An Efficient Context-Free Backbone for Natural Language Analyzers

Manuel Vilares Ferro

Facultad de Informática
Campus de Elviña, s/n
15071 La Coruña, Spain

Abstract

Parsing efficiency is crucial when building practical natural language systems. This paper introduces an efficient context-free parsing algorithm based on dynamic programming techniques[1], and focuses on its practical application to natural languages interfaces. Our analyzer takes a general class of context-free grammars as drivers, using the same user interface applied by the standard generators of parsers in UNIX, which facilitates its use in relation to another context-free analyzers.

To assure computational efficiency, we consider the concept of non-deterministic push-down transducer as operational model simulating all possible computations at worst in cubic time and space complexity. So, the essential feature of our context-free parser is that we do not interpret the grammars, but we first compile them. In addition, we solve the problem of the optimal sharing of computations during the parse process introducing very simple techniques of dynamic programming.

Another important question relies on the treatment of parse forest construction. In effect, the parse is represented by a sequence of rules to be used in a left-to-right reduction of the input sentence to the start symbol of the grammar, which allow us to gracefully solve the problem of the optimal sharing of the output syntactic structure.

The new system has been baptized ICE, after Incremental Context-Free Environment. In an empirical comparison, ICE appears to be superior to the others context-free analyzers and comparable to the standard generators of deterministic parsers, when the input string is not ambiguous.

Incremental facilities provided by ICE will not be commented in the present work. This aspect of the tool will be explained separately in a next paper because its complexity.

Key Words and Phrases: Context-Free Parsing, Dynamic Programming, Dynamic Frames, Earley Parsing, Parse Forest, Push-Down Automata.

1 Introduction

At the outbreak, interest in non-deterministic parsing was due to the fact that it is the context-free backbone of some natural language analyzers. In effect, several of them start with a purely context-free parsing phase which generates all possible parses as some kind of sharing structure. Later, in a second phase, a more elaborate analyzer that takes into account the finer grammatical structure is applied in order to filter out undesirable parses. So, we can benefit from the experience accumulated along the years in parsing technology.

In the present work, the problem is stated in the context of the application of deterministic techniques to generate very efficient non-deterministic parsers, as proposed by Lang in [8]. Essentially, we consider a simple variation of Earley's dynamic programming construction [4], where in order to solve the problems derived from grammatical constraints, we extend it to push-down transducers (PDTs), separating the execution strategy from the implementation of the push-down automaton (PDA) interpreter. So, we can obtain a family of general context-free parallel parsers for each family of deterministic context-free push-down ones, simulating all possible computations of any PDT, at worst in cubic time. However, the method is linear in a large class of grammars.

[1] this work was partially supported by the Eureka Software Factory project, and integrally developed at INRIA, Institut National de Recherche en Informatique et en Automatique, Rocquencourt, France.

In section 2 of this paper, we present and justify the implementation of our general context-free parser. Section 3 is a study of the formal efficiency properties of the algorithm. In section 4, we give an extensive range of comparative tests between ICE and the best known deterministic and non-deterministic parsers. Finally, section 5 reports a short conclusion about the work presented.

2 Context-Free Parsing

The following is an informal overview of parsing by dynamic programming interpretation of PDAs, such as it is implemented in ICE. Our aim is to parse sentences in the language $\mathcal{L}(\mathcal{G})$ generated by a context-free grammar $\mathcal{G} = (N, \Sigma, P, S)$ according to its syntax, where N is the set of non-terminals, Σ the set of terminal symbols, P the rules and S the start symbol. The empty string will be represented by ε.

2.1 The operational model

The operational model of our parser is a classic PDT, in general non-deterministic. To represent it, we assume a formal definition taken from Lang in [8] that can fit most usual PDT construction techniques. It is defined as a 8-tuple $\mathcal{T}_{\mathcal{G}} = (\mathcal{Q}, \Sigma, \Delta, \Pi, \delta, q_0, Z_0, \mathcal{Q}_f)$, where: $\mathcal{Q}$ is the set of states, Σ the set of input word symbols, Δ the set of stack symbols, Π the set of output symbols, q_0 the initial state, Z_0 the initial stack symbol, and $\mathcal{Q}_f$ the set of final states. In relation to δ, it is a finite set of transitions of the form $\tau = \delta(p, X, a) \ni (q, Y, u)$ with $p, q \in \mathcal{Q}$; $a \in \Sigma \cup \{\varepsilon\}$; $X, Y \in \Delta \cup \{\varepsilon\}$, and $u \in \Pi^*$.

To represent the state of a PDT in a moment of the parse process, we define a *configuration* as a 5-tuple $(p, X\alpha, ax, u)$, where p is the current state, $X\alpha$ the stack contents with X on the top, ax the remaining input where the symbol a is the next to be shifted, $x \in \Sigma^*$, and u is the already produced output. The application of a transition $\tau = \delta(p, X, a) \ni (q, Y, v)$ results in a new configuration $(q, Y\alpha, x, uv)$ where the terminal symbol a has been scanned, X has been popped, Y has been pushed, and v has been concatenated to the existing output u. So, for example, if the terminal symbol a is ε in the transition, no input symbol is scanned. If X is ε then no stack symbol is popped from the stack. In a similar manner, if Y is ε then no stack symbol is pushed on the stack. We can distinguish, in particular, two special types of configuration:

- *Initial configurations*, from which the computation process starts. They are of the form $(q_0, \varepsilon, x, \varepsilon)$, where x represents the total input string.
- *Final configurations*, that are the result of a successful computation process. We can always assume that they are of the form $(q_f, \varepsilon, \varepsilon, w)$, where $q_f \in \mathcal{Q}_f$.

The described framework is called S^T by the dependence of the formalism of transitions on the total parse stack, which reduces the possibility of sharing computations, a fundamental problem in ambiguous parsing. We shall now reduce this dependence considering the notion of *dynamic frame*.

2.2 The concept of dynamic frame

In our context, a dynamic programming interpretation of a PDT is the systematic exploration of a space of elements called *items*. This search space is a condensed representation of all possible

computations of the PDT. It is important to guarantee that all useful parts of that space are actually explored (cf. fairness, completeness), and that useless or redundant parts are ignored as much as possible (cf. admissibility). It is also necessary to assure that the representation of configurations by items is compatible with the formalism of transitions. To formalize this idea, we introduce the concept of *dynamic frame* establishing the conditions over which correctness and completeness of computations with items are verified in relation to S^T.

