1991

J. Rekers

Generalized LR parsing for
general context-free grammars

Computer Science/Department of Software Technology Report CS-R9153 December

CWI, nationaal instituut voor onderzoek op het gebied van wiskunde en informatica

CWI is the research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a non-profit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands organization for scientific research (NWO).

Generalized LR Parsing for General Context-Free Grammars

J. Rekers

CWI
P.O.box 4079, 1009 AB Amsterdam, the Netherlands
email: rekers@cwi.nl

Abstract: Which methods for parser generation and parsing are best suited for an interactive development system of syntax definitions? In this chapter we argue that a Generalized LR parsing algorithm is the best choice. We present an enhanced version of Tomita's GLR algorithm, and compare its efficiency with two competitors, YACC and Earley's algorithm.

Keywords & phrases: Generalized LR parsing, general context-free grammars, representation of ambiguous parses, comparison between Earley, Yacc and GLR.

1991 CR Categories: D.3.1 syntax, D.3.4 parsing, E.1 graphs, E.1 trees, F.4.3 context-free languages.

1991 Mathematics Subject Classification: 68N20 compilers and generators, 68P05 data structures, 68Q50 parsing.

Note: Partial support has been received from the European Communities under ESPRIT project 2177 (Generation of Interactive Programming Environments II - GIPE II) and from the Netherlands Organization for Scientific Research – NWO, project *Incremental Program Generators.*

1 Introduction

Which methods for parser generation and parsing are best suited for an interactive development system of syntax definitions? We encountered this question in the context of the Esprit project GIPE (Generation of Interactive Programming Environments), that aims at deriving programming environments from formal language definitions.

Report CS-R9153
ISSN 0169-118X
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

We have selected a Generalized LR (GLR) method as the basis for our syntactic tools. This algorithm was originally developed by Tomita [1]. We extended it to general context-free grammars and improved the sharing in the parse forest it generates. In this paper we summarize the arguments for choosing the GLR method, we describe our extensions to Tomita's parsing algorithm and we compare the efficiency of the GLR algorithm with YACC and Earley's algorithm.

Most of the subjects discussed are of general relevance, but dependencies on the specific setting in which these questions were raised is unavoidable. In particular, our ultimate goal has been to implement SDF (Syntax Definition Formalism, [2]), a specification formalism for lexical, context-free and abstract syntax. However, the paper does not require any knowledge of SDF, and all algorithms presented are based on conventional BNF definitions.

2 Choosing a parsing method

Which requirements does SDF impose on its implementation and how do these affect the choice of a parser and parser generator?

2.1 Requirements

The parser and parser generator should accept general context-free grammars (CFGs). This class may seem overly large, as LALR(1) or LR(1) is usually a large enough class to define programming languages in, and ambiguous grammars are in most cases undesirable. We prefer the larger class of CFGs however for the following reasons

- Many parser generation systems do not allow certain kinds of rules like left-recursive, right-recursive or epsilon rules. This forces the writer of a grammar to avoid these cases, and it restricts the form of parse trees that can be built. By allowing all of these, maximal freedom is given to the writer of a specification.

- SDF allows modular composition of grammar modules. This means that if one module imports another one, their grammars are combined. The only class of context-free grammars that is closed under composition, is that class itself [3, page 131]. This is not the case for any subclass of it, like LR(k), LALR(1) or LL(k).

- It is not possible to exclude ambiguous grammars, as it is undecidable whether a grammar is ambiguous [4, page 260]. In practice, one can only ensure that a grammar is non-ambiguous by restricting it to a smaller class of grammars, like LR(k) or LL(k). This would at best mean that the parser is only allowed to use a fixed number of symbols of look-ahead, while we would like it to use arbitrary look-ahead. One can include the full class of unambiguous grammars only by allowing general CFGs.

- SDF has a quite elaborate scheme for processing the priorities between grammar rules, which is partly defined by computing which parse tree is the "largest" among the possibilities [2, section 6.2]. This means that the parser must generate all possible parse trees in order that they can be compared.

As the envisaged system is intended for the definition of programming languages, large parts of the grammars will fit in the LR(1) class. In these cases the parser should be comparable in speed to the ordinary, efficient, LR parsing techniques.

We aim at a system for the interactive development of syntax definitions. Parser generation should therefore be fast. It must be possible to make incremental updates to the parser generated, and parser generation for different modules of a modular specification should not involve duplicate generation effort. These requirements all point to a very simple parser generation algorithm, without expensive global operations on the grammar rules.

2.2 The parser

The possible algorithms we examined for the parser and its generator are:

- LR(1) algorithms

  These have an efficient parser generation (table construction) algorithm that leads to time efficient parsers. However, the class of LR(1) grammars is too restricted.

- LR(k) algorithms, with $k > 1$

  The larger k is, the larger the class of accepted grammars becomes. However, parsing in accordance with all non-ambiguous grammars is

still impossible, and parser generation (table construction) time increases exponentially with k.

- Earley's universal context-free parsing algorithm [5]

  This algorithm can handle all context-free grammars and can work with a negligible parser generation phase. However, an Earley parser is very slow on LR(1) grammars.

- Tomita's universal parsing algorithm [1]

  This algorithm can be placed between LR(k) algorithms and Earley's algorithm. The class of accepted grammars is restricted to acyclic grammars and the time complexity of the algorithm depends on the complexity of the grammar and the sentence being parsed. Tomita's algorithm can use any LR parse table constructor as a parser generator. Sikkel studied the differences between the algorithms of Earley's and Tomita's and concluded that both are remarkably similar [6].

Tomita's algorithm is both more powerful than any LR(k) algorithm as well as faster than Earley's algorithm on most grammars, but it loops on cyclic grammars. We considered this as a bug in the algorithm and have repaired it. By doing so, we have converted the algorithm to a *Generalized LR* parsing algorithm which is as strong as Earley's algorithm.

The GLR algorithm starts as an ordinary LR parser, but when it encounters a shift-reduce or reduce-reduce conflict in its parse table during parsing, it splits up in as many parsers as there are possibilities. These parsers then act in parallel; some of them may die if the conflicting entry was caused by a need for a larger look-ahead, some of them are combined again after having recognized an ambiguous part of the input. In [7], Lang described this scheme in a general manner for all kinds of table driven parsers. Our GLR algorithm is a special case of his general technique.

The generalized LR parsing algorithm can handle more deterministic grammars than any LR(k) algorithm, because for each LR(k) parsing algorithm a grammar can be constructed which needs a look-ahead of $k+1$ and hence cannot be parsed by that algorithm. The generalized LR parsing algorithm does not have such an upper limit, because it adjusts its look-ahead dynamically by using different parse stacks as a look-ahead mechanism.

The GLR parsing algorithm is called pseudo-parallel, but is clearly designed to run on one processor only. A parallel version of the algorithm that splits up at each conflict in the parse table does not induce much gain due

to the large communication overhead [8, 9]. A more successful attempt to parallelize Tomita's algorithm has been performed by Sikkel [10]. He uses a separate processor for each word of the input sentence and each processor parses all constituents that start with that particular word. See [11] for a general overview of parallel parsing algorithms.

