Generation of Incremental Parsers

M. Vilares, M.A. Alonso, and V.M. Darriba

Department of Informatics, Campus As Lagoas s/n, 32004 Orense, Spain
{vilares,darriba}@uvigo.es alonso@udc.es
Url: http://www.grupocole.org

Abstract. An incremental development environment for unrestricted context-free languages is described and tested. Our proposal includes a parse generator, an incremental facility to make the overall parsing efficient in the context of program development; and a graphical interface that provides a complete set of customization and trace facilities. The tool, baptized ICE after Incremental Context-Free Environment, appears to be superior to other general context-free parsing environments and is comparable to deterministic ones, when the context is not ambiguous.

1 Introduction

Programming requires certain characteristics making it both friendly and efficient. This implies providing the user with an interface in order to favor incremental program development in a context where several consecutive corrections of the input are usually made. After each editing operation on an input, its implementation should also be updated efficiently, which means that preparing a program requires significantly less effort than developing it from scratch. We are interested in language design applications, where modifications of the grammatical structure are frequent. In this context, parser generation is inspired by BISON [2], which we have extended in order to deal with general context-free grammars (CFGs). Parsing is stated in the context of parallel methods, a variation of Earley's construction [3] proposed by Lang [4] that separates the execution strategy from the implementation of the push-down automaton (PDA).

Finally, incremental parsing within general context-free parsing has been addressed by van den Brand [9] and Rekers [5]. Both authors take the variable covering the modification as a parameter to which the text that is to be parsed should be reduced, to prevent the system from doing unnecessary work during the search for this minimal node, for example, when the input contains an error. Instead, we update runs in parallel to the parsing [10], which ensures the earlier detection of errors, thus avoiding any unnecessary work.

2 Parser generation

Parser construction is an extension from BISON [2] in order to permit the generation of extended LALR(1) PDAs. More explicitly, the generation of tables has been

re-implemented in order to deal with both incremental and non-deterministic parsing. We direct our attention to constraining the space bounds for the generation process, which involves to the consideration of default actions as well as array compacting methods in the PDA.

The language representing all possible elementary actions in the PDA allows us to decompose complex actions in terms of simple push and pop transitions, which constitutes the basis for introducing dynamic programming in the parsing process. In order to take care of the trace of pop transitions in reduce actions, possibly in the context of non-deterministic interpretation, the system also provides the facility to go back over the schema in the automata. What follows is a short description of tables and functionalities:

- *yytranslate*: vector mapping *yylex's* tokens into user's token numbers. The token translation table is indexed by a token number as returned by the user's *yylex* routine. It yields the internal token number used by the parser.
- *yyr1[r]*: symbol that rule r derives.
- *yyr2[r]*: number of symbols composing the right hand side of rule r.
- *yydefact[s]*: default rule to reduce within state s, when *yytable* does not specify something else to do.
- *yydefgoto[i]*: default state to go to after a reduction that generates variable YYNTBASE $+ i$, except when *yytable* specifies something else to do.
- *yypact[s]*: index in *yytable* of the portion describing state s. The lookahead token type is used to index that portion to find out what to do. If the value in *yytable* is positive, it indexes the corresponding elementary action on *yyautomaton*. If the value is zero, the default action from *yydefact[s]* is used. We can avoid the access to *yytable* in the following cases:
  - If the only action in state s is the default one. This case can be detected in three different forms without accessing to *yytable*:
    - When *yypact[s]* = YYFLAG.
    - When *yypact[s]* + *lookahead* < 0.
    - When *yypact[s]* + *lookahead* $>$ YYLAST.
  - If the current action is the default one, we can also avoid the access to *yytable* when $yycheck[yypact[s] + lookahead] \neq lookahead$, as will be explained below.
- *yypgoto[i]*: the index in *yytable* of the portion describing what to do after reducing a rule that derives variable YYNTBASE $+ i$. This portion is indexed by the parser state number as if the text for this non-terminal had been previously read. The value from *yytable* is the state to go to. We can avoid the access to *yytable* when the action to apply is the default one:
  - When *yypgoto[i]* + *state* ≥ 0.
  - When *yypgoto[i]* + *state* $\leq$ YYLAST.
  - When *yycheck[yypact[s]* + *state]* $\neq$ *state*.
- *yytable*: vector with portions for different uses, found via *yypact* and *yypgoto*.