Definition 2.1 *Let $\mathcal{T}_\mathcal{G} = (\mathcal{Q}, \Sigma, \Delta, \Pi, \delta, q_0, Z_0, \mathcal{Q}_f)$ be a PDT. We define a* dynamic frame *as a pair $(\Re, Op)$ where:*

- *$\Re$ is an equivalence relation on the stacks, whose classes are named* items. *We denote:*
  - *The class of ξ as $\bar{\xi}$.*
  - *The set of items as $It_{\mathcal{T}_\mathcal{G},\Re}$, or simply It when the context is clear.*
- *Op is an operator of the form:*

$$\begin{array}{rcl} Op: \; \delta & \rightarrow & \{It^+ \times \Sigma \cup \{\varepsilon\} \rightarrow It^+ \times \Pi^*\} \\ \tau & \rightsquigarrow & Op(\tau): It^n \times \Sigma \cup \{\varepsilon\} \rightarrow It^m \times \Pi^* \end{array}$$

  verifying the following conditions:

  - Compatibility*: all computation in S^T must have its corresponding counterpart in the dynamic frame.*
  - Completeness*: all final configuration in S^T has its corresponding counterpart in the dynamic frame.*
  - Correctness*: all final configuration in the dynamic frame has its corresponding counterpart in S^T.*

  where n and m depend on the nature of the transition. More exactly, they represent respectively the number of items over which we apply the transition, and the number of items resulting of this application.

□

Dynamic frames were originally introduced by Villemonte de la Clergerie in [14] to formalize the notion of item in relation to the use of *logical push-down automata*[2] for constructing efficient and complete definite clause programs compilers. The consideration to our particular case was proposed by the same author of this paper in [13].

In practice, only two dynamic frames are considered: S^1 and S^2. Both of them construct items from the notion of *mode*. A mode is a 4-tuple (p, X, S_i^w, S_j^w), where p is the current state in our PDT, X the last recognized symbol from state p, S_i^w a pointer, called *back-pointer*, to the position i in the input string w containing the first token derived from the symbol X, and S_j^w is a pointer to the position j of the token currently analized. The only difference between S^1 and S^2 consists in the number of modes considered. So, S^1 considers an item as an structure composed by an only mode while S^2 considers items constructed by two modes in the form $[(p, X, S_i^w, S_j^w)(q, Y, S_k^w, S_i^w)]$. This implies, in essence, that S^1 represents the current

[2] essentially, PDAs that store atoms and substitutions on their stack, and use unification to apply transitions. They are due to Lang [9], which obtains an exponential reduction in complexity over the traditional resolution methods.

configuration of the PDT by the top of the stack while S^2 uses also the preceding element in the stack.

At this point, the choice of a particular dynamic frame is central to assure a good sharing computation process, essential to guarantee efficiency in a non-deterministic context. In relation to this, S^2 cannot be considered as optimal because the continuous dependence on the context represented by the second mode of the items. This is the reason for which we have adopted S^1 as dynamic frame.

2.3 The parser

We assume that, using a standard technique, we produce from the grammar $\mathcal{G}$ a LALR(1) recognizer, possibly non-deterministic, for the language $\mathcal{L}(\mathcal{G})$.

The algorithm proceeds by building a collection of items. We associates a set of items S_i^w, habitually called *itemset*, for each word symbol w_i at the position i in the input string of length n, $w_{1..n}$.

New items are produced by applying transitions to existing ones, until no new application is possible. An item represents the current stack configuration in the form $[p, X, S_i^w, S_j^w]$, where p is the current state in our extended LALR(1) automaton, X the last recognized symbol from state p, S_i^w the itemset containing the first token derived from the symbol X, and S_j^w the current itemset. From a technical point of view, this implies that we work in a *dynamic frame* S^1 in opposition to the classic one S^T, where no compact representation is considered. That is an important point, since the more compact are the representations, the more successful will be the sharing of computations, and later of syntactic structures. Given a transition $\tau = \delta(p, X, a) \ni (q, Y, u)$ in S^T, we translate it to S^1 in the form of a transition $Op(\tau)$ given by:

$$
\begin{array}{llllll}
1. & \tilde{\delta}([p, X, S_j^w, S_i^w], a) & \ni & ([q, \varepsilon, S_i^w, S_i^w], & \varepsilon) & \text{if } Y = X \\
2. & \tilde{\delta}([p, X, S_j^w, S_i^w], a) & \ni & ([p, Y, S_i^w, S_{i+1}^w], & I_0 \rightarrow a) & \text{if } Y = a \\
3. & \tilde{\delta}([p, X, S_j^w, S_i^w], a) & \ni & ([p, Y, S_i^w, S_i^w], & I_1 \rightarrow I_2) & \text{if } Y \in N \\
4. & \tilde{\delta}([p, \varepsilon, S_j^w, S_i^w], a) & \ni & \tilde{\delta}_d([q, \varepsilon, S_l^w, S_i^w], a) \ni ([q, \varepsilon, S_l^w, S_j^w], & I_3 \rightarrow I_4 I_5) & \text{if } Y = \varepsilon \\
 & & & \forall q \in \mathcal{Q} \text{ such that: } \exists\, \delta(q, X, \varepsilon) \ni (p, X, \varepsilon) & &
\end{array}
$$

where:

$$\tilde{\delta} : \text{It} \times \Sigma \cup \{\varepsilon\} \longrightarrow \{\text{It} \cup \tilde{\delta}_d\} \times \Pi^* \qquad \tilde{\delta}_d : \text{It} \times \Sigma \cup \{\varepsilon\} \longrightarrow \text{It}$$

and

$$
\begin{array}{lll}
I_0 = [p, Y, S_i^w, S_{i+1}^w] & I_1 = [p, Y, S_i^w, S_i^w] & I_2 = [p, X, S_j^w, S_i^w] \\
I_3 = [q, \varepsilon, S_l^w, S_i^w] & I_4 = [q, X, S_l^w, S_j^w] & I_5 = [p, \varepsilon, S_j^w, S_i^w]
\end{array}
$$

with *It* the set of all items developed in the parsing process and $\tilde{\delta}_d$ is called the set of *dynamic transitions.*

It is important to comment the behavior of the algorithm face to a pop action, the last case represented. In effect, given that our compact representation of the stack is its top, we must consider a protocol to treat the absence of information about the rest of the stack. The solution relies to the concept of *dynamic transition.* Briefly, it consists in generating a new transition from that implying the pop action. This new transition must be built in such a manner that it is applicable not only to the configuration resulting of the first one, but also on those to be generated and sharing the same syntactic structure.