2.3 The parser generator

Having decided to use the Generalized LR parsing algorithm, we still have to choose which parse table constructor to use, as the GLR parsing algorithm can work with LR(0), SLR(1), LALR(1) and LR(1) tables. Unlike the conventional situation, these tables are allowed to contain multiple entries (shift-reduce and reduce-reduce conflicts) when used in combination with the GLR algorithm.

An LR(0) parse table constructor generates a reduce action for each rule that has been recognized completely, without checking if the look-ahead is right for it. LR(1) parse table constructors, on the other hand, only generate a reduce action if the look-ahead is right. So, the GLR parsing algorithm will start more parsers when controlled by an LR(0) table than when controlled by an LR(1) table for the same grammar.

A disadvantage of the LR(1) technique is that an LR(1) parse table contains more states than an LR(0) table for the same grammar, as, in the LR(1) technique, states are considered different if their items have different look-ahead information. If the GLR parser is controlled by an LR(1) table it will therefore be able to join less parsers, as parsers are joined only if they have the same state on top of their stack. From measurements described in [12] and [13], it turns out that this disadvantage often outweighs the advantage of running fewer parsers.

SLR(1) and LALR(1) parse tables contain as many states as LR(0) parse tables, while they do apply look-ahead information to limit the number of reductions. SLR(1) tables generate a reduce action for a rule $A ::= \alpha$ only if the next input symbol is in $FOLLOW(A)$. LALR(1) tables even generate less reduce actions, by using a LR(1) construction scheme in which states are joined as if no look-ahead information was present.

If we order the different table generators in accordance with the number of useless reduce actions generated, LR(0) is on top, next come SLR(1), LALR(1) and LR(1). It is to be expected, and verified by measurements, that the GLR algorithm will be most efficient with LALR(1) tables. However, in the measurements performed in [12], SLR(1) and LALR(1) have

about equal effect, and their gain in speed over LR(0) is only 10%.

We have decided to use an LR(0) table generation algorithm, as this is the simplest generator, and will be the easiest one to extend both to incremental parser generation [14] and to parser generation for modular grammars [15, chapter 3].

3 Generalized LR recognition

A Generalized LR parser runs several simple LR parsers in parallel. It starts as a single LR parser, but, if it encounters a conflict in the parse table, it splits in as many parsers as there are conflicting possibilities. These independently running simple parsers are fully determined by their parse stack. If two parsers have the same state on top of their stack, they are joined in a single parser with a forked stack. A reduce action which affects a part of the parse stack containing a fork, splits the corresponding parser again into two separate parsers. If a parser encounters an error entry in the parse table, it is killed by removing it from the set of active parsers.

The algorithm we describe differs slightly from the original Tomita algorithm, mainly to allow it to handle the full class of context-free grammars.

3.1 Description

The joined stacks maintained by the algorithm have a graph-like form and are implemented using *stack nodes* that contain a state and a set of links to stack nodes one level lower on the stack.

If a state must be pushed on a stack which has stack node p^- on top, a new stack node p is created which gets a link back to p^-, and p becomes the top of the stack. A pop-action is not performed physically, the top of the stack pointer is just moved one level lower on the stack. A pop action results in a *set* of new top nodes.

During parsing, the variable *active-parsers* contains all stack nodes which have been on top of a stack during the processing of the current input token. This set never contains two stack nodes with the same state. When a parser with top node p^- must push a state s, while there is already a stack node p in *active-parsers* which contains state s, then the links of p are extended with a link to p^-.

The GLR recognizer creates and maintains these graph-like stacks while it processes its input sentence. Initially, the set of active parsers just consists of a single stack node having as state the start state of the parse table.

The input sentence is extended with an end-of-sentence marker, EOF. Next, routine PARSEWORD is called repeatedly to process each token in the input sentence. The Boolean *accept-sentence*, initially "false", indicates whether the sentence has been recognized or not. If the parse tables prescribe an accept action at the processing of EOF, this variable is set to "true".

For each of the active parsers, PARSEWORD consults the parse table by means of routine ACTION. This routine returns a set of actions to perform with the state on top of the stack and the current input token. A "(shift *state'*)"-action means that the parser has to push *state'* on the stack and has to move to the next input symbol. A "(reduce $A ::= \alpha$)"-action means that the parser has to pop $|\alpha|$ states off the stack, has to use routine GOTO to obtain a new state *state'*, and has to push *state'* on the stack again.

Shift actions are postponed until all parsers are ready to shift and they are performed by routine SHIFTER. On a reduction of "$A ::= \alpha$" in a parser with top node p, all stack nodes at $|\alpha|$ links distance from p are given to REDUCER for further processing. Both SHIFTER and REDUCER have to push new nodes on the stack, so here it may happen that the links of other stack nodes must be extended in order to join two parsers. In REDUCER the matter is even more complicated. If the links of a stack node are extended, all previously performed reductions must be re-checked as new paths may have become possible over the link just created.

This re-checking of the reductions that have already been performed is a modification to the original Tomita algorithm, and is due to Nozohoor-Farshi [16]. In the original algorithm only those paths were reconsidered which had the new link as first step. However, in order to take ϵ-reductions seriously, all paths which contain the new link must be reconsidered.

The modification of Nozohoor-Farshi affects the way in which ϵ-symbols between adjacent input symbols are treated. In the original algorithm as many ϵ-symbols as needed are put between them, while in the variant of Nozohoor-Farshi only one ϵ is used, which is shared as many times as needed. This subtle difference avoids looping on cyclic grammars (cf. section 4.1) and on grammars in which there exists a non-terminal A, such that $A \stackrel{+}{\Longrightarrow} \alpha A \beta$ where $\alpha \stackrel{+}{\Longrightarrow} \epsilon$ but not $\beta \stackrel{*}{\Longrightarrow} \epsilon$. We refer to [16] for the full explanation of this extension, which allows the GLR recognizer to handle the full class of context-free grammars.

3.2 Algorithm of the recognizer

The GLR recognizer for general context-free grammars described above is implemented by the following functions. The Lisp version of this algorithm can be found in [15, Appendix A.1].

```
PARSE(Grammar, a_1 ... a_n) :
  a_{n+1} := EOF
  global accept-sentence := false
  create a stack node p with state START-STATE(Grammar)
  global active-parsers := { p }
  for i := 1 to n + 1 do
    global current-token := a_i
    PARSEWORD
  return accept-sentence

PARSEWORD :
  global for-actor := active-parsers
  global for-shifter := ∅
  while for-actor ≠ ∅ do
    remove a parser p from for-actor
    ACTOR(p)
  SHIFTER

ACTOR(p) :
  forall action ∈ ACTION(state(p), current-token) do
    if action = (shift state' ) then
      add <p, state' > to for-shifter
    else if action = (reduce A::=α) then
      DO-REDUCTIONS(p, A::=α)
    else if action = accept then
      accept-sentence := true

DO-REDUCTIONS(p, A::=α) :
  forall p' for which a path of length(α) from p to p' exists do
    REDUCER(p', GOTO(state(p' ), A))

REDUCER(p⁻, state) :
  if ∃p ∈ active-parsers with state(p) = state then
    if there is no direct link from p to p⁻ yet then
      add a link link from p to p⁻
      forall p' in (active-parsers - for-actor) do
        forall (reduce rule) ∈ ACTION(state(p' ), current-token) do
          DO-LIMITED-REDUCTIONS(p', rule, link)
  else
```

```
        create a stack node p with state state
        add a link from p to p⁻
        add p to active-parsers
        add p to for-actor

DO-LIMITED-REDUCTIONS(p, A::=α, link) :
    forall p' for which a path of length(α) from p to p' through link exists do
        REDUCER(p', GOTO(state(p' ), A))

SHIFTER :
    active-parsers := ∅
    forall <p⁻, state' > ∈ for-shifter do
        if ∃p ∈ active-parsers with state(p) = state' then
            add a link from p to p⁻
        else
            create a stack node p with state state'
            add a link from p to p⁻
            add p to active-parsers
```

