \rightarrow \alpha \bullet \beta, i, j \rangle$ where $A \rightarrow \alpha \bullet \beta$ is a single dotted rule and not a set of dotted rules.

The following lemma shows how one can map the items that the parser produces to the items that Earley's parser produces for the same grammar and input:

Lemma 1 If $\langle s, i, j \rangle \in \mathcal{C}$ then we have:

(i) for all kernel dotted rules $A \rightarrow \alpha \bullet \beta \in s$, we have $\langle A \rightarrow \alpha \bullet \beta, i, j \rangle \in \mathcal{C}_{earley}$

(ii) and for all non-kernel dotted rules $A \rightarrow \bullet\beta \in s$, we have $\langle A \rightarrow \bullet\beta, j, j \rangle \in \mathcal{C}_{earley}$

The proof of the above lemma is by induction on the number of items added to the chart $\mathcal{C}$.

This shows that an item is mapped into a set of items produced by Earley's parser.

By construction, in a given state $s \in \mathcal{S}$, non-kernel dotted rules have been introduced before run-time by the closure of kernel dotted rules. It follows that Earley's parser can require $O(|G|n)$ more space since all Earley's items of the form $\langle A \rightarrow \bullet\alpha, i, i \rangle$ $(i \in [0, n])$ are not stored separately from the kernel dotted rule which introduced them.

Conversely, each kernel item in the chart created by Earley's parser can be put into correspondence with an item created by the parser we propose.

Lemma 2 If $\langle A \rightarrow \alpha \bullet \beta, i, j \rangle \in \mathcal{C}_{earley}$ and if $\alpha \neq \varepsilon$, then $\langle s, i, j \rangle \in \mathcal{C}$ where $s = closure(\{A \rightarrow \alpha \bullet \beta\})$.

The proof of the above lemma is by induction on the number of kernel items added to the chart created by Earley's parser.

The correctness of the parser follows from Lemma 1 and its completeness from Lemma 2 since it is well known that the items created by Earley's parser are characterized as follows (see, for example, page 323 in Aho and Ullman [1973] for a proof of this invariant):

Lemma 3 The item $\langle A \rightarrow \alpha \bullet \beta, i, j \rangle \in \mathcal{C}_{earley}$ if and only if, $\exists \Upsilon \in (V_{NT} \cup V_T)^*$ such that $S \stackrel{*}{\Longrightarrow} w_{]0,i]} X \Upsilon$ and $X \Longrightarrow \Gamma\Delta \stackrel{*}{\Longrightarrow} w_{]i,j]}\Delta$.

The parser we propose is therefore more efficient than Earley's parser since it has compiled out prediction before run time. How much more efficient it is, depends on how prolific the prediction is and therefore on the nature of the grammar and the input string.

6 Optimizations

The parser can be easily extended to incorporate standard optimization techniques proposed for predictive parsers.

The closure operation which defines how a state is constructed already optimizes the parser on chain derivations in a manner very similar to the techniques originally proposed by Graham et al. (1980) and later also used by Leiss (1990).

In addition, the closure operation can be designed to optimize the processing of non-terminal symbols that derive the empty string in manner very similar to the one proposed by Graham et al. (1980) and Leiss (1990). The idea is to perform the reduction of symbols that derive the empty string at compilation time, i.e. include this type of reduction in the definition of *closure* by adding (iii):

If s is a state, then *closure*(s) is now the state constructed from s by the three rules:

(i) Initially, every dotted rule in s is added to *closure*(s);

(ii) if $A \rightarrow \alpha \bullet B\beta$ is in *closure*(s) and $B \rightarrow \gamma$ is a production, then add the dotted rule $B \rightarrow \bullet\gamma$ to *closure*(s) (if it is not already there);

(iii) if $A \rightarrow \alpha \bullet B\beta$ is in *closure*(s) and if $B \stackrel{*}{\Longrightarrow} \varepsilon$, then add the dotted rule $A \rightarrow \alpha B \bullet \beta$ to *closure*(s) (if it is not already there).

Rules (ii) and (iii) are applied until no more new dotted rules can be added to *closure*(s).

The rest of the parser remains as before.