In relation to fairness and completeness, an equitable selection order must be established to treat the items. We use a technique of *merit ordering* [7]. In essence, we process the items in an itemset in order, performing none or some transitions on each one depending on the form of the item. These operations may add more states to the current itemset and may also put states in the itemset corresponding to the following token to be analyzed from the input string. To ignore redundant items we put in place a simple subsumption relation based in the equality.

2.4 The shared forest

A major difference with other analyzers is that we represent a parse as the string of the context-free grammar rules used in a leftmost reduction of the parsed sentence, rather than as a parse tree. When the sentence has several distinct parses, the set of all possible parse strings is represented in finite shared form by a context-free grammar that generates that possibly infinite set.

However, this difference is only appearance. In effect, context-free grammars can be represented by AND-OR graphs that in our case are precisely the shared-forest graph. To generate it, items are used as non-terminals of an output contex-free grammar $\mathcal{G}_o = (N_o, \Sigma_o, P_o, S_o)$, where N_o is the set of all items, Σ_o the set of input symbols of the original grammar[3] $\mathcal{G}$, and the rules in P_o are constructed together with their left-hand-side item I by the algorithm. More exactly, we generate a rule for the output grammar each time a reduce or a shift action from the grammar defining the language is applied on the stack. In both cases, the left-hand-side of this rule is the new item describing the resulting configuration. In relation to the right-hand-side, it is composed by the token recognized in the case of a shift and by the items popped from the stack in that action and the number of the reduced rule in the original grammar, in the case of a reduction. So, the start symbol S_o derives of the last item produced by a successful computation.

In this graph, AND-nodes correspond to the usual parse-tree nodes, while OR-nodes correspond to ambiguities. Sharing of structures is represented by nodes accessed by more than one other node and it may correspond to sharing of a complete subtree, but also sharing of a part of the descendants of a given node, in fact, a consequence of the binary nature of the transition protocol preceding described. This feature allows us to obtain a cubic space complexity for the shared forest in the worst case, whichever it is the form of the grammar.

2.5 The choice of the kernel

The choice of the deterministic parsing scheme to be the kernel of ICE depends *in fine* on: the grammar used, the corpus of sentences to be analyzed, and a balance between computational and sharing efficiency, and parser size. As a consequence, the problem is a practical one and the best basis to decide, experimental. In this sense, the results achieved by Billot and Lang in [1], and earlier by Bouckaert, Pirotte and Snelling in [2] show, on an experimental basis, that techniques close to straightforward bottom-up methods, such as LALR(1), are the most appropriate.

From an intuitive point of view, that conclusion was expected. Effectively, those algorithms combine certain characteristics making them good candidates for a non-deterministic approach:

- A very large deterministic domain. In this direction, it should be good to remember that user often write grammars sufficiently close to deterministic ones. This phenomenon,

[3] for readability in the graphic representation of the shared forest, we shall also consider as terminals, the integers corresponding to rule numbers of the input language grammar $\mathcal{G}$.

called *local determinism*, has already been observed by Lang in [8]. Therefore, the use of the above parallel parsing techniques allows an important economy in time and space.

- An efficient treatment of the sharing problem of computations and structures:
  - The consideration of efficient determinization techniques such as the classic LR(k) ones, in principle the best, has as a consequence the distortion of the initial grammar due to the application of a predictive technique in the generation of the parser in relation to the lookahead facility. This leads to the well known *state splitting phenomenon*. Sharing of syntactic structures can then be affected.
  - In the case of LALR(1) parsers, not only the necessary tests to implement lookahead facility are easier to perform, but the state splitting phenomenon remains reasonable. That allows us to assure a good sharing of computation and parsing structures.

3 Time and Space Bounds

The time bounds for the parser are the same as those of the recognizer. So, our algorithm is in general, a $\mathcal{O}(n^3)$ recognizer, where n represents the length of the current input string $w_{1..n}$. The reasons for this are the following:

- The number of items in any itemset S_i^w is $\mathcal{O}(i)$, because the ranges of $\mathcal{Q}$ and Δ are bounded, while only the back-pointer depends on i, and it is bounded by n.
- In consequence, the maximum number of items is given by

$$| It | \leq \Sigma_{i=0}^{n} | S_i^w | = \frac{(n+1)(n+2)}{2} | \Delta || \mathcal{Q} |$$

which implies that the space bounds are in the worst case $\mathcal{O}(n^2)$, for the recognizer.

In the case of the parser, space bounds are cubic. To justify that, we shall prove that the maximum number of rules generated is $\mathcal{O}(n^3)$. Given that from an item $I = [p, A, S_i^w, S_j^w]$ and an applicable transition $\tau = \delta(p, A, a) \ni (q, B, x)$, the parser produces exactly one output rule except in the case of a pop transition, it will be sufficient to do it for this particular case. So, let's assume that $B = \varepsilon$, then the number of rules created is equal to the number of existing items of the form $[q, C, S_l^w, S_i^w]$, $0 \leq l \leq i \leq j$, such that there exists a transition $\delta(q, A, \varepsilon) \ni (p, A, \varepsilon)$. For a given item I, there are at most $(l+1) \mid \Delta \mid\mid \mathcal{Q} \mid$ such items. Summing over all items, we obtain a complexity $\mathcal{O}(\Sigma_{i=0}^{n} \Sigma_{l=0}^{i} (l+1)) = \mathcal{O}(n^3)$, for the space.

As a consequence, time bound is $\mathcal{O}(n^3)$ in the general case. However, we can characterize classes of grammars with inferior time bound. In particular, if we can assure that there is a fixed bound on the size of any itemset, then the algorithm will only execute $\mathcal{O}(n)$ steps. Observe that it is the case of LALR(1) grammars, which assures a good behavior in the case of grammars sufficiently close to this deterministic model.