3.3 An example

We illustrate the recognizer using the following grammar with only one rule:

S ::= S S (Grammar G_{SS})

and let it parse the sentential form "S S S", which is ambiguous according to the grammar. It is possible to parse sentential forms with the recognizer, as the algorithm makes no distinction between terminals and non-terminals. We could, of course, also add a rule "S ::= a" to the grammar and parse the sentence "a a a", but that would only introduce additional, and less interesting, reduce actions. The LR(0) parse table of G_{SS} is

	transitions		reductions
state	EOF	S	
0		shift 1	
1	accept	shift 2	
2		shift 2	reduce S ::= S S

In the trace we denote the stack nodes by little boxes, which contain a state number and can have links to other stack nodes. For example,

represents a stack node containing state 1, that has a link to another stack node containing state 0. Now, we show the step by step execution of the recognition algorithm.

Initially

active-parsers := { [0] }

The first token

current-token := S
PARSEWORD
  ACTOR([0])
    ACTION(0, S) = { shift 1 }, so *for-shifter* := { < [0] , 0 > }
  SHIFTER
    active-parsers := { [1] → [0] }

The second token

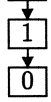
current-token := S
PARSEWORD
  ACTOR([1] → [0])
    ACTION(1, S) = { shift 2 }, so *for-shifter* := { < [1] → [0] , 2 > }
  SHIFTER
    active-parsers := { [2] → [1] → [0] }

The third token

current-token := S
PARSEWORD
  ACTOR([2] → [1] → [0])
    ACTION(2, S) = { shift 2, reduce S ::= S S }

the shift action is performed by setting *for-shifter* to { < [2] , 2 > },

the reduce action by DO-REDUCTIONS([2] , S ::= S S)



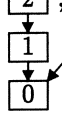
pop two nodes off the stack, and

REDUCER([0] , GOTO(0, S))

GOTO(0, S) = 1

there is no parser yet in *active-parsers* with state = 1,

so we extend *active-parsers* to { [2] , [1] },



and add [1] to *for-actor*

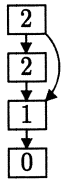
ACTOR([1])

ACTION(1, S) = { shift 2 }

so *for-shifter* is extended to { < [2] , 2 >, < [1] , 2 > }

SHIFTER

active-parsers := { [2] }

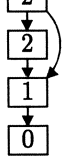


The last token

current-token := EOF

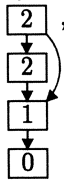
PARSEWORD

ACTOR([2])



ACTION(2, EOF) = { reduce S ::= S S }

DO-REDUCTIONS([2] , S ::= S S)



there are two ways to pop two nodes off the stack, via [2] and [2]

via the first path:

REDUCER ([1] , GOTO(1, S))

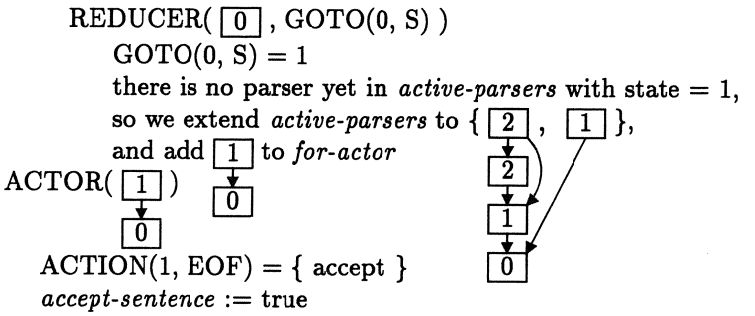
GOTO(1, S) = 2

there is a parser in *active-parsers* with state = 2

a link from this parser to [1] already exist,

so do nothing

via the second path:



Finally

return true

3.4 Cycles in the parse stack

The graph of stack nodes, as generated by the recognizer of section 3.2 may in some cases become cyclic. To explain how and why this happens, we use the following grammar

S ::= A S b (Grammar G_1)
S ::= x
A ::= ϵ

of the language $xb^n, n \geq 0$. The LR(0) parse table of this grammar is:

	transitions					reductions
state	x	b	EOF	A	S	
0	shift 1			shift 2	shift 3	reduce A ::= ϵ
1						reduce S ::= x
2	shift 1			shift 2	shift 4	reduce A ::= ϵ
3			accept			
4		shift 5				
5						reduce S ::= A S b

On parsing a sentence xb^n in accordance with G_1, the parser needs to introduce just as many ϵ's before the x, as there are b's after it. The original Tomita algorithm loops on this grammar, as an additional ϵ can always be inserted. We avoid this loop in our algorithm by sharing ϵ symbols, but by doing so, the graph of parse stacks becomes cyclic. This is necessary, as for

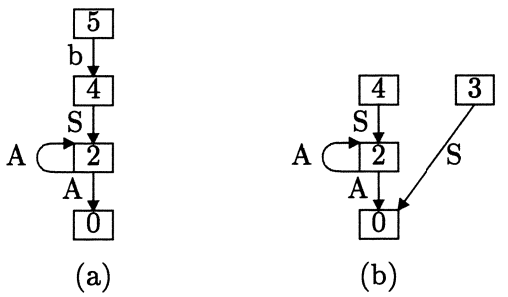


Figure 1: The parse stack before and after reduction of S ::= A S b

every number of *b*'s, enough *A*'s should be available to reduce A ::= A S b repeatedly.

Just before a reduction of the rule S ::= A S b, the parse stacks looks like in Fig. 1(a).[1] Popping off the nodes for the symbols on the right-hand side can be done over two paths; one that goes straight down and ends in stack node [0], and the other that goes over the cycle and ends [2]. Pushing the states GOTO(0, *S*) and GOTO(2, *S*) on both stack nodes, leads to the graph of stack nodes as in Fig. 1(b). It depends on the next input symbol, *b* or EOF, which of these two parsers will survive.

This example shows how the parser uses a cyclic parse stack to introduce just as many ϵ symbols as there will be needed afterwards.

4 Generalized LR parsing

If we generate a tree for the input sentence, we extend the GLR *recognizer* of the previous section in a GLR *parser*. In the ordinary LR case, a parser generates a tree by not only pushing states on its parse stack, but also subtrees. On a shift-action it pushes a terminal node on the stack, and on a reduce action it pops the subtrees of the right-hand side of the rule off the stack, takes these together in a new subtree, and pushes this subtree on the stack again. When the parser encounters the accept action, the stack contains the parse tree for the whole sentence.

In the GLR case the input sentence may be ambiguous and several trees must be built for it. In order to do so, we build a parse forest which splits at ambiguous points and shares common subtrees. This parse forest may

[1] For clarity, we have annotated the links in Fig. 1 with symbols, while these are actually not present in the algorithm.

become cyclic (and is thus, in fact, a graph) as a result of cycles in the grammar.