- *yycheck*: vector indexed in parallel with *yytable*. It indicates, in a roundabout way, the bounds of the portion you are trying to examine. Suppose that the portion of *yytable* starts at index p and the index to be examined within the portion is i. Then if *yycheck[p+i]* $\neq i$, i is outside the bounds of what is actually allocated, and the default from *yydefact* or *yydefgoto* should be used. Otherwise, *yytable[p+i]* should be used.
- *yyautomaton*: vector containing the descriptors for actions in the automaton: block $\rightarrow 0$, halt $\rightarrow 1$, non-determinism $\rightarrow 2$, reduce $\rightarrow 3$, shift $\rightarrow 4$.
- *yystos[s]*: the accessing symbol for the state s. In other words, the symbol that represents the last thing accepted to reach that state.
- *yyreveal_map[s]*: index in *yyreveal* of the portion that contains all the states having a transition over the state s.
- *yyreveal*: vector indexed by *yyreveal_map* that groups together all the states with a common transition.

where we have considered the following set of constants:

- YYFINAL*:* the termination state. The only state where a *halt* is possible.
- YYFLAG*:* the most negative short integer. Used to flag in *yypact*.
- YYLAST*:* the final state, whose accessing symbol is the end of input. It has a only one transition, over YYFINAL. So, we obey the parser's strategy of making all decisions one token ahead of its actions.
- YYNTBASE*:* the total number of tokens, including the end of input.

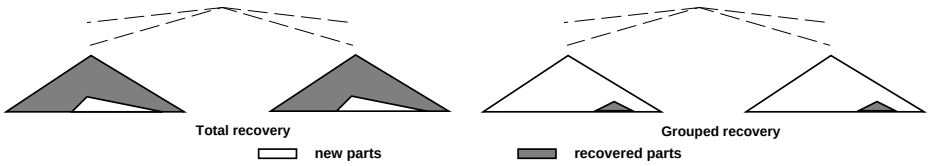

Fig. 1. Practical incremental recovery

3 Standard parsing

Our aim is to parse a sentence $w_{1\ldots n} = w_1 \ldots w_n$ of length n, according to a CFG $\mathcal{G} = (N, \Sigma, P, S)$, 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. We generate from $\mathcal{G}$ a PDA having as finite-state control a LALR(1) automaton built as indicated in Sect. 2. The direct execution of PDAs may be exponential with respect to the length of the input string and may even loop. To get polynomial complexity, we must avoid duplicating stack contents when several transitions may be applied to a given configuration. Instead of storing all

the information about a configuration, we must determine the information we need to trace in order to retrieve that configuration. This information is stored into a table $\mathcal{I}$ of *items*:

$$\mathcal{I} = \{\, [st, X, i, j],\ st \in \mathcal{S},\ X \in N \cup \Sigma \cup \{\nabla_{r,s}\},\ 0 \leq i \leq j \,\}$$

where $\mathcal{S}$ is the set of states in the LALR(1) automaton. Each configuration of the PDA is represented by an item storing the current state st, the element X placed on the top of the stack and the positions i and j indicating the substring $w_{i+1} \ldots w_j$ spanned by X. The symbol $\nabla_{r,s}$ indicates that the final part $A_{r,s+1} \ldots A_{r,n_r}$ of a context-free rule $A_{r,0} \rightarrow A_{r,1} \ldots A_{r,n_r}$ has been recognized.

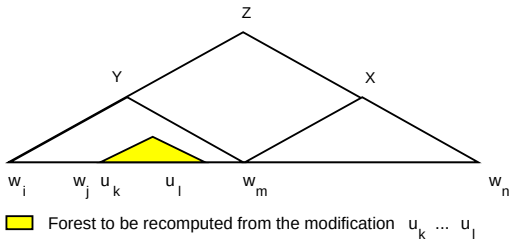

Fig. 2. A pop transition $XY \mapsto Z$ totally recovering a modification

We describe the parser using Parsing Schemata, a framework for high-level description of parsing algorithms [7]. A *parsing scheme* is a triple $\langle \mathcal{I}, \mathcal{H}, \mathcal{D} \rangle$, with $\mathcal{I}$ a set of items, $\mathcal{H} = \{[a, i, i+1],\ a = w_i\}$ an initial set of special items called *hypothesis* that encodes the sentence to be parsed[1], and $\mathcal{D}$ a set of *deduction steps* that allow new items to be derived from already known items. Deduction steps are of the form $\{\eta_1, \ldots, \eta_k \vdash \xi \mid conds\}$, meaning that if all antecedents η_i of a deduction step are present and the conditions *conds* are satisfied, then the consequent ξ should be generated by the parser[2]. In the case of the parsing algorithm we propose, the set of deduction steps is the following one