4 Experimental Results

We use the syntax of complete Pascal, an universally known language, to show the efficiency of ICE comparing it with: YACC [6], the standard generator of parsers in UNIX, when the context is deterministic. Otherwise, we have chosen SDF [5], in the best of our knowledge the most efficient generator of non-deterministic parsers which is inspired in the work of Tomita

in [12]. Finally, we have also compared ICE with an implementation of the classic Earley's algorithm proposed by the same author of this paper in [13]. Results are shown in figure 1, where:

1. All tests have been performed using the same input programs for each one of the parsers and the time needed to "print" parse trees was not measured.

2. We include the time corresponding to lexical analysis, since it is not possible in SDF to differentiate that from the parsing. In all other cases, LEX [10] has been used as lexical analyzer. To minimalize the time corresponding to this analysis, we have considered very short programs.

3. All the measurements have been performed on a *Sun SPARCstation 2*, weakly loaded.

4. The ambiguous version for Pascal includes the *dangling else* and the ambiguity for the arithmetic expressions. To illustrate non-determinism, we measure the time needed to parse programs of the form:

   program P **(input, output)**;
   var a, b : **integer**;
   begin
   $a := b\{+b\}^i$
   end.

   where i is the number of +'s. As the grammar contains a rule "Expression ::= Expression + Expression", these programs have a number of ambiguous parses which grows exponentially with i. This number is:

$$C_i = \begin{cases} 1 & \textbf{if } i \in \{0,1\} \\ \sum_{k=0}^{i-1} C_k C_{i-k-1} = \binom{2i}{i} \frac{1}{\text{i+1}} & \textbf{if } i > 1 \end{cases}$$

5. SDF works on an extended LR(0) machine, and ICE uses an extended LALR(1) one. So, the ambiguous Pascal grammar used for SDF has 357 shift/reduce conflict, while that used for ICE contains only 305.

6. Given that in both, SDF and ICE, mapping between concrete and abstract syntax is fixed, we have generated in the case of YACC, a simple recognizer.

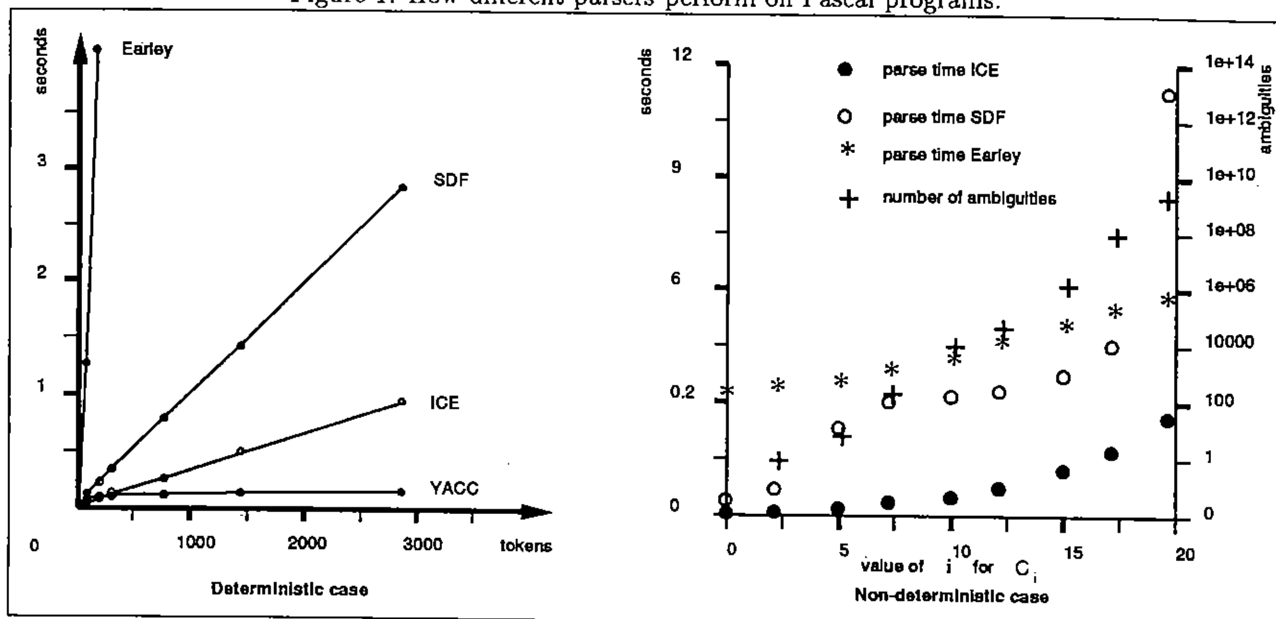
7. ICE and SDF have been written in Le_Lisp [3], while YACC is written in C.

5 Conclusion

This work applies to the construction of very efficient context-free parsers. At this point, ICE has proved its practical validity in comparison with the best deterministic and non-deterministic generators of parsers. In this sense, our algorithm not only surpasses the best previous results, but it does this with one single algorithm which does not have specified to it the class of grammars it is operating on and does not require the grammar in any special form.

The modularity of the approach emphasizes the possible variations in the operational strategies, showing the importance of the choice of the dynamic framework in relation to the efficiency of the parse process.

Figure 1: How different parsers perform on Pascal programs.



A balance between sharing of computations and sharing of syntactic structures seems to favor a LALR(1) approach to implement the operational kernel of ICE. In connection with this, we must remark that there exists a close relation between the principle of sharing and the amount of work developed by the parser. From an intuitive point of view, the sharing of forest reflects the sharing of computations and therefore time and space bounds are at stake. However, this relation is not always complete. In effect, the best sharing quality for syntactic structures is obtained using algorithms taking only into account grammatical features as it is the case of Earley in [4]. The reverse of the coin is represented by its more restrictive deterministic domain, which implies higher time and space bounds. On the other hand, more efficient deterministic parsers as LR(k) have an important state splitting phenomenon, and as a consequence sharing of structures is less efficient.

Finally, ICE is compatible with YACC [6], which permits to freely use all the input that were developed for this generator.

References

[1] **Billot, S. and Lang, B.**
The Structure of Shared Forest in Ambiguous Parsing.
1989. Research Report n^o1038, INRIA Rocquencourt, France.

[2] **Bouckaert, M.; Pirotte. A. and Snelling, M.**
Efficient Parsing Algorithms for General Context-Free Grammars.
1975. Information Sciences, vol. 8 (pp. 1-26).

[3] **Chailloux, J.**
Le_Lisp. Version 15.23. Reference Manual.
1990. INRIA. Rocquencourt. France.

