Parser Generation for Interactive Environments

Jan Rekers

University of Amsterdam, 1992

Parser Generation for Interactive Environments

Academisch Proefschrift

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam,
op gezag van Rector Magnificus
Prof. dr. P.W.M. de Meijer,
in het openbaar te verdedigen in de Aula der Universiteit
(Oude Lutherse kerk, ingang Singel 411, hoek Spui),
op 31 januari 1992, des namiddags te 15.00 uur,

door

Joan Gerard Rekers

geboren te Amsterdam

Promotor: prof. dr. P. Klint.
Faculteit: Wiskunde en Informatica.

Notes

Partial support for this research was received from the European Communities under ESPRIT project 348 (GIPE - Generation of Interactive Programming Environments) and under ESPRIT project 2177 (GIPE II - Generation of Interactive Programming Environments II).

Partial support for this research was also received from the Netherlands Organization for Scientific Research – NWO, project *Incremental Program Generators*.

Parts of chapter 1 — Generalized LR Parsing — have been published as [BHK89, chapter 8]. These parts are reprinted with the kind permission of Addison-Wesley.

Chapter 2 — Incremental Parser Generation — is joint work with J. Heering and P. Klint. It has been published as [HKR89] and as [HKR90] and is reprinted with the kind permission of IEEE.

Chapter 3 — Restricting a Parser to a Subgrammar — is a major revision of [Rek89].

Chapter 4 — Substring Parsing — is joint work with Wilco Koorn. It has earlier been published as [RK91a] and as [RK91b].

Chapter 6 — Implementation of SDF — has, in an earlier form, appeared as part of deliverable D1.3 of the 3$^{\text{rd}}$ review report of the GIPE II project.

Cover design by Okkie Mathijsen.

Acknowledgements

I am very grateful to Paul Klint, my promotor. He always stimulated me and pushed me in the right direction. Because of the trust he had in me from the start, this dissertation was brought to completion. I also want to thank Jan Heering for his many contributions and his endless patience with my English.

Working at CWI was very pleasant, especially because of the kind colleagues I had. Without listing them all, I would like to mention Paul Hendriks, Emma van der Meulen, Wilco Koorn, Arie van Deursen, Pum Walters, Jasper Kamperman, and Frank Tip. I would also like to add Dominique Clémènt (from SEMA-METRA) to this list.

CWI, with its ample modern equipment and technical support, provided an excellent research environment. I have enjoyed to implemement my research on modern workstations, formatting my thesis was easy, and communicating with people all over the world through Mail and News has had a very positive influence on my work.

I would like to thank the referees for their willingness to review this thesis: Prof. Dr. J.A. Bergstra, Prof. Dr. F.E.J. Kruseman Aretz, Prof. Dr. B. Lang, Prof. Dr. A. Nijholt, and Prof. Dr. R.J.H. Scha.

I also want to thank some people for the support they gave in my private life. Monique Megens, although sometimes reluctant, certainly contributed to my work. Apart from her, I especially want to thank my father, for whom I started doing research in the first place, my mother, and my sisters Claar and Manès. Finally, I want to thank Okkie Mathijsen, who gave this thesis its nice jacket.

Opdracht

Voor mijn vader, die me liet zien hoe leuk wetenschap kan zijn.

Voor Monique, die me gelukkig nooit enig "*growing neglect during the work on this thesis*" heeft toegestaan.

Contents

Overview

The work described in this thesis has been performed in the context of the Esprit projects GIPE and GIPE II (Generation of Interactive Programming Environments). The goal of these projects is to develop a system which is able to generate interactive programming environments from formal language definitions. One of the objectives was to design a powerful and easy-to-use formalism for the definition of syntax. The resulting syntax definition formalism SDF has the following properties:

- lexical, context-free and abstract syntax are defined simultaneously,
- SDF supports general context-free grammars,
- it has powerful disambiguation constructs,
- it has list constructs,
- it supports modular grammar definitions,
- it can easily be coupled to semantic formalisms in order to provide them with user-definable syntax,
- its implementation is fully incremental, and
- it provides all information needed by a syntax-directed editor for the language described.

This thesis describes how the problems in the implementation of SDF, related to parsing and parser generation, have been solved. Although the prime motivation for this research was to implement SDF, we attempt to remain as general as possible and we deal in most cases with BNF grammars only.

All algorithms described in this thesis are also provided in pseudo code. This facilitates translation of the algorithms into a real programming language. In addition to this, appendix A contains versions of the algorithms in LISP which are available via electronic mail, as well. This allows experiments to be performed with the algorithms without any implementation effort.

The parsing algorithm

Chapter 1 – *Generalized LR Parsing* – deals with choosing a suitable parsing algorithm. This parser should accept general context-free grammars, and should be as efficient as an ordinary LR parser on LR(1) grammars. We have selected a Generalized LR (GLR) method as the basis for our syntactic tools. The theoretical framework for GLR parsing was introduced by Lang [Lan74], and worked out for LR parsing by Tomita [Tom85]. Our contribution is that we extended it to the full class of general context-free grammars and that we improved the sharing in the parse forest.

In fact, we have been quite fortunate in that our investments in the GLR algorithm remained of value in the sequel of the project, in which more and more elaborate parser generation schemes were developed, which were not foreseen at the time the GLR method was selected.

Parser generation

The GLR parser needs LR parse tables. A generator for these parse tables is straightforward, as the GLR algorithm works quite well with simple LR(0) parse tables. However, we do not only want to generate interactive programming environments, but we also wish to provide facilities for interactive grammar development. As a consequence the parser generator should be incremental.

In chapter 2 – *Incremental Parser Generation* – we describe a lazy and incremental parser generator IPG:

- The parser is generated in a lazy fashion from the grammar. There is no separate parser generation phase, but the parser is generated by need while parsing input. If typical input sentences need only a small part of the grammar, a faster response is achieved than in the greedy case: the parser generation phase does not introduce a noticeable delay and parsing can start immediately. If the input sentences do not

use the whole grammar, work is saved on the generation process as a whole. It turns out that in comparison with conventional techniques, the overhead introduced by this lazy technique is small.

- The parser generator is incremental. A change in the grammar produces a corresponding change in the already generated parser. Parts of the parser that are not affected by the modification in the grammar are re-used. Hence, the effort spent in generating them is re-used as well.
- The efficiency of the parsing process itself remains unaffected, in the sense that once all required parts of the parser have been generated, the parser will be as efficient as a conventionally generated one.

A similar technique (for the more limited class of LALR(1) grammars) has been proposed by Horspool [Hor89, Hor90].

Modular grammars

Not only the syntax, but also the *semantics* of programming languages need to be defined. To this end, the algebraic specification formalism ASF has been developed in the GIPE project. The main property of ASF, in relation to the work described in this thesis, is that it is a modular formalism. This means that by combining ASF and SDF (resulting in the ASF+SDF formalism), SDF has to become modular as well. This introduces the question of how to generate parsers for modular grammar definitions.

A modular grammar consists of a number of grammar modules each containing a set of grammar rules and a set of names of other modules to be imported. Each module defines a (possibly incomplete) grammar, which has to be completed by the rules in the imported modules. A modular grammar consisting of n modules thus defines n ordinary grammars. In most cases, these grammars will have large parts in common. If the parsers defined by these modules are all needed, n parsers will have to be generated.

It is, of course, possible to use a non-modular parser generation technique to generate these n parsers. This would, however, induce much duplicate generation effort for the common parts of the grammars. Furthermore, a modification in a module at the bottom of the import hierarchy would cause many parsers to be invalidated.

The obvious approach to parser generation for modular grammars would be to generate an incomplete parser for the rules in each module and translate

the import relation between modules to an import relation between parsers. This solution, however, rules out all optimizations available in the LR parsing technique of chapter 1 and 2, as these optimizations are based on knowledge of the complete grammar.

In chapter 3 – *Restricting a Parser to a Subgrammar* – we introduce a technique for restricting a parser to a subgrammar of the grammar it was generated for. The resulting parser behaves like a parser specially generated for the subgrammar, but making the restriction is much cheaper than generating a specific parser.

By means of this technique we are able to solve the problem of generating parsers for modular grammars. We do this by using IPG (chapter 2) to generate *one* parser for the union of all grammar rules of all modules, and restrict this parser n times according to the n grammars defined by the modules. In this way, no duplicate generation work is done, modifications are processed incrementally, and the generated parsers are reasonably efficient. A drawback of this approach is that it is not possible to develop parsers separately and combine them later on. However, this limitation is not too severe for the grammar development system envisaged.

Other approaches to parser generation for modular grammars are reported in [Voi86, Kos90]. To our knowledge the restricted parsing technique itself has never been proposed in the literature. Klint applied the same idea to scanner generation [Kli91a].

Substring parsing

Chapter 4 – *Substring Parsing* – addresses the problem whether a string can be a substring of some sentence in a language. The proposals for substring parsing reported in the literature [Cor89, Ric85] only work for a limited class of grammars and with specially generated parse tables. Our substring parser is based on general context-free grammars and uses the same parse tables as the original parser.

Substring parsing could be used to support incremental parsing in a syntax-directed editor, but we finally decided not to do so for reasons of efficiency. Substring parsing can also be used for noncorrecting syntax error recovery: if an ordinary parser detects a syntax error on some symbol, the substring parser can be started on the next symbol to discover additional syntax errors.

SDF

Now that all basic problems have been solved, we proceed with SDF itself. Chapter 5 – *From BNF to SDF* – contains an introduction to writing SDF definitions and describes the development of an SDF definition for a subset of Pascal. Our main points of interest are the modular decomposition of the grammar, the readability of the definition and the behaviour of the editor generated from it. SDF has been introduced in [HHKR89] and several SDF definitions have been published, but a tutorial on how to design an SDF definition did not yet exist.

Chapter 6 – *An Implementation of SDF* – describes the implementation of SDF itself. The purpose of this chapter is to document the current implementation, to guide programmers who have to deal with it, and to give an impression of the software infrastructure still needed to ensure proper operation of the underlying algorithms.

At the time of finishing this thesis, the GIPE group has successfully implemented a system for interactive development of specifications in the ASF+SDF formalism. When specifying a programming language, the system incrementally generates a programming environment for it. The semantic features of the ASF+SDF system have not been addressed in this thesis, but many of the syntactic features in ASF+SDF are the result of the research described here. How the implementation of SDF fits in the ASF+SDF system is described in [Kli91b, Hen91].

Chapter 1

Generalized LR Parsing

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.

1.1 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.

We have selected a Generalized LR (GLR) method as the basis for our syntactic tools. This algorithm was originally developed by Tomita [Tom85]. 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, [HHKR89]), 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.

1.2 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?

1.2.1 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 [HU79, 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 [Har78, 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 [HHKR89, 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.

1.2.2 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 [Ear70]

  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 [Tom85]

  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 [Sik90].

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 [Lan74], 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.

Another interesting approach to general context-free parsing is recursive ascent parsing [KA88, Lee91], which should beat both the Earley and the Tomita parsing algorithm in speed. We have not investigated this technique into any depth, however.

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 [TN89, NT90]. A more successful attempt to parallelize Tomita's algorithm has been performed by Sikkel [Sik91]. 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 [Nij91] for a general overview of parallel parsing algorithms.

1.2.3 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 [Lan91] and [BL89], 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 [Lan91], 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 [Chapter 2 of this thesis] and to parser generation for modular grammars [Chapter 3 of this thesis].

1.3 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.

1.3.1 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 [NF89]. 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 1.4.1) 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 [NF89] for the full explanation of this extension, which allows the GLR recognizer to handle the full class of context-free grammars.

1.3.2 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 [Appendix A.1 of this thesis].

```
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
```

1.3.3 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

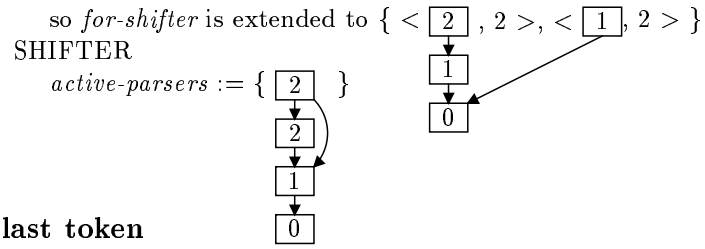
current-token := S
PARSEWORD
ACTOR([0])
ACTION(0, S) = { shift 1 }, so *for-shifter* := { < [0] , 0 > }
SHIFTER
active-parsers := { [1] → [0] }

The second token

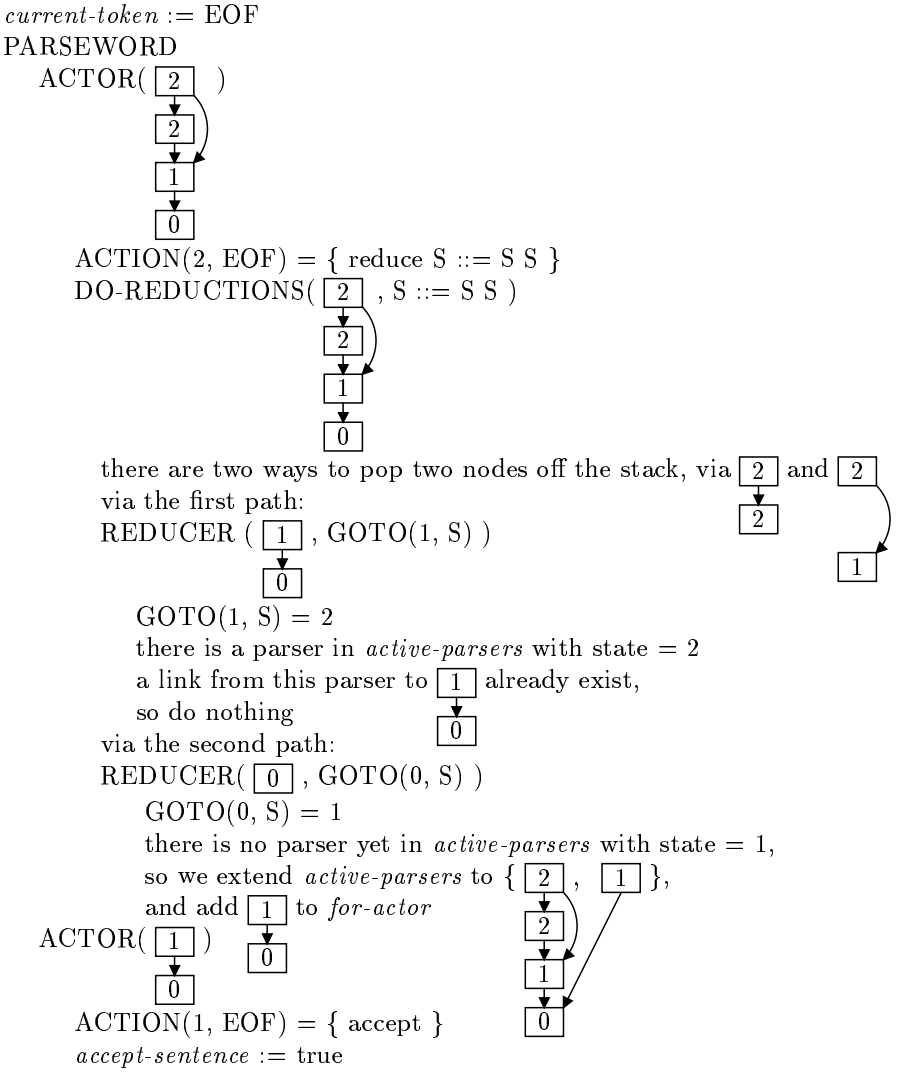
current-token := S
PARSEWORD
ACTOR([1] → [0])
ACTION(1, S) = { shift 2 }, so *for-shifter* := { < [1] → [0] , 2 > }
SHIFTER
active-parsers := { [2] → [1] → [0] }

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] → [1] → [0] , 2 > },
the reduce action by DO-REDUCTIONS([2] → [1] → [0] , 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] → [0] , [1] → [0] },
and add [1] → [0] to *for-actor*
ACTOR([1] → [0])
ACTION(1, S) = { shift 2 }



The last token



Finally

return true

1.3.4 Cycles in the parse stack

The graph of stack nodes, as generated by the recognizer of section 1.3.2 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:

state	transitions					reductions
	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 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. 1.1(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 $\boxed{0}$, and the other that goes over the cycle and ends $\boxed{2}$. Pushing the states GOTO(0, S) and GOTO(2, S) on both stack nodes, leads to the

[1] For clarity, we have annotated the links in Fig. 1.1 with symbols, while these are actually not present in the algorithm.

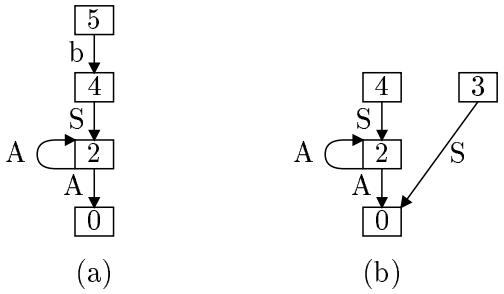


Figure 1.1: The parse stack before and after reduction of S ::= A S b

graph of stack nodes as in Fig. 1.1(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.

1.4 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 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.

1.4.1 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.

1.4.2 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 [Har69, 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.

1.4.3 Algorithm of the parser

The algorithms of the recognizer (section 1.3.2) 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) :
```

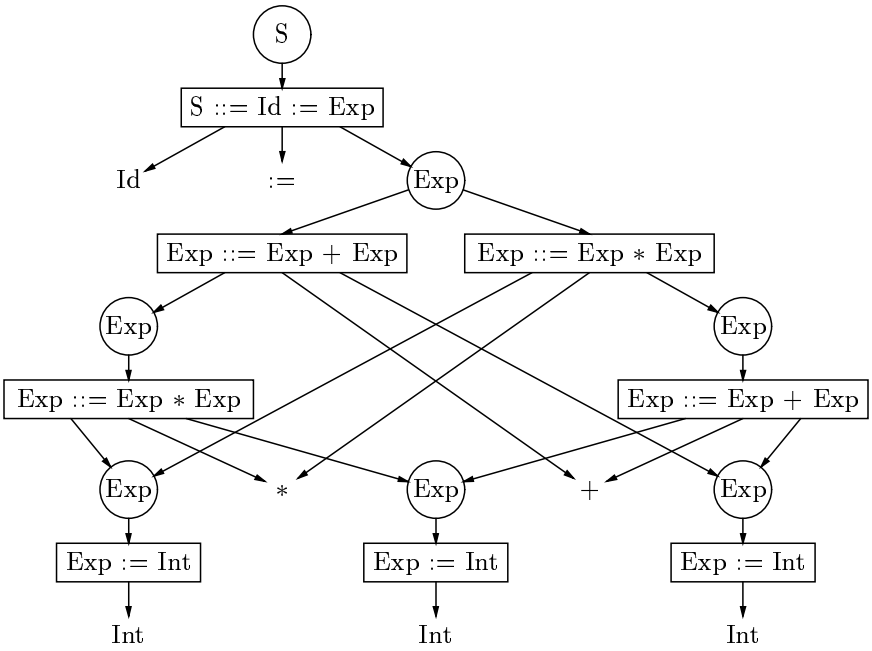


Figure 1.2: The (ambiguous) tree of "Id := Int * Int + Int"

add *rulenode* to the possibilities of *symbolnode*

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

1.4.4 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. 1.2. 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. 1.2 and is called *subtree sharing*.

- 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. 1.2 is such a locally ambiguous point.

Our trees contain more nodes than the trees in, for example, [Tom85]. 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 [Tom85]. And, with our representation, it is possible to obtain better sharing.

[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.

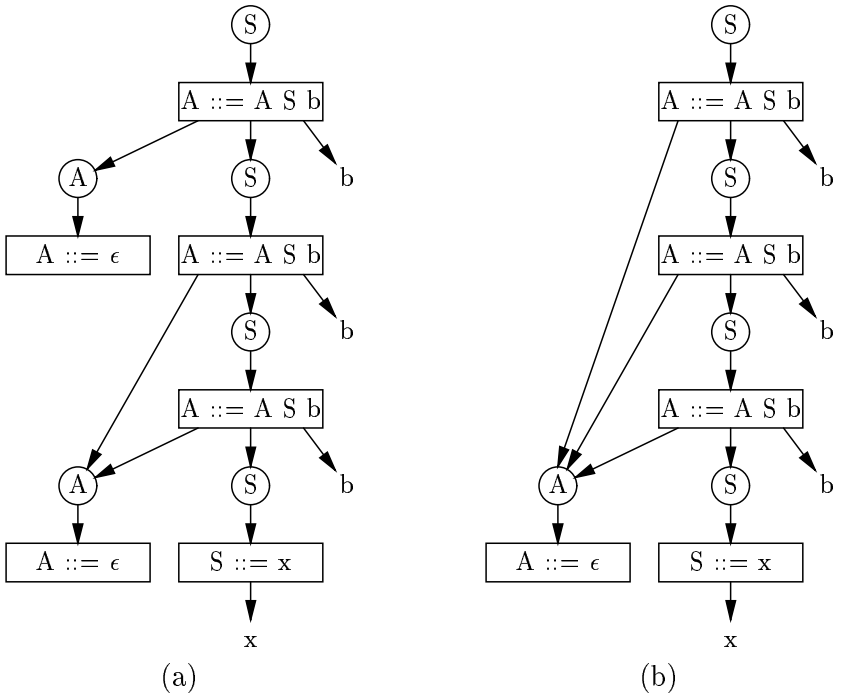


Figure 1.3: Two trees for "x b b b"

1.5 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 1.3.4, and use the parsing algorithm of Section 1.4.3 on the sentence "x b b b", the tree of Fig. 1.3(a) is generated. One would expect a tree like that of Fig. 1.3(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 [NF89], 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 [BL89], 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

  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 1.4.3, (b) with rule node sharing alone, (c) with symbol node sharing alone, and (d) with both methods of sharing applied. Fig. 1.4 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).

1.5.1 Algorithm of the parser with improved sharing

This is an extension of the algorithm of Section 1.4.3. The Lisp version of this GLR parsing algorithm can be found in [Appendix A.2 of this thesis].

The main difference with the algorithm of Section 1.4.3 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 1.4.3 are marked by a bar in the right

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

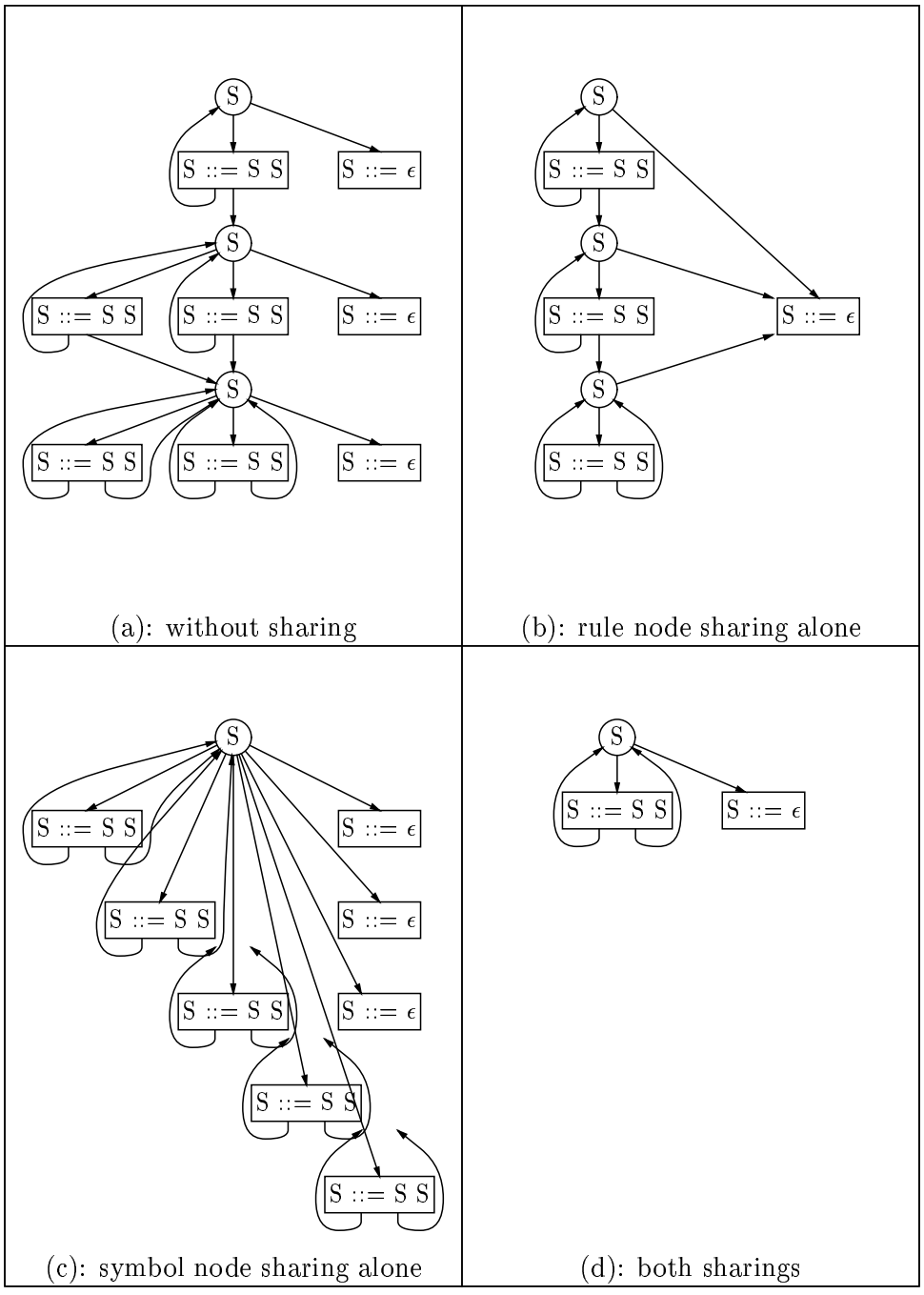


Figure 1.4: Four parse trees for ϵ according to grammar G_3

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
```

1.6 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 [HHKR89, 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 [Appendix A.2 of this thesis]. This implementation is written in LeLisp, and the code has been compiled with the LeLisp compiler "Complice" [LeL87]. The parse table generator used is the incremental parser generator IPG [Chapter 2 of this thesis], 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 [Joh86]

  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.