7 Variants on the basic machine

In the previous section we have constructed a machine whose number of states is in the worst case proportional to the size of the grammar. This requirement is essential to guarantee that the complexity of the resulting parser with respect to the grammar size is not exponential or worse than $O(|G|^2)$-time as other well known parsers. However, we may use some non-determinism in the machine to guarantee this property. The non-determinism of the machine is not a problem since we have shown how the non-deterministic machine can be efficiently driven in pseudo-parallel (in $O(|G|^2 n^3)$-time).

We can now ask the question of whether it is possible to determinize the finite state control of the machine while still being able to bound the complexity of the parser to $O(|G|^2 n^3)$-time. Johnson (1989) exhibits grammars for which the full determinization

of the finite state control (the LR(0) construction) leads to a parser with exponential complexity, because the finite state control has an exponential number of states and also because there are some input string for which an exponential number of states will be reached. However, there are also cases where the full determinization either will not increase the number of states or will not lead to a parser with exponential complexity because there are no input that require to reach an exponential number of states. We are currently studying the classes of grammars for which this is the case.

One can also try to determinize portions of the finite state automaton from which the control is derived while making sure that the number of states does not become larger than $O(|G|)$.

All these variants of the basic parser obtained by determinizing portions of the basic non-deterministic push-down machine can be driven in pseudo-parallel by the same pseudo-parallel driver that we previously defined. These variants lead to a set of more efficient machines since the non-determinism is decreased.

8 Conclusion

We have introduced a shift-reduce parser for unrestricted context-free grammars based on the construction of a non-deterministic machine and we have formally proven its superior performance compared to Earley's parser.

The technique which we employed consists of constructing before run-time a parsing table that encodes a non-deterministic machine in the which the predictive behavior has been compiled out. At run time, the machine is driven in pseudo-parallel with the help a chart.

By defining two kinds of shift moves (on kernel dotted rules and on non-kernel dotted rules) and two kinds of reduce moves (on kernel and non-kernel dotted rules), we have been able to efficiently evaluate in pseudo-parallel the non-deterministic push down machine constructed for the given context-free grammar.

The same worst case complexity as Earley's recognizer is achieved: $O(|G|^2 n^3)$-time and $O(|G|n^2)$-space. However, in practice, it is superior to Earley's parser since all the prediction steps and some of the completion steps have been compiled before run-time.

The parser can be modified to simulate other types of machines (such LR(k)-like or SLR-like automata). It can also be extended to handle unification based grammars using a similar method as that employed by Shieber (1985) for extending Earley's algorithm.

Furthermore, the algorithm can be tuned to a particular grammar and therefore be made more efficient by carefully determinizing portions of the non-deterministic machine while making sure that the number of states in not increased. These variants lead to more efficient parsers than the one based on the basic non-deterministic push-down machine. Furthermore, the same pseudo-parallel driver can be used for all these machines.

We have adapted the technique presented in this paper to other grammatical formalism such as tree-adjoining grammars (Schabes, 1991).

Bibliography

A. V. Aho and J. D. Ullman. 1973. *Theory of Parsing, Translation and Compiling. Vol 1: Parsing.* Prentice-Hall, Englewood Cliffs, NJ.

Jay C. Earley. 1968. *An Efficient Context-Free Parsing Algorithm.* Ph.D. thesis, Carnegie-Mellon University, Pittsburgh, PA.

Jay C. Earley. 1970. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102.

S.L. Graham, M.A. Harrison, and W.L. Ruzzo. 1980. An improved context-free recognizer. *ACM Transactions on Programming Languages and Systems*, 2(3):415–462, July.

Mark Johnson. 1989. The computational complexity of Tomita's algorithm. In *Proceedings of the International Workshop on Parsing Technologies*, Pittsburgh, August.

T. Kasami. 1965. An efficient recognition and syntax algorithm for context-free languages. Technical Report AF-CRL-65-758, Air Force Cambridge Research Laboratory, Bedford, MA.

James R. Kipps. 1989. Analysis of Tomita's algorithm for general context-free parsing. In *Proceedings of the International Workshop on Parsing Technologies*, Pittsburgh, August.

D. E. Knuth. 1965. On the translation of languages from left to right. *Information and Control*, 8:607–639.

Bernard Lang. 1974. Deterministic techniques for efficient non-deterministic parsers. In Jacques Loeckx, editor, *Automata, Languages and Programming, 2nd Colloquium, University of Saarbrücken.* Lecture Notes in Computer Science, Springer Verlag.

Hans Leiss. 1990. On Kilbury's modification of Earley's algorithm. *ACM Transactions on Programming Languages and Systems*, 12(4):610–640, October.