[4] **Earley, J.**
An Efficient Context-Free Parsing Algorithm.
1970. Communications of the ACM, vol. 13, n^o2 (pp. 94-102).

[5] **Heering, J. and Klint, P.**
A Syntax Definition Formalism.
1987. ESPRIT '86: Results and Achievements, North-Holland Publishing Company, New York, U.S.A (pp. 619-630).

[6] **Johnson, S. C.**
YACC. Yet Another Compiler Compiler.
1975. Computer Sciences Technical Report n^o32, AT&T Bell Laboratories, Murray Hill, New Jersey, U.S.A.

[7] **Kowalski, R. A.**
Logic for Problem Solving.
1980. North-Holland Publishing Company, New York, U.S.A.

[8] **Lang, B.**
Deterministic Techniques for Efficient Non-deterministic Parsers.
1974. Research Report n^o72, INRIA Rocquencourt, France.

[9] **Lang, B.**
Complete Evaluation of Horn Clauses, an Automata Theoretic Approach.
1988. Research Report n^o913, INRIA Rocquencourt, France.

[10] **Lesk, M. E. and Schmidt, E.**
Lex: *a Lexical Analyzer Generator.*
1975. UNIX Programmer's Manual 2. Murray Hill, N.J.: AT&T Bell Laboratories,

[11] **Sheil, B. A.**
Observations on Context-Free Grammars.
1976. Statistical Methods in Linguistics, Stockholm, Sweden (pp. 71-109).

[12] **Tomita, M.**
An Efficient Augmented-Context-Free Parsing Algorithm.
1987. Computational Linguistics, vol. 13, n^o1.2 (pp. 31-36).

[13] **Vilares Ferro, M.**
Efficient Incremental Parsing for Context-Free Languages.
1992. Doctoral thesis, University of Nice, France.

[14] **Villemonte de la Clergerie, E.**
DyALog. Une Implementation des Clauses de Horn en Programmation Dynamique.
1990. Actes du 9th Séminaire de Programmation en Logique, CNET, Lannion, France (pp. 207-228).

A A simple example of dynamic frames

To illustrate the following discussion, we shall assume the grammar $\mathcal{G}$ given by the productions:

$$\begin{array}{llll} (0)\ \Phi \rightarrow A \dashv & (1)\ A \rightarrow a\,b\,B & (2)\ B \rightarrow C\,F \\ (3)\ B \rightarrow D\,F & (4)\ B \rightarrow E\,F & (5)\ C \rightarrow \varepsilon \\ (6)\ D \rightarrow \varepsilon & (7)\ E \rightarrow \varepsilon & (8)\ F \rightarrow d\,e \end{array}$$

whose LALR(1) finite state machine is represented in figure 2.

To prove the influence of the choice of the dynamic frame in the sharing of computations and syntactic structures, we shall consider the grammar $\mathcal{G}$ and the input string $w = abde \dashv$. In relation with this, figure 3 shows the state of the LALR(1) stacks after the scan of the token e from the original input string. Each atom of these stacks represents a symbol and the state of the LALR(1) where it was recognized. It is evident that the best sharing corresponds to S^1 and the worst to S^T where the concept of item is not considered.

B A simple example for the parser

To show the application of the formalism of transitions in dynamic programming, we shall consider the grammar $\mathcal{G}$, the dynamic frame S^1 and the input string $w = abde\dashv$. All the results can be verified by the reader from table 1 and figure 4. At this point, it is important to remark that we only effectively save in the itemsets those items representing the recognition of some syntactic category in the initial grammar $\mathcal{G}$. In fact, we shall see that they are sufficient to built the resulting parse forest.

To start with, the algorithm automatically adds the item $I_0^0 = [0, \varepsilon, S_0^w, S_0^w]$ to S_0^w. To follow the construction, it is sufficient to apply transitions from the original PDT, whose LALR(1)

Figure 2: The LALR(1) machine for the $\mathcal{G}$ grammar.