- Earley

  We have used an implementation of Earley's parsing algorithm written by Mark Freeley in Scheme[Dyb87]. 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"[Mad91]. 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.

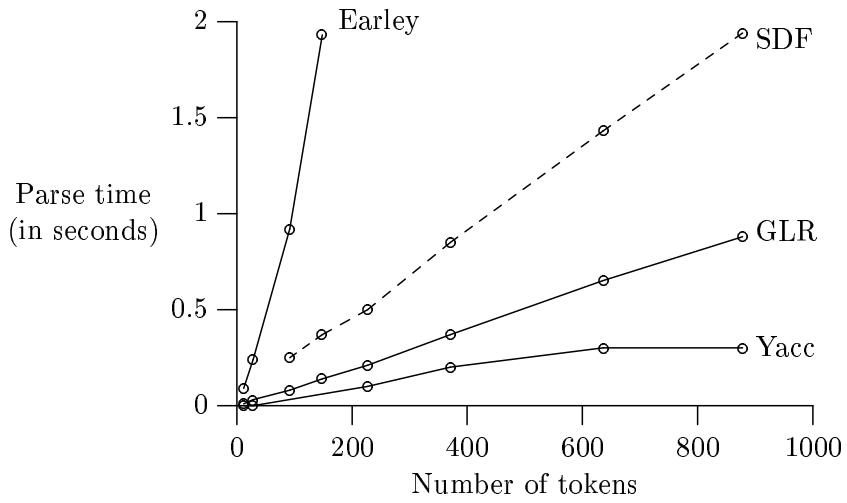


Figure 1.5: How different parsers perform on Pascal programs

The results of the measurements are depicted in Fig. 1.5. 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. 1.5 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. 1.5 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 [GKP89, 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. 1.6 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 grammars are known to be in the LALR(1) class, it would, obviously, be more appropriate to use YACC.

1.7 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

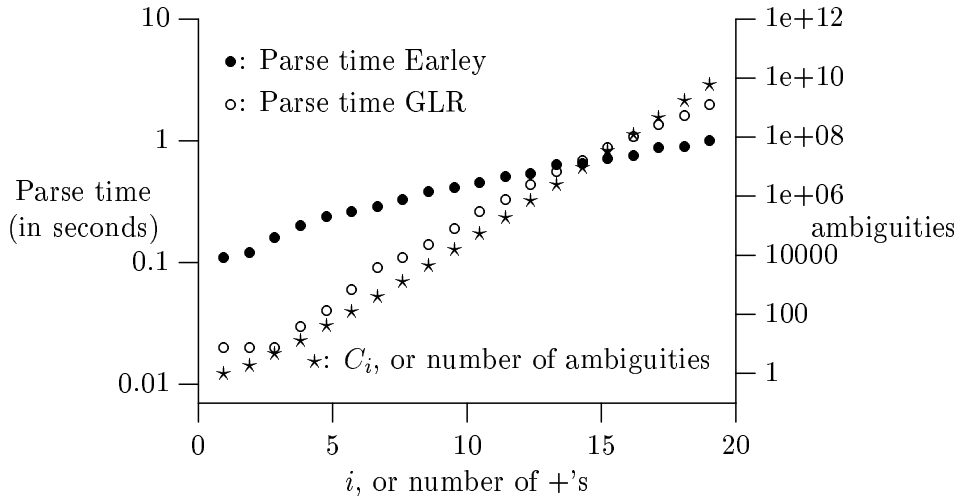


Figure 1.6: How different parsers perform on highly ambiguous programs

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.

Chapter 2

Incremental Parser Generation

An LR-based parser generator for arbitrary context-free grammars is described that generates parsers by need and handles modifications to its input grammar by updating the parser it has generated so far. The need for these techniques is motivated in the context of interactive language definition environments. We present all required algorithms, and give measurements comparing their performance with that of conventional techniques.

2.1 Introduction

The design of parser generators is usually based on the assumption that the generated parsers are used many times. If this is indeed the case, a sophisticated, possibly inefficient, parser generator can be used to generate efficient parsers. There are applications, however, to which this assumption does not apply:

- When a language is being designed, its grammar is not yet completely fixed. After each change of the grammar, a (completely) new parser must be generated, but there is no guarantee that it will be used sufficiently often. Three observations can be made here:
  - The time needed to parse the input is determined by the efficiency of both the parser and the parser generator.

  - Some parts of the grammar may not be needed by any of the sentences actually given to the parser; the effort spent on such parts by the parser generator is wasted.
  - In general only a small part of the grammar is modified. One would like to exploit this fact by making a correspondingly small modification to the parser, rather than generating an entirely new one.

- There is a trend towards programming/specification languages that allow general user-defined syntax (LITHE [San82], OBJ [FGJM85], Cigale [Voi86], ASF+SDF [BHK89]). In such languages each module defines its own syntax, and each import of a module extends the syntax of the importing module with the (visible) syntax of the imported module. For efficient parsing and syntax-directed editing of these languages, it is of great importance to use a parser generator that can handle a large class of context-free grammars, and that can incorporate modifications of the grammar in the parser incrementally.

We describe a lazy and incremental parser generator IPG, which is specially tailored towards the highly dynamic applications sketched above:

- The parser is generated in a lazy fashion from the grammar. There is no separate parser generation phase, but the parser is generated *by need* while parsing input. If typical input sentences need only a small part of the grammar, a faster response is achieved than in the greedy case: the parser generation phase does not introduce a noticeable delay and parsing can start immediately. If the input sentences do not use the entire grammar, work is saved on the generation process as a whole. It turns out that in comparison with conventional techniques, the overhead introduced by this lazy technique is small.

- The parser generator is incremental. A change in the grammar produces a corresponding change in the already generated parser. Parts of the parser that are not affected by the modification in the grammar are re-used. Hence, the effort spent in generating them is re-used as well. This has clear advantages for interactive language definition systems.

- The efficiency of the parsing process itself remains unaffected, in the sense that once all required parts of the parser have been generated, the parser will be as efficient as a conventionally generated one.

- The parsing algorithm is capable of handling general context-free grammars, inclusive ambiguous grammars.

For a description of the general principles underlying our method, see [HKR91]. In [HKR87] a lazy/incremental *lexical scanner* generator ISG is described. The combination ISG/IPG is used in an interactive development environment for the ASF+SDF specification language mentioned above. The universal syntax-directed editor of this environment is parametrized with a grammar written in SDF [HHKR89], and uses ISG/IPG as its parsing component. The response time of the editor is acceptable, even though the lexical scanner and the parser are generated and modified on the fly during editing.

In Section 2.2 we discuss related algorithms and explain how our technique evolved from them. In Section 2.3 we present an LR parser and a conventional LR(0) parser generation algorithm. We extend this into a lazy parser generation algorithm in Section 2.4. In Section 2.5 we extend it once again into an incremental parser generation algorithm. Finally, Section 2.6 gives the results of efficiency measurements, and Section 2.7 contains some concluding remarks.

2.2 Choosing a parsing algorithm

We compare some existing parsing algorithms with our own algorithm from the perspective of highly dynamic applications like the ones discussed in the previous section:

- LR(k) and LALR(k) algorithms [ASU86, chapter 4.7]

  These algorithms are controlled by a parse table that is constructed beforehand by a table generator. The table is constructed top-down, whereas the parser itself works bottom-up. The parser works in linear time. When the look-ahead k is increased, the class of recognizable languages becomes larger (but will always be limited to non-ambiguous grammars), and the table generation time increases exponentially. With conventional LR or LALR table generation algorithms it is difficult to update an already generated parse table incrementally if the grammar is modified (see below).

- Recursive descent and LL(k) algorithms [ASU86, chapter 4.4]

A recursive descent parser generator builds a parsing program, whereas an LL generator builds a parse table that is interpreted by a fixed parser. In both algorithms the parsers work top-down. The class of accepted languages depends on the look-ahead k, but is always limited to non-left-recursive, non-ambiguous grammars.

- Earley's general context-free parsing algorithm [Ear70]

  Earley's algorithm can handle all context-free grammars. It works by attaching to each symbol in the input a set of "dotted rules". A dotted rule consists of a syntax rule with a cursor ($\bullet$) in it and the position in the input where the recognition of the rule started. The set of dotted rules for symbol $n+1$ is computed at parse time from the set for symbol n. Earley's algorithm does not have a separate generation phase, so it adapts easily to modifications in the grammar. It is this same lack of a generation phase that makes the algorithm too inefficient for interactive purposes.

- Cigale [Voi86]

  Cigale uses a parsing algorithm that is specially tailored to expression parsing. It builds a *trie* for the grammar in which production rules with the same prefix share a path. During parsing this trie is traversed recursively. A trie can easily be extended with new syntax rules and tries for different grammars can be combined just like modules. The class of grammars is somewhat larger than LR(0) grammars, as the parser does not use look-ahead in a general manner and cannot backtrack.

- OBJ [FGJM85]

  OBJ uses a recursive descent parsing technique with backtracking. OBJ itself does not allow ambiguous grammars, but the backtrack-parser does detect all ambiguous parses. This makes the parsing system suitable for finitely ambiguous grammars, but as mentioned in [FGJM85, page 60] "parsing can be expensive for complex expressions", which makes the algorithm less suitable for large input sentences.

- Pseudo-parallel LR parsing [Lan74, Tom85]

  This is an extended LR parsing algorithm that requires a conventional (but possibly multi-valued) LR(0), LR(1) or LALR(1) parse table. The

parser starts as an LR parser, but when it encounters a multi-valued entry in the parse table (conventionally known as a table conflict), it splits up in several LR parsers that work in parallel. The theoretical framework for pseudo-parallel LR and LL parsing was introduced by Lang [Lan74]. It was optimized independently by Tomita for LR parsing [Tom85]. Grammars are restricted to the class of finitely ambiguous (or acyclic) context-free grammars. We discuss this algorithm in detail and extend it to general CF grammars in [Chapter 1 of this thesis]. As Tomita's parsing technique uses the same table generation phase as conventional LR algorithms, modifying the grammar is an expensive operation with this algorithm.

- Incremental parser generator IPG

  We developed this method on the basis of the Tomita parsing algorithm, but provided the algorithm with an incremental LR(0) parse table generator. Parsing starts with an empty parse table, which is expanded by need during parsing. A change in the grammar is handled incrementally by removing those parts of the parse table that are affected by the change; these parts are recomputed for the modified grammar when the parser needs them again. The parse table is constructed during parsing, so after a certain time, depending on the input given to it, the system will become as fast as a conventionally generated Tomita parser.

- Incremental LALR(1) parser generation [Hor89, Hor90]

  At the time we wrote this paper, a very similar approach was proposed independently by Horspool. His point of departure is a conventional LR parser rather than a parallel one and he considers incremental generation of LALR(1) parse tables. This is more difficult than incremental generation of LR(0) tables: look-ahead sets have to be taken into account whose incremental generation and modification turn out to be problematic. As a consequence, his system has a less efficient incremental table generation phase, but generates more efficient LALR(1) parsers. We opted for a more efficient LR(0) table generation phase at the expense of some loss in parsing efficiency for non-LR(0) languages (but without restricting the class of acceptable grammars in any way).

Fig. 2.1 assigns a rating to the above-mentioned algorithms for each of the following properties: capability of handling arbitrary context-free grammars

	powerful	fast	flexible	modular
LR(k), LALR(k)	−	+ +	− −	− −
recursive descent, LL(k)	−	+ +	− −	− −
Earley	+ +	− −	+ +	+ +
Cigale	− −	−	+ +	+ +
OBJ	+	− −	+	+
Tomita	+ +	+ +	− −	− −
IPG	+ +	+ +	+	+
Horspool	−	+ +	+	−

Figure 2.1: Comparison of various parsing algorithms

(powerful), efficiency on large input sentences (fast), possibility for processing of modifications of the grammar (flexible), and possibility for modular composition of parsers (modular).

2.3 LR parsing and parser generation

In this section we describe an LR parser, the associated parse tables, and an LR(0) parser generator. We assume the reader to be reasonably familiar with the subject. This section just serves to refresh the reader's memory.

The basic reference for LR parsing and parser generation is [ASU86]. [GJ90] and [KP90] are also interesting as these contain an up-to-date annotated bibliography of related algorithms.

2.3.1 LR parsing

We use a pseudo-parallel parsing algorithm, developed by Tomita [Tom85]. It runs several ordinary LR parsers in parallel and can handle arbitrary context-free grammars. To give an idea of how our parser works, we present a non-parallel LR parser. It maintains a parse stack which initially contains the start-state. The state on top of the stack is the current state of the parser. The parser repeatedly consults its parse table for actions to be performed in the current state and with the current input symbol. This is done by routines *ACTION* and *GOTO*. If there are several possible actions, it chooses one of them.

Push the *start-state* on the stack

```
while true do
    The current state of the parser is the state on top of the stack
    The current input symbol is the first symbol of sentence
    Choose an arbitrary action from those returned by ACTION
    if there is no action then
        Reject the sentence
    else
        if it is a shift action to a state then
            Push the current symbol and the new state on the stack
            Remove the current symbol from the head of sentence
        elseif it is a reduce action of rule A ::= β then
            Replace β on the stack by A
            Call GOTO for a new state
            Push the new state on the stack
        elseif it is an accept action then
            Accept the sentence
```

Provided with a parse table generated by an LR(0) parser generator, this algorithm yields unique parses for any LR(0) grammar, but it may fail for other grammars if it chooses the wrong action at any point at which *ACTION* returns multiple actions. The pseudo-parallel version of the algorithm [Tom85], [Chapter 1 of this thesis] explores all actions returned by *ACTION* by splitting in multiple parsers, one for each possibility. In this way it yields unique parses for any unambiguous grammar, and all possible parses for ambiguous grammars. So, this technique enables us to recognize the full class of context-free grammars, while using a simple LR(0) parse table generator.

We do not reject ambiguous sentences, but let the parsing algorithm return *all* possible parses. This leaves room for a postprocessor, such as the algorithm described in [HHKR89, section 6] which selects parse trees on the basis of priority declarations.

2.3.2 The parse table or graph of itemsets

The notion of *item* or *dotted rule* is basic to an understanding of the LR method. An item is a grammar rule with a dot in its right-hand side indicating how far the parse according to that particular rule has progressed. A set of items is an *itemset*. A parse table is a graph whose nodes are itemsets and whose (labeled) edges are transitions between itemsets. A state of the parser is an itemset in this graph. For example, the graph generated for the following grammar

Figure 2.2: Graph of itemsets for the Booleans

B ::= true (Grammar of the Booleans)
B ::= false
B ::= B or B
B ::= B and B
START ::= B

is given in Fig. 2.2. Fig. 2.3 shows the steps done by the parser when parsing the sentence "true or false".

Each itemset in the graph has the fields *kernel*, *transitions*, *reductions* and *type*. The *kernel* is a set of dotted rules that are recognized by the parser in the corresponding state. The *transitions* form the labeled edges of the graph; each itemset contains the transitions originating from it. The *reductions* are the rules that are recognized completely by the parser in the corresponding state. The *type* can be *initial* or *complete*. When *initial* the transitions and reductions have not yet been computed.

The *reductions* form a separate set of rules, but for the sake of compactness we represent them in the diagrams by underlining the corresponding dotted rules in the kernel. Fortunately, there are no rules of the form $A ::= \epsilon$ in the example, as these would introduce reductions without corresponding dotted rules in the kernel. Such reduction rules would need a different representation in the diagrams.

Parse stack	Input
0	true or false $
0 true 3	or false $
0 B 2	or false $
0 B 2 or 6	false $
0 B 2 or 6 false 4	$
0 B 2 or 6 B 8	$
0 B 2	$
0 START 1	$

Figure 2.3: Steps of the parser for input "true or false"

The parser moves through the graph of itemsets: shift actions cause the parser to move forward along a transition labeled with the current input symbol, while reduce actions first cause a move backward along the path stored on the parse stack, and then a move forward along a transition labeled with the non-terminal that was the result of the reduction.

ACTION and *GOTO* obtain their information from the graph in the following manner:

ACTION(*state*, *symbol*) :
  actions :=
    { reduce $A ::= \beta \mid A ::= \beta \in state.reductions$ } ∪
    { shift $state' \mid (symbol\ state') \in state.transitions$ } ∪
    { accept $\mid (symbol$ accept$) \in state.transitions$ }
  return *actions*

GOTO(*state*, *symbol*) :
  return $state'$: $(symbol\ state') \in state.transitions$

2.3.3 Parse table generation

The graph of itemsets is generated by the following LR(0) parse table generator:

PG(*Grammar*) :
  Grammar := *Grammar* ∪ { START′ ::= START}
  Generate a start-itemset with { START′ ::= • START} as kernel
  while there is an initial itemset **do**
    Complete it using *EXPAND*

return the start-itemset

The real generation work is done by *EXPAND*, which computes the transitions and reductions of an itemset. It starts by using *K-CLOSURE* to generate a new set *Closure* of dotted rules (which is an extension of the kernel) containing all rules that may become applicable in this state/itemset. *Closure* is then partitioned in subsets of rules having the same symbol S after the dot. On shifting S (or reducing to S), the parser will have advanced one step recognizing a rule in the subset associated with S. For each S the associated subset is transformed into a new kernel by moving the dot over S. When an itemset with that kernel does not yet exist, it is generated as an initial one. A transition to that itemset labeled with S is added to *transitions*. A rule in the extended kernel that ends on a dot has been recognized completely. This implies implies an accept or a reduce action.

EXPAND(*itemset*) :
  Closure ::= *K-CLOSURE*(*kernel*)
  for each distinct symbol S that occurs just after a dot in *Closure* **do**
    Generate a new kernel K consisting of all dotted rules in *Closure*
      which have their dot before S, and move the dot over S in each rule
    if there does not yet exist an itemset with kernel K **then**
      Generate an initial itemset with kernel K
    Add to *transitions* a shift action labeled with S to that itemset
  for each rule in *Closure* that ends on a dot **do**
    if it is START′ ::= START • **then**
      Add an accept action to *transitions*
    else
      Add a reduce action of the rule to *reductions*
  The *type* of the itemset is now set to "complete"

K-CLOSURE(*kernel*) :
  Closure := *kernel*
  while there is a rule in *Closure* with its dot before an S **do**
    Extend *Closure* with all rules that derive S
      and add a dot before their leftmost symbol
  return *Closure*

2.4 Lazy parser generation

The parser generation algorithm described so far generates the parser completely before it is used. This is a good method when a parser is generated only once for a stable grammar after which it is used relatively often.

In applications where the grammar is subject to modification, this approach causes the parse time of the first sentence to be effectively increased by the parser generation time. Clearly, it would be preferable to spread the generation time over the parsing of many sentences to obtain a better response time. Lazy parser generation has this property. It generates only those parts of the parser that are really needed to parse the sentences given to it. Both of these arguments in favor of lazy parser generation are valid only when typical input sentences need a relatively small part of the parser. See [HKR91] for an in-depth discussion of the advantages and disadvantages of lazy program generation. In our specific application, we use the lazy parser generation algorithm mainly as a step towards *incremental* parser generation (Section 2.5).

We have to adjust the LR(0) parser generator of the previous section only a little to obtain a lazy parser generator. We move the parser generation phase into the parsing phase by moving the expansion of initial itemsets from routine *PG* to *ACTION*. This means that the state with which *ACTION* is called can now be either complete or initial. When it is still initial, it is expanded first by *EXPAND*. The lazy parser generator LPG now generates only *start-itemset* as an initial itemset. The rest of the parser generation will be taken care of by *ACTION*.

```
LPG( Grammar) :
    Grammar := Grammar ∪ { START′ ::= START }
    Generate a start-itemset with { START′ ::= • START } as kernel
    return the start-itemset

ACTION(state, symbol) :
    if state.type = initial then EXPAND(state) fi
    actions :=
        { reduce A ::= β | A ::= β ∈ state.reductions } ∪
        { shift state′ | (symbol state′) ∈ state.transitions } ∪
        { accept | (symbol accept) ∈ state.transitions }
    return actions
```

Like *ACTION*, *GOTO* uses information that is only available in complete itemsets, so one might be inclined to think that the same test for initial itemsets has to be added to *GOTO* as well. Due to the the particular way in which the parsing algorithm works, however, *GOTO* will only be called with itemsets that have already been completed. The parser asks *GOTO* for information about a state when it reduces a rule. The parser obtains this state from its parse stack of previously visited states. The fact that the state

has been visited previously, implies that *ACTION* has already been called on it. During that call the state will have been expanded.

Consider, for example, the grammar of the Booleans of Fig. 2.2. The graph of itemsets generated by the lazy parser generator initially consists only of the start-itemset (with type *initial*) shown in Fig. 2.4. The fact that this itemset has type *initial* is expressed by a white disk. If an itemset has type *complete*, a black disk is used.

When the parser is given its first sentence, its first step will be to ask what actions it has to perform in *start-state*. Hence, *ACTION* is called with initial itemset *start-state* which will then be expanded to the graph shown in Fig. 2.5. Fig. 2.6 shows the graph of itemsets after the sentence "true and true" has been parsed.

The overhead in time introduced by this lazy technique is small, as only the test on the type of the itemset takes some extra time in *ACTION*. The use of memory increases, as the kernels are now needed during parsing as well.

2.5 Incremental parser generation

The lazy parser generator can react to modifications of the grammar only by throwing away the parser it has already generated and starting from scratch, which is rather wasteful.

In this section we describe an incremental parser generator that retains those parts of the old graph of itemsets that can still be used in the graph for the modified grammar. How much has to be thrown away depends not only on the "size" of the modification, but also on how much of the graph had already been generated for the old grammar. When the graph of itemsets is already highly specialized towards the old grammar, chances are that a relatively large part of it has to be removed.

2.5.1 An algorithm for incremental parser generation

The incremental parser generator retains only that part of the (possibly incomplete) graph that can still be used in the graph of itemsets for the new grammar. It does this by returning those itemsets in the graph that were (from the viewpoint of the new grammar) expanded incorrectly to their initial state. The lazy parser generator LPG will then, when needed, re-expand these itemsets in accordance with to the new grammar. The incremental

Figure 2.6: The graph of itemsets after “true and true” has been parsed

parser generator IPG consists of LPG plus routines *ADD* and *DELETE* which update an existing parser.

Suppose a rule $A ::= \beta$ is added to the grammar. We then have to find the states (itemsets) in which recognition of the new rule should start. In the new graph the *closure* of the kernel of these itemsets would contain $A ::= \bullet\beta$. How can we find these itemsets in the existing graph without recomputing the closure of every kernel? Initial itemsets can easily be dealt with because they do not have to be *re*-expanded, but complete ones present a problem. Fortunately, we can be sure that $A ::= \bullet\beta$ will be added to the closure only when the latter contains at least one dotted rule with its dot before an A. But if there was a rule with its dot before an A in the closure, *EXPAND* must already have added a transition for A to the transitions of the itemset in question. So we can recognize all complete itemsets that should have $A ::= \bullet\beta$ in the closure of their kernel by the presence of $(A\ itemset')$ in their transitions.

Similarly, if we *delete* a rule $A ::= \beta$ from the grammar, we have to find the states (itemsets) in the existing graph in which recognition of this rule started. These are the itemsets that had $A ::= \bullet\beta$ in the closure of their kernel. As in the case of addition, these are the complete itemsets having a transition $(A\ itemset')$ among their transitions.

These itemsets with a transition for A in their kernel, which are the first ones affected by the modification of the grammar, have to be re-expanded. This can be achieved simply by returning them to their initial state and let the lazy parser generator re-expand them when needed.

So routines *ADD* and *DELETE* are very simple: they just update the grammar and make all complete itemsets with a transition for A "initial" again. When the parser needs those itemsets again, they will be re-completed by the lazy parser generator in accordance with the modified grammar.

```
ADD(A ::= β) :
    Add A ::= β to the Grammar
    for each itemset with a transition on A do
        Return the itemset to its initial state

DELETE(A ::= β) :
    Delete A ::= β from the Grammar
    for each itemset with a transition on A do
        Return the itemset to its initial state
```

If, for example, the rule "B ::= nil" is added to the grammar of the Booleans, and the graph of itemsets for the grammar of Fig. 2.2 is updated

Figure 2.7: Graph for the Booleans after addition of "B ::= nil"

by *ADD*, the itemsets 0, 5, and 6 are returned to their initial state, because they had a transition for "B" among their transitions. the graph of itemsets is thus transformed into the unconnected graph of Fig. 2.7.

When the lazy parser generator now expands set 0 again, its former connections with 1, 2, 3, and 4 are re-established, and the initial itemset 9 is generated with kernel "B ::= nil •". The resulting graph is shown in Fig. 2.8.

