Friendly Incremental Prototyping

Manuel Vilares Ferro[1] Miguel Angel Alonso Pardo[2]

[1]Departamento de Ciencias de la Computación e Inteligencia Artificial, Universidad de A Coruña, Campus de Elviña S/N, 15071 A Coruña, Spain. E-mail: vilares@dc.fi.udc.es.
[2]Instituto de Investigaciones Lingüísticas y Literarias Ramón Piñeiro, Road Santiago-Noya, Km. 3, A Barcia, 15896 Santiago de Compostela, Spain. E-mail: alonso@dc.fi.udc.es.

Abstract. A development environment for interactive systems devoted to generate formal languages is described. Our system is organized around three cooperative modules. The first is a generator of general context-free parsers, following different parsing schemes among several available. The second is a generic incremental parsing facility that can be used to make the overall parsing process efficient in the context of program development. The third is an extensible user graphical interface that provides a complete set of customization and trace facilities for the system.

The final tool has been baptized Ice, after **I**ncremental **C**ontext-Free **E**nvironment. All components in Ice show a reasonable grade of efficiency both, in space and time. In an empirical comparison, it appears to be superior to other general context-free parsing environments and is comparable to the classic deterministic ones, when the context is not ambiguous. The cooperative architecture allows the modification of the environment, by adding or redesigning modules, with low impact on other components.

Key Words & Phrases: Incremental Parsing, Interactive Programming, Parser Generation, Prototyping, Reuse of Components.

Note: This work was partially supported by the **E**ureka **S**oftware **F**actory project, and by the Autonomous Government of Galicia under project XUGA10501A93.

1 Introduction

Programming requires certain characteristics making it both friendly and efficient, including the possibility to describe which are the components of the user interface or simply alter the appearance of these. However, a simple interface does not ensure an efficient treatment of the information. In effect, the environment should also provide the user with the possibility to create and test new programs, modifying them easily and efficiently. So, for example, programming environments should offer high level interactive facilities to favor incremental program development in a context where several consecutive corrections of the input are usually made. To do this, after each editing operation on an input, its implementation should also be updated efficiently. In this manner, both the scanning and the parsing process should be incremental, which means that preparing a program requires significantly less effort than developing it from scratch.

In fact, we are interested in those environments capable of intergrating the language design capability, which allows us to obtain a very interesting feature: We can test a language in the same framework we have used to implement it. In effect, during the language design process, modifications of the grammar are frequent and this approach seems to be the most advised to validate prototypes. On the other hand, language design is a particular case of programming and it does not make sense to consider it separately.

Proceedings of ASF+SDF95. A workshop on Generating Tools from Algebraic Specifications. May 11 & 12, 1995, CWI, Amsterdam, M.G.J. van den Brand, A. van Deursen, T.B. Dinesh, J.F.Th. Kamperman & E. Visser (eds.) Technical Report P9504, Programming Research Group, University of Amsterdam

1.1 Previous work

In order to provide flexibility for the parsing process, the problem is stated in the context of parallel methods, also called *Earley-like* algorithms. We consider a simple variation of Earley's dynamic programming construction [1] proposed by Lang in [2], where in order to solve the problems derived from grammatical constraints, the author extends it to push-down transducers, separating the execution strategy from the implementation of the push-down automaton interpreter. So, Lang obtains a family of general context-free parallel parsers for each family of deterministic context-free push-down ones, simulating all possible computations of any push-down transducer, in the worst of cases in cubic time[1]. Recently published variants of Earley's algorithm may be viewed as this construction applied to some specific model of push-down transducer. This is the explicit strategy of Tomita [3] in the special case of LR(0) parsers, which was later retaken to implement some of the most efficient general context-free parsing environments such as SDF [4] or GLR [5]. We have applied the Lang's construction to implement the extended LALR(1) parsing algorithm serving as a kernel for ICE.

To increase the efficiency in a context where the number of possible parse trees may become very large when the size of sentences increases, parse trees should be merged as much as possible into a single structure that allows them to share common parts. This sharing saves on the space needed to represent trees, and also on the later processing of these since it may allow the sharing among the different trees in the process of some common parts, which is of importance in practical systems because that impacts the performances. A lot of research in this domain was developed by Villemonte de la Clergerie in [6] for constructing efficient and complete definite clause programs compilers. Although this work is not directly related to the problem of parsing, the techniques described on it can be easily adapted to give an adequate treatment to the posed problem, as it is shown in [7].

In relation to incremental parsing, to the best of our knowledge, the problem has not previously been addressed within general context-free parsing except for van den Brand in [8], J. Rekers in [5] and the first author of this paper in [9], even if the approach is different. In effect, the incremental parsing routines used in [8] and [5] take the non-terminal as a parameter to which the text that is to be parsed should be reduced. Although in most cases the focused node in the original program will cover the alterations, for a few of them the reparsing will have to start in an ancestor node of the focus, which is found by a sequence of trial and error. At this point, it is not possible to prevent the system from doing unnecessary work during the search for this minimal node covering the complete syntactical effect of the modification, for example, when the text to be parsed contains an error. In the case of [9], the update of the parse forest is run parallel to the parsing process itself which ensures the earlier detection of parse errors, avoiding unnecessary work. To be more exact, when the parsing process for the introduced modification begins, the system has previously verified if this is viable in relation to the current syntactic context. This work may be easily extended to Earley's classic algorithm, which leads us to conjecture that the technique described is at the heart of incremental parsing constructions in dynamic programming.