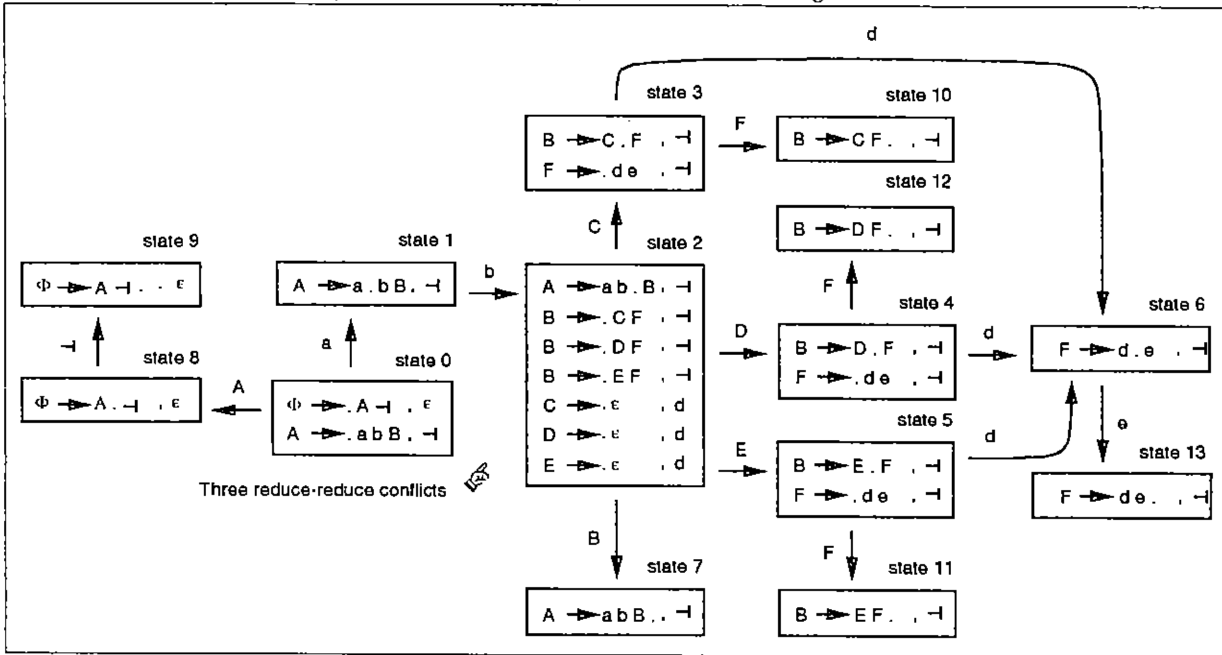
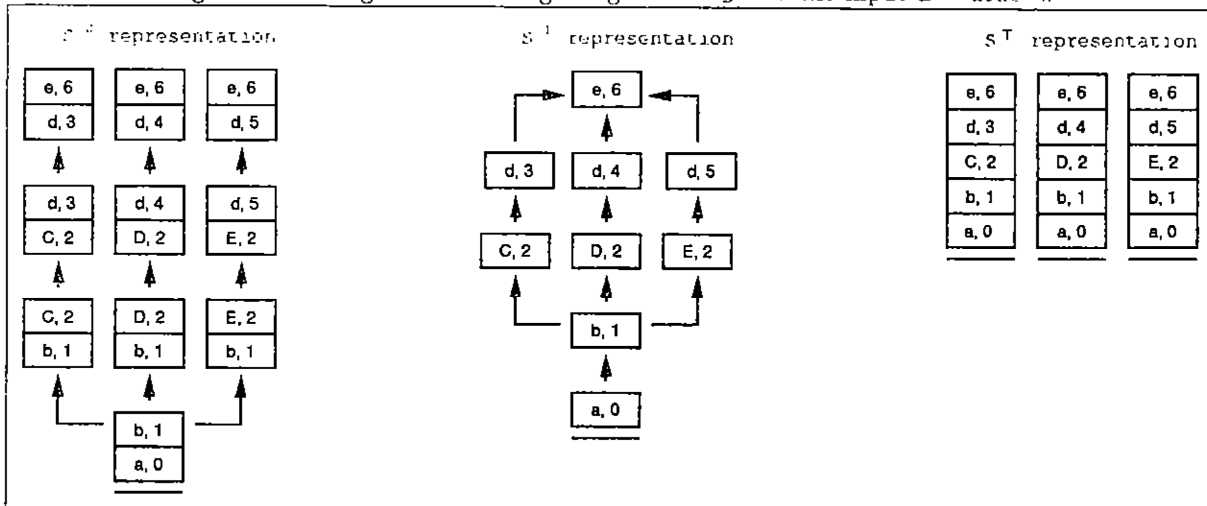


Figure 3: Sharing of stacks using the grammar $\mathcal{G}$ and the input $w = abde \dashv$.



finite state machine is shown in figure 2. At this point, there exists an only action to consider in state 0 when the lookahead is b, in fact a shift action on this token that we can translate in the application of the transitions:

$$\begin{array}{lllll}\delta(0,\varepsilon,a) & \ni & (0,a,a) & \text{to produce} & I_0^1=[0,a,S_0^w,S_1^w], & \text{output} & I_0^1 \rightarrow a \\ \delta(0,a,\varepsilon) & \ni & (1,a,\varepsilon) & \text{to produce} & [1,\varepsilon,S_1^w,S_1^w], & \text{output} & \varepsilon\end{array}$$

The last transition does not generate any output structure because it does not express the recognition of a syntactic category. Now, on state 1 we can perform a shift action on the token b applying:

$$\begin{array}{lllll}\delta(1,\varepsilon,b) & \ni & (1,b,b) & \text{to produce} & I_0^2=[1,b,S_1^w,S_2^w], & \text{output} & I_0^2 \rightarrow b \\ \delta(1,b,\varepsilon) & \ni & (2,b,\varepsilon) & \text{to produce} & [2,\varepsilon,S_2^w,S_2^w], & \text{output} & \varepsilon\end{array}$$

In state 2, the LALR(1) machine whose LALR(1) kernel is shown in figure 2 has three reduce-reduce conflicts, which result in three ambiguities. These ambiguities are represented by the transition:

$$\tau_0=\delta(2,\varepsilon,d)=\{(2,C,C),(2,D,D),(2,E,E)\}$$

that produces respectively the items

$$\begin{array}{llll}I_1^2=[2,C,S_2^w,S_2^w] & \in & S_2^w, & \text{output} \quad I_1^2 \rightarrow 5 \\ I_2^2=[2,D,S_2^w,S_2^w] & \in & S_2^w, & \text{output} \quad I_2^2 \rightarrow 6 \\ I_3^2=[2,E,S_2^w,S_2^w] & \in & S_2^w, & \text{output} \quad I_3^2 \rightarrow 7\end{array}$$

We can now apply to these, the following transitions from top to down, representing the goto actions after the empty reduces precedently applied:

$$\begin{array}{llllll}\delta(2,C,\varepsilon) & \ni & (3,C,\varepsilon) & \text{to produce} & [3,\varepsilon,S_2^w,S_2^w], & \text{output} \quad \varepsilon \\ \delta(2,D,\varepsilon) & \ni & (4,D,\varepsilon) & \text{to produce} & [4,\varepsilon,S_2^w,S_2^w], & \text{output} \quad \varepsilon \\ \delta(2,E,\varepsilon) & \ni & (5,E,\varepsilon) & \text{to produce} & [5,\varepsilon,S_2^w,S_2^w], & \text{output} \quad \varepsilon\end{array}$$

Table 1: Development for the input $w = abde\dashv$ and the grammar $\mathcal{G}$.

S_0^w		$I_0^0=[0,\varepsilon,S_0^w,S_0^w]$			
S_1^w	$(w_1=a)$	$I_0^1=[0,a,S_0^w,S_1^w]$	S_4^w	$(w_4=e)$	$I_0^4=[6,e,S_3^w,S_4^w]$
S_2^w	$(w_2=b)$	$I_0^2=[1,b,S_1^w,S_2^w]$			$I_1^4=[3,F,S_2^w,S_4^w]$
		$I_1^2=[2,C,S_2^w,S_2^w]$			$I_2^4=[4,F,S_2^w,S_4^w]$
		$I_2^2=[2,D,S_2^w,S_2^w]$			$I_3^4=[5,F,S_2^w,S_4^w]$
		$I_3^2=[2,E,S_2^w,S_2^w]$			$I_4^4=[2,B,S_2^w,S_4^w]$
S_3^w	$(w_3=d)$	$I_0^3=[3,d,S_2^w,S_3^w]$			$I_5^4=[0,A,S_0^w,S_4^w]$
		$I_1^3=[4,d,S_2^w,S_3^w]$	S_5^w	$(w_5=\dashv)$	$I_0^5=[8,\dashv,S_4^w,S_5^w]$
		$I_2^3=[5,d,S_2^w,S_3^w]$			$I_1^5=[9,\Phi,S_0^w,S_5^w]$