2.5.2 Garbage collection

The incremental parser generator causes some itemsets to become unreachable from the start-state. As frequent modification of a grammar can produce many unreachable itemsets, the algorithm has to be extended with some kind of garbage collection. For the sake of efficiency, however, it is essential to retain unreachable itemsets for some time. Otherwise major part of the graph of itemsets would have to be regenerated (this would occur in the example of Fig. 2.7). Clearly, a compromise has to be found between removing unreachable itemsets immediately, and retaining them forever. To this end, we attach to each itemset a count of the number of itemsets referring to it. Routine *EXPAND* sets and increments the reference count of the

Figure 2.8: The graph of Fig. 2.7 after re-expansion of set 0

itemsets it creates transitions to. Furthermore, *ADD* and *DELETE* should return itemsets to a "dirty" rather than "initial" state. A dirty itemset is an initial itemset with a history (its former transitions). It is expanded in the same way as an initial set, but after its expansion the reference counts of those itemsets to which it no longer refers are decreased. When the reference count of a itemset becomes zero, it is removed. Using this method the removal of an unused itemset is postponed until the chance is better that it will not be used again.

2.6 Performance and efficiency

We have compared the efficiency of the lazy and incremental parser generator IPG with that of the non-incremental version PG of Section 2.3.3. We also compared IPG and PG with the LALR(1) parser generator Yacc [Joh86]. A comparison of IPG with Earley's parsing algorithm would have been appropriate here, because both systems recognize the same class of context-free grammars. As we did not have access to a good implementation of the algorithm, and a quick and mediocre implementation made by us would not be a fair match, we have not included such a comparison. From a theoretical viewpoint, we expect Earley's algorithm to have better

generation performance, but a much inferior parsing performance.

Both PG and IPG generate parse tables (or graphs of itemsets) that are interpreted by Tomita's context-free parsing algorithm. Since these are the only grammars accepted by Yacc, the test grammar had to be LR(1). The grammar we used is an LR(1) version of the grammar of the syntax definition formalism SDF [HHKR89]. The reason for choosing SDF is its reasonably sized grammar. The fact that it also happens to be the language in which grammars for PG and IPG have to be expressed is purely coincidental. It only means that the grammar of SDF has to be expressed in SDF itself to be acceptable to PG and IPG.

We measured the time in seconds CPU time used by the three parser generators and the generated parsers to:

- construct a parse table for SDF;
- parse an input sentence (SDF definition) twice;
- modify the grammar and reconstruct the parse table;
- parse the same sentence twice.

The measurements have been repeated on input texts of different length and complexity, namely four SDF definitions of which the smallest is 15 lines and the largest 142 lines long. The modification of the grammar consisted of the addition of a rule that extends the possible elements in the priority and function declarations of SDF. We added rather than deleted a rule in order to be able to use the same input sentences again after the modification. Other experiments showed that addition or deletion of a rule roughly takes the same amount of time.

To prevent the lexical scanner and the file system from influencing the measurements, the input of all parsers was a stream of lexical tokens already in memory, and the parsers constructed a parse tree but did not print it. All measurements have been carried out on a SUN 3/60 with a low workload (no swapping). Yacc generates C-code, which was compiled in 68020 machine code by the C-compiler. PG and IPG ran in the LeLisp environment and were compiled by the LeLisp compiler "Complice" [LeL87]. LeLisp garbage collections were only allowed *between* measurements.

The results of the measurements are given in Fig. 2.9. They show the following:

Figure 2.9: Efficiency measurements for Yacc, PG and IPG

- Yacc

  Yacc generates parsers that are about twice as fast as the parsers generated by PG and IPG, but the generation time for a Yacc parser is unacceptably high for an interactive language definition environment. This time consists of: 1.3 sec for Yacc to generate the parser in C; 7.6 sec for the C compiler to compile the parser; 0.7 sec to link the compiled parser into the rest of the code.

- PG

  The fact that PG generates parsers in the same (Lisp) environment in which the parsers are used has great advantages, as is shown by the relatively small construction and modification times of PG. The second reason that PG uses less generation time than Yacc, is that PG generates LR(0) tables, whereas Yacc generates LALR(1) tables. The parse times of both PG and IPG are larger than that of Yacc. There are two reasons for this difference: Yacc uses LALR(1) tables and generates parsers in C, whereas PG and IPG use LR(0) tables and generate parsers in Lisp.

  The difference between LR(0) and LALR(1) tables is the amount of information pre-computed for the grammar. LR(0) tables demand a reduction whenever a rule has been recognized, whereas LALR(1) tables only demand a reduction when the look-ahead is right. Tomita's parsing algorithm can use both, but leads to more failing parses with LR(0) tables than with LALR(1) tables.

- IPG

  In this case the time needed for constructing the parse table is almost zero. The lazy parser generator produces the requisite parts of the parse table while parsing the input, which explains why the second parse always takes less time than the first one. This difference is not as large as the generation time taken by PG, indicating that only a part of the parse table had to be generated for parsing the input. The modification time used by IPG is negligible. Only the first parse of "Exam.sdf" after the modification of the SDF grammar shows that some time was used for regenerating affected parts of the parse table.

In our opinion, the measurements convincingly show the benefits of lazy and incremental parser generation. IPG uses twice as much parse time as

Yacc, but since we expect grammars that are much larger than the grammar of SDF and input sentences to be quite small (the parser will mainly be used in conjunction with a syntax-directed editor), we consider IPG to be an excellent choice for interactive language definition systems and other highly dynamic applications.

2.7 Conclusions and future work

Although incremental generation of LR parse tables may have seemed a difficult problem, we were able to present all algorithms for incremental parser generation in this paper. We kept the complexity of the algorithms low by building the incremental generator on top of the lazy one, which in turn is an easy derivative of a conventional LR(0) parser generator. As is shown by the measurements in Section 2.6, IPG is an efficient parser generator suitable for use in interactive language definition systems. One might doubt the usefulness of the incremental behaviour of IPG as the non-incremental version of IPG is already 30 times faster than Yacc. We need incrementality however for languages that allow general user-defined syntax.

Future work related to IPG will include:

- Simultaneous editing of language definitions and programs.

  As has been explained in the introduction, we currently have an operational prototype of a universal syntax-directed editor parametrized with a syntax definition written in SDF. It is our aim to allow simultaneous editing of both this syntax definition as well as the program/specification written in the language defined by it.

- Syntax-directed editing of programs/specifications defining their own syntax [Chr90, Bur90b, Bur90a].

  An extreme case of the simultaneous definition, modification, and use of syntax occurs in languages that can define their own syntax Limited forms of user-defined syntax appear under various disguises. such as operator declarations, macros and user-defined function denotations. Clearly, the modification capability of IPG can be used to implement these syntax changes as well. What part of the already generated parse tree remains valid after a modification of the syntax is also a subject for future research. Pettersson [Pet90] did already use the IPG algorithms to implement an extension of ML with user-defined syntax.

- Modular composition of parsers.

  IPG does not yet support the composition of parsers that are generated for different modules. Although it would be possible to use the incremental modification capability of IPG in this case by adding the grammar of one module to the parser of the other, this is an asymmetrical operation, which, we believe, is not satisfactory. A different approach to modular parser generation based on IPG based on *restricted parsing* is described in [Chapter 3 of this thesis].

Chapter 3

Restricting a Parser to a Subgrammar

A technique is introduced for restricting a parser to a subgrammar of the grammar it was generated for. The resulting parser behaves like a parser generated for the subgrammar, but restricting an existing parser is much cheaper than generating a new parser for the subgrammar. Restricted parsing can be used to avoid repeated parser generation for individual modules in a modular grammar definition. We present the algorithms for restricted parsing and compare the efficiency of conventional parser generation with that of restricted parsing.

3.1 Introduction

A technique is introduced for restricting a parser to a subgrammar R' of the grammar R it was generated for. The resulting parser behaves as if it was generated for R', but, given a parser for R, making the restriction is much cheaper than generating a new parser for R'.

Parsers may be needed for n different subgrammars R_i of R and these subgrammars may have large parts in common. In such a case, generating a new parser for each R_i would lead to much duplication of generation effort. It might then be more time and space efficient to invest in the generation of a parser for R, and restrict this parser n times. For grammars having a modular structure, the sets of grammar rules common to several modules will in most cases be quite large. In Section 3.2.1 we describe how restricted parsing can be used to implement a parser generator for modular grammars.

A parser restricted to R' will only accept sentences that are in the lan-

guage described by R', but, unfortunately, it does not have the *correct prefix property* possessed by other LR parsing techniques. This property says that, if a parser is able to read v of a sentence vw, there always exists a w' such that vw' is a correct sentence. A parser restricted to R' may read more of an erroneous sentence than a parser specially generated for R' would have done, and will therefore be less exact in indicating the location of the erroneous token in a faulty sentence. Because of the lack of the correct prefix property the substring parse technique introduced in [Chapter 4 of this thesis] cannot use restricted parse tables either.

Our restricted parsing technique is based on LR(0) parse table generation, but can easily be extended to LALR(1) or LR(1) tables. The restricted parsing technique is an extension of the lazy and incremental parser generator IPG [Chapter 2 of this thesis]. Like IPG, it generates parsers in a lazy way, it is able to update a parser incrementally, accepts general context-free grammars and generates efficient parsers. The parse tables generated are used by a Generalized LR parsing algorithm [Chapter 1 of this thesis][Tom85].

The paper is organized as follows. First, in Section 3.2 we sketch two applications for the restricted parsing technique. Next, in Section 3.3 we discuss the technique in detail and present all algorithms. In Section 3.4 we analyze the behaviour of a restricted parser and compare it with conventionally generated ones. In Section 3.5 we present results of some measurements on the implementation of restricted parsing, and in Section 3.6 we finish with some concluding remarks.

3.2 Applications

3.2.1 Parser generation for modular grammars

A modular grammar consists of a number of grammar modules each containing a set of grammar rules and a set of names of other modules to be imported. Each module defines a (possibly incomplete) grammar, which has to be completed by the rules in the imported modules. A modular grammar consisting of n modules thus defines n ordinary grammars. In most cases, these grammars will have large parts in common. If the parsers defined by these modules are all needed, n parsers will have to be generated.

It is, of course, possible to use a non-modular parser generation technique to generate these n parsers. This would, however, induce much duplicate generation effort for the common parts of the grammars. Furthermore, a

modification in a module at the bottom of the import hierarchy would cause many parsers to be invalidated. Regenerating them all is unacceptable if one is interested in an interactive development and testing system for modular grammars.

The obvious approach to parser generation for modular grammars would be to generate an incomplete parser for the rules in each module and translate the import relation between modules to an import relation between parsers. This is done in Cigale [Voi86]. This solution, however, rules out all optimizations available in the efficient LR and LL parsing techniques, as these are based on knowledge of the complete grammar.

The approach followed by Koskimies [Kos90] is not satisfactory either, as his technique only works if all rules for a non-terminal are defined in the same module. We do not want to impose this restriction on the modular composition mechanism of the formalism.

We propose a parser generator for modular grammars based on restricted parsing. We do this by using IPG [Chapter 2 of this thesis] to generate *one* parser for the union of all grammar rules of all modules, and restrict this parser n times according to the n grammars defined by the modules. In this way no duplicate generation work is done, modifications are processed incrementally, the generated parsers are reasonably efficient, and no restrictions are imposed on the contents of the modules. A drawback of this approach is that it is not possible to develop parsers separately and combine them later at will. This limitation is, however, not too severe for the grammar development system envisaged.

ASF+SDF [BHK89] is a modular formalism for the definition of syntax and semantics of programming languages. Its implementation, the ASF+SDF system [Kli91b, Hen91], is highly incremental and applies the restricted parsing technique to generate the different parsers defined by a specification.

3.2.2 Incremental LALR(1) parser generation

Restricted parsing could also be used to solve a problem present in the incremental LALR(1) parser generator of Horspool [Hor89, Hor90].

Addition of rules works satisfactorily in Horspool's system: the underlying LR(0) states are updated in a manner similar to that of IPG, except that the automaton is expanded immediately, and the effect on the look-ahead sets is propagated through the automaton. However, on deleting a grammar rule, the effect on the look-ahead sets cannot be computed incrementally[1],

[1] It is unclear for symbols in the look-ahead set associated with the deleted rule, whether

and all sets must be removed and recomputed from scratch. This makes deletion of a rule an expensive operation.

Restricted parsing could in this case also yield the desired parser, by just removing the rule from the selection parsed for. The resulting parser would then behave as if the rule were deleted, but this computation is much cheaper. Pending deletions should, of course, be actually carried out at a certain moment, but using this scheme most of the time consuming recomputations can be avoided.

3.3 Restricted Parsing

As already mentioned, the restricted parsing method is based on the lazy and incremental parser generator IPG. We assume the reader to have some familiarity with that work, as we will build the restricted parsing method on top of the IPG algorithms.

Section 3.3.1 serves to give an intuitive idea of the method and shows how a simple parser can be restricted to a subgrammar. Next, in Section 3.3.2, the method is roughly sketched and, finally, all algorithms for restricted parsing are given in Section 3.3.3.

3.3.1 Restricting the grammar of the Booleans

The graph of itemsets as generated by IPG for the following grammar is shown in Fig. 3.1.

B ::= true (Grammar of the Booleans)
B ::= false
B ::= B or B
B ::= B and B
START ::= B

The steps of the parser on the sentence "true or false" are shown in Fig. 3.2. While parsing, the parser moves through the graph of itemsets: shift actions cause the parser to move forward along a transition labeled with the current input symbol, while reduce actions first cause a move backward along the path stored on the parse stack, and then a move forward along a transition labeled with the non-terminal that was the result of the reduction.

---

they are only there due to that rule or also due to other ones.

Figure 3.1: The graph of itemsets of the Booleans

Now, we make a selection of the grammar of the Booleans from which the rule "B ::= B or B" is excluded. A first approach for restricting the parser is to inhibit the *reduction* of "B ::= B or B". Figure 3.3 shows that the parser now fails on the reduction of this rule in state ⑧. The restriction thus causes only parses that need to reduce "B ::= B or B" to fail; the other parses will not be affected.

However, the parser could have failed much earlier. Already on the step from state ② to state ⑥ it is clear that the parser is on its way to recognize "B or B", while this rule is not in the selection. Therefore, it can already be stopped in ② by a restriction on the *transitions*, as is shown in Fig. 3.4.

The restrictions on the transitions and reductions can also be remembered, instead of re-computed on each visit by the parser to a state. In Fig. 3.5 the edges in the graph to state ⑥ are temporarily removed. As a result, state ⑥ and state ⑧, which deal specifically with the rule "B ::= B or B", have become unreachable for any parser.

3.3.2 The restricted parsing method

The basis of the restricted parsing method is that reduction of rules that are not selected is simply inhibited. This has as effect that all parsers that try such a reduction die, and only parses consisting entirely of valid rules will

Figure 3.3: Moves of the parser using restricted *reductions*

Figure 3.5: The restricted graph

succeed.

In addition to this, we only allow transitions to states which have a rule in their kernel which is in the selection. The kernel of a state is the set of rules possibly being recognized by the parser; if a kernel does not contain any rule belonging to the selected rules, each of its possibilities will eventually be forbidden by the restrictions on reductions. So we can already forbid such a parse in this stage.

The computation of restrictions affects the time needed by the parser for each step. To avoid this overhead, we compute the restrictions only once and save them in the form of trimmed versions of the actions for each state.

3.3.3 Algorithms

Before parsing starts, the set of selected rules is communicated to the generator with the call *RESTRICT-PARSER*(*rules*). This routine removes the possibly existing old restrictions and stores the new set for use by routine *RESTRICT-STATE*. This set is always extended with the rule "START′ ::= START".

RESTRICT-PARSER(*rules*) :
  forall *state* **do**
    if *state.type* = restricted **then** *state.type* := complete
  selected-rules := rules ∪ { START′ ::= START}

Routine *ACTION* now checks states for having type "restricted"; if not, routine *RESTRICT-STATE* is first called. The actions returned are derived from the fields *restricted-transitions* and *restricted-reductions*, which are set by *RESTRICT-STATE*.

ACTION(*state*, *symbol*) :
  if *state.type* ≠ restricted **then**
    if *state.type* = initial **then** *EXPAND*(*state*)
    RESTRICT-STATE(*state*)
  actions :=
    { reduce A ::= β | A ::= β ∈ *state.restricted-reductions* } ∪
    { shift *state′* | (*symbol state′*) ∈ *state.restricted-transitions* } ∪
    { accept | (*symbol* accept) ∈ *state.transitions* }
  return *actions*

Routine *RESTRICT-STATE* computes the actions which are valid for the current set of rules. It only allows reductions according to rules in the current set of selected rules, and it only allows transitions to states which contain at least one rule in their kernel that belongs to the current selection.

```
RESTRICT-STATE(state) :
    state.restricted-reductions := state.reductions ∩ selected-rules
    restricted-transitions := ∅
    forall (symbol state') ∈ state.transitions do
        if ∃ A ::= α•β ∈ state'.kernel : A ::= α β ∈ selected-rules then
            restricted-transitions := restricted-transitions ∪ (symbol state')
    state.restricted-transitions := restricted-transitions
    state.type := restricted
```

Routine *GOTO* remains unchanged, except that it uses *restricted-transitions* instead of the ordinary *transitions*. Unlike conventional parse tables, it may now happen that *GOTO* does not return a state, as the expected transition may have been removed by the restrictions. The parsing algorithm has to be adjusted to take care of this case.

3.4 Evaluation

Restricting a parser as sketched above is a simple and cheap extension of IPG. It has the advantage that the parser can switch to another set of rules with little overhead, and that all lazy and incremental properties of IPG are retained.

A drawback of the method is that a parser for a subgrammar can be (partly) invalidated by a modification in the grammar which does not affect that subgrammar itself. The required recomputation of the graph has then no effect on the behaviour of that parser, as all effects will be filtered out again by the restrictions.

It may turn out that the complete parser for R is never needed. Since we use a lazy parser generation technique, the parts of the parser that have not yet been needed by any subgrammar of R, are not generated at all.

Restrictions on *reductions* are the basic feature of the technique, but the parser will in most cases never reach states with such restricted reductions, as the restrictions on *transitions* prevent this.

3.4.1 Restricted parsing versus conventional parsing

We would like to compare a parser restricted to a certain grammar with a parser specially generated for that grammar. It will turn out that the restricted parser can, for certain grammars and sentences, be later to discover an error than the conventional parser. The efficiency of the Tomita parsing algorithm [Tom85] may also be influenced by using restricted parse

tables. As errors are discovered later, the Tomita algorithm has to pursue alternatives longer to determine which of them is correct. This means that on average more parallel parsers will have to be maintained.

This is the price paid for the saving on parse table generation offered by the restricted parsing method. Experience with restricted parsing in the ASF+SDF system [Hen91] shows that delayed error detection seldomly occurs in practice.

Delayed error detection

If we apply the restricted parsing method only using restrictions on reductions, it can easily happen that the parser fails too late on an erroneous sentence. Consider the following example: R contains both an "if-then"-rule and an "if-then-else"-rule, and R' does not contain the latter one. A restricted parser would only discover that the sentence "if a then print a else print b" is faulty on the reduction of the "if-then-else"-rule, after having read the input up to the "b".

The restriction on the transitions has been introduced to avoid this undesirable behaviour, and it solves the problem of delayed error detection in most cases. For the above example, the error is now discovered on reading the "else", just like the conventional parser would have done. However, the behaviour is not perfect yet, as is shown in the following example.

START ::= b C	(Rules R)
START ::= b a	
C ::= c c	
START ::= b a	(Selection R')
C ::= c c	

The parse graph for R, restricted to R', is shown in Fig. 3.6. If it is used to parse the sentence "b c c", which is erroneous according to R', the parser would first have to recognize the rule "C ::= c c" to discover that the "C" may not be used. This means that it would only fail after having read the second "c".

This problem can be solved by removing all useless rules from R'. According to [HU79, page 88], a symbol X is called *useful* if there is a derivation $START \stackrel{*}{\Longrightarrow} \alpha X \beta \stackrel{*}{\Longrightarrow} w$ for some α, β and w, where w is in T^*. Useless symbols can be identified easily, and all rules in which they appear are useless rules and can be removed from a selection without affecting the language

Figure 3.6: Restricted graph of itemsets

recognized. In the above example, this condition would cause R' to contain only "START ::= b a", and the parser would fail on the first "c", as desired.

Removing useless rules enhances the error detection capability of the restricted parsing method again, but there remain cases in which a restricted parser will discover an error later than a conventionally generated parser. If we take this grammar

```
START ::= Stat                          (Statement grammar)
START ::= Exp
Stat ::= if Exp then Stat
Stat ::= Id
Exp ::= Exp + Exp
Exp ::= Id
```

and select all rules but "START ::= Exp", the restricted parser will on a sentence "a + b + c" still read up to the "c", before it discovers that the reduction of "START ::= Exp" may not be applied.

This example shows that the restricted parsing method lacks the *correct prefix property*. This is also the reason why the substring parsing method [Chapter 4 of this thesis] cannot use restricted parse tables.

3.4.2 Simulating a larger parse table with a smaller one

In [Hor90] Horspool discusses a grammar, originally proposed by Alan Demers, which has an interesting property. This grammar G_n (Fig. 3.7) pro-

$$
\begin{array}{lcl}
S_0 & ::= & \text{a } S_0 \mid \text{b } S_0 \mid \text{a } S_1 \mid \text{b } S_1 \\
S_1 & ::= & \text{a } S_2 \mid \text{b } S_2 \\
S_2 & ::= & \text{a } S_3 \mid \text{b } S_3 \\
 & \vdots & \\
S_{n-1} & ::= & \text{a } S_n \mid \text{b } S_n \\
S_n & ::= & \text{d}
\end{array}
$$

Figure 3.7: Grammar G_n of language "$(a|b)^*(a|b)^n d$"

duces the language "$(a|b)^*(a|b)^n d$". An LR(0) parse table for G_n consists of $4n+5$ states. If we remove the rule "S_0 ::= b S_1" from G_n, yielding grammar G_n^-, it produces the language "$(a|b)^*a(a|b)^{n-1}d$". However, the number of states now explodes to $2^n + 2n + 4$. This is of course a challenge for the restricted parsing technique: how can it simulate a parse table of $2^n + 2n + 4$ states, while it can only address $4n + 5$ states?

The grammar used by Horspool in [Hor90] contains an additional rule "S_0 ::= c S_1", but as this rule does not affect the problem as far as restricted parsing is concerned, we have left it out for the sake of simplicity. The absence of this rule explains why our number of states differs slightly from those of Horspool. The number of states needed by our two versions of the grammar for different values of n are given in the following table.

	G_n	G_n^-
n	$4n + 5$	$2^n + 2n + 4$
1	9	7
2	13	12
3	17	18
4	21	28

To keep the example as simple as possible, we will work it out for $n = 3$. We will first show the parse tables for the two versions of this grammar as generated by a conventional parse table generator. Next, we show the parse table as used by the restricted parsing technique.

The conventional parse tables

Fig. 3.8 shows a graph-like description of the parse table generated for grammar G_n with $n = 3$:

S_0 ::= a S_0 | b S_0 | a S_1 | b S_1 (Grammar G_3)
S_1 ::= a S_2 | b S_2

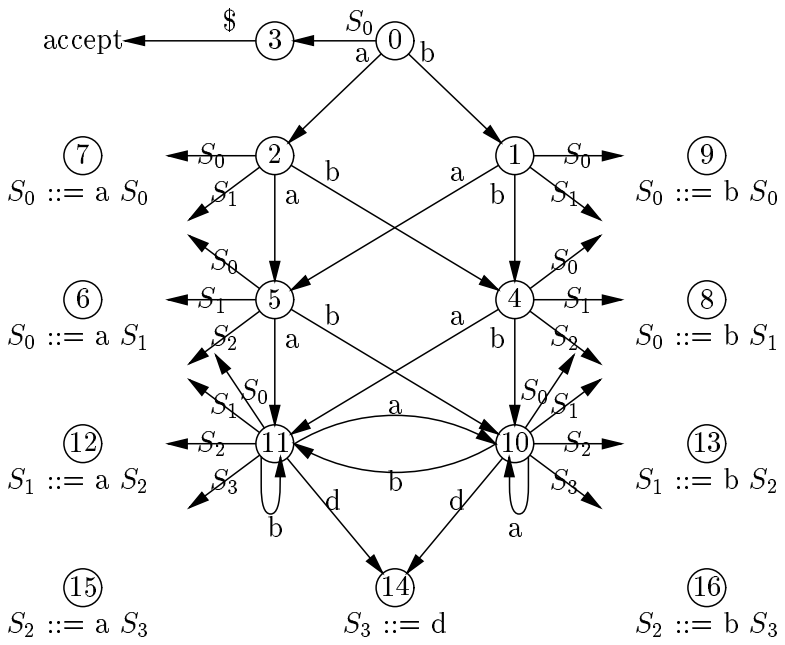


Figure 3.8: The parse graph of G_3

S_2 ::= a S_3 | b S_3
S_3 ::= d

This parse graph can best be used to understand the following moves of the parser on some sentence in the language.

- Parsing starts in state ⓪.
- On the first a or b the parser goes to state ② or ①.
- On the second a or b to ⑤ or ④.
- On the third a or b to ⑪ or ⑩.
- All following a's and b's are processed by alternating between the states ⑪ and ⑩.
- On the final d, the parser goes to state ⑭. From now on the parser will only execute reductions.