$$\mathcal{D} = \mathcal{D}^{\text{Init}} \cup \mathcal{D}^{\text{Shift}} \cup \mathcal{D}^{\text{Sel}} \cup \mathcal{D}^{\text{Red}} \cup \mathcal{D}^{\text{Head}}$$

where

$$\mathcal{D}^{\text{Init}} = \{\, \vdash [st_0, -, 0, 0] \,\}$$

$$\mathcal{D}^{\text{Shift}} = \{ [q, X, i, j] \vdash [q', a, j, j+1] \,/\, \begin{array}{l} \exists\, [a, j, j+1] \in \mathcal{H} \\ shift_{q'} \in action(q, a) \end{array} \}$$

[1] The empty string, ε, is represented by the empty set of hypothesis, $\emptyset$. An input string $w_{1\ldots n}$, $n \geq 1$ is represented by $\{[w_1, 0, 1],\ [w_2, 1, 2],\ \ldots\ ,\ [w_n, n-1, n]\}$.

[2] Parsing schemata are closely related to *grammatical deduction systems* [6], where items are called *formula schemata*, deduction steps are *inference rules*, hypothesis are *axioms* and final items are *goal formulas*.

$$\mathcal{D}^{\text{Sel}} = \{[st, X, i, j] \vdash [st, \nabla_{r,n_r}, j, j] \Big/ \begin{matrix} \exists\, [a, j, j+1] \in \mathcal{H} \\ \mathit{reduce}_r \in \mathit{action}(st, a) \end{matrix} \}$$

$$\mathcal{D}^{\text{Red}} = \{[st, \nabla_{r,s}, k, j][st, X_{r,s}, i, k] \vdash [st', \nabla_{r,s-1}, i, j], st' \in \mathit{reveal}(st)\}$$

$$\mathcal{D}^{\text{Head}} = \{\, [st, \nabla_{r,0}, i, j] \vdash [st', A_{r,0}, i, j], st' \in \mathit{goto}(st, A_{r,0}) \,\}$$

with $X \in N \cup \Sigma$, st referring to the initial state and *action*, *goto* and *reveal* referring to the tables that encode the behavior of the LALR(1) automaton:

- The action table determines what action should be taken for a given state and lookahead. In the case of shift actions, it determines the resulting new state and in the case of reduce actions, the rule which is to be applied for the reduction.
- The goto table determines what the state will be after performing a reduce action. Each entry is accessed using the current state and the non-terminal, which is the left-hand side of the rule to be applied for reduction.
- The reveal table is used to traverse the finite state control of the automaton backwards: $st^i \in \mathit{reveal}(st^{i+1})$ is equivalent to $st^{i+1} \in \mathit{goto}(st^i, X)$ if $X \in N$, and is equivalent to $\mathit{shift}_{st^{i+1}} \in \mathit{action}(st^i, X)$ if $X \in \Sigma$.

As is shown in [1], this set of deduction steps is equivalent to the dynamic interpretation of non-deterministic PDAs:

- A deduction step *Init* is in charge of starting the parsing process.
- A deduction step *Shift* corresponds to pushing a terminal a onto the top of the stack when the action to be performed is a shift to state st'.
- A step *Sel* corresponds to pushing the ∇_{r,n_r} symbol onto the top of the stack in order to start the reduction of a rule r.
- The reduction of a rule of length $n_r > 0$ is performed by a set of n_r steps *Red*, each of them corresponding to a pop transition replacing the two elements $\nabla_{r,s}\, X_{r,s}$ placed on the top of the stack by the element $\nabla_{r,s-1}$.
- The reduction of a rule r is finished by a step *Head* corresponding to a swap transition that replaces the top element $\nabla_{r,0}$ by the left-hand side $A_{r,0}$ of that rule and performs the corresponding change of state.