Before we continue the description of the parse forest, we have to spend a few words on the nature of these *cyclic grammars* and the problems they introduce for a parser.

4.1 Cyclic grammars

A cyclic grammar is a grammar containing a non-terminal which can derive itself, e.g. $S \stackrel{*}{\Longrightarrow} \alpha A \beta \stackrel{+}{\Longrightarrow} \alpha A \beta$. These grammars are problematic because the derivation $A \stackrel{+}{\Longrightarrow} A$ can be repeated infinitely many times in any derivation that contains an A. This may cause parsers to loop forever and gives rise to infinitely many different parse trees. Most parsing systems do not allow cyclic grammars, as these can always be rewritten into non-cyclic ones that recognize the same language. This limits the expressive power of context-free grammars, as a cyclic grammar can be the most compact and natural way to describe a language. Therefore, we prefer to deal with cyclic grammars in the parser itself. By doing so, the parse forest built becomes cyclic.

4.2 The structure of the parse forest

The parse forest which is built by our GLR parsing algorithm consists of instances of three structures: *symbol node*, *term node* and *rule node*.

- *Symbol nodes* are labeled with a non-terminal of the grammar. Edges that depart from a symbol node are called *possibilities*, and point to a rule node whose rule has the non-terminal of the symbol node as its left-hand side. If a symbol node has more than one possibility, there are several applicable production rules. This multiplicity represents an ambiguity in the parse.

- *Term nodes* are labeled with a terminal. Term nodes do not have outgoing edges and are leaves of the parse graph.

- *Rule nodes* are labeled with a rule of the grammar. A rule node has as many outgoing edges as it has elements in the right-hand side of the rule, and these edges are ordered. If the associated element of an edge is a terminal, the edge goes to a term node labeled with that terminal; if it is a non-terminal, the edge goes to a symbol node labeled with the non-terminal. In the case of an ϵ-rule the rule node does not have any outgoing edge and constitutes a leaf of the parse graph.

Note that the parse forest thus organized forms a bipartite graph [17, p.17], in which the rule nodes are in one partition, and the symbol nodes and term nodes in the other.

In the GLR parser, the links between the nodes of the parse stack are extended with term nodes and symbol nodes. On a shift action, a term node is created which is used in the links of all parsers that are ready to shift. On a reduce action, the term nodes and symbol nodes of each stack path are assembled in a rule node. Next, a symbol node is created with the new rule node as its only possibility. This symbol node is then attached to the link between the associated nodes in the parse stack. If such a link already exists, however, it already has a symbol node. In that case, the possibilities of that symbol node are extended with the rule node, and an ambiguous point in the parse forest is introduced.

4.3 Algorithm of the parser

The algorithms of the recognizer (section 3.2) and the parser are quite similar. They only differ in the fact that a parse forest is built. The differences are marked by a bar in the right margin.

```
PARSE(Grammar, a_1 ... a_n) :
    a_{n+1} := EOF
    global accepting-parser := ∅                                    |
    create a stack node p with state START-STATE(Grammar)
    global active-parsers := { p }
    for i := 1 to n + 1 do
        global current-token := a_i
        PARSEWORD
    if accepting-parser ≠ ∅ then                                    |
        return the tree node of the only link of accepting-parser   |
    else                                                            |
        return ∅                                                    |

PARSEWORD :
    global for-actor := active-parsers
    global for-shifter := ∅
    while for-actor ≠ ∅ do
        remove a parser p from for-actor
        ACTOR(p)
    SHIFTER

ACTOR(p) :
```

```
    forall action ∈ ACTION(state(p), current-token) do
        if action = (shift state' ) then
            add <p, state' > to for-shifter
        else if action = (reduce A::=α) then
            DO-REDUCTIONS(p, A::=α)
        else if action = accept then
            accepting-parser := p

DO-REDUCTIONS(p, A::=α) :
    forall p' for which a path of length(α) from p to p' exists do
        kids := the tree nodes of the links which form the path from p to p'
        REDUCER(p', GOTO(state(p' ), A), A::=α, kids)

REDUCER(p⁻, state, A::=α, kids) :
    rulenode := GET-RULENODE(A::=α, kids)
    if ∃p ∈ active-parsers with state(p) = state then
        if there already exists a direct link link from p to p⁻ then
            ADD-RULENODE(treenode(link), rulenode)
        else
            n := GET-SYMBOLNODE(A, rulenode)
            add a link link from p to p⁻ with tree node n
            forall p' in (active-parsers − for-actor) do
                forall (reduce rule) ∈ ACTION(state(p' ), current-token) do
                    DO-LIMITED-REDUCTIONS(p', rule, link)
    else
        create a stack node p with state state
        n := GET-SYMBOLNODE(A, rulenode)
        add a link from p to p⁻ with tree node n
        add p to active-parsers
        add p to for-actor

DO-LIMITED-REDUCTIONS(p, A::=α, link) :
    forall p' for which a path of length(α) from p to p' through link exists do
        kids := the tree nodes of the links which form the path from p to p'
        REDUCER(p', GOTO(state(p' ), A), A::=α, kids)

SHIFTER :
    active-parsers := ∅
    create a term node n with token current-token
    forall <p⁻, state' > ∈ for-shifter do
        if ∃p ∈ active-parsers with state(p) = state' then
            add a link from p to p⁻ with tree node n
        else
            create a stack node p with state state'
```

add a link from p to p^- with tree node n
add p to *active-parsers*

GET-RULENODE(r, *kids*) :
return a rule node with rule r and elements *kids*

ADD-RULENODE(*symbolnode*, *rulenode*) :
add *rulenode* to the possibilities of *symbolnode*

GET-SYMBOLNODE(s, *rulenode*) :
return a symbol node with symbol s and possibilities { *rulenode* }

4.4 Example of a tree built by the parser

We parse the sentence "Id := Int * Int + Int" according to

S ::= Id := Exp (Grammar G_2)
Exp ::= Exp + Exp
Exp ::= Exp * Exp
Exp ::= Int

to give an example of the forest generated by the parser. This sentence is ambiguous according to grammar G_2.

The forest generated by the parser is given if Fig. 2. Rule nodes are in boxes, symbol nodes in circles and term nodes are just represented by their tokens. The parser uses two methods to compactify the forest generated, *subtree sharing* and *local ambiguity packing*. Both methods are in fact a direct consequence of the sharing of parse stacks already performed in generalized LR parsing.

- subtree sharing

  If two parsers are combined and act for a while as a single parser, they generate tree nodes on their common part of the parse stack. At the moment a reduction is performed that goes beyond the common part, the parser splits again. As a result, the tree nodes which where on the common part will be used in two different contexts. This is what happened to the subtrees at the bottom of Fig. 2 and is called *subtree sharing*.