- The reduction of the rule "S_3 ::= d" brings the parser to state ⑮ or ⑯, dependent of the history which is stored on the parse stack.
- Next, the reduction of "S_2 ::= a S_3" or "S_2 ::= b S_3" brings the parser to ⑫ or ⑬.
- The reduction of "S_1 ::= a S_2" or "S_1 ::= b S_2" to ⑥ or ⑧.
- The reduction of "S_0 ::= a S_1" or "S_0 ::= b S_1" to ⑦ or ⑨.
- All subsequent reductions of "S_0 ::= a S_0" and "S_0 ::= b S_0" keep the parser in states ⑦ and ⑨, until the bottom of the stack is reached.
- The *GOTO*-transition under S_0 in state ⓪ at the bottom of the stack brings the parser to state ③ where the input is accepted.

If we remove the rule "S_0 ::= b S_1" from G_3, it becomes G_3^-:

S_0 ::= a S_0 | b S_0 | a S_1 (Grammar G_3^-)
S_1 ::= a S_2 | b S_2
S_2 ::= a S_3 | b S_3
S_3 ::= d

A graph-like representation of the parse table as generated by a conventional parser generator for G_3^- is shown in Fig. 3.9. To represent the full table as a graph would be difficult to interpret and we only represent the part visited by the parser during its shift transitions. The states that are visited while performing the reductions after reading symbol "*d*" are left out. For this parse table holds the same as for that of G_3: as from the moment the parser enters state ⑭ under the final symbol "*d*", it will only perform reductions and arrives finally in an accepting state.

The parse graph of G_3^- as displayed in Fig. 3.9 shows that states ⑪, ⑲, ⑳ and ㉑ do have a transition under "*d*", while states ②, ⑤, ⑰ and ⑱ do not have such a transition. This difference expresses whether three symbols ago an "*a*" was seen or not. As a consequence, on encountering a "*d*", the parser can continue in the first four states, while it will fail in the last four.

On reading *a*'s and *b*'s, the parser moves around between these states. This means that each state must express which were the last three symbols seen. This explains the factor 2^n in the number of states needed for grammar G_n^-. To clarify this, we have attached to each state in Fig. 3.9 a triple

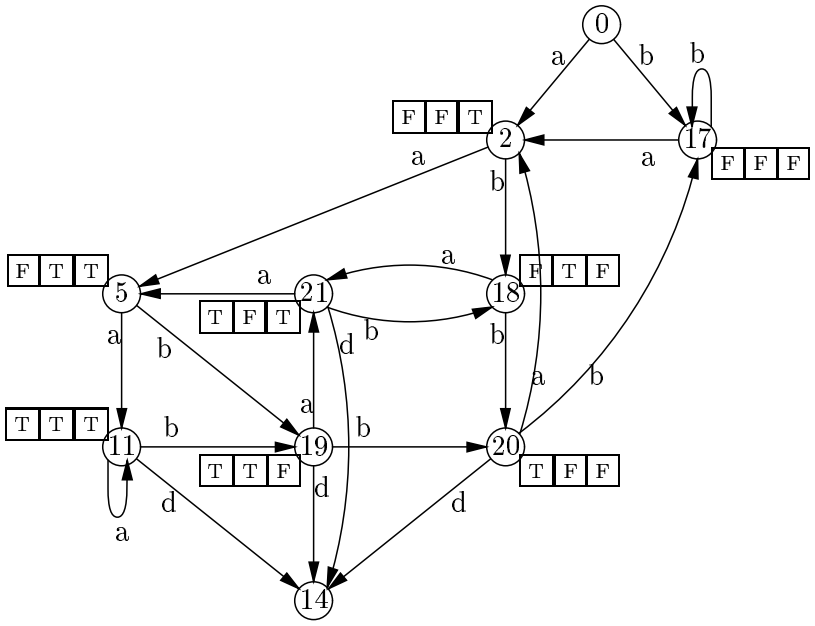


Figure 3.9: Part of the parse graph of G_3^-

describing the last three symbols encountered by the parser in that state. Going from left to right, the positions in the triple have indices -3 (least recent symbol) to -1 (most recent symbol). An "F" appearing at some position stands for "not-an-a", while "T" stands for "an-a". In the original graph of itemsets of G_3^-, this is of course not denoted by such a triple, but by the combination of the dotted rules in the kernel of the state.

Only those states with "T" at position -3 in their triple, do have a transition under d to the final state ⑭. On reading an "a" or a "b", the parser goes to a state of which the triple is shifted one position to the left and the Boolean at position -1 becomes "T" if the symbol read was an "a", "F" if it was a "b".

The restricted parse table

Fig. 3.10 shows the parse table of G_3 restricted to rules in G_3^-. The transitions to state ⑧ are removed by the restrictions, as state ⑧ does not contain any selected rule in its kernel. We have denoted the restricted transitions by dashing them.

Now, if a parser works according to this restricted parse table, the transi-

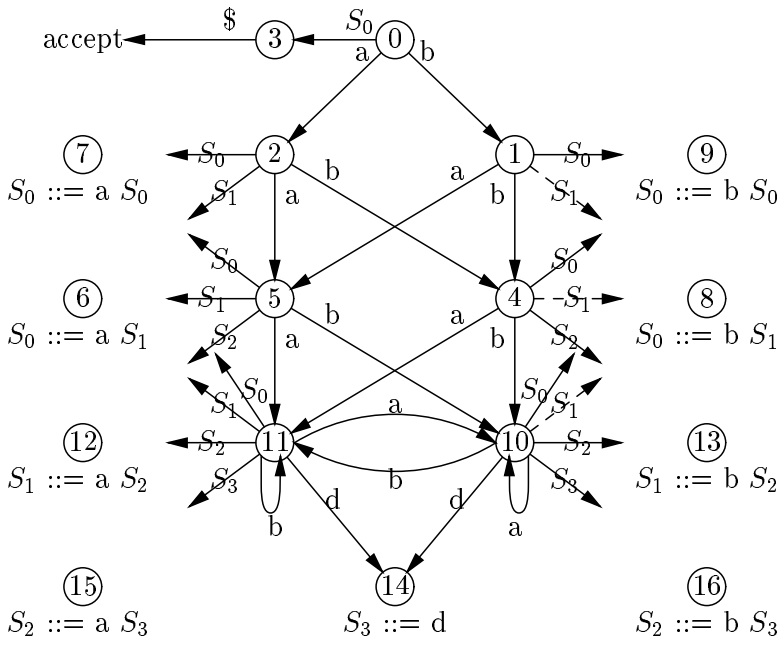


Figure 3.10: The parse graph of G_3 restricted to G_3^-

tion under d succeeds always and the parser starts reducing rules. However, on reducing "S_1 ::= a S_2" or "S_1 ::= b S_2", the parser needs a $GOTO$-transition under the result of the rule, S_1. If it needs this transition in state ①, ④ or ⑩, it is prohibited by the restrictions, as this would mean that the symbol, three symbols before the d, has been an b instead of the a needed. In that case, the parser fails after all.

This shows that the restricted parser discovers the error in the sentence later, but it still is able to discover it with the parse table of G_3 restricted to G_3^-. This example therefore shows that restricted parsing can be used to simulate a larger parse table with a smaller one.

An interesting consequence of this property of restricted parsing is that it becomes thus possible to add some rules to a grammar in order to make the parse table smaller. However, the cases in which this might apply are so unlikely in practice, that we will not investigate this possibility any further.

3.5 Measurements

3.5.1 Time consumption in a restricted parser

To give an idea of the relative time consumption of the parser, the parser generator and parser restrictor, we did some measurements on a grammar that defines two subgrammars, *Stacks* (12 rules) and *Stats* (21 rules), which have *Expressions* (8 rules) in common. We performed the following measurements in succession:

1	Assemble	Assemble all rules in a grammar structure
2	*Stats*	Restrict parser to *Stats* and parse a sentence (of 110 tokens)
3	*Stats*	Parse the same sentence once more
4	*Stacks*	Restrict parser to *Stacks* and parse a sentence (of 60 tokens)
5	*Stacks*	Parse the same sentence again
6	*Stats*	Restrict parser to *Stats* and re-parse the *Stats* sentence
7	Assemble	Start from scratch and re-assemble all rules in a new grammar structure
8	*Stacks*	Restrict parser to *Stacks* and re-parse the *Stacks* sentence

As IPG generates parsers by need, the time taken by the first parse of a sentence is always augmented with the generation time of the part of the

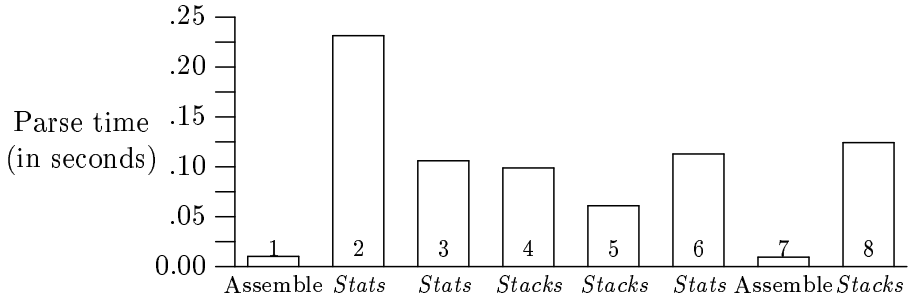


Figure 3.11: Eight successive measurements

parser needed to parse it. The results of these measurements are given in Fig. 3.11 and show the following:

- The difference in parse time used between measurements 2 and 3 shows how much time was spend to generate the needed part of the parser for *Stats*.
- The difference in time consumed between measurements 3 and 6 for *Stats*, shows that re-specializing the already generated parser from *Stacks* to *Stats* takes little time.
- The time needed to parse for *Stacks* differs between measurements 4 and 8: during the generation for *Stats* in 2, part of the parser for *Expressions* has already been generated, which was of use for *Stacks* also.

3.5.2 Measurements on Pascal

A restricted parser can always be simulated by just using a conventional parser generator to generate a separate parser for each subgrammar. In this section we will compare the efficiency of these methods with each other.

As test grammar we take the grammar of Pascal, divided into three subgrammars: One subgrammar *Exp* that describes the syntax of Pascal expressions, one for its statements *Stm* and one for complete Pascal programs *Prog*. Subset *Stm* includes *Exp*, and *Prog* consists of all grammar rules and includes thus both *Stm* and *Exp*. We want to parse sentences according to each of these subgrammars. We use IPG to generate a parser for each of the subgrammars, and we use it to generate a parser for *Prog* which is then restricted to the three subgrammars.

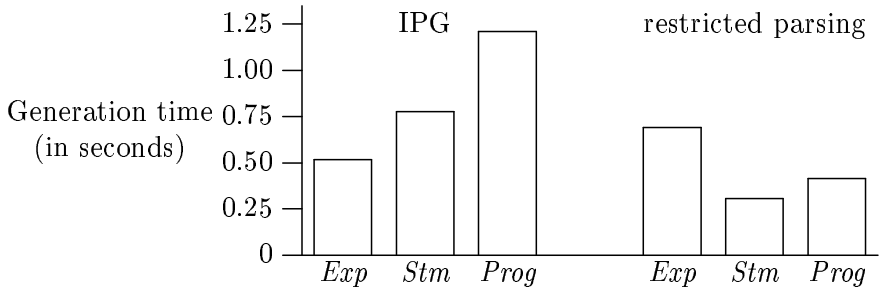


Figure 3.12: Generation times for subgrammars of Pascal

We performed the following measurements: take three input sentences, one for each subgrammar, and let them parse in succession by the three IPG parsers and by the parser restricted to the appropriate subgrammar. These sentences are chosen in such a way that they cover their grammars reasonably well, and large part of the parser has to be generated to parse them. The generation time needed by the two parser generators is shown in Fig. 3.12.

The generation time used by IPG increases with the size of the grammar. In the restricted parsing method however, the generation work done for *Exp* is re-used while generating for *Stm*, which work is used again in *Prog*. It clearly uses less generation time for *Stm* and *Prog* as IPG needs to do, however the larger generation time of *Exp* is the price paid. It is larger because the parser for *Exp* is generated in an environment of other rules (those of *Stm* and *Prog*), which are taken into account while generating for *Exp*.

3.6 Conclusions and future work

Restricting a parser to a subgrammar turns out to be a fast and easy way to obtain the parser for that subgrammar. The parser thus generated is not guaranteed to be as efficient as a conventionally generated one, but in most cases the speed will be comparable. Moreover, it is guaranteed to recognize exactly the language defined by the grammar. The restricted parsing technique is already in use as implementation for a modular grammar formalism and works satisfactorily.

We may extend the technique from LR(0) parse tables to LALR(1) tables. Both kind of tables can be used by the Tomita parsing algorithm to parse for general context-free grammars, but parsers will be more efficient with

the latter kind of tables. We would also like to use the technique to realize an incremental LALR(1) parse table generator, along the lines sketched in Section 3.2.2.

Chapter 4

Substring Parsing

A substring recognizer for a language L determines whether a string s is a substring of a sentence in L, i.e., *substring-recognize*(s) succeeds if and only if $\exists v, w$: $vsw \in L$. The algorithm for substring recognition presented here accepts general context-free grammars and uses the same parse tables as the general context-free parsing algorithm from which it is derived. Substring recognition can be useful for noncorrecting syntax error recovery and for incremental parsing. By extending the substring *recognizer* with the ability to generate trees for the possible contextual completions of the substring, we obtain a substring *parser*, which can be used in a syntax-directed editor to complete fragments of sentences.

4.1 Introduction

A recognizer for a language L determines whether a sentence s belongs to L. A substring recognizer performs a more complicated job, as it determines whether s can be *part* of a sentence of L.

A recently developed substring recognition algorithm [Cor89] uses an ordinary LR parsing algorithm with special parse tables. For ordinary parsing, this parsing algorithm is limited to LR(1) grammars, but the more complicated nature of substring recognition limits it to bounded-context grammars (see Section 4.3).

We describe a substring recognition algorithm that does not suffer from this drawback. It accepts general context-free grammars and uses the same parse tables as our ordinary parser. Our algorithm is based on the pseudo-parallel parsing algorithm of Tomita [Tom85], [Chapter 1 of this thesis], which runs a dynamically varying number of LR parsers in parallel and ac-

cepts general context-free grammars. Next, we extend the substring *recognizer* into a substring *parser* that generates trees for the possible completions of the substring.

4.2 Applications

Before discussing existing proposals to substring parsing (Section 4.3) and our approach to it (Section 4.4), we mention some possible applications of the technique.

4.2.1 Syntax error recovery

In its simplest form, a parser stops at the first syntax error found. If it has to find as many errors in the input as possible, it can try to correct the error in order to continue parsing. Spurious errors are easily introduced, however, if the parser makes false assumptions about the kind of error encountered.

Substring parsing can be used to implement *non*correcting syntax error recovery. If an ordinary parser detects a syntax error on some symbol, the substring parser can be started on the next symbol to determine whether the rest of the sentence could be a legal substring. It can thus discover additional syntax errors. Using this method, it is not necessary to let the parser make any assumption about how to correct the error, or to let it skip input until a trusted symbol is found. However, no guarantee is given that the substrings sequentially found will match with each other.

Richter defines noncorrecting syntax error recovery with the aid of substring parsing and interval analysis in a formal framework [Ric85]. He proves that his technique does not generate spurious errors, but is not explicit about its implementation. He notes, however, that there are difficulties in keeping the substring parser deterministic due to a limitation on the class of grammars accepted. Our technique could be useful here as it implements the required substring analysis for general context-free grammars.

4.2.2 Completion tool

In Section 4.5 we will show how the substring recognizer can be extended such that it generates parse trees for the possible completions of a substring. As the total number of possible completions will often be infinite, only generic completions are generated. A syntax-directed editor could use

these to complete fragments of sentences in accordance with the grammar used, or to guess the continuation of what the user is typing.

Snelting presents a technique to complete the right-hand side of unfinished sentences [Sne90]. We will discuss parts of his method in Section 4.5.3.

4.2.3 Incremental parsing

Another application for substring parsing is incremental parsing. An incremental parser builds the parse tree for the current version of its input text while it re-uses the parse tree generated for the previous version as much as possible. We will first sketch two possible solutions for the problem of incremental parsing, and next suggest a third solution based on substring parsing.

Re-use parser states

Incremental parsing can be performed by attaching parser states to tokens [Cel78, AD83, Yeh83]. After a modification has been made, the parser is restarted in a saved state, at a point in the text just before the modification. Parsing stops when the parser reaches a token after the modification in an old configuration (if ever).

These methods are very good as to minimizing the amount of recomputation after a modification, but require a huge amount of memory for storing the states of the parser (parse stacks with partial parse trees as elements).

Abbreviate sentence

Ghezzi and Mandrioli present an alternative technique for incremental parsing [GM79, GM80]. If the string $x\tilde{x}z\tilde{y}y$ is modified to $x\tilde{x}\hat{z}\tilde{y}y$, where $\tilde{x}$ and $\tilde{y}$ have length k, with k the look-ahead used by the parser, then the parse trees previously generated for x and y are still valid after the modification. All subtrees previously generated for x and y can thus be abbreviated by their top non-terminals, which minimizes the length of the string to be reparsed.

This technique is both time and space efficient, but is not applicable to general context-free parsing as it requires a fixed look-ahead. In our particular case, we need incremental parsing in a syntax-directed editor that uses the Tomita parser. By running a varying number of LR-parsers in parallel, the Tomita parser adjusts its look-ahead dynamically to the amount needed, and is thus not limited to an a priori known k.

Reparse a subtree only

Incremental parsing can also be achieved in another manner: after a modification has been made in the text, find the substring s' belonging to the smallest subtree that contains the modification in the stored parse tree. If the type of this subtree is T and s' can be parsed as a tree of type T also, the old subtree can be replaced by the new one. If s' fails to parse, it may be the case that the modification introduced a syntax error, or that the subtree has been chosen too small. These two cases must be distinguished, as the incremental parser proceeds in a different way in each case.

A substring parser can provide a hint as to which of the two possibilities is actually the case. If the substring parser fails on s', the modification will be syntactically incorrect in any context, and an error message can be given. If the substring parser succeeds, a larger subtree is chosen and parsing is retried.

This can be more time consuming than remembering parser states, but the amount of memory needed is far less. We consider using this scheme in the syntax-directed editor GSE [Koo], but it has to be investigated further as the technique still performs a lot of work twice.

4.3 Related work

Cormack [Cor89] describes a substring parse technique for Floyd's class of bounded context or BC(1,1) grammars [Flo64], and implements the substring parser Richter mentions [Ric85]. A grammar is BC(1,1) if for every rule $A ::= \alpha$, if some sentential form contains $a\alpha b$ where α is derived from A then α is derived from A in *all* sentential forms containing $a\alpha b$. This class is smaller than LR(1). The solution of Cormack consists in using an ordinary LR automaton, but a special parse table constructor. The sets of items generated do not only contain items of the form $A ::= \alpha \bullet \beta$ but also "suffix items" of the form $A ::= \cdots \bullet \beta$. These suffix items denote partial handles whose origins occur before the beginning of the input. The generated parse tables are deterministic, provided that the grammar is BC(1,1). This substring parser is used for noncorrecting error recovery in a parser for Pascal. The limitation on the class of the grammar caused problems in the definition of Pascal, which where alleviated by permitting the parse table generator to rewrite the grammar if necessary.

Lang describes a method for parsing sentences containing an arbitrary number of unknown parts of unknown length [Lan88]. The parser produces

a finite representation of all possible parses (often infinite in number) that could account for the missing parts. The implementation of this method is based on Earley parsing [Ear70]. The basic idea of Lang's method is that "in the presence of the unknown subsequence "$*$", scanning transitions may be applied any number of times to the same computation thread, without shifting the input stream." This process terminates, as parsers in the same state are joined and the number of states is finite.

This method is very elegant and powerful, and can be used as a substring parser (by providing it with the string "$*s*$"). We will not use it, however, as it is more general than what we need. Whether it would be efficient enough for interactive purposes is unclear.

4.4 Substring Recognition

4.4.1 Tomita parsing

We base the implementation of our substring parser on Generalized LR parsing [Tom85], [Chapter 1 of this thesis]. This technique 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. When 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 goes back over a fork in a parse stack, splits the corresponding parser again into two separate parsers. If a parser hits an error entry in the parse table, it is killed by removing it from the set of active parsers. The possibility to run several parsers in parallel makes the Tomita algorithm very well suited for substring parsing.

For a full description of the GLR parsing algorithm we refer to Tomita [Tom85], to Nozohoor-Farshi who corrected an error in the algorithm concerning ϵ-productions [NF89], or to Rekers who extended the algorithm to the full class of context-free grammars by including cyclic grammars[1] [Chapter 1 of this thesis]. For a detailed explanation of LR parsing [ASU86, chapter 4.7] is recommended.

[1] Grammars in which $A \stackrel{+}{\Longrightarrow} A$ is a possible derivation.

4.4.2 The grammar

The grammar according to which the substring recognition algorithm works, should not contain useless symbols. According to [HU79, page 88], a symbol X is called *useful* if there is a derivation $S \stackrel{*}{\Longrightarrow} \alpha X \beta \stackrel{*}{\Longrightarrow} w$ for some α, β and w, where w is in T^*. Useless symbols can be identified easily, and all rules in which they appear should be removed from the grammar. Such a clean-up operation does not affect the language recognized.

Unreachable symbols and rules do not influence our method of substring parsing, as these are already ignored by the parse table generator. This is due to the fact that LR parse tables are generated top-down, starting with the start symbol of the grammar, and that unreachable symbols and rules are, by definition, unreachable from the start symbol.

Symbols and rules which cannot produce any terminal string should be removed from the grammar however. These can cause the substring parser to succeed on a string s, while no string vsw exists in L.

4.4.3 The algorithm

If we have to determine whether a string $a_0 \cdots a_n$ is a substring of a sentence in a language L, we start the substring recognition process by generating, for each state directly reachable under a_0, a parser with this state on its stack. These parsers will process $a_1 \cdots a_n$.

We will show how an individual parser processes an action, but we will not discuss the management of the different parsers, as this is done in the same way as in ordinary Tomita parsing.

The parser obtains an action from the parse table with the state on top of its stack and with input symbol a_k. This can be a *shift*, *error* or *reduce*-action, and is processed in the following manner:

- A (shift $state'$)-action is processed as in normal parsing: $state'$ is pushed on the stack and the parser is ready to process a_{k+1}.
- An (error)-action removes the parser from the set of active parsers.
- A (reduce $A ::= \alpha\beta$)-action is processed as follows:
  - If there are at least $|\alpha\beta| + 1$ entries on the parse stack the reduce action is performed as in normal parsing: $|\alpha\beta|$ entries are popped off the stack, and the parse table is consulted, with the state remaining on top of the stack and A, to obtain a state to push

on the stack again. The parser is now ready to continue the processing of a_k.

- If there are only $|\beta|$ entries on the stack, only β has been recognized of $A ::= \alpha\beta$; α lies before a_0 and should produce (a part of) a prefix of a_0. This is possible, as all non-terminals in α can produce some terminal string, and all terminals in α trivially do. So the reduction $A ::= \alpha\beta$ may be performed, and parsing may continue in the states which can be reached directly by a transition under A. For each of these valid states a new parser is started with that state on the stack. These parsers all proceed to process a_k.
- If there are exactly $|\alpha\beta|$ entries on the stack, $a_0 \cdots a_{k-1}$ reduces to $\alpha\beta$, but the context in which A is to be used is unknown. This is handled in the same way as the previous case.

If there are no parsers left alive after the processing of a_n, the substring parser fails. If there are parsers left, these are currently recognizing rules $A ::= \alpha\beta$, of which (a part of) α has been recognized. As every β can produce some terminal string, these rules can all be finished. This means that the substring parser succeeds if there are parsers remaining after the processing of a_n.

4.4.4 The parse table generator

The substring parser is controlled by the same parse table as our ordinary parser. To generate this parse table we use an extended version of the lazy and incremental parser generator IPG [Chapter 2 of this thesis]. The extension concerns the need of the substring parser to know all states which can be reached by a transition under a given symbol. This function needs global information about the parse table, which means that the whole parse table must be known. As a consequence, the lazy aspect of IPG cannot be exploited here and the parse table must be fully expanded. The expanded parse table can of course also be used by the ordinary parser.

4.5 Substring Parsing

We extend the substring recognizer into a substring parser by generating parse trees for substrings. The possible parse trees for a substring s are

the parse trees of all sentences vsw for which $vsw \in L$ holds. To limit the number of completions we allow v and w to consist both of *terminals* and *non-terminals*, and we generate a parse tree, corresponding to a sentential form $\sigma_1 s \sigma_2$, only when the frontier of each of its *subtrees* contains at least one symbol of s; i.e., we do not generate subtrees whose frontier lies entirely within σ_1 or σ_2. The trees that we generate are the most general trees, as it is not possible to replace any of their subtrees by a non-terminal without removing part of the substring s. Even so, the number of completions can still be infinite. In Section 4.5.3 we will discuss how to limit this number still further.

4.5.1 Example of a completion

For the following grammar

(A small grammar)

start ::= Stat
start ::= Exp
Stat ::= if Exp then Stat
Stat ::= if Exp then Stat else Stat
Stat ::= Id := Exp
Exp ::= Id
Exp ::= Int
Exp ::= Exp + Exp
Exp ::= Exp ∗ Exp
Exp ::= (Exp)

and the string "`) + 5 then if`", a possible completion is the sentential form

$$\underbrace{\texttt{if (}\ \mathit{Exp}}_{\sigma_1}\ \underbrace{\texttt{) + 5 then if}}_{s}\ \underbrace{\mathit{Exp}\ \texttt{then}\ \mathit{Stat}}_{\sigma_2}$$

whose parse tree is given in Figure 4.1. To distinguish the leaves of s from those of σ_1 and σ_2, the former are printed in boldface.