1.2 A simple road map

In section 2 of this paper, we give an informal overview of parser generation, justifying the choice of the different parsing schemes proposed. In section 3, we introduce the most relevant elements in order to improve the performance of standard parsing, focusing our attention on LALR(1) extended parsers. In

[1] The method is linear in a large class of grammars, including all programming languages.

section 4, we describe the essential features of the incremental parsing algorithm included in the ICE system, justifying tactical decisions from a practical point of view and differentiating two cases: *Total* and *grouped recovery*, with respect to the nature of the modified shared parse forest. Section 5 shows the programming environment and an overview of the system at work. In section 6, we give an extensive range of comparative tests between ICE and the best deterministic and non-deterministic context-free parsing environments. Section 7 is a conclusion about the work presented.

2 Parser generation

Parser generation [10] is inspired by BISON [11], which we have extended in order to deal with general context-free grammars. As a consequence, we can compare ICE with the standard parser generators in UNIX using an uniform framework, which represents a valid point of reference for the quality of the tool.

Our system can generate parsers following several parsing schemes with a generic incremental facility, including Earley's method [1], and deterministic and non-deterministic LALR(1) parsing schemes. The description formalism used by ICE is actually a subset of the one used by BISON.

Earley's algorithm is a general context-free grammar oriented parsing method. It adapts easily to changes in the grammar, but it is very inefficient because for each parsing step all the information must be recomputed from the grammar. In spite of its lack of efficiency, Earley's method was extended for spoken sentence recognition [12], and logic formalisms [13]. This simple and effective algorithm has founded a school in the domain of general context-free parsing [2, 3] and related formalisms [14], and is at the heart of parallel parsing. At this point, the algorithm has both an experimental interest and also an interest as point of reference, that justifies its inclusion in ICE.

The use of LALR(k), $k \geq 1$ parsing techniques requires the generation of a parse table in advance. The time used by the resulting parser is linear and does not depend on the used look-ahead k. The only difference between deterministic and non-deterministic parse tables is given by the number of actions to be considered for each state and look-ahead. When the look-ahead is increased the class of recognizable languages becomes larger, but the parser generation time increases exponentially because of the state splitting phenomenon. For extended LALR(k) parsers, state splitting phenomena not only involves an increased parser generation time, but also the multiplication of parsing schemes preventing the sharing of locally identical subcomputations because of differences in syntactic context analysis. In this sense, from the point of view of the sharing quality, pure bottom-up techniques as simple precedence are more efficient since it only takes grammatical features into account. The other side of the coin is represented by its more restrictive deterministic domain, which we can translate into an inefficient treatment of the local determinism phenomenon[2]. At this point, the problem is a practical one and experimenting is the best basis to decide upon it. So, we have concluded [15, 16] that techniques close to straightforward bottom-up methods, as LALR(1), are often the most appropriate. That justifies the consideration of such a parsing scheme in ICE.

[2] Ambiguities have usually a local behavior since the user often write grammars sufficiently close to deterministic ones, and during most of the time deterministic parsing would be possible.

3 Standard parsing

We assume that by using a standard technique we produce a recognizer for the context-free language $\mathcal{L}$, based on any of the parsing schemes available on Ice. Our aim is to parse sentences in this language according to its syntax.

3.1 The descriptive model

There is an apparent major difference with other parsers in the kind of structure we use to represent the output shared forest, by using context-free grammars. When the sentence has several distinct parses, the set of all possible parse chains is represented in finite shared form by a context-free grammar that generates that possible infinite set, such as was proved by Lang in [2].

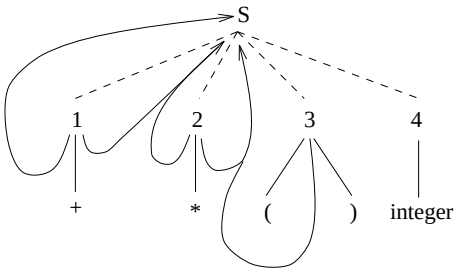

Figure 1: AND-OR graph for ambiguous arithmetic expressions

This difference is only apparent since the formalism habitually used to represent parse forest, AND-OR graphs with different labels for each node, may be translated into a context-free grammar such that AND-node labels are rule names, OR-node labels represent non-terminal categories, and leaf-node labels are terminals.

Conversely, context-free grammars can be represented by AND-OR graphs. To be more exact, OR-nodes are represented by the non-terminal categories, and AND-nodes are represented by the rules of the grammars. Leaf-nodes are terminal categories. The OR-node corresponding to a non-terminal X has exiting arcs leading to each AND-node n representing a rule that defines X. If there is only this arc, then it is represented by placing n immediately under X. The sons of an AND-node are the grammatical categories found in the right-hand-side of the rule, in that order. The convention for orienting the arcs is that they leave a node from below and reach a node from above. As example, Fig. 1 shows a representation for the grammar of ambiguous arithmetic expressions by using an AND-OR graph. The set of rules considered is the following:

$$(0)\quad S \to S\ +\ S \qquad (1)\quad S \to S\ *\ S \qquad (2)\quad S \to (\ S\ ) \qquad (3)\quad S \to \mathit{integer}$$

In relation to other representations, this one proposed by Lang is, in the best of our knowledge, the only one for which the correctness of the shared-forest has been proved. This model also ensures an optimal sharing of syntactic structures without imposing constraints on the form of the input grammar. This feature justifies the adoption of this description formalism in our system, in which implementation the problem of sharing was an essential issue.