Deduction steps are applied until new items cannot be generated. The splitting of reductions into a set of *Red* steps allow us to share computations corresponding to partial reductions of rules, attaining a worst case time complexity $\mathcal{O}(n^3)$ and a worst case space complexity $\mathcal{O}(n^2)$ with respect to the length n of the input string. The input string has been successfully recognized if the final item $[st_f, S, 0, n]$, with st_f final state of the PDA, has been generated.

Following [4], we represent the shared parse forest corresponding to the input string by means of an output grammar $\mathcal{G}_o = (N_o, \Sigma_o, P_o, S_o)$, where N_o is the set of all items, Σ_o is the set of terminals in the input string, the start symbol S_o corresponds to the final item generated by the parser, and a rule in P_o is generated each time a deduction step is applied:

- For Shift, a production $[st', a, j, j+1] \to a$ is generated.
- For Sel, a production $[st, \nabla_{r,n_r}, j, j] \to \varepsilon$ is generated .
- For Red, a production $[st', \nabla_{r,s-1}, i, j] \to [st, \nabla_{r,s}, k, j]\,[st, A_{r,s}, i, k]$ is generated.
- For Head, a production $[st', A_{r,0}, i, j] \to [st, \nabla_{r,0}, i, j]$ is generated.

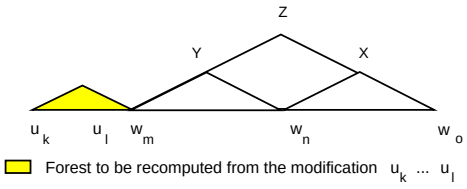

Fig. 3. A pop transition $XY \mapsto Z$ independent of the modification

4 Incremental parsing

Incremental parsing has been attempted in two senses: firstly, as an extension of left-to-right editing, and secondly, in relation with the full editing capability on the input string. We are interested in the latter, called full incrementality, in the domain of general CFGs, without editing restrictions, guaranteeing the same level of sharing as in standard mode, but without any impact. In practice we have focused on two cases, shown in Fig. 1:

- *Total recovery*, when recovery is possible on all the syntactic context once the modification has been parsed.
- *Grouped recovery*, when recovery is possible for all branches on an interval of the input string to be re-parsed.

which allows us to increase the computational efficiency by avoiding the recovery of isolated trees in a forest corresponding to an ambiguous node. We consider a simplified text-editing scenario with a single modification, in order to favor understanding. Let's take a modified input string from a previously parsed initial one. We must update the altered portion of the original shared forest. To do so, it is sufficient to find a condition capable of ensuring that all possible transitions to be applied from a given position in an interval in the input string are independent from the introduced modification. We focus our attention on those transitions dependent on the past of the parsing, that is, on pop transitions. If the portion of the input to be parsed is the same, and the parts of the past to be used in this piece of the process are also the same, the parsing will be also the same in this portion. That corresponds to different scopes in this common past: when this extends to the totality of the structures to be used in the remaining parsing

process, we have total recovery, as is shown in Fig. 2. If it only extends to a region after the modification, we have grouped recovery, as is shown in Fig. 3.

To ensure that pop transitions are common between two consecutive parses, in an interval of the unchanged input string, we focus on the set of items which are arguments of potential future pops. This is the case of items resulting from non-empty reductions before a shift. These items can be located in a simple fashion, which guarantees a low impact in standard parsing.