4.5.2 Generating the completions of a substring

LR parsers generate parts of parse trees during a reduction step. On reducing $A ::= \alpha$, the parse stack contains the subtrees created for α. These are assembled in a new node of type A and the subtree created in this way is pushed on the stack. In the substring parser ordinary reductions are treated in the same way.

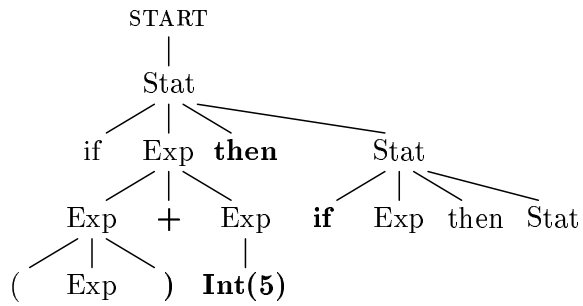


Figure 4.1: A completion of "`) + 5 then if`"

If the rule $A ::= \alpha\beta$ is reduced with only nodes for β on the stack, however, additional nodes are created for α. In this way, the parse trees for the possible prefixes of s are created.

Parse trees for postfixes of s are created in the same way: after processing s, the parser has to finish all rules which are in the process of being recognized. These are the rules in the kernel of the current state of the parser. If only α has been seen from a rule $A ::= \alpha\beta$, the rule is reduced and additional nodes are created for β. It can even be the case that only β has been recognized from a rule $A ::= \alpha\beta\gamma$, and that nodes must be created for both α and γ.

4.5.3 Further reduction of the number of possible completions

By producing only parse trees that are most general, the number of possible completions is reduced, but it is often still too large and not even always finite. We propose the following rules to limit this number still further:

1. The parse trees generated are kept as compact as possible by disallowing reductions of rules of the form $A ::= \alpha A$, $A ::= \alpha A\beta$, and $A ::= A\beta$, where only A has actually been recognized and all elements of α and β would produce elements in σ_1 or σ_2. Clearly, such reductions can be repeated infinitely often. They are undesirable as they only enlarge σ_1 or σ_2.

   For example, the substring "`) + 5 then if`" also has a possible completion

$$\underbrace{\texttt{if } \textit{Exp} \texttt{ + (} \textit{Exp}}_{\sigma_1} \underbrace{\texttt{) + 5 then if}}_{s} \underbrace{\textit{Exp} \texttt{ then } \textit{Stat}}_{\sigma_2}$$

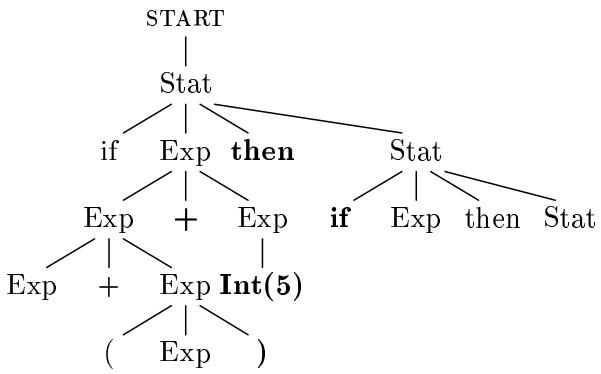


Figure 4.2: Another possible completion of "`) + 5 then if`"

whose parse tree is given in Figure 4.2. In this tree a subtree for the rule $Exp ::= Exp + Exp$ has been inserted in the prefix.

2. The number of possible sentential forms for which parse trees are generated is now finite, but these can still have infinitely many parse trees as the grammar may be cyclic. Rekers describes how to parse and generate parse *graphs* for cyclic grammars [Chapter 1 of this thesis]. The cycles generated in this graph can be removed by his routine *remove-cycles*. The same approach can be used for substring parsing, and this results in a finite number of most general completions.

3. In the generation of the postfixes of s a choice can be made for the "simplest" completion. That is, if a substring can be completed according to both $A ::= \alpha\beta$ and $A ::= \alpha\gamma$, and $|\beta| < |\gamma|$, we prefer $A ::= \alpha\beta$. In the example of Figure 4.1 this rule forbids the choice of the "if-then-else" rule, as the "if-then" rule already applies. Snelting's rule "*prefer reduce items over shift items*" [Sne90] is similar to ours. His rule can also be formulated as: if completion according to both $A ::= \alpha$ and $B ::= \alpha\gamma$ ($\gamma \neq \epsilon$) is possible, then prefer $A ::= \alpha$. We consider our rule more appropriate, as we take the case of β being non-empty but shorter than γ into account as well, and we only make the choice if the two rules reduce to the same non-terminal. Otherwise, the rule $A ::= \alpha$ might be preferred over $B ::= \alpha\gamma$, whereas the environment in which the substring is completed needs a tree of type B.

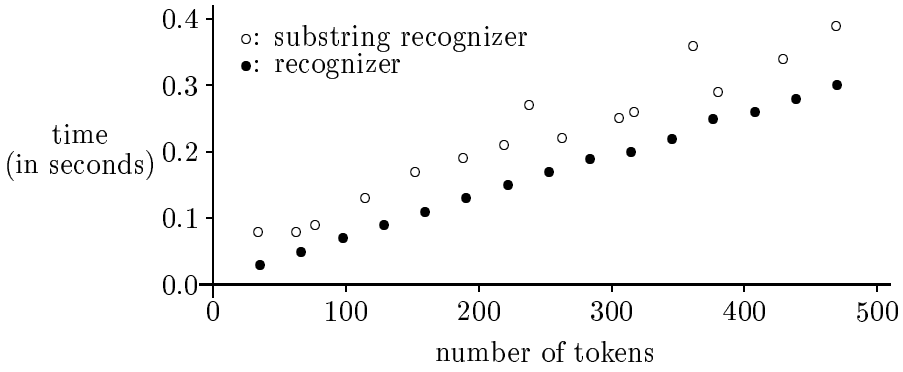


Figure 4.3: Comparison of the substring recognizer with an ordinary recognizer

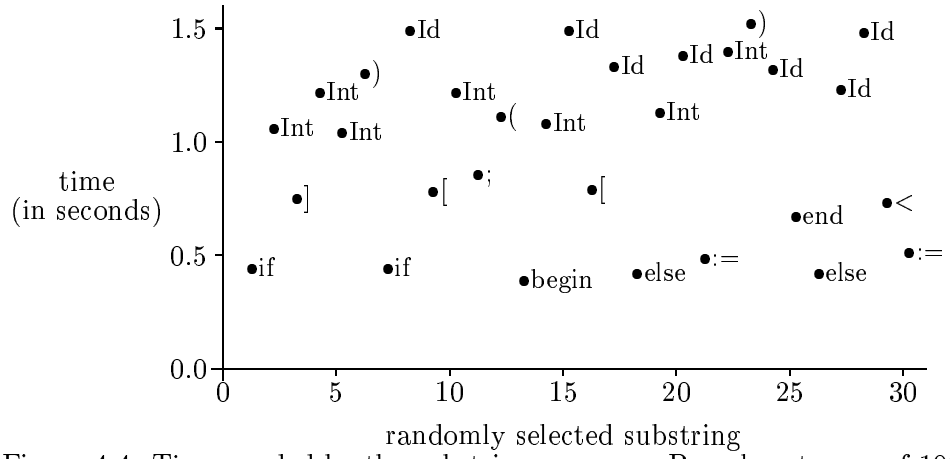


Figure 4.4: Time needed by the substring parser on Pascal sentences of 100 tokens

4.6 Measurements

Our first measurement compares the substring recognizer with the Tomita recognizer from which it was derived to learn the additional costs of substring parsing.[1]

We have taken a grammar of about twenty rules and sentences of increasing length. These were parsed by the Tomita recognizer first. The resulting parse times are indicated in Figure 4.3 with a "•". Next, the same strings

[1]The measurements have been performed on a SUN SPARC station. The programs have been written in Lisp. The time used by the lexical scanner has not been taken into account.

minus a randomly chosen prefix were given to the substring parser. The required times are indicated in Figure 4.3 with a "o".

The measurements show that the substring parser has a moderate overhead with respect to the normal parser. This overhead can be interpreted as the time needed for the substring parser to get on the "right track". As our next measurements show, the variations in this overhead are caused by the random cutting of the string. For some strings it takes longer than for others to determine of which language construct it can be a substring. The larger the grammar is, the more alternatives are available and therefore the higher the variation.

In Figure 4.4 we compared the time taken by the substring parser on 30 randomly chosen parts of Pascal sentences of 100 tokens. The dots indicate the amount of time needed and they are attributed with the first symbol of the substring. These measurements show that sentences starting with a token that can appear in many differents contexts, like "`Id`" or "`)`", take more time to recognize than sentences starting with a disambiguating token like "`:=`" or "`else`".

4.7 Conclusions

The adaptation of the Tomita algorithm to substring parsing results in a very elegant and powerful algorithm. The main advantage of the fact that it accepts general context-free grammars and uses ordinary LR parse tables is that substring parsing can now be applied in a very general manner, instead of only to carefully written grammars and at the cost of an extra generation phase.

Substring parsing is slower than ordinary parsing, but this will not be a serious drawback for its application as an error recovery technique or as a completion tool. The use of the substring parser in incremental parsing, however, has to be investigated further.

Chapter 5

From BNF to SDF

The syntax definition formalism SDF is introduced by developing an SDF definition for a subset of Pascal. The main points of interest are the modular decomposition of the grammar, the readability of the definition, and the behaviour of the editor generated from this definition.

5.1 Introduction

SDF is a formalism for the definition of syntax, which is comparable to BNF in some respects, but has a wider scope, in that it also covers the definition of lexical and abstract syntax and the behaviour of a syntax directed editor. Its design and implementation are tailored towards the language designer who wants to develop new languages as well as implement existing ones in a highly interactive manner.

SDF emphasizes compactness of syntax definitions by offering (a) a standard interface between lexical and context-free syntax; (b) a standard correspondence between context-free and abstract syntax; (c) powerful disambiguation constructs; (d) list constructs; and (e) an efficient incremental implementation which accepts arbitrary context-free syntax definitions.

In this paper we discuss a number of points which should be considered while writing an SDF definition. We do this in a tutorial manner, by translating a given BNF grammar, in a number of steps, into an SDF definition. Aho, Sethi and Ullman in [ASU86, appendix A] define a subset of Pascal by means of a BNF grammar and some explanations in English. We have chosen this grammar because it is well known, not too large, the language defined needs little explanation and it is not completely trivial.

We will develop a parser and a syntax directed editor for the language described by using the SDF formalism and implementation. First, we will decompose the grammar into modules and next translate the BNF grammar into an SDF definition. Finally, we will make several modifications to the SDF definition in order to improve the readability of the grammar, make it shorter, and enhance the quality of the syntax directed editor derived.

The syntax definition formalism SDF has been introduced in [BHK89, chapter 6] and is fully described in the SDF reference manual [HHKR89]. The implementation of SDF used is the ASF+SDF system [Kli91b, Hen91], but we will not use the semantic part, ASF, of this system [BHK89, Wal91]. The ASF+SDF system heavily depends on the generalized syntax directed editor GSE [Koo].

Interesting SDF definitions can be found in [BHK89, chapter 9], [Hen91, chapter 4], [Meu88] and [Deu91]. SDF is also used to define the syntax of the language μCRL in [Gro91, chapter 7] and for that of LOTOS in [JJWW90b, JJWW90a].

5.2 Deciding on a modular decomposition

The BNF grammar in [ASU86, appendix A] is a flat list of grammar rules, while SDF allows modular grammar definitions. An SDF module contains local definitions and may import other modules. The import relation between modules is transitive. It is of course possible to use just one SDF module for the entire grammar, but a modular structure makes it easier to understand and test the definition.

How do we split the BNF grammar into modules such that they represent clear conceptual entities of the language described? A good way to start this decomposition is to look at the grammar as a graph. Each non-terminal (or lexical token) N is a node, and for each rule that defines N and uses non-terminals $N_1 \cdots N_n$, edges are added from each N_i to N. Edges that go from N to N are left out.

We will not show this graph for the entire grammar, but only for the part of the grammar defining non-terminal *statement*. These grammar rules are shown in Fig. 5.1.

The graph for this part of the grammar is shown in Fig. 5.2. We will gradually transform this dependency graph between non-terminals into an import graph between modules.

The graph thus obtained contains cycles (or strongly-connected compo-

statement → *variable* **assignop** *expression*
| *procedure_statement*
| *compound_statement*
| **if** *expression* **then** *statement* **else** *statement*
| **while** *expression* **do** *statement*

variable → **id** | **id** [*expression*]

procedure_statement → **id** | **id** (*expression_list*)

compound_statement → **begin** *optional_statements* **end**

optional_statements → *statement_list* | ϵ

statements_list → *statement* | *statements_list* **;** *statement*

expression_list → *expression* | *expression_list* **,** *expression*

expression → *simple_expression* | *simple_expression* **relop** *simple_expression*

simple_expression → *term* | *sign term* | *simple_expression* **addop** *term*

term → *factor* | *term* **mulop** *factor*

factor → **id** | **id** (*expression_list*) | **num** | (*expression*) | **not** *factor*

sign → + | −

Figure 5.1: The grammar for non-terminal *statement*

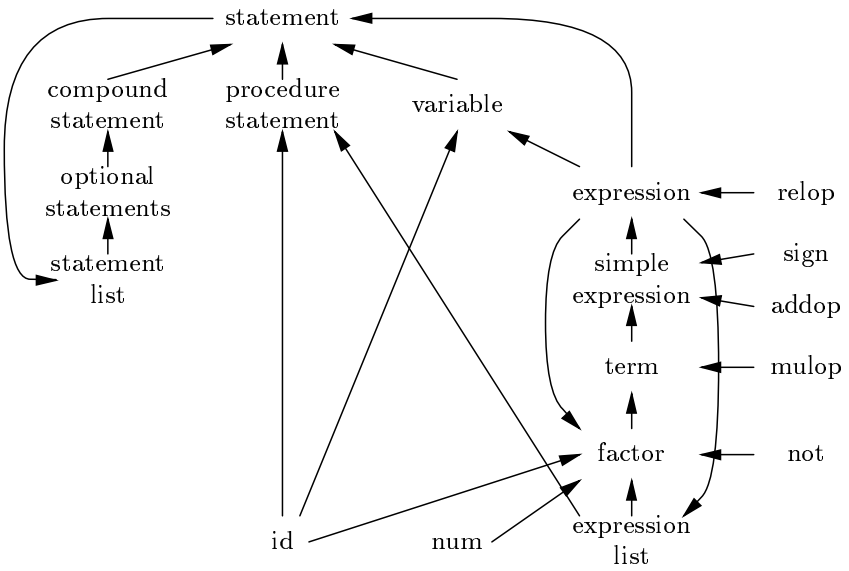


Figure 5.2: The use-def graph of the grammar for *statement*

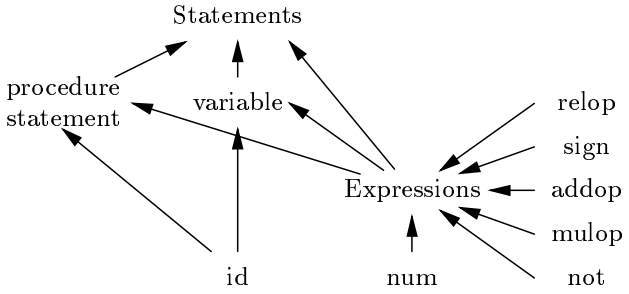


Figure 5.3: The graph of Fig. 5.2 after removing cycles

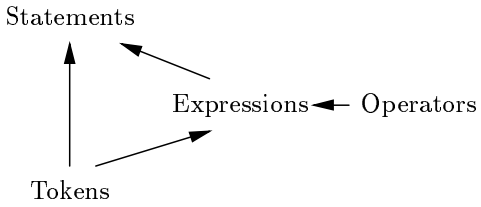


Figure 5.4: The graph of Fig. 5.3 with unified nodes

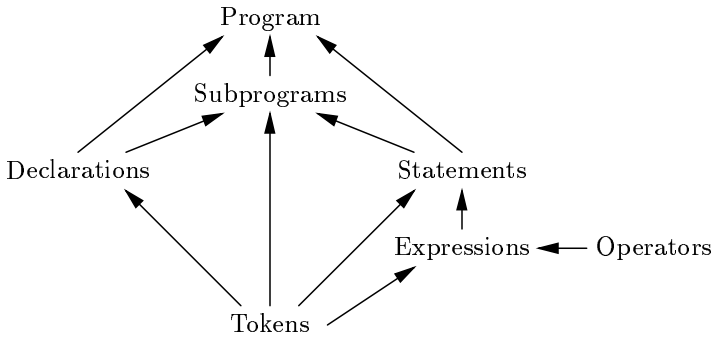


Figure 5.5: The modular structure for the entire grammar

nents). If nodes $N_1 \cdots N_n$ are strongly-connected, we take them together into one node. As a result, the nodes for the non-terminals *statement* and *expression*, and all nodes in the cycle emanating from them, are replaced by the modules *Statements* and *Expressions*.[1] Having done this, the graph of Fig. 5.2 collapses to the one of Fig. 5.3.

The graph as it is now introduces many small and uninteresting modules. We prefer to take nodes together which describe objects of the same kind. For example, *relop*, *sign*, *addop*, *mulop* and *not* describe expression operators and will be taken together into one module *Operators*. The non-terminals *procedure-statement* and *variable* are only needed to describe the more important non-terminal *statement*, and do not really represent a new concept. So we add these to module *Statements*. Finally, we take *id* and *num* together into one module *Tokens* as they describe closely related objects. The graph of Fig. 5.3 then becomes as in Fig. 5.4.

One could argue that the link between *Tokens* and *Statements* could be discarded, as module *Tokens* is already imported in *Statements* via *Expressions*. We prefer to keep the link explicit, however, as it represents the fact that objects of module *Tokens* are used by module *Statements* directly. This additional import relation does not have any effect in the ASF+SDF formalism.

If we perform the same process for the complete grammar of the subset of Pascal, we end up with the modular structure as depicted in Fig. 5.5. This subdivision results in reasonably sized modules, which all describe clear subconcepts of the entire language.

[1] We remove the cycles only for aesthetic reasons: The ASF+SDF system handles them by automatically combining all modules in a cycle and displaying a warning message.

The graph could be simplified even further by joining modules *Program* and *Subprograms*. We prefer to keep these distinct however, as a separate module *Subprograms* implies that a separate editor will be generated which only allows function and procedure declarations to be edited.

5.3 Translating BNF rules into SDF functions

The translation from a BNF grammar into an SDF definition is quite easy. For example, the BNF rule

statement → **while** *expression* **do** *statement*

just corresponds to an SDF context-free function

```
while Expression do Statement -> Statement.
```

Non-terminals, which are called *sorts* in SDF, must start with an uppercase letter and be declared in a sorts section. Terminals must be quoted, unless they consist entirely of letters and start with a lowercase character.

In the lexical syntax section of an SDF definition one defines the internal structure of tokens and the characters that will serve as layout. In most cases it is harder to translate lexical conventions into SDF, as SDF only knows a few regular expression operators and forbids ϵ-rules in the lexical syntax part.

5.3.1 Module *Tokens*

The rules given in [ASU86, appendix A] for tokens *id* and *num* are the following:

letter	→	[a-zA-Z]
digit	→	[0-9]
id	→	letter (letter | digit)*
digits	→	digit digit*
optional-fraction	→	. digits | ϵ
optional-exponent	→	(E (+ | − | ϵ) digits) | ϵ
num	→	digits optional-fraction optional-exponent

These rules can be expressed in SDF as follows:

```
exports
    sorts
        Id Num Letter Digit Letter-or-Digit Digits Fraction Exponent
    lexical syntax
        [a-zA-Z]                -> Letter
        [0-9]                   -> Digit
        Letter                  -> Letter-or-Digit
        Digit                   -> Letter-or-Digit
        Letter Letter-or-Digit* -> Id

        Digit+                  -> Digits
        "." Digits              -> Fraction
        "E" Digits              -> Exponent
        "E" [+\-] Digits        -> Exponent
        Digits                  -> Num
        Digits Fraction         -> Num
        Digits Exponent         -> Num
        Digits Fraction Exponent -> Num
```

This SDF definition of module *Tokens* is still quite hard to read and contains many sorts which could be circumvented easily. We consider the following SDF definition of module *Tokens* more appropriate.

```
exports
    sorts
        Id Digits Num
    lexical syntax
        [a-zA-Z] [a-zA-Z0-9]*              -> Id

        [0-9]+                             -> Digits

        Digits                             -> Num
        Digits "." Digits                  -> Num
        Digits "." Digits "E" Digits       -> Num
        Digits "." Digits "E" [+\-] Digits -> Num
        Digits "E" Digits                  -> Num
        Digits "E" [+\-] Digits            -> Num
```

Layout

The rules given in [ASU86, appendix A] for comments and blanks are:

1. Comments are surrounded by { and }. They may not contain a }. Comments may appear after any token.

2. Blanks between tokens are optional, with the exception that keywords must be surrounded by blanks, newlines, the beginning of the program or the final dot.

We express these in SDF as follows:

```
lexical syntax
    [ \t\n]                 -> LAYOUT
    "{" ~[}]* "}"           -> LAYOUT
```

The predefined sort `LAYOUT` can be used to declare which parts of the input have no meaning as separate lexical constructs. The first rule defines the space, the tab (`\t`) and the newline (`\n`) as layout. The second rule defines the following sequences as layout: A `{`, followed by zero or more characters which may not be `}`, finished by a `}`. This definition carefully disallows `}` inside comments, as the scanner generated always tries to find the longest match. A sequence "`{jan} 7 {rekers}`" would otherwise be recognized as one long comment string, instead of two short ones with an integer in between.

We add these two layout-rules to module *Tokens*, as the modular structure as designed in Section 5.2 will export these to all modules which need to separate lexical constructs.

5.3.2 Module *Operators*

The main question to answer for module *Operators* is whether the rules should be in the lexical or in the context-free syntax part. The difference is that literals in the context-free syntax part are given priority over tokens described by the lexical part. These two SDF definitions differ in the place where "mod" is declared:

```
exports
    sorts Id Mulop
    lexical syntax
        [a-z]+ -> Id
        mod    -> Mulop
```

```
exports
    sorts Id Mulop
    lexical syntax
        [a-z]+ -> Id
    context-free syntax
        mod    -> Mulop
```

The first one recognizes "mod" both as *Id* and *Mulop*, the second one as *Mulop* alone.

It is not entirely clear from [ASU86, appendix A] whether the expression operators should be interpreted as keywords or not. Here we have made the choice that they are reserved words and we place them in a context-free syntax section.

```
exports
    sorts
        Sign Relop Addop Mulop Not
    context-free syntax
        "+"  -> Sign        "-"  -> Sign
        "="  -> Relop       "<>" -> Relop       "<"  -> Relop
```

```
"<=" -> Relop       ">=" -> Relop       ">"  -> Relop
"+"  -> Addop      "-"  -> Addop       or   -> Addop
"*"  -> Mulop      "/"  -> Mulop       div  -> Mulop
mod  -> Mulop      and  -> Mulop
not  -> Not
```

5.3.3 Module *Expressions*

The first translation of the rules for the expressions in [ASU86, appendix A] would be the following SDF definition. This definition however can be improved on several points.

```
imports
    Tokens Operators

exports
    sorts
        Expression-list Expression Simple-expression Term Factor
    context-free syntax
        Expression                                      -> Expression-list
        Expression-list "," Expression                  -> Expression-list

        Simple-expression                               -> Expression
        Simple-expression Relop Simple-expression       -> Expression

        Term                                            -> Simple-expression
        Sign Term                                       -> Simple-expression
        Simple-expression Addop Term                    -> Simple-expression

        Factor                                          -> Term
        Term Mulop Factor                               -> Term

        Id                                              -> Factor
        Id "(" Expression-list ")"                      -> Factor
        Num                                             -> Factor
        "(" Expression ")"                              -> Factor
        Not Factor                                      -> Factor
```

Improving the behaviour of the editor created

The syntax directed behaviour of the editor created for an SDF module is completely determined by the abstract syntax defined in that module. This behaviour concerns the manner in which the focus can be moved through the text and the possibilities offered by the expand menu.

A consequence of the current version of module *Expressions* is, for example, that two expand steps are needed to insert a +-expression. First a not yet filled in `<Expression>`-hole must be expanded to

```
<Simple-expression> <Addop> <Term>,
```

and next `<Addop>` to `+`.[1]

To improve this expand-behaviour, we rewrite the SDF definition of module *Expressions* such that the intermediate steps for the operators are omitted. This has as consequence that module *Operators* disappears.

```
imports
    Tokens

exports
    sorts
        Expression-list Expression Simple-expression Term Factor
    context-free syntax
        Expression                                      -> Expression-list
        Expression-list "," Expression                  -> Expression-list

        Simple-expression                               -> Expression
        Simple-expression "=" Simple-expression         -> Expression
        Simple-expression "<>" Simple-expression        -> Expression
        Simple-expression "<" Simple-expression         -> Expression
        Simple-expression "<=" Simple-expression        -> Expression
        Simple-expression ">=" Simple-expression        -> Expression
        Simple-expression ">" Simple-expression         -> Expression

        Term                                            -> Simple-expression
        "+" Term                                        -> Simple-expression
        "-" Term                                        -> Simple-expression
        Simple-expression "+" Term                      -> Simple-expression
        Simple-expression "-" Term                      -> Simple-expression
        Simple-expression or Term                       -> Simple-expression

        Factor                                          -> Term
        Term "*" Factor                                 -> Term
        Term "/" Factor                                 -> Term
        Term div Factor                                 -> Term
        Term mod Factor                                 -> Term
        Term and Factor                                 -> Term

        Id                                              -> Factor
        Id "(" Expression-list ")"                      -> Factor
        Num                                             -> Factor
        "(" Expression ")"                              -> Factor
        not Factor                                      -> Factor
```

This results in just one step to expand `<Expression>` to

```
<Simple-expression> + <Term>.
```

[1] The step from *Simple-expression* to *Expression* is taken automatically by the editor. This happens for all injections of the form $A \rightarrow B$.