From here, there exists an only action to apply to each one of the stack configurations represented by the preceding items, in order to shift the token d. To perform that, we respectively apply the following groups of transitions:

1. The first one is given by:

$$\begin{array}{llllll}\delta(3,\varepsilon,d) & \ni & (3,d,d) & \text{to produce} & I_0^3=[3,d,S_2^w,S_3^w], & \text{output} \quad I_0^3 \rightarrow d \\ \tau_1=\delta(3,d,\varepsilon) & \ni & (6,d,\varepsilon) & \text{to produce} & J_1=[6,\varepsilon,S_3^w,S_3^w], & \text{output} \quad \varepsilon\end{array}$$

2. The second group is composed by:

$$\begin{array}{llllll}\delta(3,\varepsilon,d) & \ni & (3,d,d) & \text{to produce} & I_1^3=[4,d,S_2^w,S_3^w], & \text{output} \quad I_1^3 \rightarrow d \\ \tau_2=\delta(4,d,\varepsilon) & \ni & (6,d,\varepsilon) & \text{to again produce} & J_1, & \text{output} \quad \varepsilon\end{array}$$

3. The last group of transitions is the following:

$$\begin{array}{llll} \delta(3,\varepsilon,d) & \ni\ (3,d,d) & \text{to produce} & I_2^3=[5,d,S_2^w,S_3^w],\ \text{output}\ I_2^3\to d \\ \tau_3=\delta(5,d,\varepsilon) & \ni\ (6,d,\varepsilon) & \text{to again produce} & J_1,\ \text{output}\ \varepsilon \end{array}$$

At this point, we can apply to J_1 the set of transitions representing a shift action on the token e in state 6:

$$\begin{array}{llll} \delta(6,d,e) & \ni\ (6,e,e) & \text{to produce} & I_0^4=[6,e,S_3^w,S_4^w],\ \text{output}\ I_0^4\to e \\ \delta(6,e,\varepsilon) & \ni\ (13,e,\varepsilon) & \text{to produce} & W=[13,\varepsilon,S_4^w,S_4^w],\ \text{output}\ \varepsilon \end{array}$$

In state 13 of figure 2, the only action to perform is a reduce corresponding to rule numbered (8) in the grammar $\mathcal{G}$. To compute it, we firstly pop the symbol e by the transition:

$$\tau_4=\delta(13,\varepsilon,\dashv)\ni(6,\varepsilon,\varepsilon)\text{ to produce }\tau_d=\delta([6,\varepsilon,S_3^w,S_3^w],\dashv)\ni(U,\ U\to I_0^4W)$$

where τ_d is a dynamic transition applicable to the result of transitions $\tau_i,\ i\in\{1,2,3\}$ and $U=[6,\varepsilon,S_3^w,S_4^w]$, which explains the sharing of the node labeled by I_0^4 in figure 4. Continuing with the process, we can apply to the item U the following transitions:

$$\tau_{5,i}=\delta(6,d,\dashv)=\{(i,\varepsilon,\varepsilon)\},\text{ to give}\left\{\begin{array}{l} J_2=[3,\varepsilon,S_2^w,S_4^w],\ i=3,\ \text{rule}\ J_2\to I_0^3U \\ J_3=[4,\varepsilon,S_2^w,S_4^w],\ i=4,\ \text{rule}\ J_3\to I_1^3U \\ J_4=[5,\varepsilon,S_2^w,S_4^w],\ i=5,\ \text{rule}\ J_4\to I_2^3U \end{array}\right.$$

$$\tau_{6,i}=\delta(i,\varepsilon,\dashv)=\{(i,F,F)\},\text{ to give}\left\{\begin{array}{l} I_1^4=[3,F,S_3^w,S_4^w],\ i=3,\ \text{rule}\ I_1^4\to 8J_2 \\ I_2^4=[3,F,S_3^w,S_4^w],\ i=4,\ \text{rule}\ I_2^4\to 8J_3 \\ I_3^4=[3,F,S_3^w,S_4^w],\ i=5,\ \text{rule}\ I_3^4\to 8J_4 \end{array}\right.$$

At this point, we can simplify the output grammar by eliminating those items which do not represent the recognition of a symbol in the grammar $\mathcal{G}$. So, from the preceding rules corresponding to the translation of the reduce of the production numbered (8) in $\mathcal{G}$, and eliminating items of the form $J_i,\ i\in\{2,3,4\}$, U and W, we obtain the new simplified rules:

$$I_1^4\to 8I_0^3I_0^4,\ I_2^4\to 8I_1^3I_0^4,\ I_3^4\to 8I_2^3I_0^4$$

which are the final rules used to represent the final parse forest in figure 4. In the same way, henceforth we shall omit the generation of dynamic transitions during pop actions in order to simplify the exposition.

An important point is that we cannot sharing the nodes labeled by I_0^3, I_1^3 and I_2^3, even they represent the recognition of the same token d. The reason of this relies on the state splitting phenomenon that generates different contexts to analyze the same syntactic category, and it was easily foreseeable from figure 3.