3.2 The operational model

Standard parsing using the Earley's classic algorithm has been implemented, without introducing relevant changes, following the original method described by

the author in [1]. At this point, we shall turn our attention to the implementation of the extended LALR(1) scheme, which has required the greatest effort.

A dynamic programming interpretation of a transducer is the systematic exploration of a space of elements called *items*. This search space is a condensed representation of all possible computations of the transducer. 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 ensure 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 the classic framework, that we call S^T.

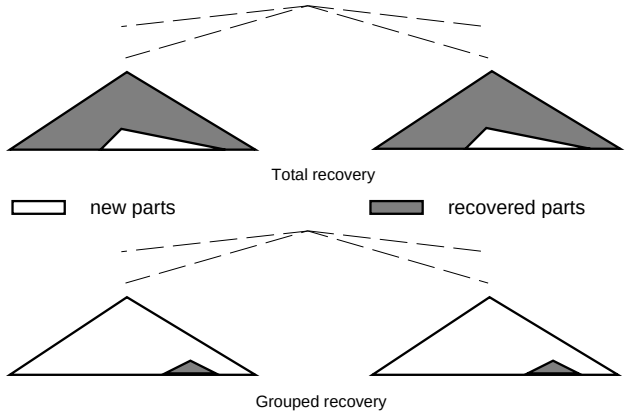

Figure 2: Practical incremental recovery

3.2.1 The concept of dynamic frame

Given a transducer, we define a *dynamic frame* as a pair $(\Re, \mathrm{Op})$ where $\Re$ is an equivalence relation on the stacks, whose classes are named *items*, and Op is an operator that translates transitions in S^T to the new framework; and verifies the following conditions:

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

Dynamic frames were originally introduced by Villemonte de la Clergerie in [6] to formalize the notion of item in relation to the use of *logical push-down automata*[3].

In practice, only two dynamic frames are considered: S^1 and S^2, whose only difference is the extension of the stack which is considered to represent configurations in the transducer. To be more precise, S^1 uses only the top, while S^2 uses also the previous element.

[3]Essentially, automata that store atoms and substitutions on their stack, and use unification to apply transitions. They are due to Lang [17], which obtains an exponential reduction in complexity over the traditional resolution methods.

Correctness and completeness of S^2 is directly derived from S^T since in the worst case transitions in S^T depend on the first two elements in the stack. This is not the case of S^1, where we must take into account the absence of information about the rest of the stack during pop transitions. In order to solve this, we generate a new transition such that it is applicable not only to the configuration generating the current one on which the pop is applied[4], but also on those to be generated and sharing the same syntactic context.

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

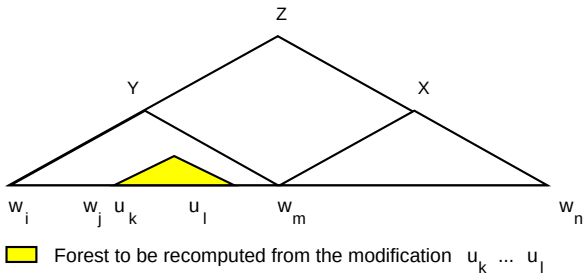

Figure 3: A pop transition $XY \mapsto Z$ totally recovering a modification

3.2.2 The parsing algorithm

The algorithm proceeds by building a collection of items. New items are produced by applying transitions to existing ones, until no new application is possible. We associate a set of items, usually called *itemset*, for each word symbol in the input string. Items in an itemset are processed in order, performing none or some transitions on each one depending on the form of the item. To ignore redundant computations we put a simple subsumption relation in place in the set of items.

Items are also used as non-terminals of the output grammar, for which rules are constructed together with their left-hand-side item. Each time a pop or a scan is applied, we generate a rule. 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 scan and by the items popped from the stack in that action, in the case of a pop. The start symbol is the last item produced by a successful computation. At this point, items are not only elements of the computation process, but also non-terminals of the output grammar. That allows us to identify items with nodes in the resulting parse forest.

4 Incremental parsing

Usually, *incremental parsing* has been attempted in two different senses: First, as an expression of the left-to-right extension of editing. Secondly, in relation with the full editing capability on the input string. We are interested in full incrementality[5], in the domain of general context-free grammars, without restrictions in the number

[4]This configuration agrees with that resulting from the pop action in S^T, and it is already included, *a fortiori*, in the computation process.

[5]Full incrementality includes left-to-right incrementality.

of editing operations to be considered simultaneously, guarantying the same level of sharing as in standard mode, and this last being without any impact.

Although from a theoretical point of view all cases of full incrementality have been considered, in practice we have focused our attention on two cases, shown in Fig. 2:

- *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 reparsed.

where the portion of the input string to be reparsed is probably, in both cases, a superset of the substring modified. The reason here being efficiency. In effect, to recover a proper subset of branches in the interval intuitively is equivalent to the recovery of isolated trees in a forest corresponding to an ambiguous node. At this point, to continue guarantying the best sharing of the resulting parse forest, a more complex algorithm to handle the reconstruction of the parse forest would then be necessary, which would imply an important increase in time and space. In fact, the problem has to do with the real necessity of such a facility since, in most practical cases, non-determinism is well located and this class of phenomena is limited.