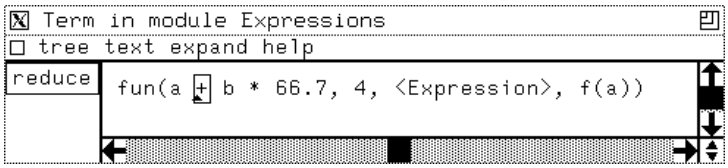


Figure 5.6: A click on the "+" just places the focus on the "+"

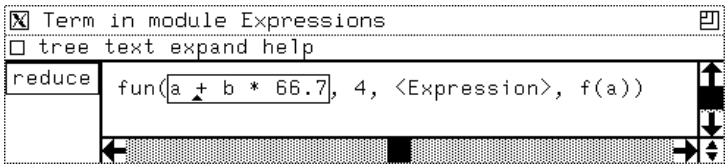


Figure 5.7: A click on the "+" now shows the complete expression

Another consequence of the modification is that the so-called "clicking behaviour" improves. With the former definition, a click at the "+" in

```
a + b * 66.7
```

resulted in a focus around the "+" alone (see Fig. 5.6). The user of the editor has to zoom out one level in order to see that the "+" takes "a" on the one side and "`b * 66.7`" on the other side as its arguments. With the modified definition, this is displayed directly (as is shown in Fig. 5.7).

The syntax directed editor always tries to place the focus around the smallest subtree containing the character clicked at. In the previous definition, the "+" introduced a separate subtree, which is no longer the case with the modified definition.

Using the SDF priority declarations

The SDF definition we now have, codes the priority relations between the different expression operators by using four expression levels: *Expression*, *Simple-expression*, *Term* and *Factor*. This is quite cumbersome to write and can be expressed much easier by using the priority mechanism of SDF.

There are three different disambiguation constructs which may be used in an SDF definition.

- Context-free functions may be provided with an associativity attribute. For example, `E "+" E -> E {left}` states that this rule is left asso-

ciative, and that a sentence `a + b + c` should be read as `(a + b) + c`.

- In the priorities section one can define the relative priority between rules. For example,

```
priorities
    E "*" E -> E > E "+" E -> E
```

states that the `"*"` binds stronger than the `"+"`, and that `a + b * c` should be read as `a + (b * c)`, and `a * b + c` as `(a * b) + c`.

- One can also define associativities between rules in the priorities section. For example,

```
priorities
    {left: E "+" E -> E, E "-" E -> E}
```

states that these rules are mutual left associative. This means that `a + b - c` should be interpreted as `(a + b) - c`, and `a - b + c` as `(a - b) + c`.

The rules in the priorities section may be abbreviated to their keyword skeleton, provided that this skeleton is unique.

The translation from priority relations coded in a grammar to SDF priority declarations is not trivial and should be performed with great care. General rules cannot be given for this translation, as the two mechanism have different expressive power. We refer to [HHKR89, section 6] for a more in depth explanation of the disambiguation constructs of SDF.

```
imports
    Tokens

exports
    sorts
        Expression Expression-list
    context-free syntax
        Expression                              -> Expression-list
        Expression-list "," Expression          -> Expression-list

        Expression "=" Expression               -> Expression {non-assoc}
        Expression "<>" Expression              -> Expression {non-assoc}
        Expression "<" Expression               -> Expression {non-assoc}
        Expression "<=" Expression              -> Expression {non-assoc}
        Expression ">=" Expression              -> Expression {non-assoc}
```

```
        Expression ">" Expression              -> Expression {non-assoc}

        "+" Expression                         -> Expression
        "-" Expression                         -> Expression
        Expression "+" Expression              -> Expression {left}
        Expression "-" Expression              -> Expression {left}
        Expression or Expression               -> Expression {left}

        Expression "*" Expression              -> Expression {left}
        Expression "/" Expression              -> Expression {left}
        Expression div Expression              -> Expression {left}
        Expression mod Expression              -> Expression {left}
        Expression and Expression              -> Expression {left}
        not Expression                         -> Expression

        Id                                     -> Expression
        Id "(" Expression-list ")"             -> Expression
        "(" Expression ")"                     -> Expression {bracket}
        Num                                    -> Expression

priorities
    {non-assoc: "=", "<>", "<", "<=", ">=", ">"} <
    { "-" Expression -> Expression,
      "+" Expression -> Expression,
      Expression "+" Expression -> Expression,
      Expression "-" Expression -> Expression,
      or } <
    {left: "*", "/", div, mod, and}
priorities
    {left: Expression "+" Expression -> Expression,
           Expression "-" Expression -> Expression,
           or }
```

Using the priorities makes the definition easier to read and results in entries in the expand menu which are just

```
Expression "*" Expression -> Expression
```

instead of

```
Term "*" Factor -> Term.
```

The latter contains too much low level information with which a user should not be troubled.

A second benefit of using the priority declarations is that the editor is able to decide for itself when it is necessary to insert brackets around expressions in order to avoid priority conflicts. Fig. 5.8 shows an edit session in which this happens.

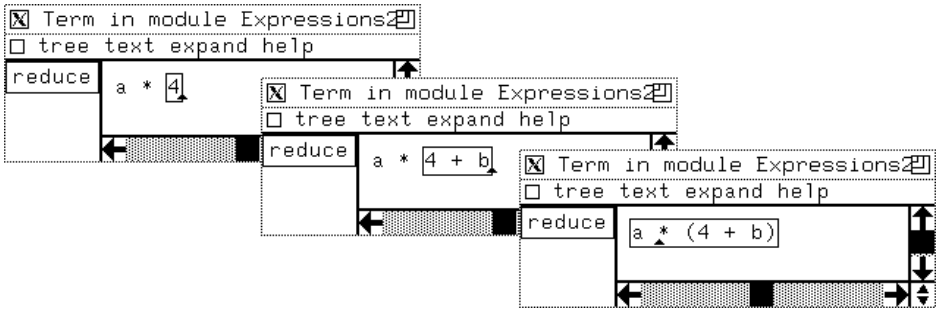


Figure 5.8: Automatic bracket insertion

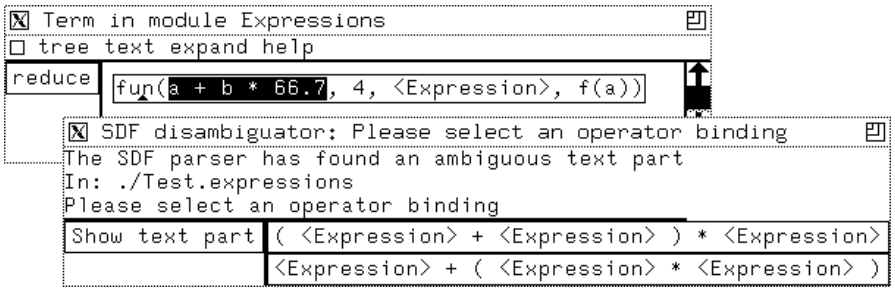


Figure 5.9: Choosing between ambiguous possibilities

Unsolved ambiguities

The internal conduct of the ASF+SDF system regarding disambiguation is the following. First, the parser generates *all* parse trees that are possible according to the definition, disregarding the priority and associativity declarations. Next, these possibilities are weeded, by first removing trees that contain a priority conflict, and next by ordering the trees in accordance to the priorities and choosing the "largest" one. Finally, the remaining parse tree is converted into an abstract syntax tree, which is used in the editor.

In the case that the definition is ambiguous and there is more than one remaining parse tree, the user of the editor is asked to make a choice. Such a question indicates that an additional priority declaration might be needed to disambiguate these automatically in the future.

For example, if we take the most recent definition of module *Expressions* and leave the priority section out, a sentence like "`a + b * 66.7`" would be ambiguous. The user is then asked for a choice as is shown in Fig. 5.9.

5.3.4 Module *Statements*

The rules for the statements in [ASU86, appendix A] can be translated directly into the following SDF module:

```
imports
    Tokens Expressions

exports
    sorts
        Compound-statement Statement
        Optional-statements Statement-list
        Variable Procedure-statement
    context-free syntax
        begin Optional-statements end          -> Compound-statement

        Statement-list                         -> Optional-statements
                                               -> Optional-statements
        Statement                              -> Statement-list
        Statement-list ";" Statement           -> Statement-list

        Variable ":=" Expression               -> Statement
        Procedure-statement                    -> Statement
        Compound-statement                     -> Statement
        if Expression then Statement
                     else Statement            -> Statement
        while Expression do Statement          -> Statement

        Id                                     -> Procedure-statement
        Id "(" Expression-list ")"             -> Procedure-statement

        Id                                     -> Variable
        Id "[" Expression "]"                  -> Variable
```

Using SDF lists

SDF supports the following list constructs which may be used in the left-hand side of context-free functions:

`S*`	Zero or more repetitions of S
`S+`	One or more repetitions of S
`{S t}*`	Zero or more repetitions of S, separated by t
`{S t}+`	One or more repetitions of S, separated by t

Advantages of using these list constructs of SDF, instead of coding lists in the grammar, are

- definitions become shorter and easier to read,

- the trees generated contain flat lists of elements, instead of nested ones, which makes it easier in the semantic part of the definition to obtain an element of a list, and
- the insert-hole functionality of the generated editors becomes available.

The rules for *Optional-statements* and *Statement-list* can then simply be replaced by `{Statement ";"}*`. The definition of module *Statements* becomes in that case:

```
imports
    Tokens Expressions

exports
    sorts
        Compound-statement Statement Variable
    context-free syntax
        begin {Statement ";"}* end              -> Compound-statement

        Compound-statement                      -> Statement
        Variable ":=" Expression                -> Statement
        if Expression then Statement
                    else Statement              -> Statement
        while Expression do Statement           -> Statement
        Id                                      -> Statement
        Id "(" {Expression ","}+ ")"            -> Statement

        Id                                      -> Variable
        Id "[" Expression "]"                   -> Variable
```

The declaration of the sort *Expression-list* and its grammar rules can also be removed from module *Expressions*.

5.3.5 Module *Declarations*

Now that we use the SDF list constructs, the translation of module *Declarations* becomes:

```
imports
    Tokens

exports
    sorts
        Declaration Type Standard-type
    context-free syntax
        var {Id ","}+ ":" Type ";"                      -> Declaration

        Standard-type                                   -> Type
        array "[" Num ".." Num "]" of Standard-type     -> Type

        integer                                         -> Standard-type
```

```
    real                                        -> Standard-type
```

One could consider to remove the sort *Standard-type* and use four rules that define *Type*, like

```
    integer                                     -> Type
    real                                        -> Type
    array "[" Num ".." Num "]" of integer       -> Type
    array "[" Num ".." Num "]" of real          -> Type
```

This would improve the "expand behaviour" again, as a user can then in one step insert an array of reals. However, the sort *Standard-type* is also used in module *Subprograms*; this would mean that the declaration of integer and real functions has to be separated also, which seems less natural. So we choose to leave module *Declarations* as it is.

This deliberation shows that with SDF, in which one defines many syntactic aspects simultaneously, an advantage for one component can be a disadvantage for another. This is in contrast to systems providing a different formalism for each of the subcomponents. These give more freedom to the writer of a specification. Definitions in SDF are, on the other hand, much more compact.

5.3.6 Module *Subprograms*

The rules for function and procedure declarations can also be expressed easily in an SDF module.

```
imports
    Declarations Statements

exports
    sorts
        Subprogram-declaration Subprogram-head Arguments Parameter
    context-free syntax
        Subprogram-head Declaration* Compound-statement ";" ->
                                                    Subprogram-declaration

        function Id Arguments ":" Standard-type ";"         -> Subprogram-head
        procedure Id Arguments ";"                          -> Subprogram-head

        "(" {Parameter ";"}+ ")"                            -> Arguments
                                                            -> Arguments

        {Id ","}+ ":" Type                                  -> Parameter
```

A consequence of this definition however is that if the user clicks at the keyword "`function`", only the function heading will be taken in the focus, instead of the entire function. See Fig. 5.10.

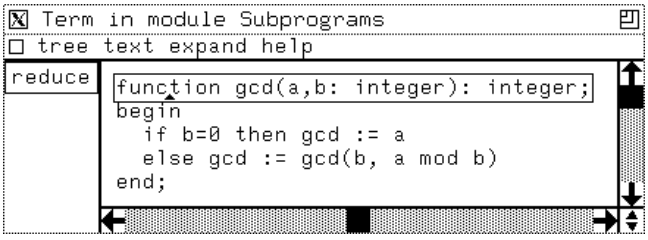


Figure 5.10: A click on "`function`" places only the heading in the focus

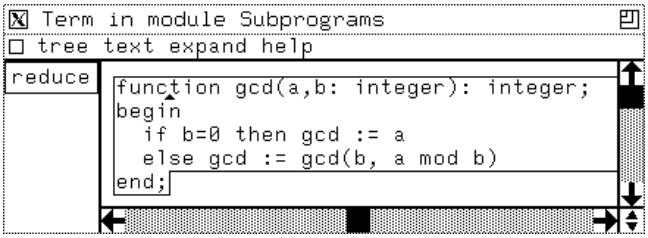


Figure 5.11: The entire function declaration is placed in the focus

We consider this undesirable and we modify the definition in order to obtain the "click behaviour" as shown in Fig. 5.11. This leads to long (and harder to read) rules, but this is the only price paid.

```
imports
    Declarations Statements

exports
    sorts
        Subprogram-declaration Arguments Parameter
    context-free syntax
        function Id Arguments ":" Standard-type ";"
          Declaration*
          Compound-statement ";"                    -> Subprogram-declaration
        procedure Id Arguments ";"
          Declaration*
          Compound-statement ";"                    -> Subprogram-declaration

        "(" {Parameter ";"}+ ")"                    -> Arguments
                                                    -> Arguments
        {Id ","}+ ":" Type                          -> Parameter
```

5.3.7 Module *Program*

The final module, *Program*, imports all other modules and contains just one rule.

```
imports
    Declarations Subprograms Statements

exports
    sorts
        Program
    context-free syntax
        program Id "(" {Id ","}+ ")" ";"
          Declaration*
          Subprogram-declaration*
          Compound-statement "."                    -> Program
```

This concludes our SDF definition of the syntax of a subset of Pascal. We have rewritten this definition several times in order to improve readability and editor behaviour. Such an interactive development of an SDF definition is typical and is supported fully by ASF+SDF system. The final definition is listed in its entirety in [Appendix B of this thesis].

5.4 Concluding remarks

An advantage of SDF over related formalisms is that one definition covers nearly all aspects of syntax analysis: lexical syntax, context-free syntax, abstract syntax and syntax directed behaviour of an editor.[1] This has as a consequence that the writer of a grammar does not have to bother with different formalisms for defining different syntactic aspects, or with exchanging information between the different definitions.

Some general hints which might be of use to writers of SDF definitions are:

- A good modular structure is important, but difficult to achieve for a real language. Invest effort in this structure, as a well designed modular structure results in elegant specifications.
- In the example given in this paper, all rules that define a sort are in the module in which the sort is declared. This is a very strict organization, which is not per se required by the ASF+SDF formalism.

[1] We also automatically derive a pretty-printer from an SDF definition. This turns out to be reasonably easy, and the resulting pretty-printer is of acceptable quality.

One can imagine situations where another organization would be more appropriate. For example, a module *Integers* which imports *Booleans* and defines a comparison function `INT ">" INT -> BOOL`. This rule, although it has a Boolean as result, clearly belongs in module *Integers* and not in module *Booleans*.

- Write the specification bottom-up and test modules with some typical sentences before importing them in others. Also observe the effect of the grammar design on the syntax directed behaviour of the editors generated.
- Use the SDF features like the lists and priority declarations. They make a specification shorter and easier to understand.
- Avoid intermediate sorts. They easily introduce unexpected ambiguities in the semantic part and they make the parser big and parsing slow.
- It is difficult to express complicated regular expressions in the lexical syntax part. In most cases, it seems best to circumvent all intermediate sorts. This will speed up the interaction between scanner and parser considerably anyway.

Chapter 6

An Implementation of SDF

6.1 Introduction

On the basis of the incremental scanner generator for modular regular grammars [Kli91a], the generalized LR parser [Chapter 1 of this thesis], the incremental parser generator for modular grammars [Chapter 2 of this thesis], the restricted parsing method [Chapter 3 of this thesis] and the virtual tree processor [CIL89], we are now able to implement SDF [HHKR89].

We will describe this implementation with two objectives in mind: (1) to document the current implementation and to guide programmers who use it, and (2) to give an impression of the software infrastructure needed to ensure proper operation of the algorithms we use.

The implementation of SDF is based on the notion of a *Syntax Manager*, which is a large data structure containing all information derived from the SDF definition (like, for instance, sorts, lexical functions and context-free functions), all its mutual dependencies, as well as the actual "implementation" derived from it (scanner, parser and abstract syntax definition).

A Syntax Manager is capable of incrementally adding or deleting parts of an SDF definition, of parsing texts in accordance with the current definition, of selecting a subset of the SDF definition to use when parsing, and it can provide information about the SDF definition.

The SDF implementation is currently in use as parsing component of the generic syntax-directed editor GSE [Koo], which in its turn is used in the ASF+SDF system [Kli91b, Hen91, chapter 5]. SDF can also be used to define syntax for the Centaur system [BCD+88]; the generated parser is in this case used by the editor ctedit of Centaur.

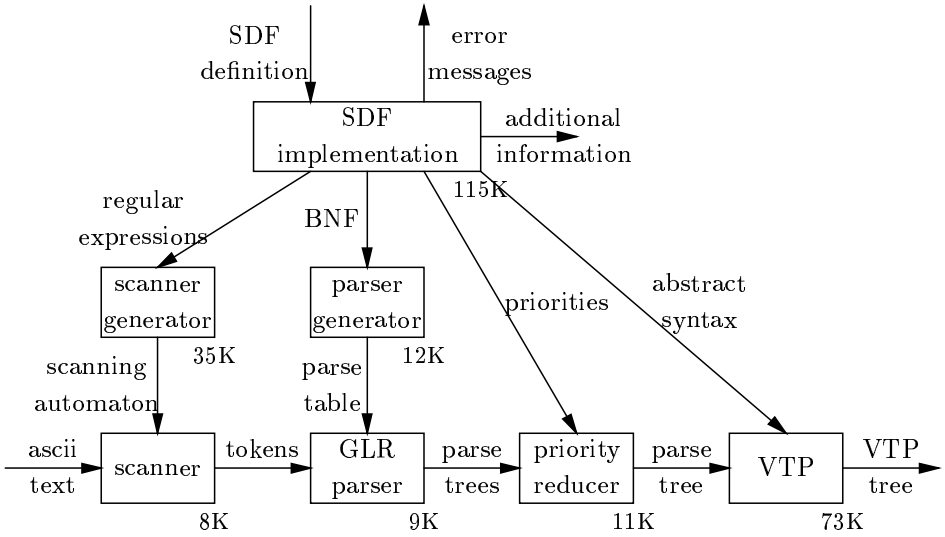


Figure 6.1: The internal structure of the implementation

6.1.1 Internal structure of the implementation

The implementation is written in LeLisp [LeL87], and uses itself to parse SDF definitions. The flow of information through the implementation is roughly as shown in Fig. 6.1. The SDF definition is split into regular expressions, grammar rules, priority declarations and an abstract syntax definition. The regular expressions are compiled into a scanning automaton by the scanner generator MSG [Kli91a]. The grammar rules are compiled into a parse table by a parser generator which combines the features of the incremental parser generator IPG [Chapter 2 of this thesis] and the restricted parsing method described in [Chapter 3 of this thesis]. The abstract syntax is implemented with the VTP [CIL89].

An input text is tokenized by the scanner and translated into (possibly several) parse trees by the Generalized LR parser [Chapter 1 of this thesis]. These trees are then reduced to a single parse tree and converted into an abstract syntax tree. To solve ambiguities the priority declarations are used and the user may be asked to make choices.

To give an idea of the size of the components, we have annotated the boxes in Fig. 6.1 with the size of their corresponding lisp sources in kilobytes. The size of the "SDF implementation", in comparison to the others, gives a good indication of the complexity of the translation of SDF definitions into

definitions for the subcomponents.

6.2 Parsing with a Syntax Manager

6.2.1 The abstract syntax

The trees generated by the parser are in accordance with an abstract syntax which is implemented with the VTP [CIL89]. In VTP, an abstract syntax is called "*formalism*" and mainly consists of a set of "*phyla*" and "*operators*". Phyla roughly correspond to the sorts in an SDF definition. An operator corresponds to a context-free function, a context-free list or a lexical token. A phylum is a named set of operators and each operator belongs to at least one phylum.

Each node in a VTP tree is labeled with an operator. This operator determines to which phylum the operator of each subtree of the node must belong. For example, the SDF context-free function

```
if Exp then Stat -> Stat
```

introduces a binary operator. A tree node typed by this operator must have two subnodes of which the operator of the first one must belong to phylum *Exp*, and the second to *Stat*. The operator of the node itself belongs to (at least) phylum *Stat*.

An SDF list construct, such as `{Stat ";"}+`, defines a variadic operator of which the arguments must all belong to the same phylum, here *Stat*.

If a sort has one or more lexical definitions, an atomic operator is created for that sort. A tree node typed by such an operator does not have children, but an atomic value which contains the string recognized.

An SDF injection does not introduce an operator by itself, but causes other operators to belong to more than one phylum. For example, an injection "`A -> B`" makes A a subsort of B, which means that all operators contained in phylum A are contained in phylum B as well.

6.2.2 Parsing text

The most simple use of a syntax manager to parse a text is by calling the following function:

(#:SDFimpl:parse
SM
kind-of-text

text
requested-phyla) → *VTPtree*

It tries to parse its input text in *text* in accordance with the Syntax Manager *SM* and returns a VTP tree if the parse succeeds; if not, it prints an error message and returns (). The argument *kind-of-text* may be 'string, 'strings or 'filename. The arguments of #:SDFimpl:parse are a subset of the arguments of #:SDFimpl:parse-text and will be explained in more detail in the sequel.

The parsing capabilities of the Syntax Manager will mostly be used from within a syntax-directed editor. Such an editor needs a tree representation of the text in its buffer. On textual modifications it updates the tree incrementally by selecting some subtree that covers the modification. The text belonging to this subtree is parsed and the subtree is replaced by the result of the parse. This kind of use imposes additional constraints on the trees the parser returns

- The tree will, in most cases, have to be of the same phylum as the tree it is replacing.
- To inform the editor about the relationship between text and tree, the tree has to be annotated with position information indicating for each subtree the part of the text it covers.
- The editor may need to know whether a tree replacement will cause a priority conflict.
- If an ambiguity arises during parsing, which can not be solved by the priority declarations, the parser has to ask the calling editor to disambiguate it. This is necessary as an abstract syntax tree can only be generated for a non-ambiguous parse tree.

In order to meet these requirements the SDF implementation also offers the more sophisticated function #:SDFimpl:parse-text, which allows for more options than #:SDFimpl:parse above.

(#:SDFimpl:parse-text

SM	the Syntax Manager
kind-of-text	'string, 'strings or 'filename
text	the text corresponding to *kind-of-text*
requested-phyla	a list of phyla or ()
surrounding-operator	an operator or ()

rank-in-operator	the rank in *surrounding-operator* or ()
start-column	a column number or ()
start-line	a line number or ()
caller	an identification of the caller or ()
) → *parse-result*	

These arguments have the following meaning

SM

The Syntax Manager containing the SDF definition to be used.

kind-of-text

The format of the text, which may be 'string, 'strings or 'filename.

text

The text to be parsed in a format in accordance with *kind-of-text*. If it is 'filename, *text* is interpreted as the name of the file where the actual text is to be found.

requested-phyla

The operator of the resulting VTP tree must belong to one of the phyla in this list. If *requested-phyla* is () all phyla are allowed.

surrounding-operator and *rank-in-operator*

If *surrounding-operator* and *rank-in-operator* are not nil, a check is performed whether the new tree might cause a priority conflict.

Surrounding-operator is the operator of the tree in which the new tree is to be inserted, and *rank-in-operator* indicates at which rank in *surrounding-operator* it will be inserted.

start-column and *start-line*

These two indicate the column and line number which should be assigned to the first character of *text*. This makes it easier for the calling editor to provide the parser with a part of its complete text buffer and still obtain correct position information.

caller

An identification of the editor calling the parser. If ambiguities arise the parser asks *caller* to solve them.

6.2.3 The result of a parse

The result of #:SDFimpl:parse-text is not a VTP tree (like #:SDFimpl:parse of Section 6.2.2 returns), but is an object *parse-result* that contains more information. For each call to #:SDFimpl:parse-text a new instance of this structure is created in order to leave results of previous calls to the parser intact.

```
(defstruct #:SDFimpl:parse-result
    status
    message
    data
    phylum
    layout-only)
```

The field *status* of #:SDFimpl:parse-result indicates to what extent parsing succeeded. In case parsing did not succeed, the *message* field contains a list of strings to inform the user about the kind of error. In appropriate cases *data* contains position information which the calling editor might add to the message given to the user. The *phylum* field indicates to which phylum the generated tree belongs. The *layout-only* field is a Boolean that is set to true if the input text of the parser was found empty (or consisted entirely of layout characters).