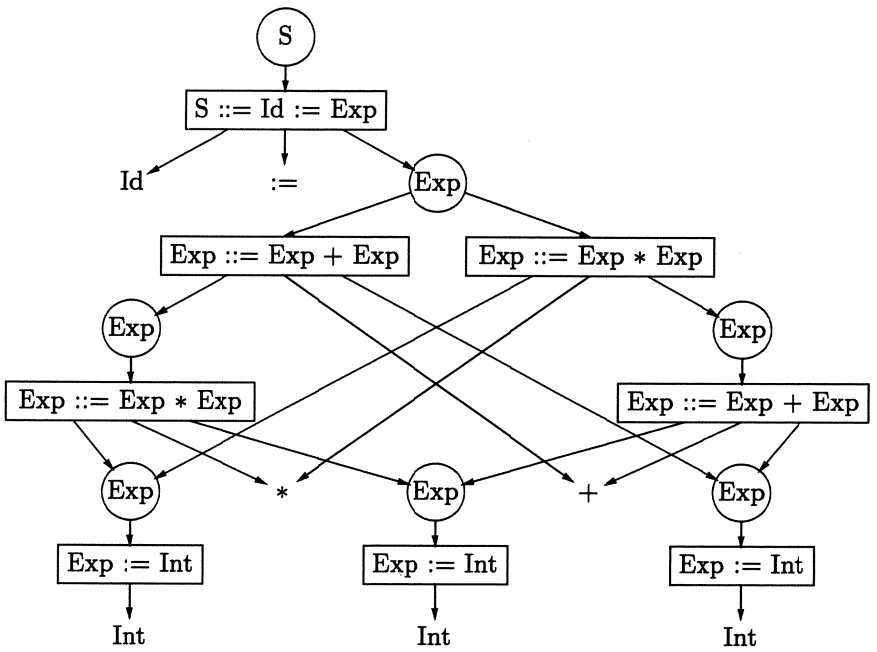


Figure 2: The (ambiguous) tree of "Id := Int * Int + Int"

- local ambiguity packing

  A sentence is said to have a local ambiguity if one of its proper subsentences can be reduced to the same non-terminal in two or more ways. If a sentence has many local ambiguities, the total number of ambiguities would grow exponentially. To avoid this, the top nodes of the subtrees that represent local ambiguities are merged and they are treated as a single node by the higher level nodes.

  Local ambiguity packing is performed by the routines REDUCER and ADD-RULENODE in the parsing algorithm. If there already exists a parser p in the state to go to, and p already has a link back to p^-, the newly found rule node can just be added to the symbol node associated to this link.[2] The highest "Exp" node in Fig. 2 is such a locally ambiguous point.

[2]The new rule node covers the same part of the input sentence as the other rule nodes in the symbol node, because a link between two stack nodes p and p^- describes what happened between the moment that p^- was top of the stack, and the moment that p is. This means that, if a new link between p and p^- is found, they cover the same part of the input.

Our trees contain more nodes than the trees in, for example, [1]. This is due to the fact that we use distinct nodes for symbols and rules. Symbol nodes with multiple outgoing edges represent ambiguity, while the outgoing edges of a rule node merely represent the arity of the rule. We consider a tree representation less clear if this kind of information must be guessed from the proximity of edges, as in Fig. 2-12 of [1]. And, with our representation, it is possible to obtain better sharing.

5 Improving the sharing in the parse forest

In the GLR parsing algorithm, the sharing in the parse forest is directly derived from that of the parse stacks. However, it might be that the parse table contains several states in which the reduction of the same rule is prescribed. In that case, these states are not shared in the parse stack, while the nodes generated by their reductions could be shared in the parse forest.

As an example, if we take grammar G_1 of Section 3.4, and use the parsing algorithm of Section 4.3 on the sentence "x b b b", the tree of Fig. 3(a) is generated. One would expect a tree like that of Fig. 3(b), however, with only a single node for the rule A ::= ϵ.

The first tree is, from the viewpoint of the grammar, a weird tree. Why is the node for ϵ-rule re-used at one point and not at another point? This can only be understood with the parse table of G_1 in mind, which contains two states with a reduction of the rule A ::= ϵ. Improved sharing in the parse tree would remove this generator dependent information and generates a more compact tree.

The sharing we propose is, again, an extension of the original Tomita algorithm. In that algorithm rule nodes do not appear as separate entities, and sharing cannot be performed easily. Furthermore, from the trees drawn in [16], it appears that Nozohoor-Farshi does not exploit this kind of sharing either. The sharing in the representation we propose is nearly as strong as that in the grammar representation of Billot and Lang [13], except that we do not allow sharing of the *tail* of a list of sons between different nodes. Billot and Lang generate nodes with maximally two subnodes in their grammar representation, and are thus able to achieve cubic space complexity.

We use the following two methods to improve the sharing of nodes in the parse tree:

- rule node sharing

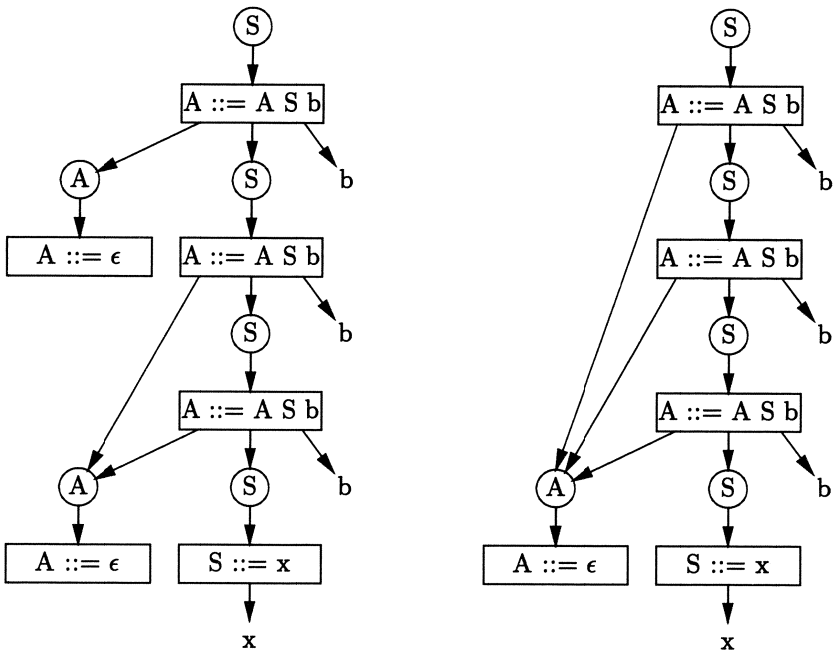


Figure 3: Two trees for "x b b b"

Check at the creation of a rule node whether a rule node with the same rule and children already exists. If so, re-use this rule node.

- symbol node sharing

  Check at the creation of a symbol node if there already exist a symbol node with the same symbol which covers the same part of the input. If so, re-use this symbol node.

To illustrate the effect of these two measures, we take the following (cyclic) grammar

S ::= S S
S ::= a
S ::= ϵ,

Grammar G_3

of which the LR(0) parse table is

state	transitions			reductions
	a	EOF	S	
0	shift 1		shift 2	reduce S ::= ϵ
1				reduce S ::= a
2	shift 1	accept	shift 3	reduce S ::= ϵ
3	shift 1		shift 3	reduce S ::= ϵ reduce S ::= S S

We use grammar G_3 to parse an empty sentence in four different ways: (a) with the GLR parsing algorithm as presented in Section 4.3, (b) with rule node sharing alone, (c) with symbol node sharing alone, and (d) with both methods of sharing applied. Fig. 4 shows the parse trees generated.