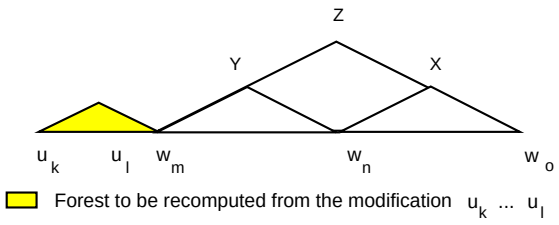

Figure 4: A pop transition $XY \mapsto Z$ independent of the modification

4.1 An overview of full incrementality in ICE

The goal of the incremental context-free parser is to recover stable parts of a shared forest between consecutive parsing steps. Although ICE has been designed to deal with several simultaneous modifications, we consider a simplified text-editing scenario with a single modification, in order to favor understanding.

Let's take a modified input string from an initial one previously parsed. We must update the altered portion of the original shared forest. To do it, it is sufficient to find a condition capable to ensure that all possible transitions to be applied from a given position in an interval in the input string are independent on the introduced modification. At this point, we focus our attention on those transitions dependent on the past of the parsing, that is, on pop transitions. In effect, 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 parsing process are also the same, the parsing will be also the same in this portion. In relation to the practical cases of incrementality considered in ICE, 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. 3. If it only extends to a well located region after the modification, we have grouped recovery, as is shown in Fig. 4.

To ensure that pop transitions are common between two consecutive parsing processes, in an interval of the unchanged input string, we shall focus on the set of items for which we are sure that there exists a successful continuation of the

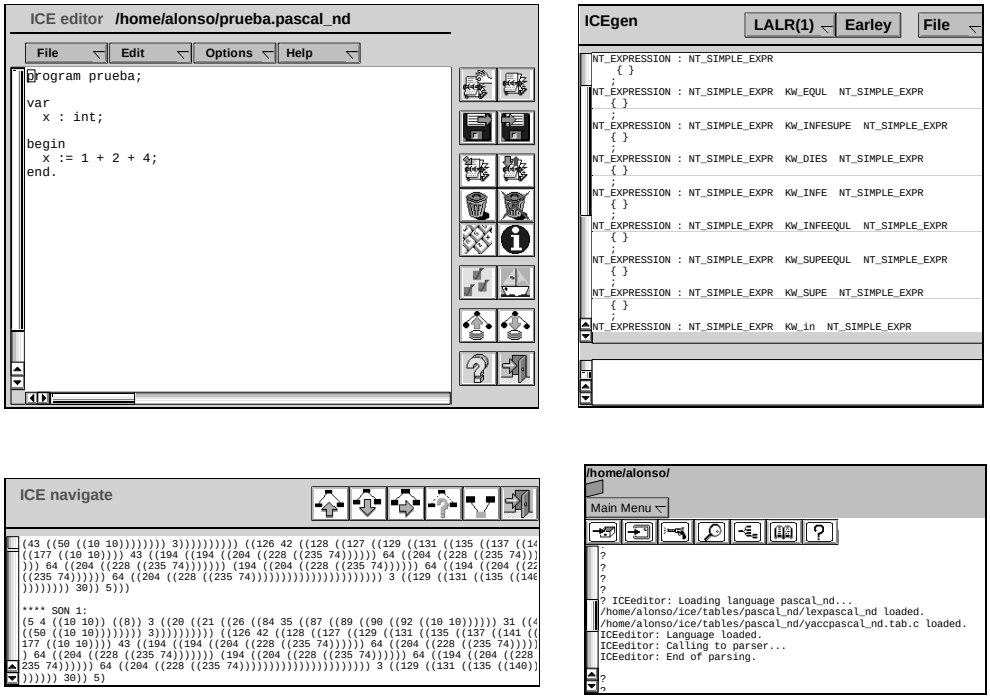

Figure 5: ICEgen and ICEeditor main windows

parse, for which they will be arguments for pop actions. This is the case of those items resulting from proper reduce actions before a shift[6]. Although this does not guarantee the recovery of all common computations in a given interval, it ensures an increased efficiency of the incremental process in practice. These kind of items can be located in a simple fashion, which guarantees a low impact in standard parsing. To establish now a condition ensuring that all pop transitions from one, taking one of these items as argument, are commons in an interval, is also simple task. It is sufficient to consider the semantics of the *back pointer* in the item notion in *Earley-like*: A pointer to the itemset at which we began to look for the configuration of the transducer represented by the item. In effect, when corresponding items between two consecutive parsing processes have *equivalent* back pointers, then incremental recovery is possible. Here, the notion of equivalency has two meanings:

1. When the itemsets pointed are the same.
2. When the itemsets pointed are associated to a same token, once the modification has been parsed.

The first meaning includes the second one, which has a direct translation on the kind of recovery they characterize. 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.

4.2 Comparing with other approaches

Here, we focus our attention in the full incremental parsing algorithm suggested by van den Brand in [8]. The author proposes a sequence of four steps to get full

[6]Empty reductions may be applied between this proper reductions and the corresponding shifts.

incrementality for an isolated modification:

1. The system focuses on the node to be updated.
2. Prune this node in order to replace it later, if possible, by the new one resulting from the parse of the modification.
3. Reparse the substring representing the modification.
4. If this reparse is successful, we recover the resulting tree. If we can translate it into the place of the old pruned node, the incremental process is finished. Otherwise, we must focus on an ancestor of the old node to restart the process from the first step.