The following values are possible for *status*

input-error

One of the arguments of #:SDFimpl:parse-text was of wrong type or could not be used. *Message* contains a detailed error message and the other fields are not set.

scanner-fails

The scanner failed during its scan of the input and returns the token ERROR to the parser. *Data* contains the area in the text (of the size of one character) where the error occurred. See Section 6.2.4 for a description of the position information contained in an area.

parser-fails

No parse tree could be generated for *text*. *Data* contains the area of the last token read by the parser. *Message* will (in the near future) be extended with information about what the parser expected at the point of failure.

internal-conflict

All possible parses contained a priority conflict and were therefore rejected (a function in the SDF definition with a *non-assoc* attribute can cause this quite exceptional status). *Data* contains the area of the conflict.

wrong-phylum

It is possible to parse the text, but there is no tree whose top operator belongs to one of the *requested-phyla*. *Data* contains a list of the phyla which were possible. Section 6.2.5 suggests some precautions an editor can take to avoid this status as much as possible.

unsolved-ambiguity

During the generation of the VTP tree an ambiguity occurred which could not be solved by *caller*. *Data* contains the area in the text where the ambiguity occurred. See Section 6.2.6 for a description on how the calling editor *caller* is asked to solve an ambiguity.

external-conflict

The parse succeeded but the resulting VTP tree will introduce a priority conflict with its environment. *Data* contains the VTP tree. The editor can respond to this status by surrounding the text with brackets (see also Section 6.3.4), it can order a reparse at a higher level in the tree, or it can just let the parse fail.

parse-succeeded

The parse succeeded completely. *Data* contains the VTP tree, and *phylum* contains the phylum chosen.

6.2.4 Position information

If both *start-column* and *start-line* are provided in the call to the parser, the parser returns position information in *area* data structures. Areas can be contained in the *data* field of #:SDFimpl:parse-result (as described in Section 6.2.3), and each subtree in the VTP tree is annotated with areas denoting which part of the input text is covered by the subtree. The name of the *decor* of these annotations is "area".[1] Position information will not be computed if *start-column* or *start-line* is ().

[1]For an explanation of the use of annotations and decors we refer to [CIL89].

An area contains a start and an end coordinate of an area in the text, and is computed with respect to *start-column* and *start-line*. The contents of an *area* can be accessed with the following functions:

(#:SDFimpl:area:bline *area*) → *line number begin*
(#:SDFimpl:area:bcol *area*) → *column number begin*
(#:SDFimpl:area:eline *area*) → *line number end*
(#:SDFimpl:area:ecol *area*) → *column number end*

6.2.5 The phylum of the resulting VTP tree

The top operator of the VTP tree as returned by the parser will always belong to one of the *requested-phyla* provided in the call to the parser. If this is not possible, the parser fails with status *wrong-phylum*. The editor calling the parser should avoid this status as much as possible, by only using *requested-phyla* if it is needed and by providing the most general phyla possible.

If the phyla requested are (), the parser makes a random choice among the possibilities and status *wrong-phylum* will never occur. Which phylum is chosen, is always communicated to the calling editor via the field *phylum* of *parse-result*.

6.2.6 Solving ambiguities

Before the parser can generate an abstract syntax tree, all ambiguities must have been solved. If ambiguities are still present after the priority declarations have been applied, the caller of the parser is asked to make a choice in the following way:

(send 'solve-ambiguity *caller area VTPtrees*) → *VTPtree*

Area contains the sub-area in the text which is ambiguous and *VTPtrees* contains a list of the VTP trees which are possible for that area. The *caller* must return one of these trees. It could in one way or another ask the user to solve the conflict (and maybe re-use previous answers), but could also make a random choice between the possibilities.

If something goes wrong in this communication, the parser fails with status *unsolved-ambiguity*.

6.3 Getting information from a Syntax Manager

A Syntax Manager can provide information about the SDF definition used and about the abstract syntax trees generated.

6.3.1 Abstract syntax

As explained in Section 6.2.1, the trees generated by the parser are in accordance with an abstract syntax which is coded in a *VTP formalism*. This formalism can be obtained by

(#:SDFimpl:get-vtpdef *SM*) → *VTP formalism*

The following function returns the phylum created for sort *sortname*.

(#:SDFimpl:get-phylum *SM sortname*) → *phylum*

To know which operator was created for an SDF construct the following function can be used.

(#:SDFimpl:get-operator *SM kind string*) → *operator*

Kind must be one of 'fun, 'list or 'lex and *string* is as the string returned by #:SDFimpl:get-menu-string (see Section 6.3.4).

An operator can belong to several phyla, but there is always one phylum it was initially created for. One can obtain this "lowest" phylum with

(#:SDFimpl:lowest-phylum *SM operator*) → *phylum*

6.3.2 Trees

To distinguish the different kinds of trees, a function is provided to ask for the "kind" of a tree and its operator.

(#:SDFimpl:tree:kind *tree*) → *kind*

The returned *kind* can be one of *list*, *function*, *constant*, *lexical*, *variable* or *metavar*.

For trees with kind *list* or *function*, the VTP function ({tree}:sons *tree*) will return a list of its children. For trees with kind *lexical*, the VTP function ({tree}:atom_value *tree*) will return the atomic value of *tree*. Trees of kind *constant* do not have children or atomic values. For more information about trees with kind *variable* or *metavar*, the following functions can be used:

(#:SDFimpl:var:name *tree*) → name
(#:SDFimpl:var:class *tree*) → {tree} or {sublist}

(#:SDFimpl:var:phylum *tree*) → phylum

It is also possible to ask an operator for its kind with

(#:SDFimpl:operator:kind *operator*) → *kind*

For this function it is however not possible to make a distinction between kinds *variable* and *metavar*, and these two cases are mapped to *var*.

6.3.3 Metavariables

To allow parsing of incomplete sentences, each phylum has an associated *metavariable representation*. This is a string which may be used as placeholder for a subtree of kind *phylum*.

(#:SDFimpl:get-metavar-string *SM phylum*) → *string*
(#:SDFimpl:get-metavar-tree *SM phylum*) → *tree*

The second function returns a VTP tree for the metavariable.

If a syntax-directed editor wants to provide a facility to expand a metavariable, it needs to know all operators that belong to the phylum of the metavariable. To generate a menu with the different possibilities it needs a string representation of each operator which is recognizable for the user. Also, if the user makes a choice, the editor needs a string representation of the operator that can be parsed. As an example, for the operator of an if-statement the menu-string representation will be something like "`if Exp then Stat -> Stat`" and the expand-string representation "`if <Exp> then <Stat>`".

To obtain the list of the operators that may replace a tree of kind *metavar* the following function can be used

({phylum}:contents (#:SDFimpl:var:phylum *tree*))

For a string representation of *operator* that looks like the construct in the SDF definition it was created for, use

(#:SDFimpl:get-menu-string *SM operator* [*phylum*]) → *string*

For a string representation of *operator* that can be parsed, use

(#:SDFimpl:get-expand-string *SM operator* [*phylum*]) → *string*

If *string* is parsed again, it results in a subtree with top-operator *operator* of which all subtrees are metavariables of the appropriate kind.

6.3.4 Pretty printing

To allow for the derivation of default pretty-printers from an SDF definition, the following functions are provided.

(#:SDFimpl:get-pp-list *SM operator* [*phylum*]) → *pp-list*
(#:SDFimpl:get-brackets *SM phylum*) → *pp-list*

Both functions return a list *pp-list* which contains strings for the keywords and the atom *son* as placeholder for each child. For example, the pp-list of a typical if-statement would be "`("if" son "then" son)`".

For pretty printing trees with kind *list*, the separator of the list is needed. The following function, which may only be used for list operators, returns this.

(#:SDFimpl:get-list-separator *SM operator*) → *string*

It returns an empty string for lists which do not have a separator.

Also, a pretty-printer should know if brackets surrounding a construct are needed in order to ensure that the generated string will, if re-parsed, result in the same tree. For this purpose the following function is available

(#:SDFimpl:priority-conflict
SM
father-operator
kid-operator
rank-in-father) → *Bool*

The function returns *true* if *kid-operator* is in conflict with *father-operator*, and therefore brackets would be needed for the corresponding subtree.

6.3.5 The no-operator

The functions

#:SDFimpl:get-menu-string,
#:SDFimpl:get-expand-string and
#:SDFimpl:get-pp-list

allow an optional phylum argument. Since an operator can have different representations depending on the phylum it has to belong to, this argument serves to discriminate between them. These different representations are caused by SDF functions with a no-operator argument. Take for example this SDF definition

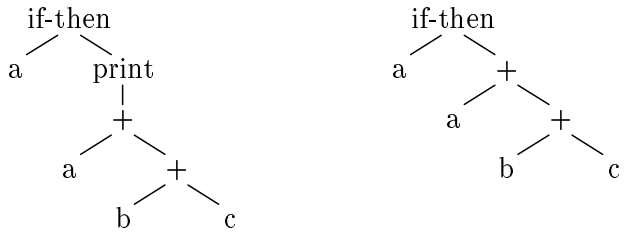


Figure 6.2: Two trees of "if a then print a+b+c"

```
context-free syntax
    if Exp then Stat        -> Stat
    print Exp               -> Stat {no-operator}
    Id                      -> Exp
    Exp "+" Exp             -> Exp {left}
```

Fig. 6.2 shows two abstract syntax trees of the string "if a then print a+b+c". The one in Fig. 6.2(a) is the tree ordinarily generated, the one in (b) is generated if the no-operator attribute is present.

The representation of the operator for "`Exp "+" Exp -> Exp`" now depends on the context in which it is used. The pp-list (see Section 6.3.4) of the highest "+" in Fig. 6.2 should be "`("print" son "+" son)`", while that of the lowest is "`(son "+" son)`". This difference lies in the phylum the operator must belong to, *Stat* or *Exp*. The same holds for the expand-strings and the menu-strings. This means that if a representation for an operator is needed, its phylum should be provided in order to get a correct result. The phylum argument is optional for cases where it is unknown.

6.3.6 CHAR variables

If the SDF definition contains one or more variable declarations over the pre-defined sort "CHAR", a context free function

```
"sort" "(" CHAR+ ")" -> SORT
```

is generated by the implementation for every sort *SORT* that has a lexical definition. These context-free functions make it possible in a semantic formalism to access the contents of lexical tokens (see [HHKR89, section 7.2] for more information).

For the implementation of such a semantic component it is of importance to be able to match the lexical operator of a sort with the operator of the generated context-free function, and vice versa. We therefore introduce the notion of "charpartner". This is an annotation on operators which is used to link the two operators together. If a CHAR variable is present, the charpartner annotation of each lexical operator is set to the operator of its context-free function, and the other way round.

The decor (see [CIL89]) of this charpartner annotation is available through

(#:SDFimpl:SDFlang:charpartner *SM*) → *decor*

6.4 Generating a Syntax Manager

The simplest way to create a Syntax Manager for an SDF definition is to call

(#:SDFimpl:gen-SM *filename*) → *SM*

This function reads an SDF definition from file and generates a Syntax Manager *SM* that implements the definition. Clearly, this function is non-incremental.

6.4.1 Incremental generation

It is also possible to create an empty Syntax Manager, to add various SDF constructs to a Syntax Manager, and to remove them again. After each modification the Syntax Manager can be used immediately.

An empty Syntax Manager is created with

(#:SDFimpl:init *caller*) → *SM*

Its argument *caller* must provide functions add-message, del-message and name, which will be called in the following manner:

(send 'add-message *caller string*) → *MESShandle*
(send 'del-message *caller MESShandle*)
(send 'name *caller*) → *atom*

These are needed for displaying and removing messages describing eventual errors found in the SDF definition.

The functions to add parts of an SDF definition to a Syntax Manager are:

(#:SDFimpl:add-sort *SM caller VTPtree*) → *SMhandle*

(#:SDFimpl:add-lex-function *SM caller VTPtree*) → *SMhandle*
(#:SDFimpl:add-cf-function *SM caller VTPtree*) → *SMhandle*
(#:SDFimpl:add-variable *SM caller VTPtree*) → *SMhandle*
(#:SDFimpl:add-prior-chain *SM caller VTPtree)* → *SMhandles*

(#:SDFimpl:add-prior-relation *SM caller SMhandle1 SMhandle2)* → *SMhandle*
(#:SDFimpl:add-assoc-relation *SM caller kind SMhandles)* → *SMhandle*

The first five of these add functions have a VTP tree as argument. These VTP trees must be generated with the Syntax Manager of SDF itself, which is stored in the variable

#:SDFimpl:stdSDFlang.

Only trees generated by parsing SDF definitions with this Syntax Manager are accepted by the add-functions. The result of all add-functions are *SMhandles* which can be used to identify added constructs later on.

The functions #:SDFimpl:add-prior-relation and #:SDFimpl:add-assoc-relation can be used instead of #:SDFimpl:add-prior-chain. The *SMhandle*'s which are provided to them must have been obtained with the function #:SDFimpl:add-cf-function, as the arguments of priority declarations are context-free functions.

All added constructs can be removed again from a Syntax Manager with the following function:

(#:SDFimpl:del-from-SM *SM SMhandle*)

If an error is found in one of the SDF constructs given to the Syntax Manager, an error message is sent to its *caller*. On removal of the *SMhandle* in question the message will be removed as well. It depends on the error made how much of the construct still is processed.

6.4.2 Facilities for modular syntax analysis

In modular SDF, each module needs a parser which only works in accordance with the module itself and the exported parts of the modules imported in it. As described in [Chapter 3 of this thesis] an efficient implementation for generating and updating a number of parsers which import each other is to generate one parser for the union of all modules and to use a *selection* of this parser to work for each individual module. A selection of a parser only works according to the rules which are enabled in it.

The following functions exist to implement selections:

(#:SDFimpl:new-selection *SM caller*) → *Selection*

(#:SDFimpl:del-selection *Selection*)
(#:SDFimpl:enable Selection *SMhandle*)
(#:SDFimpl:disable Selection *SMhandle*)

In the ASF+SDF system (which implements modular SDF) all modules have their own *Selection*. This system adds the SDF constructs found in each module to one general *SM*, but each *SMhandle* (see Section 6.4) is only enabled in the *Selection* of its own module and, if exported, enabled in the *Selection*'s of the modules which import the module.

The *caller* in #:SDFimpl:new-selection must again be able to respond to the calls *add-message*, *del-message* and *name* (see Section 6.4).

The information functions and the parse functions both accept an *SM* and a *Selection* as Syntax Manager. Some information functions however do not yet use the subset defined in the *Selection*, but use the entire definition.

6.5 Assessment

As might be gathered from the description of its interface, the implementation of SDF is a complicated piece of software. Several reasons can be given for this complexity:

- SDF is a high level syntax specification language. It combines the description of lexical syntax, context-free syntax, abstract syntax and priority declarations in one single formalism. The different components which implement these sub-formalisms all have their own interface and peculiarities. Only the top-level part knows the original SDF definition, and can maintain the cross-references between the objects in the different sub-formalisms.

- The implementation is incremental. As a consequence, all commands given to the sub-components must be stored in some way, in order to be able to undo them again. Next to this, the system must be able to work with (temporarily) incorrect definitions.

- As SDF is used interactively, efficient implementation techniques must be used. These are in general more complex than straightforward solutions.

- SDF contains a reasonable amount of "bells and whistles" which have to compensate, for example, for the fact that the abstract syntax is

derived automatically from the context-free syntax. Suffice to mention injections, bracket rules, no-operator rules, operator names and lists.

- SDF allows modular definitions, while the sub-components that implement the abstract syntax and the priority declarations do not. This shortcoming has to be compensated for by the top-level part.

The complexity of the implementation makes it difficult to maintain the software, and is also reflected in its speed and consumption of memory. As a consequence, large SDF definitions are not handled satisfactorily in the ASF+SDF system.

Should SDF ever be re-implemented in order to comply with new demands, we suggest the following measures to decrease the complexity of the implementation:

- A modular version of the abstract syntax component would circumvent many problems present in the current implementation.
- The priority declarations should be handled by a separate component offering a clear interface.
- The implementation would become simpler if context-free lists would be declared in a separate section, and would no longer appear on themselves in context-free functions or variable declarations.
- The automatic introduction of context-free functions in the case of CHAR variables (see Section 6.3.6) is complicated and could also be left to the writer of a specification.
- Less interesting features, like, for instance, naming of operators and no-operator functions, should be removed from SDF.

On the whole, by being able to use a tool like the SDF implementation, a great burden has been taken from the writer of a specification, who used to have the sub-formalisms only and had to maintain all cross-references by himself. The fact that SDF is incremental makes it very easy to tune a definition interactively.

Appendix A

The algorithms in Lisp

To make it as easy as possible for the reader to experiment with the algorithms discussed in this thesis, we present the following algorithms in their lisp implementation.[1]

- Appendix A.1: The GLR recognizer of chapter 1.
- Appendix A.2: The GLR parser of chapter 1.
- Appendix A.3: The incremental parser generator IPG of chapter 2, extended with the restricted parsing facilities of chapter 3.
- Appendix A.4: Some utilities to read in a grammar, print a parse tree and print a parse table.

In this appendix, we do not explain how the algorithms *work*, but only how they can be *used*. The corresponding papers are recommended for an in depth explanation.

The implementations presented here are much more concise than the versions used in the actual implementation of SDF:

- All SDF specific parts have been removed, the input grammars just consist of BNF rules.
- The BNF rules are labeled. These labels can be used to select a group of rules for restricted parsing.

[1] To obtain the code in electronic form a request can be mailed to *rekers@cwi.nl*.

- The input for the parsers and recognizers is a list of tokens, as opposed to the algorithms in the implementation of SDF which use a lexical scanner.
- The output of the parser is a, possibly ambiguous, parse tree.

LeLisp

The lisp dialect used is LeLisp [LeL87], which is said to be close to Common Lisp. We have tried to use only a small subset of the functionality of LeLisp, in order to facilitate porting the code to other lisp dialects. We expect the reader to be reasonably familiar with lisp, but some explanations of specific LeLisp constructs are necessary.

"#:sys-package:colon"

All our lisp files start with a declaration like

(defvar #:sys-package:colon 'recognizer).

This serves to structure the global name space, as it has the effect that all symbols in the file starting with a colon will be prefixed by "#:" and the name given. In the above example the prefix will be "#:*recognizer*". We use this mechanism to keep function names short within the file, while name clashes with functions in other files are less likely to occur. Also, if a function from another file is used, it stands out because of the prefix needed.

For example, in a file with the above declaration,
(defun :PARSEWORD ($\cdots$) $\cdots$)
stands for
(defun #:recognizer:PARSEWORD ($\cdots$) $\cdots$),
and
(setq parsers (:PARSEWORD $\cdots$))
will be interpreted as
(setq parsers (#:recognizer:PARSEWORD $\cdots$)).

Structures

Structures can be declared with "defstruct". For example,

(defstruct *stacknode state backlinks*)

declares a *stacknode*-object with two fields, *state* and *backlinks*. An instance of this object is created with "(omakeq *stacknode*)". The value of, for example, the field *state* of a *stacknode*-object *object* can be set with

(#:stacknode:state *object value*)

and obtained with

(#:stacknode:state *object*).

It is also possible to set fields when an object is created.

These structures also allow for an object-oriented programming style. For example, for an object with type *stacknode*, "(send 'show *object*)" will result in a call of the function "(#:stacknode:show *object*)". The type of an object can be inspected with "(type-of *object*)".

As we often use structures to create cyclic data structures, it is very convenient that, if we define the function "#:stacknode:prin", this function will be used when printing an object with type *stacknode*. This allows to print an abbreviation of an object instead of its full contents.

"Mapc", "mapcar", "any" and "every"

These are functions to apply a function *f* to all elements of a list *l*.

- "(mapc *f l*)" applies *f* to each element of *l*.
- "(mapcar *f l*)" applies *f* to each element of *l* and returns a list containing the result of the applications.
- "(any *f l*)" returns true if *one* of the applications of *f* to an element of *l* returns true.
- "(every *f l*)" returns true if *all* applications of *f* to the elements of *l* return true.

The function *f* may be defined with "defun", but may also be a *lambda expression*. For example, the following function

```
(defun :GET-STACKNODE (stacknodes state)
  (any
    (lambda (sn)
      (when (eq (#:stacknode:state sn) state)
        sn ))
    stacknodes ))
```

uses *any* with a lambda expression as function argument.

A.1 The GLR Recognizer

This is the LeLisp version of the Generalized LR recognizer as presented in pseudo-code in chapter 1, section 3.

The recognizer can be characterized as follows:

(#:recognizer:PARSE *grammar tokens*) → *Boolean.*

It should be called with a *grammar*-object as generated by the parser generator of appendix A.3, and with a list of tokens. It returns `t` if the tokens could be recognized, `()` otherwise.

```
(defvar #:sys-package:colon 'recognizer)

(defstruct stacknode state backlinks)

(defun :PARSE (grammar tokens)
  (let ( parsers result )
    (setq parsers
      (list (omakeq stacknode state (#:RPG:GET-START-STATE grammar))) )
    (while parsers
      (setq parsers (:PARSEWORD parsers (or (nextl tokens) 'EOF))) )
    result ))

(defun :PARSEWORD (active-parsers token)
  (let ( (for-actor active-parsers)
         for-shifter )
    (while for-actor
      (:ACTOR (nextl for-actor)) )
    (:SHIFTER for-shifter token) ))

(defun :ACTOR (parser)
  (mapc
    (lambda (action)
      (selectq (car action)
        (shift  (newl for-shifter (cons parser (cdr action))) )
        (reduce (:DO-REDUCTIONS parser
                                (#:RPG:rule:length (cdr action))
                                (#:RPG:rule:result (cdr action))
                                t t) )
        (accept (setq result t)) ))
    (#:RPG:ACTION (#:stacknode:state parser) token) ))

(defun :DO-REDUCTIONS (stacknode length result backlink-to-see backlink-seen)
    (if (zerop length)
      (when backlink-seen
        (:REDUCER stacknode (#:RPG:GOTO (#:stacknode:state stacknode) result)))
;   else
      (mapc
        (lambda (stacknode-1)
          (:DO-REDUCTIONS
            stacknode-1
            (1- length)
```

```
            result
            backlink-to-see
            (or backlink-seen (eq stacknode-1 backlink-to-see)) ))
        (#:stacknode:backlinks stacknode) )))

(defun :REDUCER (stacknode-1 gotostate)
  (let ( stacknode )
    (when gotostate
      (setq stacknode (:GET-STACKNODE active-parsers gotostate))
      (if stacknode
        (unless (memq stacknode-1 (#:stacknode:backlinks stacknode))
          (#:stacknode:backlinks stacknode
                    (cons stacknode-1 (#:stacknode:backlinks stacknode)))
          (mapc
            (lambda (sn)
              (unless (memq sn for-actor)
                (mapc
                  (lambda (action)
                    (when (eq (car action) 'reduce)
                      (:DO-REDUCTIONS sn
                          (#:RPG:rule:length (cdr action))
                          (#:RPG:rule:result (cdr action))
                          stacknode-1 ()) ) )
                  (#:RPG:ACTION (#:stacknode:state sn) token) )))
            active-parsers ) )
;     else
        (setq stacknode
          (omakeq stacknode state gotostate backlinks (list stacknode-1)))
        (newl active-parsers stacknode)
        (newl for-actor stacknode) ) )))

(defun :SHIFTER (for-shifter token)
  (let ( stacknode new-active-parsers )
    (mapc
      (lambda ( (stacknode-1 . state) )
        (setq stacknode (:GET-STACKNODE new-active-parsers state))
        (if stacknode
          (#:stacknode:backlinks stacknode
                  (cons stacknode-1 (#:stacknode:backlinks stacknode)))
;       else
          (setq stacknode
            (omakeq stacknode state state backlinks (list stacknode-1)))
          (newl new-active-parsers stacknode) ))
      for-shifter )
    new-active-parsers ))

(defun :GET-STACKNODE (stacknodes state)
  (any
    (lambda (sn)
      (when (eq (#:stacknode:state sn) state)
        sn ))
    stacknodes ))
```

A.2 The GLR parser

This is the LeLisp version of the Generalized LR parser as presented in pseudo-code in chapter 1, section 5, and is an extension of the recognizer in appendix A.1.

The parser can be characterized as follows:

(#:parser:PARSE *grammar tokens*) → *parse-graph*

It should be called with a *grammar*-object as generated by the parser generator of appendix A.3, and with a list of tokens. It returns a graph-like representation of all possible parses in *parse-graph* (or `()` if parsing failed), which can be printed with routine *SHOW-TREE* of appendix A.4.