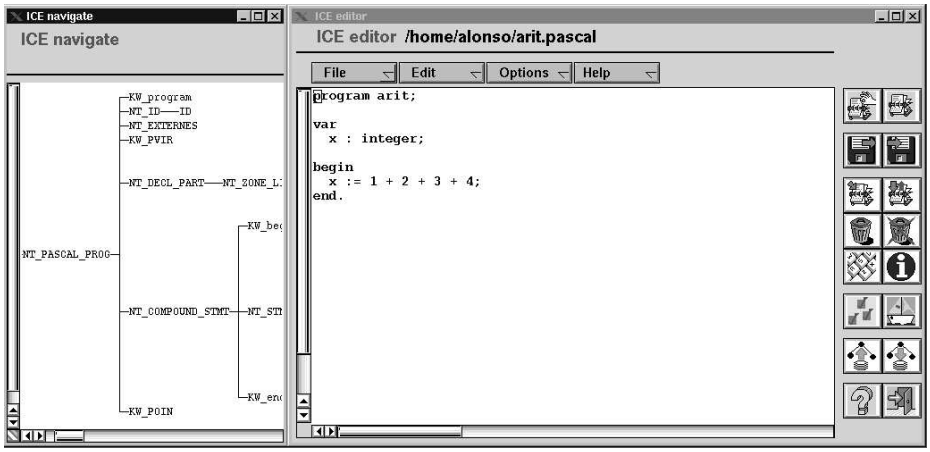

Fig. 4. Analyzing a program

To now find a condition ensuring that all pop transitions from any given transition, taking one of these items as argument, are common in an interval, we use the notion of the *back pointer* i of items $[st, X, i, j]$. When corresponding items between consecutive parses have equivalent back pointers, incremental recovery is possible. Back pointers are equivalent iff they point to a position of the input string not modified since the previous parsing, or when they are associated to a position corresponding to a token belonging to a modified part of the input string. In the first case, we can ensure total recovery since both parses have returned to a common past. In the second one, we can only ensure grouped recovery since common computations are only possible while there are no pop transitions returning on the scope of the modification, which limits the extension of the interval to be recovered.

5 User interface

The tool helps the language designer in the task of writing grammars, with a dedicated editor. At any moment the user can request a parse of the grammar, which is done according to the parsing scheme chosen in advance, from an input file written in a BISON-*like* format. A view of the interface is shown in Fig. 4.

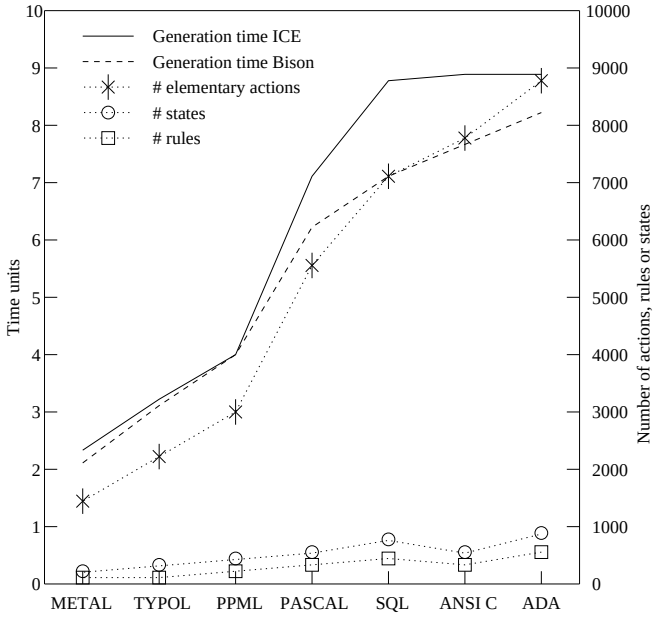

Fig. 5. Results on parser generation

The interface for the programming environment allows the user to choose the parsing mode, standard or incremental, and load a language generated in advance. A set of options allows the user to choose the class of information reported: conflicts that have been detected, statistics about the amount of work generated and so on. Debugging facilities also incorporate information about the recovery process during incremental parsing, and errors are always reported. The interface allows parse forests to be recovered and manipulated. We can also select the language in which the system interacts with us: English, French and Spanish are currently available. A help facility is always available to solve questions about the editors and the incremental facilities.

6 Experimental results

We have compared Ice with Bison [2], Glr [5] and Sdf [8], which are to the best of our knowledge some of the most efficient parsing environments, from two different points of view: parser generation and parsing process. We also show the efficiency of incremental parsing in relation to the standard one, and the capability of Ice to share computations. In order to provide a classical point of reference, we have also included the original Earley's parsing scheme [3] in Ice. All the measurements have been performed using generic time units.

In relation to parser generation, we took several known programming languages and extracted the time used to generate parser tables, comparing Bison