The tree of (a) clearly contains too many nodes. In (b) the rule nodes of (a) with the same rule and the same children have been combined, which removes 5 superfluous rule nodes. In (c) all symbol nodes of (a) were joined into one as they all contained the same symbol S, and covered the same ϵ-symbol. If the two methods are both applied, the tree shown in (d) is the result, which is the smallest and most natural representation of all possible parse trees of ϵ according to grammar G_3.

In order to realize this sharing, the parser has to remember the rule nodes and symbol nodes generated during the processing of the current input symbol.

Each node stores the frontier it covers in a tuple $< s, e >$, with s the position of the first token covered and e the position of the last one. This information can easily be propagated bottom-up during the generation of the parse tree. Term nodes created for a token at position i obtain $< i, i >$ as cover. Rule nodes obtain $< s, e >$ as cover, with s the start position of the first child of the rule node and e the end position of its last child. Symbol nodes inherit their cover from the rule node they are created for.[3]

ϵ-Rules form a problem in this scheme, as rule nodes for them do not have children. These rule nodes obtain an empty cover, with the consequence that symbol nodes may also get an empty cover. This means again that computing the frontier covered by rule nodes higher in the tree becomes slightly more complicated (see routine COVER for the actual implementation).

[3]Other rule nodes are only added to a symbol node if they cover the same frontier.

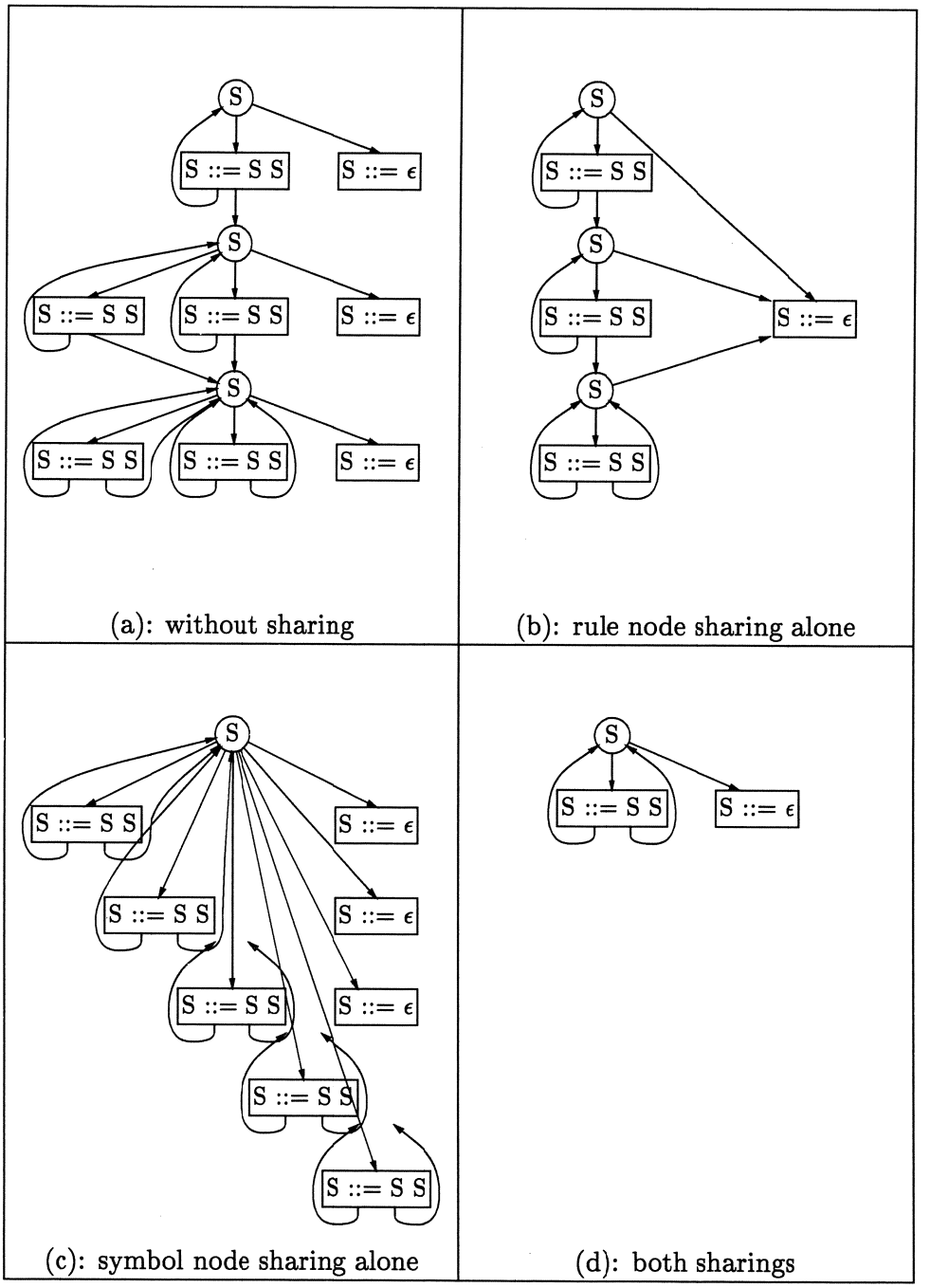


Figure 4: Four parse trees for ϵ according to grammar G_3

5.1 Algorithm of the parser with improved sharing

This is an extension of the algorithm of Section 4.3. The Lisp version of this GLR parsing algorithm can be found in [15, Appendix A.2].