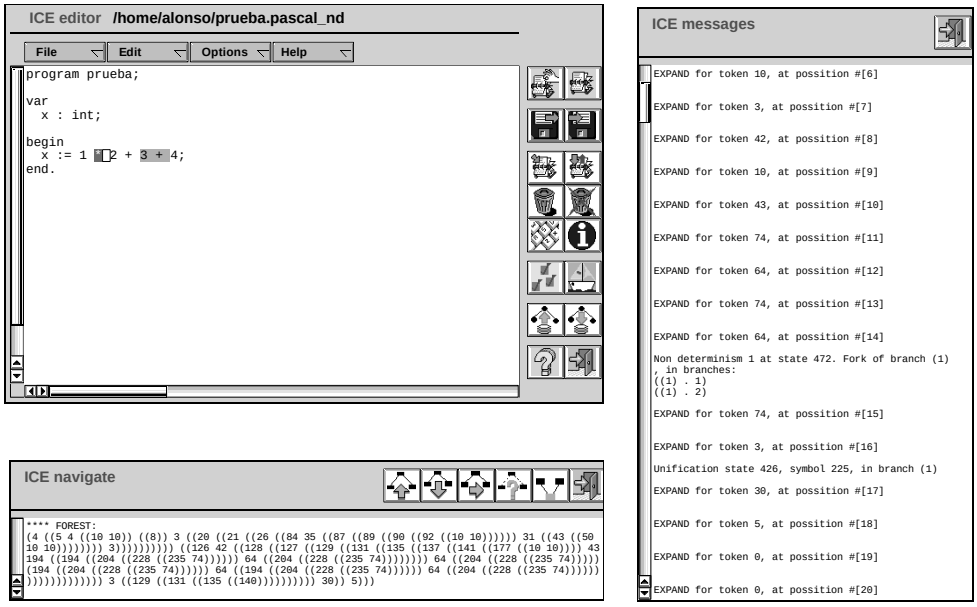

Figure 6: Analyzing a non-deterministic Pascal program

In comparison with the method applied by Ice, previously described, this approach seems to be less general. In effect:

- The concept of total recovery, the most advantageous case of incrementality, cannot be considered.
- If the reparse of the modification does not succeed, the complete program is reparsed.
- If the reparse of the modification succeeds, but the label of the node does not agree with the old one, the system can make some unnecessary work searching for the minimal node which covers the complete syntactical effect of the modification. At worst making a complete reparse of the program. In practice, to reduce the impact of this problem van den Brand proposes a set of heuristic rules to be applied, but results are not guaranteed.

A similar idea is applied in the case of the Sdf [4] environment, such as described by Rekers and Koorn in [18]. In this case, only the extent of the node focused is reparsed, which represents an additional constraint in relation to the algorithm presented by van den Brand. Here, if the parse of the modification finds an

error, the user of the system has to move the focus explicitly. Consequently, incremental parsing consists of a sequence of deriving, pruning and grafting operations interleaved with cursor movements that shift the focus of attention in the forest.

5 User interface

The ICE system has a multi-window, menu-driven user interface [19] based on the image description language AÏDA [20], and running under X11.

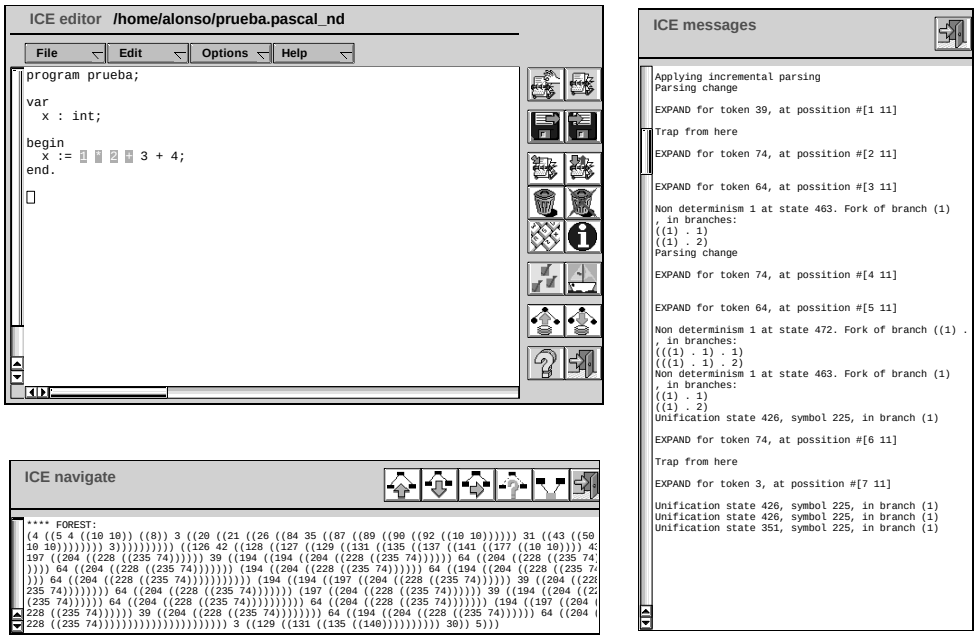

Figure 7: Incremental analysis over a modified program

5.1 An informal overview

The user interface associates with the parser generator allowing the appearance of all parser generation algorithms available in the system to unify. The tool helps the language designer in the task of writing grammars, with a dedicated editor[7]. 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.

The interface corresponding to the programming environment permits the user to choose between the parsing algorithms offered by the system and load a language generated in advance. At this point, the user can design the program and test it by invoking the corresponding parser in two modes: Standard or incremental. In addition, 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, allowing comparative tests to be done among

[7] All editors used in the system are inspired by EMACS [22] and follow this text editing standard in UNIX.

different parsing algorithms and errors always been reported. In the same way, the user interface allows parse forest to be easily recovered and manipulated.