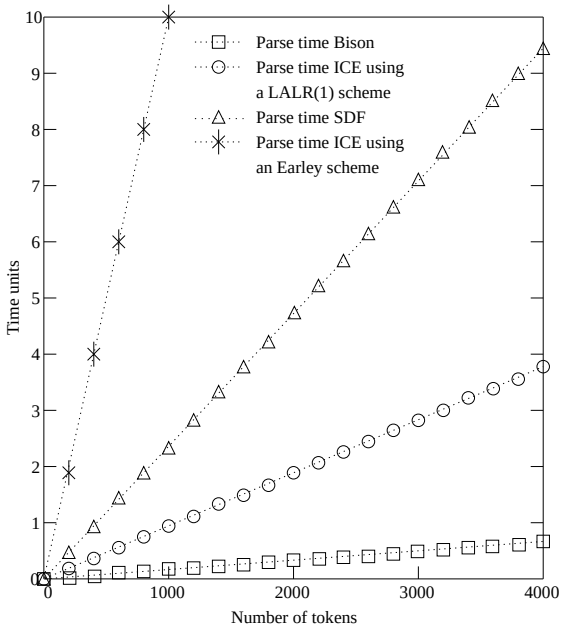

Fig. 6. Results on deterministic parsing

with the generation of LALR(1) schema in ICE[3]. Results are given in relation to different criteria. So, Fig. 5 shows these according to the number of rules in the grammar, and to the number of states associated with the finite state machine generated from them[4]. At this point, it is important to note that the behavior of ANSI-C does not seem to correspond to the rest of the programming languages considered in the same test. In effect, the number of rules in the grammar, and the number of states in the resulting PDA may not be in direct relation with the total amount of work necessary to build it. In order to explain this, we introduce the concept of *elementary building action* as an action representing one of the following two situations: the introduction of items in the base or in the closure of a state in the PDA, and the generation of transitions between two states.

We use the syntax of complete PASCAL as a guideline for parsing tests. In Fig. 6 comparisons are established between parsers generated by ICE[5], BISON and SDF, when the context is deterministic. We consider ICE, SDF and GLR when the context is non-deterministic, as is shown in Fig. 7. We have considered two versions for PASCAL: deterministic and non-deterministic, the latter including ambiguity for arithmetic expressions. Given that in the case of ICE, SDF and GLR mapping between concrete and abstract syntax is fixed, we have generated in the case of BISON, a simple recognizer. To reduce the impact of lexical time,

[3] Earley's algorithm is a grammar oriented method.
[4] BISON and ICE generate LALR(1) machines, SDF LR(0) ones.
[5] Using both, LALR(1) and Earley's schema.

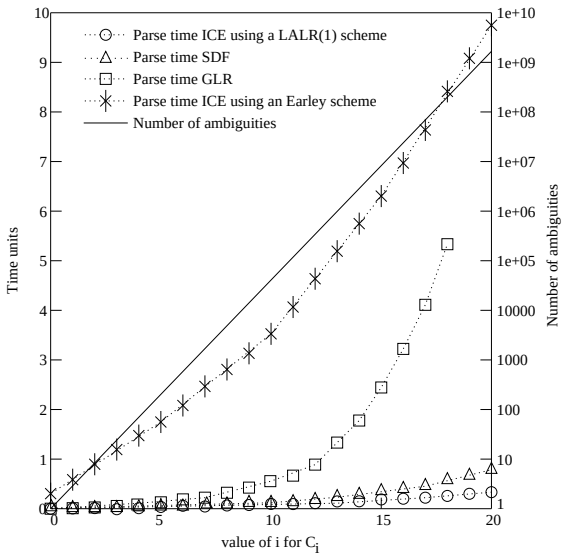

Fig. 7. Results on non-deterministic parsing

we have considered, in the case of non-deterministic parsing, programs of the form:

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

where i is the number of +'s. The grammar contains a rule `Expr ::= Expr + Expr`, therefore these programs have a number of ambiguous parses which grows exponentially with i. This number is:

$$C_0 = C_1 = 1 \quad \text{and} \quad C_i = \binom{2i}{i} \frac{1}{i+1}, \text{ if } i > 1$$

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. To illustrate incrementality, we analyze the previous programs in which we substitute expressions $b + b$ by b. Results corresponding to incremental and standard parsing are shown in Fig. 8, and those related to sharing in Fig. 9.

7 Conclusions

The ICE system is devoted to simultaneous editing of language definitions and programs, in the domain of unrestricted context-free languages. The modular composition includes a parser generator, standard and incremental parse interpretation and a graphic interface, where customizations can be carried out