The main difference with the algorithm of Section 4.3 is in routines GET-RULENODE and GET-SYMBOLNODE which try to re-use previously generated nodes. Also, all nodes in the parse tree contain a reference to the part of the frontier they cover. Finally, routine ADD-RULENODE has to check whether the rule node to add is not already contained in the symbol node. The differences with Section 4.3 are marked by a bar in the right margin.

```
PARSE(Grammar, a_1 ... a_n) :
   a_{n+1} := EOF
   global accepting-parser := ∅
   create a stack node p with state START-STATE(Grammar)
   global active-parsers := { p }
   for i := 1 to n + 1 do
      global current-token := a_i
      global position := i                                          |
      PARSEWORD
   if accepting-parser ≠ ∅ then
      return the tree node of the only link of accepting-parser
   else
      return ∅

PARSEWORD :
   global for-actor := active-parsers
   global for-shifter := ∅
   global rulenodes := ∅; global symbolnodes := ∅                  |
   while for-actor ≠ ∅ do
      remove a parser p from for-actor
      ACTOR(p)
   SHIFTER

ACTOR(p) :
   forall action ∈ ACTION(state(p), current-token) do
      if action = (shift state' ) then
         add <p, state' > to for-shifter
      else if action = (reduce A::=α) then
         DO-REDUCTIONS(p, A::=α)
      else if action = accept then
         accepting-parser := p
```

```
DO-REDUCTIONS(p, A::=α) :
  forall p' for which a path of length(α) from p to p' exists do
    kids := the tree nodes of the links which form the path from p to p'
    REDUCER(p', GOTO(state(p' ), A), A::=α, kids)

REDUCER(p⁻, state, A::=α, kids) :
  rulenode := GET-RULENODE(A::=α, kids)
  if ∃p ∈ active-parsers with state(p) = state then
    if there already exists a direct link link from p to p⁻ then
      ADD-RULENODE(tree node(link), rulenode)
    else
      n := GET-SYMBOLNODE(A, rulenode)
      add a link link from p to p⁻ with tree node n
      forall p' in (active-parsers - for-actor) do
        forall (reduce rule) ∈ ACTION(state(p' ), current-token) do
          DO-LIMITED-REDUCTIONS(p', rule, link)
  else
    create a stack node p with state state
    n := GET-SYMBOLNODE(A, rulenode)
    add a link from p to p⁻ with tree node n
    add p to active-parsers
    add p to for-actor

DO-LIMITED-REDUCTIONS(p, A::=α, link) :
  forall p' for which a path of length(α) from p to p' through link exists do
    kids := the tree nodes of the links which form the path from p to p'
    REDUCER(p', GOTO(state(p' ), A), A::=α, kids)

SHIFTER :
  active-parsers := ∅
  create a term node n with token token and cover <position, position>
  forall <p⁻, state' > ∈ for-shifter do
    if ∃p ∈ active-parsers with state(p) = state' then
      add a link from p to p⁻ with tree node n
    else
      create a stack node p with state state'
      add a link from p to p⁻ with tree node n
      add p to active-parsers

GET-RULENODE(r, kids) :
  if ∃n ∈ rulenodes with rule(n) = r and elements(n) = kids then
    return n
  else
    create a rule node n with rule r, elements kids and cover COVER(kids)
```

```
        add n to rulenodes
        return n

COVER(kids) :
    if kids = ∅ or ∀kid ∈ kids : cover(kid) = empty then
        return empty
    else
        begin := the start position of the first kid with a non-empty cover
        end := the end position of the last kid with a non-empty cover
        return <begin, end>

ADD-RULENODE(symbolnode, rulenode) :
    if rulenode ∉ the possibilities of symbolnode then
        add rulenode to the possibilities of symbolnode

GET-SYMBOLNODE(s, rulenode) :
    if ∃n ∈ symbolnodes with symbol(n) = s and
                             cover(n) = cover(rulenode) then
        ADD-RULENODE(n, rulenode)
        return n
    else
        create a symbol node n with symbol s,
                                 possibilities { rulenode } and
                                 cover cover(rulenode)
        add n to symbolnodes
        return n
```

6 Measurements

We use the syntax of Pascal to compare the efficiency of our GLR parsing algorithm with that of YACC and Earley's parsing algorithm.

In order to do so, we took the SDF definition of Pascal [2, appendix 2], and extracted the BNF definition generated by the implementation of SDF. This BNF definition is intended to be used in a syntax-directed editor; it is able to recognize any Pascal construct separately and allows holes in the input. By removing these extensions from the BNF definition, we obtained the grammar used in the measurements. This grammar contains 178 rules and allows complete Pascal programs only. The grammar is ambiguous, as priority declarations were used in the SDF definition to express the priority ordering of the Pascal operators, instead of coding the priority ordering in the grammar itself.

Using this grammar, we have compared the time needed to generate parse trees for Pascal programs up to three pages in length.

Measurements like these are easily influenced by factors not related to actual parsing; we have taken the following precautions to avoid these as much as possible.

- All measurements were performed on the same SUN SPARCstation 1.
- As input for the parsers, we used actual Pascal programs, in the form of streams of lexical tokens which were generated by a lexical scanner beforehand.
- These streams were all loaded into core before parsing started to avoid influences of the speed of the file system on the measurements.
- The time needed to *print* parse trees was not measured.

We compared implementations of the following parsing algorithms:

- GLR

  The implementation of the GLR parsing algorithm we used is the one in [15, Appendix A.2]. This implementation is written in LeLisp, and the code has been compiled with the LeLisp compiler "Complice" [18]. The parse table generator used is the incremental parser generator IPG [14], which generates LR(0) parse tables. IPG generates the needed parts of the parse table lazily, during parsing. To ensure that all needed parts of the parse table were present, we have parsed each input stream twice, and did only time the second parse.

- YACC [19]

  This is the standard parser generator available under Unix. YACC generates a parser and its LALR(1) tables in the form of a C program, which is subsequently compiled into machine code by a C-compiler. As YACC only allows non-ambiguous parse tables, we had to add disambiguation constructs to represent the priorities of Pascal expressions. This was not necessary for the two other parsing systems, which use the full, ambiguous, Pascal grammar. By adding these disambiguation constructs, we have solved 357 shift/reduce conflicts, leaving only a single conflict for the well known if-then-else ambiguity in Pascal. The actions associated with each rule build a tree representation of the input.

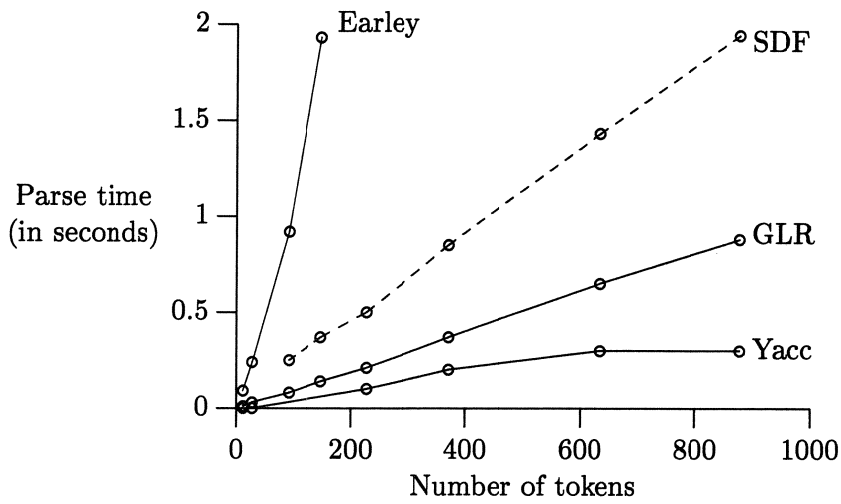


Figure 5: How different parsers perform on Pascal programs

- Earley [5]

  We have used an implementation of Earley's parsing algorithm written by Mark Freeley in Scheme[20]. As Scheme implementation we have used T, of which William Maddox remarks that "the code quality of the T compiler is among the best for any dialect of Lisp"[21]. Compiling the Earley parser resulted in a speed-up factor of about 20 compared to interpreted Scheme. Still, we have not been able to perform all planned measurements for Earley's algorithm, as long input sentences caused an apparently infinite number of garbage collections.

The results of the measurements are depicted in Fig. 5. They show that the GLR parser is about three times as slow as the YACC parser, which is mainly due to the following factors:

- The GLR algorithm is driven by LR(0) parse tables, versus the more sophisticated LALR(1) tables used by the YACC parser.
- The YACC parse tables did not contain conflicts, thanks to the disambiguation constructs that had to be added to the grammar. The GLR algorithm used parse tables that did contain conflicts, and had to build larger parse trees representing the ambiguities.
- The YACC parser is implemented in C, the GLR parser in LISP.

- The GLR method allows a larger class of grammars than YACC does. This leads to additional work during parsing.

Fig. 5 contains an additional line marked "SDF". This measurement serves to give an idea how the GLR algorithm performs within the SDF environment. In that case, the job to perform is extended with lexical scanning, solving priority conflicts, and the transformation of the parse tree into an abstract syntax tree. The grammar used in the SDF case allows incomplete programs too. This additional work about doubles the total execution time.