Finally, the user-interface can be customized by adding new graphic objects whose activation starts the evaluation of a function which is defined in Le_Lisp [21].

5.2 The system at work

To explain the behavior of the environment at work, we shall build an analyzer for Pascal, for which we include ambiguity for arithmetic expressions. To illustrate this discussion, we shall refer to Fig. 5, 6 and 7, representing the external appearance of Ice at different moments of the process described. As first step, we must create the file describing the grammar. For this purpose, we use the ICEgen tool. The main window of this tool, shown in the right upper corner of Fig. 5, contains an editor for editing grammars. To parse the grammar, we dispose of buttons for each one of the algorithms available. Error messages from parsing are shown in a box under the editor. If we click in a message, the editor will scroll the text to make the line in which the error is located visible. When no errors arise, an usable parser will have been generated.

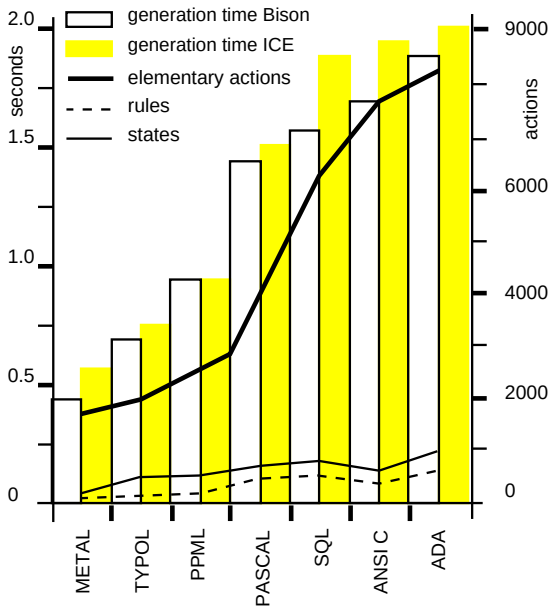

Figure 8: Results on parser generation

Once we have generated the parser, it is time to test it. The ICEeditor tool has an editor to write the source code, which is shown in the left upper corner of Fig. 5, 6 and 7. When a parser is made, the editor knows the token's structure of the text and the following editions will be made according to this structure. Graphical resources as fonts and colors are used as guidelines for the edition. Buttons for elemental incremental operations like insert, delete, undelete and modify are available. Each time we parse a program, a window with the resulting messages is activated, as shown in the right-hand-side of Fig. 6. The user can choose the level of information that will be displayed in this window, from nothing at all to have every elementary action available[8]. The user can also navigate in the shared forest, which is shown in

[8]Like, for example, push and pops from the stack, when the chosen parsing algorithm works on a push-down transducer.

the ICEnavigate window, in the bottom left-hand-corner of Fig. 5, 6 and 7. In each move around the tree, information is given about the shared branches, the number of sons and the number of ambiguities. If necessary, the structures generated during the parsing process can be saved on disk and recovered in following sessions.

In this example, we have used a small program that we have edited using the ICEeditor tool, as can be seen in Fig. 5. If we do not specify another thing, the system automatically loads the parser considering the extension from the source file. In this case the parser loaded is that corresponding to ambiguous Pascal that had previously been created using the ICEgen tool. The ICEmessages window in Fig. 6 shows a trace of the corresponding parsing process, where the user has chosen an intermediate level of information for the debugging facility. We can observe the branches corresponding to ambiguities in the arithmetic expression 1+2+4 and how they are unified later. If we want detailed information about tokens, we can use a button in ICEeditor for this purpose. Now, we change the first + in the expression for * and we insert a new summand before 4. If we see the ICEmessages window in Fig. 7, we can observe the incremental recoveries for these modifications, the new branches created by new ambiguities that have risen and how they are unified later.

To get a more friendly environment, we can select the language in which the system interacts with us: English, French, Spanish and Galician[9] are currently available. A help facility is available every time to solve questions about the editors and the incremental facilities.

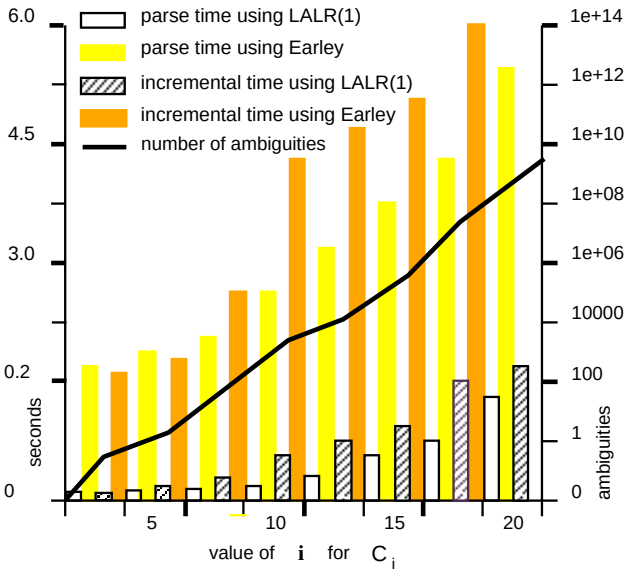

Figure 9: Parse time using Ice

6 Experimental results

We have compared Ice with Bison [11], Glr [5] and Sdf [4], 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 show also the efficiency of incremental parsing in relation to the standard one, and the capability

[9]The co-official language, together with Spanish, in the Autonomous Community of Galicia, Spain.

of ICE to share computations. All the measurements have been performed on a *Sun SPARCstation 10*, weakly loaded.