Yves Schabes. 1991. Polynomial time and space shift-reduce parsing of context-free grammars and of tree-adjoining grammars. In preparation.

Stuart M. Shieber. 1985. Using restriction to extend parsing algorithms for complex-feature-based formalisms. In *23rd Meeting of the Association for Computational Linguistics (ACL'85)*, Chicago, July.

Masaru Tomita. 1985. *Efficient Parsing for Natural Language, A Fast Algorithm for Practical Systems.* Kluwer Academic Publishers.

Masaru Tomita. 1987. An efficient augmented-context-free parsing algorithm. *Computational Linguistics*, 13:31–46.

D. H. Younger. 1967. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208.

A An Example

We give an example that illustrates how the recognizer works. The grammar used for the example generates the language $L = \{a(ba)^n | n \geq 0\}$ and is infinitely ambiguous:

$S \rightarrow S\ b\ S$

$S \rightarrow S$

$S \rightarrow a$

The set of states and the goto function are shown in Figure 1. In Figure 1, the set of states is $\{0, 1, 2, 3, 4, 5\}$. We have marked with a sharp sign ($\sharp$) transitions on a non-kernel dotted rule. If an arc from s_1 to s_2 is labeled by a non-sharped symbol X, then s_2 is in $goto_k(s_1, X)$. If an arc from s_1 to s_2 is labeled by a sharped symbol $X\sharp$, then s_2 is in $goto_{nk}(s_1, X)$.

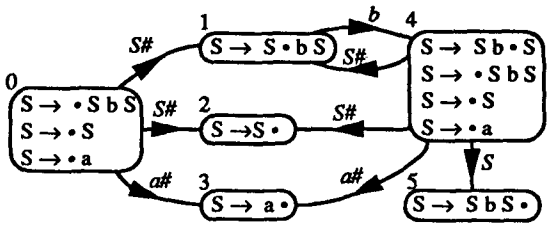

Figure 1: Example of set of states and goto function.

The parsing table corresponding to this grammar is given in Figure 2.

State	ACTION			GOTOk	GOTOnk
	a	*b*	$	*S*	*S*
0	*nksh*(3)				{1,2}
1		*ksh*(4)			
2	*red*($S \rightarrow S$)	*red*($S \rightarrow S$)	*red*($S \rightarrow S$)		
3	*red*($S \rightarrow a$)	*red*($S \rightarrow a$)	*red*($S \rightarrow a$)		
4	*nksh*(3)			{5}	{1,2}
5	*red*($S \rightarrow SbS$)	*red*($S \rightarrow SbS$)	*red*($S \rightarrow SbS$)		

Figure 2: An LR(0) parsing table for $L = \{a(ba)^n \mid n \geq 0\}$. The start state is 0, the set of final states is $\{2, 3, 5\}$. $ stands for the end marker of the input string.

The input string given to the recognizer is: *ababa*$ ($ is the end marker). The chart is shown in Figure 3. In Figure 3, an arc labeled by s from position i to position j denotes the item $\langle s, i, j\rangle$. The input is accepted since the final states 2 and 5 span the entire string ($\langle 2, 0, 5\rangle \in C$ and $\langle 5, 0, 5\rangle \in C$). Notice that there are multiple arcs subsuming the same substring.

input read	*items in the chart*		
	$\langle 0,0,0\rangle$		
a	$\langle 3,0,1\rangle$	$\langle 2,0,1\rangle$	$\langle 1,0,1\rangle$
ab	$\langle 4,0,2\rangle$		
aba	$\langle 3,2,3\rangle$	$\langle 2,0,3\rangle$	$\langle 2,2,3\rangle$
	$\langle 1,0,3\rangle$	$\langle 1,2,3\rangle$	$\langle 5,0,3\rangle$
abab	$\langle 4,0,4\rangle$	$\langle 4,2,4\rangle$	
ababa	$\langle 3,4,5\rangle$	$\langle 2,0,5\rangle$	$\langle 2,2,5\rangle$
	$\langle 2,4,5\rangle$	$\langle 1,0,5\rangle$	$\langle 1,2,5\rangle$
	$\langle 1,4,5\rangle$	$\langle 5,0,5\rangle$	$\langle 5,2,5\rangle$

Figure 3: Chart created for the input $_0\,a\,_1\,b\,_2\,a\,_3\,b\,_4\,a\,_5$ $.