Again, one different transition is applicable to each one of these items. They are respectively:

$$\begin{array}{llll} \delta(3,F,\varepsilon) & \ni\ (10,F,\varepsilon) & \text{to produce} & J_5=[10,\varepsilon,S_4^w,S_4^w],\ \text{output}\ \varepsilon \\ \delta(4,F,\varepsilon) & \ni\ (12,F,\varepsilon) & \text{to produce} & J_6=[12,\varepsilon,S_4^w,S_4^w],\ \text{output}\ \varepsilon \\ \delta(5,F,\varepsilon) & \ni\ (11,F,\varepsilon) & \text{to produce} & J_7=[11,\varepsilon,S_4^w,S_4^w],\ \text{output}\ \varepsilon \end{array}$$

Taking now into account figure 2, the only possible action in states 10, 12 and 11 is respectively the reduction of the rules numbered (2), (3) and (4) in the grammar $\mathcal{G}$. We shall treat them separately:

1. In the case of J_5, we successively apply:

$$\begin{array}{llll} \delta(10,F,\dashv) & \ni\ (3,\varepsilon,\varepsilon) & \text{to produce} & J_8=[3,\varepsilon,S_2^w,S_4^w],\ \text{output}\ J_8\to I_1^4J_5 \\ \delta(3,C,\dashv) & \ni\ (2,\varepsilon,\varepsilon) & \text{to produce} & J_9=[2,\varepsilon,S_2^w,S_4^w],\ \text{output}\ J_9\to I_1^2J_8 \end{array}$$

2. In the case of J_6, we successively apply:

$$\begin{array}{llllll}\delta(12,F,\dashv) & \ni & (4,\varepsilon,\varepsilon) & \text{to produce } J_{10} & = & [4,\varepsilon,S_2^w,S_4^w], \text{ output } J_{10}\rightarrow I_2^4J_6\\ \delta(4,D,\dashv) & \ni & (2,\varepsilon,\varepsilon) & \text{to produce } J_{11} & = & [2,\varepsilon,S_2^w,S_4^w], \text{ output } J_{11}\rightarrow I_2^2J_{10}\end{array}$$

3. In the case of J_7, we successively apply:

$$\begin{array}{llllll}\delta(11,F,\dashv) & \ni & (5,\varepsilon,\varepsilon) & \text{to produce } J_{12} & = & [5,\varepsilon,S_2^w,S_4^w], \text{ output } J_{12}\rightarrow I_3^4J_7\\ \delta(5,E,\dashv) & \ni & (2,\varepsilon,\varepsilon) & \text{to produce } J_{13} & = & [2,\varepsilon,S_2^w,S_4^w], \text{ output } J_{13}\rightarrow I_3^2J_{12}\end{array}$$

We can now close the ambiguities generated by the application of the transition τ_0. In effect, we can apply the same transition $\delta(2,\varepsilon,\dashv)\ni(2,B,B)$ to J_i, $i\in\{9,11,13\}$ to generate the new item

$$I_4^4=[2,B,S_2^w,S_4^w]\in S_4^w \text{ with output} \left\{\begin{array}{l} I_4^4\rightarrow 2J_9\\ I_4^4\rightarrow 2J_{11}\\ I_4^4\rightarrow 2J_{13}\end{array}\right.$$

that we can simplify by eliminating irrelevant items, to obtain:

$$I_4^4\rightarrow 2I_1^2I_1^4,\ I_4^4\rightarrow 2I_2^2I_2^4,\ I_4^4\rightarrow 2I_3^4I_3^4$$

In relation to this, figure 4 shows that I_4^4 is the common top for three different trees representing

Figure 4: AND-OR graph for the grammar $\mathcal{G}$ and the input $w = abde\dashv$.

ambiguities in the final parse forest. To continue with the process, we apply to I_4^4 the transition $\delta(2,B,\dashv)\ni(7,B,\varepsilon)$ and we generate the item $J_{14}=[7,B,S_4^w,S_4^w]$. From here, the only action indicated in state 7 is given by the reduction of the rule numbered (1) in the grammar $\mathcal{G}$. To do it, we successively consider the transitions:

$$\begin{array}{llllll}\delta(7,B,\dashv) & \ni & (2,\varepsilon,\varepsilon) & \text{to produce } J_{15} & = & [2,\varepsilon,S_2^w,S_4^w], \text{ output } J_{15}\rightarrow I_4^4J_{14}\\ \delta(2,b,\dashv) & \ni & (1,\varepsilon,\varepsilon) & \text{to produce } J_{16} & = & [1,\varepsilon,S_1^w,S_4^w], \text{ output } J_{16}\rightarrow I_0^2J_{15}\\ \delta(1,a,\dashv) & \ni & (0,\varepsilon,\varepsilon) & \text{to produce } J_{17} & = & [0,\varepsilon,S_0^w,S_4^w], \text{ output } J_{17}\rightarrow I_0^1J_{16}\\ \delta(0,\varepsilon,\dashv) & \ni & (0,A,A) & \text{to produce } I_5^4 & = & [1,A,S_1^w,S_4^w], \text{ output } I_5^4\rightarrow 1J_{17}\\ \delta(0,A,\dashv) & \ni & (8,A,\varepsilon) & \text{to produce } J_{18} & = & [8,\varepsilon,S_4^w,S_4^w], \text{ output } \varepsilon\end{array}$$

where the last transition is a push one and it does not generate an output structure. As in preceding cases, we can simplify the output rules to obtain:

$$I_5^4\rightarrow 1I_0^1I_0^2I_4^4$$

Now, the the only action in state 8 of figure 2 is a shift on the symbol represented by $\dashv$. To do it, we apply successively:

$$\begin{array}{lllllll}\delta(8,\varepsilon,\dashv) & \ni & (8,\dashv,\dashv) & \text{to produce} & I_0^5 & = & [8,\dashv,S_4^w,S_5^w], \text{ output } I_0^5 \rightarrow \dashv \\ \delta(8,\dashv,\varepsilon) & \ni & (9,\dashv,\varepsilon) & \text{to produce} & J_{19} & = & [9,\varepsilon,S_5^w,S_5^w], \text{ output } \varepsilon\end{array}$$

And we can finally reduce the axiom of the grammar $\mathcal{G}$, which corresponds to the rule numbered (0). To perform it, we successively apply:

$$\begin{array}{lllllllll}\delta(9,\dashv,\varepsilon) & \ni & (8,\varepsilon,\varepsilon) & \text{to produce} & J_{20} & = & [8,\varepsilon,S_4^w,S_5^w], & \text{output } J_{20} & \rightarrow I_0^5 J_{19} \\ \delta(8,A,\varepsilon) & \ni & (0,\varepsilon,\varepsilon) & \text{to produce} & J_{21} & = & [0,\varepsilon,S_0^w,S_5^w], & \text{output } J_{21} & \rightarrow I_5^4 J_{20} \\ \delta(0,\varepsilon,\varepsilon) & \ni & (0,\Phi,\varepsilon) & \text{to produce} & I_1^5 & = & [0,\Phi,S_0^w,S_5^w], & \text{output } I_1^5 & \rightarrow 0J_{21}\end{array}$$

that we can simplify to obtain

$$I_1^5 \rightarrow 0 I_5^4 I_0^5$$