In relation to parser generation, we took several known programming languages and extracted the time used to generate parser tables, comparing BISON with the LALR(1) scheme in ICE[10]. Results are given in relation to different criteria. So, Fig. 8 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[11]. At this point, it is important to remark the behavior of ANSI-C that does not seem to correspond with 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 automaton may not be in direct relation with the total amount of work necessary to build it. In order to explain that, 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 automaton, and the generation of transitions between two states.

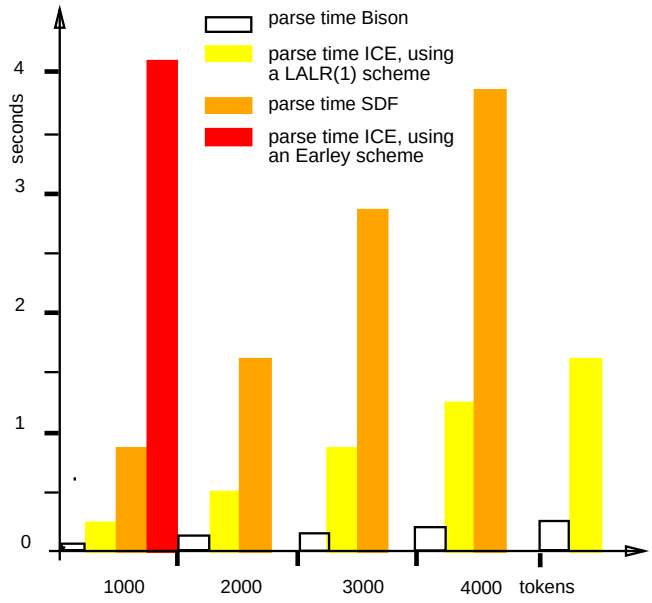

Figure 10: Results on deterministic parsing

We use the syntax of complete Pascal as a guideline for parsing tests. In Fig. 10 comparisons are established among parsers generated by ICE[12], BISON and SDF, when the context is deterministic. We consider ICE, SDF and GLR when the context is non-deterministic, as it is shown in Fig. 11. All measurements include lexical time since, for the version considered, it is not possible in SDF and GLR to differentiate it from the parsing. In all other cases FLEX [23] has been used as a lexical analyzer. We have considered two versions for Pascal: Deterministic and non-deterministic, including this last one the called *dangling else* and the 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 impact of lexical time, we have considered

[10] Earley's algorithm is a grammar oriented method.
[11] BISON and ICE generate LALR(1) machines, SDF LR(0) ones.
[12] Using both, LALR(1) and Earley's schemes.

in the case of non-deterministic parsing, 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. The grammar contains a rule

$$\text{Expression} ::= \text{Expression} + \text{Expression}$$

therefore these programs have a number of ambiguous parses which grow exponentially with i. This number is:

$$C_i = \begin{cases} 1 & \textbf{if} \quad i \in \{0, 1\} \\ \left(\begin{array}{c} 2i \\ i \end{array} \right) \frac{1}{i+1} & \textbf{if} \quad i > 1 \end{cases}$$

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. Finally, ICE, SDF and GLR are implemented in LE_LISP, and BISON in C.

To illustrate incrementality, we analize the previous programs in which we substitute expressions $b+b$ by b. Results corresponding to incremental and standard parsing are shown in Fig. 9, and those related to sharing in Fig. 12.

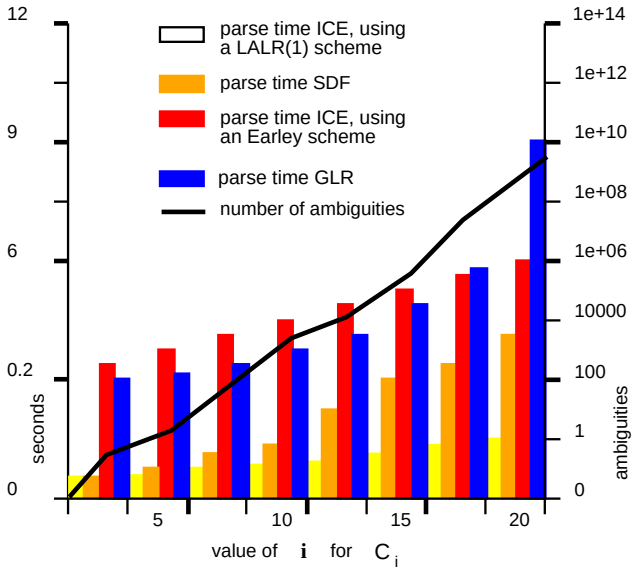

Figure 11: Results on non-deterministic parsing

7 Conclusions and future work

The ICE system is devoted to simultaneous editing of language definitions and programs. The modular composition of parsers allows the user to consider specialized algorithms if a particular case requires it, or simply to compare performances among a set of available parsing environments. In comparison with other systems, our algorithm seems to surpass the previous results.

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 was developed for these generators.

Finally, the system described includes a graphic interface, where customizations can be done either interactively, or through an initialization file.

In relation to future work, we are currently working on the implementation of a prototype extending the capabilities of ICE in three ways: Firstly, computation of abstract syntax trees and automatic error correction. In that setting, the incremental aspects of the system will be fully exploited. On the other hand, ICE has been chosen as a starting point to constitute the syntactic kernel of a generator of natural language analyzers by the *Ramón Piñeiro Linguistic Research Center*. At this point, the system is also being extended in order to deal with unification grammars.