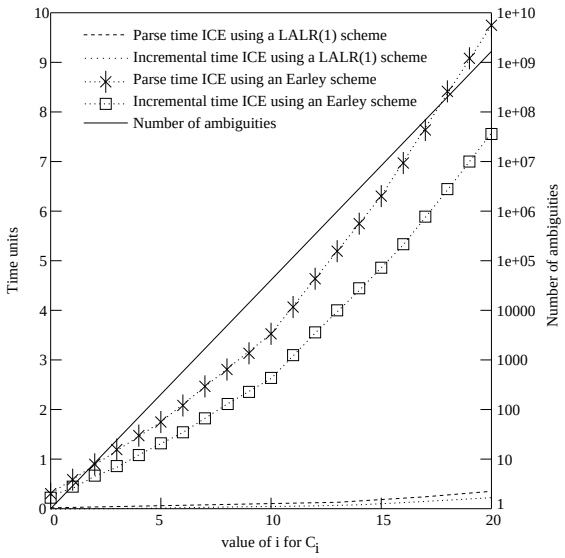

Fig. 8. Results on incremental and standard parsing using ICE

either interactively, or through an initialization file. In a practical comparison, our algorithm seems to surpass previous proposals.

Although efficient incremental parsing may have seemed a difficult problem, we were able to keep the complexity of the algorithm low. So, practical tests have proved the validity of the approach proposed when the number of ambiguities remains reasonable, as is the case in practice. In addition, ICE is compatible with the standard parser generators in UNIX, which permits a free use of all the input that has been developed for these generators.

8 Acknowledgments

This work has been partially supported by the European Union, Government of Spain and Autonomous Government of Galicia under projects 1FD97-0047-C04-02, TIC2000-0370-C02-01 and PGIDT99XI10502B, respectively.

References

1. M.A. Alonso, D. Cabrero, and M. Vilares. Construction of efficient generalized LR parsers. In Derick Wood and Sheng Yu, editors, *Automata Implementation*, volume 1436 of *Lecture Notes in Computer Science*, pages 7–24. Springer-Verlag, Berlin-Heidelberg-New York, 1998.
2. Charles Donnelly and Richard Stallman. *Bison: the YACC-compatible Parser Generator, Bison Version 1.28.* Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, USA, Tel: (617) 876-3296, USA, January 1999.

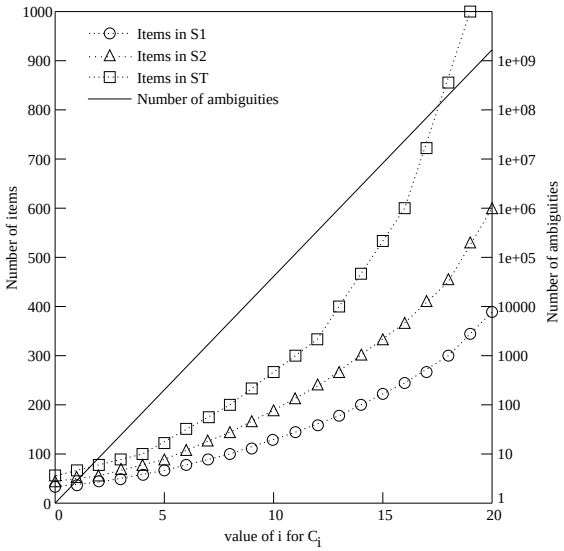

Fig. 9. Items generated using S^1, S^2 and S^T schema.

3. J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.
4. Bernard Lang. Deterministic techniques for efficient non-deterministic parsers. In J. Loeckx, editor, *Automata, Languages and Programming*, number 14 in Lecture Notes in Computer Science, pages 255–269. Springer, Berlin, DE, 1974.
5. J. Rekers. *Parser Generation for Interactive Environments.* PhD thesis, University of Amsterdam, Amsterdam, The Netherlands, 1992.
6. S.M. Shieber, Y. Schabes, and F.C.N. Pereira. Principles and implementation of deductive parsing. *Journal of Logic Programming*, 1-2:3–36, 1995.
7. K. Sikkel. *Parsing Schemata.* PhD thesis, Univ. of Twente, The Netherlands, 1993.
8. Mark van den Brand, Paul Klint, and Pieter A. Olivier. Compilation and memory management for ASF+SDF. In *93*, page 15. Centrum voor Wiskunde en Informatica (CWI), ISSN 1386-369X, February 28 1999. SEN (Software Engineering (SEN)).
9. M.G.J. van den Brand. *A Generator for Incremental Programming Environments.* PhD thesis, Katholieke Universiteit Nijmegen, Nijmegen, The Netherlands, 1992.
10. M. Vilares. *Efficient Incremental Parsing for Context-Free Languages.* PhD thesis, University of Nice. ISBN 2-7261-0768-0, France, 1992.