Fig. 5 also shows that the Earley algorithm performs quite badly on the larger input sentences, and would clearly be an undesirable choice to parse Pascal programs. It is, however, to be expected that the Earley algorithm will beat the GLR algorithm on highly ambiguous input sentences, as Earley has a worst upper bound of n^3, while GLR is exponential. To illustrate this, we measured the time needed by both algorithms to parse Pascal programs of the following form:

```
program A (input);
begin
  a := b {+ b}^i
end.
```

With i the number of +'s. As the Pascal grammar used contains a rule "Expression ::= Expression + Expression", these programs have a number of ambiguous parses which grows exponentially with i. This number, C_n, is called the Catalan number [22, p. 343-344], and is equal to:

$$C_n = \begin{cases} n = 0, 1: & 1 \\ n > 1: & \sum_{k=0}^{n-1} C_k C_{n-k-1} \end{cases} = \binom{2n}{n} \frac{1}{n+1}$$

Fig. 6 shows, for $i = 1, \ldots, 20$, the parse time taken by both algorithms and the number of ambiguous parses, C_i. This measurement confirms our expectation that the GLR algorithm generally performs better than the Earley algorithm, but loses on highly ambiguous sentences (in this example: containing more than 10^7 ambiguities).

Combining the results of the two measurements, we conclude that the GLR algorithm is a good choice for "near-LR" grammars. For these grammars it parses nearly as efficiently as YACC does, while it is able to handle ambiguous sentences reasonably well. If input sentences become highly ambiguous however, the Earley algorithm would be a better choice. If the

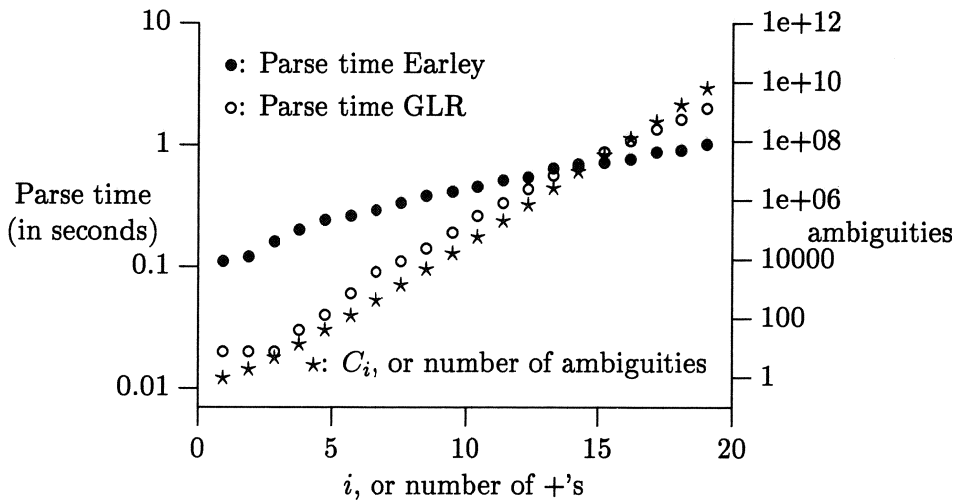


Figure 6: How different parsers perform on highly ambiguous programs

grammars are known to be in the LALR(1) class, it would, obviously, be more appropriate to use YACC.

7 Conclusions

The Generalized LR parsing algorithm covers the full range from LR grammars to general context-free grammars with acceptable efficiency. At both ends of this range it might however be advisable to use specialized algorithms, like, respectively, YACC and Earley's algorithm. Another advantage of the GLR algorithm is that it allows using the very simple LR(0) parse table generation algorithm.

Our contributions to the GLR algorithm are the following

- we present the algorithm in clear pseudo-code, which should be easy to translate to any programming language,
- we have extended the algorithm to the full class of context-free grammars,
- and we have improved the sharing in the parse forest.

Acknowledgements

I would like to thank the following people who all, in one way or another, have contributed to this paper: *Geoffrey Falk, Jan Heering, Paul Hendriks, William Maddox, Paul Klint, Bernard Lang, Mark Lankhorst, Emma van der Meulen, Rahman Nozohoor-Farshi, Frank Tip, Thomas Schoebel, Klaas Sikkel.*

References

[1] M. Tomita. *Efficient Parsing for Natural Languages.* Kluwer Academic Publishers, 1985.

[2] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[3] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[4] M.A. Harrison. *Introduction to Formal Language Theory.* Addison-Wesley, 1978.

[5] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[6] K. Sikkel. Cross-fertilization of Earley and Tomita. Memoranda Informatica 90-69, University of Twente, P.O. Box 217, 7500 AE, the Netherlands, November 1990.

[7] B. Lang. Deterministic techniques for efficient non-deterministic parsers. In J. Loeckx, editor, *Proceedings of the Second Colloquium on Automata, Languages and Programming*, volume 14 of *Lecture Notes in Computer Science*, pages 255–269. Springer-Verlag, 1974.

[8] H. Tanaka and N. Numazaki. Parallel generalized LR parsing based on logic programming. In *Proceedings of the International Workshop on Parsing Technologies (IWPT'89)*, pages 329–338, Pittsburgh, 1989.

[9] H. Numazaki and H. Tanaka. A new parallel algorithm for generalized LR parsing. In *13th. International Conference on Computational Linguistics (COLING'90)*, volume 2, pages 304–310, Helsinki, 1990.

[10] K. Sikkel. Bottom-up parallelization of Tomita's algorithm. In *Workshop on Tomita's Algorithm – Extensions and Applications*, pages 99–109. University of Twente, Computer Science Department, P.O. Box 217, Enschede, The Netherlands, 1991.

[11] A. Nijholt. The parallel approach to context-free language parsing. In U. Hahn and G. Adriaens, editors, *Parallel Natural Language Processing*, Norwood, N.J., 1991. Ablex Publishing Co.

[12] M.M. Lankhorst. An empirical comparison of generalized LR tables. In *Proceedings of the workshop on Tomita's Algorithm - Extensions and Applications*, pages 92–98. University of Twente, Computer Science Department, P.O. Box 217, Enschede, The Netherlands, 1991.

[13] S. Billot and B. Lang. The structure of shared forests in ambiguous parsing. In *Proceedings of the Twenty-Seventh Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 1989.

[14] J. Heering, P. Klint, and J. Rekers. Incremental generation of parsers. *IEEE Transactions on Software Engineering*, 16(12):1344–1351, 1990.

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

[16] R. Nozohoor-Farshi. Handling of ill-designed grammars in Tomita's parsing algorithm. In *Proceedings of the International Parsing Workshop '89*, pages 182–192, 1989.

[17] F. Harary. *Graph theory*. Addison-Wesley, 1969.

[18] INRIA, Rocquencourt. *LeLisp, Version 15.21, le manuel de référence*, 1987.

[19] S.C. Johnson. *YACC: yet another compiler-compiler*. Bell Laboratories, 1986. UNIX Programmer's Supplementary Documents, Volume 1 (PS1).

[20] K.R. Dybvig. *The Scheme Programming Language*. Prentice Hall, 1987.

[21] William Maddox. Personal communication, June 1991.

[22] R.L. Graham, D.E. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-Wesley, 1989.