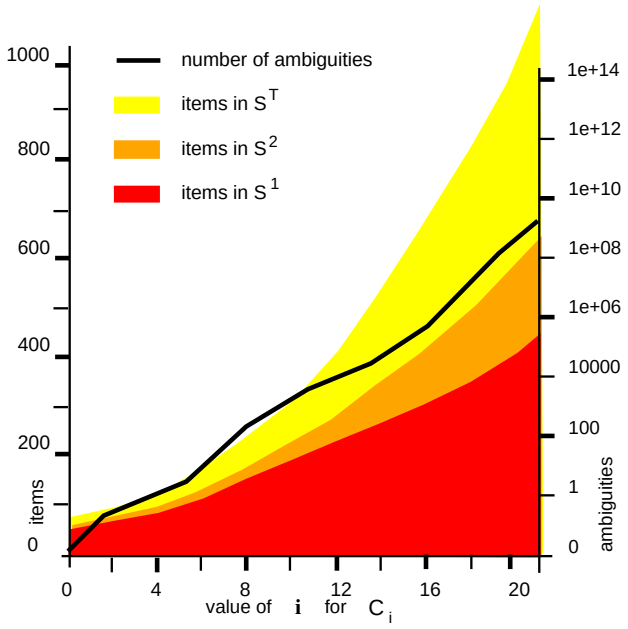

Figure 12: Items generated using S^1, S^2 and S^T schemes.

References

[1] J. Earley, "An efficient context-free parsing algorithm", *Communications of the ACM*, , no. 2, pp. 94–102, 1970.

[2] B. Lang, "Deterministic techniques for efficient non-deterministic parsers", Tech. Rep. 72, INRIA, Rocquencourt, France, 1974.

[3] M. Tomita, "An efficient augmented-context-free parsing algorithm", *Computational Linguistics*, , no. 1–2, pp. 31–36, 1987.

[4] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers, "The syntax definition formalism sdf - reference manual", *SIGPLAN Notices*, , no. 11, pp. 43–75, 1989.

[5] J. Rekers, *Parser Generation for Interactive Environments*, PhD thesis, University of Amsterdam, Amsterdam, The Netherlands, 1992.

[6] E. Villemonte de la Clergerie, *Automates à Piles et Programmation Dynamique*, PhD thesis, University of Paris VII, France, 1993.

[7] M. Vilares Ferro, "Efficient sharing in ambiguous parsing", in *Actas del X Congreso de la SEPLN*, Córdoba, España, 1994.

[8] M.G.J. van den Brand, *A Generator for Incremental Programming Environments*, PhD thesis, Katholieke Universiteit Nijmegen, Nijmegen, Nederlands, 1992.

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

[10] M. Vilares Ferro and B. A. Dion, "Efficient incremental parsing for context-free languages", in *Proc. of the* 5^{th} *IEEE International Conference on Computer Languages*, Toulouse, France, 1994, pp. 241–252.

[11] Ch. Donnelly and R.M. Stallman, *BISON. Reference Manual*, Free Software Foundation, Inc., 675 Mass Avenue, Cambridge, MA 02139, U.S.A., 1.20 edition, 1992.

[12] A. Paeseler, "Modification of Earley's algorithm for speech recognition", *NATO ASI Series*, pp. 466–472, 1988.

[13] F.C.N. Pereira and D.H.D. Warren, "Parsing as deduction", in *Proc. of the* 21^{st} *Annual Metting of the Association for Computational Linguistics*, 37-144, Ed., Cambridge, Massachusetts, U.S.A., 1984.

[14] B. Lang, "Towards a uniform formal framework for parsing", in *Current Issues in Parsing Technology*, M. Tomita ed., Ed., pp. 153–171. Kluwer Academic Publishers, 1991.

[15] S. Billot and B. Lang, "The structure of shared forest in ambiguous parsing", Tech. Rep. 1038, INRIA, Rocquencourt, France, 1989.

[16] M. Bouckaert, A. Pirotte, and M. Snelling, "Efficient parsing algorithms for general context-free grammars", *Information Sciences*, pp. 1–26, 1975.

[17] B. Lang, "Complete evaluation of horn clauses, an automata theoretic approach", Tech. Rep. 913, INRIA, Rocquencourt, France, 1988.

[18] J. Rekers and W. Koorn, "Subsring parsing for arbitrary context-free grammars", *SIGPLAN Notices*, , no. 5, pp. 59–66, 1991.

[19] M. A. Alonso Pardo, "Edición interactiva en entornos incrementales", Master's thesis, Computer Sciences Department, University of A Coruña, A Coruña, Spain, 1994.

[20] Ilog S.A., 2 Avenue Galliéni, BP 85, 94253 Gentilly, France, *Aïda: Reference Manual. Version 1.65*, 1992.

[21] Ilog S.A., 2 Avenue Galliéni, BP 85, 94253 Gentilly, France, *Le_Lisp. Version 15.25. Reference Manual*, 1992.

[22] R.M. Stallman, *GNU Emacs Manual. Version 18*, Free Software Foundation, Inc., 675 Mass Avenue, Cambridge, MA 02139, U.S.A., 1991.

[23] V. Paxson, *FLEX: Reference Manual. Release 2.4.6*, Free Software Foundation, Inc., 675 Mass Avenue, Cambridge, MA 02139, U.S.A., 1994.