The resulting structure *parse-graph* is formed by instances of objects of three types, *rule node*, *symbol node* and *term node*. All these have a field *id* which is not used in the algorithms of the parser itself, but only by routine *SHOW-TREE*. A parse graph will in most cases contain many shared objects and may even be cyclic. In all lisp routines, objects can be distinguished by using the lisp function `eq`, which compares whether two objects occupy the same memory location. This kind of comparison is less fit for humans. Therefore, routine *SHOW-TREE* assigns a unique number to the *id* field of each object, and uses these to identify the objects in the printed output.

```
(defvar #:sys-package:colon 'parser)

(defstruct stacknode    state           stacklinks )
(defstruct stacklink    treenode        backlink )

(defstruct rulenode     rule            elements        cover        id)
(defstruct symbolnode   symbol          possibilities   cover        id)
(defstruct termnode     token           string          cover        id)

(defun :PARSE (grammar tokens)
  (let ( parsers result (position 0) )
    (setq parsers
      (list (omakeq stacknode state (#:RPG:GET-START-STATE grammar))) )
    (while parsers
      (setq parsers (:PARSEWORD parsers)) )
    (when result
      (#:stacklink:treenode (car (#:stacknode:stacklinks result))) ) ))

(defun :PARSEWORD (active-parsers)
  (let ( (for-actor active-parsers)
         (token (or (nextl tokens) 'EOF))
         generated-rulenodes generated-symbolnodes
         for-shifter )
    (while for-actor
```

```
      (:ACTOR (nextl for-actor)) )
    (:SHIFTER for-shifter token (incr position)) ))

(defun :ACTOR (parser)
  (mapc
    (lambda (action)
      (selectq (car action)
        (shift          (newl for-shifter (cons parser (cdr action))) )
        (reduce         (:DO-REDUCTIONS parser ()
                                (#:RPG:rule:length (cdr action))
                                (cdr action) t t) )
        (accept         (setq result parser) )))
    (#:RPG:ACTION (#:stacknode:state parser) token) ))

(defun :DO-REDUCTIONS (#:stacknode treenodes length rule link-to-see link-seen)
    (if (zerop length)
      (when link-seen
        (:REDUCER stacknode treenodes rule) )
;   else
      (mapc
        (lambda (link)
          (:DO-REDUCTIONS
            (#:stacklink:backlink link)
            (cons (#:stacklink:treenode link) treenodes)
            (1- length)
            rule
            link-to-see
            (or link-seen (eq link link-to-see)) ) )
        (#:stacknode:stacklinks stacknode) )))

(defun :REDUCER (stacknode-1 treenodes rule)
  (let ( (symbol (#:RPG:rule:result rule))
         (rulenode (:GET-RULENODE rule treenodes))
         state stacknode link )
    (unless (setq state (#:RPG:GOTO (#:stacknode:state stacknode-1) symbol))
      (return) )
    (setq stacknode (:GET-STACKNODE active-parsers state))
    (if stacknode
      (unless
        (any
          (lambda (link)
            (when (and (eq stacknode-1 (#:stacklink:backlink link))
                       (eq (type-of (#:stacklink:treenode link)) 'symbolnode) )
              (:ADD-RULENODE (#:stacklink:treenode link) rulenode)
              t ))
          (#:stacknode:stacklinks stacknode) )
        (setq link (omakeq stacklink
                       treenode (:GET-SYMBOLNODE symbol rulenode)
                       backlink stacknode-1 ))
        (#:stacknode:stacklinks stacknode
          (cons link (#:stacknode:stacklinks stacknode)))
        (mapc
          (lambda ( sn )
            (unless (memq sn for-actor)
              (mapc
```

```
               (lambda ( action )
                 (when (eq (car action) 'reduce)
                   (:DO-REDUCTIONS sn ()
                               (#:RPG:rule:length (cdr action))
                               (cdr action) link ()) ) )
               (#:RPG:ACTION (#:stacknode:state sn) token) )))
          active-parsers ) )
;   else
      (setq stacknode
        (omakeq stacknode
            state state
            stacklinks (list (omakeq stacklink
                    treenode (:GET-SYMBOLNODE symbol rulenode)
                    backlink stacknode-1))))
      (newl for-actor stacknode)
      (newl active-parsers stacknode) )))

(defun :SHIFTER (for-shifter token position)
  (let ( termnode stacknode new-active-parsers link )
    (setq termnode (omakeq termnode
                     token token
                     string (string token)
                     cover (cons position position) ))
    (mapc
      (lambda ( (stacknode-1 . state) )
        (setq link (omakeq stacklink treenode termnode backlink stacknode-1))
        (setq stacknode (:GET-STACKNODE new-active-parsers state))
        (if stacknode
          (#:stacknode:stacklinks stacknode
            (cons link (#:stacknode:stacklinks stacknode)))
;       else
          (setq stacknode (omakeq stacknode state state stacklinks (list link)))
          (newl new-active-parsers stacknode) ))
      for-shifter )
    new-active-parsers ))

(defun :GET-RULENODE (rule treenodes)
  (let ( rulenode )
    (setq rulenode
      (any
        (lambda (r)
          (when (and (eq (#:rulenode:rule r) rule)
                     (every 'eq treenodes (#:rulenode:elements r)) )
            r ))
        generated-rulenodes ))
    (unless rulenode
      (setq rulenode (omakeq rulenode
                       rule rule
                       elements treenodes
                       cover (:COVER treenodes) ))
      (newl generated-rulenodes rulenode) )
    rulenode ))

(defun :COVER (treenodes)
  (let ( start end f )
```

```
    (mapc
      (lambda (treenode)
        (when (setq f (send 'cover treenode))
          (if start
            (setq end (cdr f))
;          else
            (setq start (car f))
            (setq end (cdr f)) )) )
      treenodes )
    (if start (cons start end) nil ) ))

(defun :GET-SYMBOLNODE (symbol rulenode)
  (let ( symbolnode )
    (setq symbolnode
      (any
        (lambda (n)
          (when (and (eq symbol (#:symbolnode:symbol n))
                     (equal (#:rulenode:cover rulenode)
                            (#:symbolnode:cover n)) )
            n ) )
        generated-symbolnodes ))
    (if symbolnode
      (:ADD-RULENODE symbolnode rulenode)
;    else
      (setq symbolnode (omakeq symbolnode symbol symbol
                               possibilities (list rulenode)
                               cover (#:rulenode:cover rulenode) ))
      (newl generated-symbolnodes symbolnode) )
    symbolnode ))

(defun :ADD-RULENODE (symbolnode rulenode)
  (unless (memq rulenode (#:symbolnode:possibilities symbolnode))
    (#:symbolnode:possibilities symbolnode
      (cons rulenode (#:symbolnode:possibilities symbolnode))) ))

(defun :GET-STACKNODE (stacknodes state)
  (any
    (lambda (sn)
      (when (eq (#:stacknode:state sn) state)
        sn ))
    stacknodes ))
```

A.3 The parse table generator

This is the lisp implementation of the lazy and incremental parser generator IPG of chapter 2, extended with the facilities for restricted parsing of chapter 3. The algorithm works for general context-free grammars, and it generates an LR(0) parse table, which may contain shift-reduce and reduce-reduce conflicts. These tables can be used to control the GLR recognizer of appendix A.1 and the GLR parser of appendix A.2.

The interface offered by the parser generator can be divided in functions for defining the grammars and the selections to use, and functions providing parse table information.

The following functions are available for the definition of the grammar:

(#:RPG:INIT-GRAMMAR *start-symbol*) → *grammar*

This function initializes a *grammar*-object and should be called with a symbol that will serve as start symbol.

Rules are added to a *grammar*-object with

(#:RPG:ADD-RULE *grammar label result elements*)

Label should be some atom and is used to refer to the rule in selections. Labels need not to be unique among rules. *Result* is the result non-terminal of the rule, and *elements* should be a list of zero or more terminals and non-terminals. Both terminals and non-terminals are atoms, and the only distinction in the parser generator between the two is that the latter appear as *result* in some rule. The tokens read by the parser may be terminals and non-terminals, which allows for parsing text containing non-terminal holes.

A rule can be removed from a *grammar*-object with

(#:RPG:DEL-RULE *grammar label result elements*)

A *grammar*-object is at all times ready to be used by a parser, but the selection to work for should be mentioned first with

(#:RPG:RESTRICT-PARSER *grammar selection*)

Selection can be `'all`, which states that all rules are selected and that the restricted parsing facility is not to be used, or it can be a list of labels. In the latter case a parser controlled by *grammar* will work as if only rules whose label is in *selection*, are part of the grammar.

The other functions in the interface of the parser generator deal with the parse table it generates. Recall that the parser and the recognizer are

called with a grammar object as argument. The parse table must therefore be accessible from this grammar object. The state in which parsing must start can be obtained by:

(#:RPG:GET-START-STATE *grammar*) → *state*

For each state, the action and goto entries in the parse table can be obtained by *ACTION* and *GOTO*.

(#:RPG:ACTION *state symbol*) → *actions*

Each of these *actions* is one of (shift *state*′), (reduce *rule*) or (accept).

(#:RPG:GOTO *state symbol*) → *state*′ or ()

By using these *state*-objects, we avoid an indirection via state numbers and an actual parse table.

The test in routine *EXPAND*, whether an itemset with a given kernel already exists, used to be an expensive one. Profiling information learned us that about half of the time taken by the parse table generator was spent in routine *GET-ITEMSET-WITH-KERNEL*, which performs this test. We therefore use a hashing technique here.

```
(defvar #:sys-package:colon 'RPG)

(defstruct :rule     ; definition of the structure of a grammar rule
        elements                ; list of grammar symbols
        result                  ; the result non-terminal
        label                   ; the label of the rule
        selected                ; set to t if label is in current set
        id)                     ; unique identifier for this rule

(defun :rule:length (rule) (length (:rule:elements rule)))

(defstruct :itemset     ; defintion of the structure of an itemset
        type                    ; can be initial, complete, dirty, specialized
        kernel                  ; this is a list of dotted rules
        trans                   ; transitions, keyed by a symbol
        reductions              ; list of rules which can be reduced
        sp-trans                ; specialized transitions
        sp-reds                 ; specialized reductions
        id                      ; a unique identifier for this itemset
        refcount                ; how many times this itemset is refered to
        grammar)                ; the grammar of the itemset

(defstruct :grammar     ; definition of the structure of a grammar
        rules                   ; assoc-list of form (result rule1 rule2 ... )
        itemsets                ; all itemsets for grammar, as a hash table
        start-symbol            ; start-symbol of the parser
        start-state             ; start state of the graph of itemsets
        current-selection       ; the current selection of labels
```

```
        highest-rule-id         ; highest used rule identifier
        highest-is-id)          ; highest used itemset identifier

(defun :INIT-GRAMMAR (external-start-symbol)
  (let ( start-state start-rule grammar )
    (setq start-rule (omakeq :rule
                        result 'START
                        elements (list external-start-symbol)
                        selected t                      ; always selected
                        id 0 ))
    (setq start-state (omakeq :itemset
                        type 'initial
                        id 0
                        kernel (list (cons start-rule 0))
                        refcount 1))
    (setq grammar (omakeq :grammar
                        start-symbol 'START
                        start-state start-state
                        itemsets (:HASH-INITG start-state)
                        highest-rule-id 0
                        highest-is-id 0 ))
    (:itemset:grammar start-state grammar)
    grammar ))

(defun :GET-START-STATE (grammar)
  (:grammar:start-state grammar) )

(defun :RESTRICT-PARSER (grammar selection)
  (:grammar:current-selection grammar selection)
  (:FOR-ALL-ITEMSETS grammar
    (lambda (is)
      (when (eq (:itemset:type is) 'specialized)
        (:itemset:type is 'complete) ) ))
  (:FOR-ALL-RULES grammar
    (lambda (rule) (:SET-SELECTED grammar rule)) ) )

(defun :SET-SELECTED (grammar rule)
  (if (or (eq (:grammar:current-selection grammar) 'all)
          (memq (:rule:label rule) (:grammar:current-selection grammar)) )
      (:rule:selected rule t)
      (:rule:selected rule ()) ))

(defun :ADD-RULE (grammar label result elements)
  (let ( (new-id (1+ (:grammar:highest-rule-id grammar)))
         (rules (:grammar:rules grammar))
         (start-state (:grammar:start-state grammar))
         rule )

    (:grammar:highest-rule-id grammar new-id)
    (setq rule (omakeq :rule
                        elements elements
                        result result
                        label label
                        id new-id ))
    (putassoc rules result rule)
```

```
    (:grammar:rules grammar rules)
    (:FOR-ALL-ITEMSETS grammar
      (lambda (is)
        (when (and (memq (:itemset:type is) '(complete specialized))
                   (assq result (:itemset:trans is)))
            (:itemset:type is 'dirty) ) ))
    (:SET-SELECTED grammar rule)
    rule ))

(defun :DEL-RULE (grammar label result elements)
  (let ( (rules (:grammar:rules grammar))
         (start-state (:grammar:start-state grammar))
         rule )
    (setq rule
      (any
        (lambda (probe)
          (when (and (equal (:rule:elements probe) elements)
                     (eq (:rule:label probe) label) )
             probe ))
        (cassq result rules) ) )
    (when rule
      (:grammar:rules grammar (remassoc rules result rule))

      (:FOR-ALL-ITEMSETS grammar
        (lambda (is)
          (when (and (memq (:itemset:type is) '(complete specialized))
                     (assq result (:itemset:trans is)))
            (:itemset:type is 'dirty) ) )) ) ))

(defun :ACTION (state symbol)
  (when (neq (:itemset:type state) 'specialized)
    (when (neq (:itemset:type state) 'complete)
      (:EXPAND-ITEMSET state) )
    (:RESTRICT-STATE state))
  (append (:itemset:sp-reds state)
          (cassq symbol (:itemset:sp-trans state)) ) )

(defun :GOTO (state symbol)
  (cdar (cassq symbol (:itemset:sp-trans state))) )

(defun :RESTRICT-STATE (itemset)
  (let ( (selection (:grammar:current-selection (:itemset:grammar itemset))) )
    (ifn (eq selection 'all)
      (let ( sp-trans sp-reds )
        (mapc
          (lambda ( reduction )
            (when (:rule:selected (cdr reduction))
              (newl sp-reds reduction) ))
          (:itemset:reductions itemset) )
        (mapc
          (lambda ( transition )
            (cond
              ((equal (cadr transition) '(accept))
                (newl sp-trans transition) )
              ((any (lambda ( (rule . dot) ) (:rule:selected rule) )
```

```
                    (:itemset:kernel (cdadr transition)) )
                  (newl sp-trans transition) )
                ) )
            (:itemset:trans itemset) )
          (:itemset:sp-reds itemset sp-reds)
          (:itemset:sp-trans itemset sp-trans)
          (:itemset:type itemset 'specialized) )

        (:itemset:sp-reds itemset (:itemset:reductions itemset))
        (:itemset:sp-trans itemset (:itemset:trans itemset))
        (:itemset:type itemset 'specialized)
      ) ))

(defun :EXPAND-ITEMSET (itemset)
  (selectq (:itemset:type itemset)
    (initial
      (:EXPAND itemset) )
    (dirty
      (let ( (refs (mapcar 'cdadr (:itemset:trans itemset))) )
        (:EXPAND itemset)
        (while refs
          (:DECR-REFCOUNT (nextl refs)) ) )) ))

(defun :EXPAND (itemset)
  (let ( (grammar (:itemset:grammar itemset))
         s for-acts acts reds new-kernel new-itemset )
    (mapc
      (lambda (item)
        (setq s (:NEXTSYMBOL item))
        (if s
          (putassoc for-acts s (:MOVE-DOT item))
          (if (eq (:rule:result (car item)) (:grammar:start-symbol grammar))
            (newl acts '(EOF (accept)) )
            (newl reds (cons 'reduce (car item))) ) ) )
      (:K-CLOSURE (:itemset:kernel itemset) grammar) )
    (mapc
      (lambda ( (symbol . items) )
         (setq new-kernel
           (sort
             (lambda ( (r1 . d1) (r2 . d2) )
               (if (= d1 d2)
                 (> (:rule:id r1) (:rule:id r2))
                 (> d1 d2) ) )
             items ) )
         (setq new-itemset (:GET-ITEMSET-WITH-KERNEL new-kernel grammar))
         (newl acts (list symbol (cons 'shift new-itemset))) )
    for-acts )
    (:itemset:trans itemset acts)
    (:itemset:reductions itemset reds)
    (:itemset:type itemset 'complete) ))

(defun :K-CLOSURE (items grammar)
  (let ( (i 0) (l (length items))
         (closure (mapcar 'identity items))
         (rules (:grammar:rules grammar))
```

```
        done s item )
    (while (< i l)
      (setq s (:NEXTSYMBOL (nth i closure)))
      (when (and s (not (memq s done)) )
        (newl done s)
        (mapc
          (lambda (rule)
            (setq item (cons rule 0))
            (when (not (member item items))
              (newr closure item)
              (incr l) ))
          (cassq s rules) ) )
      (incr i) )
    closure ))

(defun :DECR-REFCOUNT (itemset)
  (when itemset
    (let ( (grammar (:itemset:grammar itemset)) )
      (when (= 0 (:itemset:refcount itemset (1- (:itemset:refcount itemset))))
        (:HASH-DELITEMSET itemset (:grammar:itemsets grammar))
        (when (neq (:itemset:type itemset) 'initial)
          (mapc
            (lambda (ref)
              (:DECR-REFCOUNT (cdadr ref)) )
            (:itemset:trans itemset) ) )
   )) ))

(defun :GET-ITEMSET-WITH-KERNEL (k grammar)
  (prog ( (hashpos (:HASH-CODE k))
          (hashtable (:grammar:itemsets grammar))
          samehashcodes itemset new-id )
    (setq samehashcodes (vref hashtable hashpos))
    (while samehashcodes
      (setq itemset (nextl samehashcodes))
      (when (equal k (:itemset:kernel itemset))
        (:itemset:refcount itemset (1+ (:itemset:refcount itemset)))
        (return itemset) ) )
    (setq new-id (1+ (:grammar:highest-is-id grammar)))
    (:grammar:highest-is-id grammar new-id)
    (setq itemset (omakeq :itemset
                          type 'initial
                          id new-id
                          kernel k
                          refcount 1
                          grammar grammar))
    (vset hashtable hashpos (cons itemset (vref hashtable hashpos)))
    (return itemset) ))

(defun :FOR-ALL-ITEMSETS (grammar function)
  (mapvector
    (lambda (itemsets)
      (mapc function itemsets) )
    (:grammar:itemsets grammar) ) )

(defun :FOR-ALL-RULES (grammar function)
```

```
  (mapc
    (lambda ( assocentry )
      (mapc function (cdr assocentry) ) )
    (:grammar:rules grammar) ) )

(defun :MOVE-DOT ( (rule . placedot) )
  (cons rule (1+ placedot)) )

(defun :NEXTSYMBOL ( (rule . placedot) )
  (nth placedot (:rule:elements rule)) )

;
; Hash functions for the itemsets.
;

(defvar :hashtablesize 100)

(defun :HASH-INITG (start-state)
  (let ( (:hashtable (makevector :hashtablesize ())) )
    (vset :hashtable 0 (list start-state) )
    :hashtable ))

(defun :HASH-DELITEMSET (itemset hashtable)
  (let ( (hashpos (:HASH-CODE (:itemset:kernel itemset) )) )
    (vset hashtable hashpos (remove itemset (vref hashtable hashpos))) ))

(defun :HASH-CODE ( ((rule . placedot) . restofkernel) )
  (modulo (mul (:rule:id rule) (1+ placedot)) :hashtablesize) )

;
; putassoc and remassoc
;

(dmd putassoc (l key value)
  `(setq ,l
     ((lambda (l key value)
       (let ( (pair (assq key l)) )
         (ifn pair
           (acons key (list value) l)
           (rplacd pair (cons value (cdr pair)))
         l )))
      ,l ,key ,value)) )

(dmd remassoc (l key value)
  `(setq ,l
     ((lambda (l key value)
       (let ( (pair (assq key l)) )
          (rplacd pair (delq value (cdr pair)))
          (when (null (cdr pair))
            (setq l (delq pair l)) )
          l ))
      ,l ,key ,value)) )
```

A.4 Utilities

In order to facilitate using the algorithms presented in the previous appendices, we present some utilities to:

- Read a grammar from file and generate a *grammar*-object for it.
- Recognize a list of tokens according to some selection of the rules in a *grammar*-object.
- Parse a list of tokens according to some selection of the rules in a *grammar*-object, and print the resulting graph.
- Print the contents of a *grammar*-object.

The top-level functions

The package "PGtool" contains a function to create a *grammar*-object and fill it with grammar rules from a file:

(#:PGtool:GEN *filename*) → *grammar*

It is called with the name of a file containing grammar rules. It returns a *grammar*-object to which these rules have been added. The format of the grammar rules in the file must be:

(*label result* ::= $element_1$ $element_2$ ⋯)

Routine #:PGtool:GEN initializes the *grammar*-object it generates with "S" as *start-symbol*, which means that at least one of the rules should have this symbol as result.

The recognizer and the parser can be called with

(#:PGtool:RECOGNIZE *grammar selection tokens*), and
(#:PGtool:PARSE *grammar selection tokens*)

of which the arguments are like the ones of #:RPG:RESTRICT-PARSER and the underlying PARSE functions. Routine #:PGtool:RECOGNIZE returns "`t`" or "`()`". If parsing succeeds, #:PGtool:PARSE prints a linear representation of the parse graph.

```
(defvar #:sys-package:colon 'PGtool)

(defun :GEN (filename)
  (let ( grammar start-seen rules )
    (ifn (probefile filename)
```

```
      (print "Can't open file " filename)
      (with ( (inchan (openi (catenate filename))) )
        (untilexit eof (newl rules (read))) )
      (setq grammar (#:RPG:INIT-GRAMMAR 'S))
      (mapc
        (lambda ( (label result arrow . elements) )
          (when (eq result 'S)
            (setq start-seen t) )
          (ifn (eq arrow '#:user:=)
            (print "A rule should be of the form (label NT ::= el1 el2 ...)")
            (#:RPG:ADD-RULE grammar label result elements) ) )
        rules )
      (unless start-seen
        (print "No rule seen with start-symbol S as result") )
      grammar
    ) ))

(defun :RECOGNIZE (grammar selected-labels tokens)
  (#:RPG:RESTRICT-PARSER grammar selected-labels)
  (#:recognizer:PARSE grammar tokens) )

(defun :PARSE (grammar selected-labels tokens)
  (#:RPG:RESTRICT-PARSER grammar selected-labels)
  (#:parser:SHOW-TREE (#:parser:PARSE grammar tokens)) )
```

Printing a parse graph

Routine *SHOW-TREE* visits all nodes in a parse graph, assigns a unique number to them, and prints the nodes in a linear fashion. To print the rules and the parse table contained in a *grammar*-object, *SHOW-GRAMMAR* can be used.

These routines depend heavily on the *send* mechanism of LeLisp, and on the fact that the general *print* routine uses the specialized *prin*-routines of objects whenever possible.

```
(defun #:parser:SHOW-TREE (rootnode)
  (let ( to-show seen to-process node (id-counter 0) )
    (ifn rootnode
      (print "parsing failed")
      (send 'id rootnode (incr id-counter))
      (newl seen rootnode)
      (newl to-process rootnode)
      (while to-process
        (setq node (nextl to-process))
        (when (eq (type-of node) 'symbolnode)
          (mapc
            (lambda (rulenode)
              (unless (memq rulenode seen)
                (send 'id rulenode (incr id-counter))
                (newl seen rulenode) )
```

```
            (mapc
              (lambda (el)
                (when el
                (unless (memq el seen)
                  (send 'id el (incr id-counter))
                  (newl seen el)
                  (newl to-process el)) ) )
              (#:rulenode:elements rulenode) ) )
          (#:symbolnode:possibilities node) )) )
    (print "result: " rootnode)
    (mapc
      (lambda (node) (send 'show node))
      (sort
        (lambda (n1 n2) (<= (send 'id n1) (send 'id n2)))
        seen )) ) ))

(defun #:termnode:show (n)
  (print n ": " (#:termnode:token n) " " (string (#:termnode:string n)) ))
(defun #:termnode:prin (n) (prin "T" (#:termnode:id n)))

(defun #:symbolnode:show (n)
  (print n ": " (#:symbolnode:symbol n) " " (#:symbolnode:possibilities n)) )
(defun #:symbolnode:prin (n) (prin "S" (#:symbolnode:id n)))

(defun #:rulenode:show (n)
  (print n ": " (#:rulenode:rule n) " " (#:rulenode:elements n)) )
(defun #:rulenode:prin (n) (prin "R" (#:rulenode:id n)))

(defun #:RPG:rule:prin (rule)
  (prin "[" (#:RPG:rule:result rule) " ::=")
  (mapc
    (lambda (elem) (prin " " elem) )
    (#:RPG:rule:elements rule) )
  (prin "]") )


(defun #:RPG:SHOW-GRAMMAR (grammar)
  (#:RPG:FOR-ALL-RULES grammar 'print)
  (print) (print "----------------------") (print)
  (#:RPG:FOR-ALL-ITEMSETS grammar '#:RPG:itemset:show) )

(defun #:RPG:itemset:show (is)
  (prin is ": ")
  (print (#:RPG:itemset:type is))
  (selectq (#:RPG:itemset:type is)
    (complete
      (print "transitions: " (#:RPG:itemset:trans is))
      (print "reductions: " (#:RPG:itemset:reductions is)) )
    (specialized
      (print "transitions: " (#:RPG:itemset:sp-trans is))
      (print "reductions: " (#:RPG:itemset:sp-reds is)) ) )
  (print) )

(defun #:RPG:itemset:prin (i) (prin "<is-" (#:RPG:itemset:id i) ">"))
```

An example

We take a file containing the following rules:

(1 S ::= a S)
(2 S ::= S a)
(3 S ::=)

If we parse the sentence "a" according to this grammar, the (ambiguous) result would be:

```
? (#:PGtool:PARSE (#:PGtool:GEN "ASA.grammar") 'all '(a))
result: S1
S1: S (R2 R5)
R2: [S ::= S a] (S3 T4)
S3: S (R8)
T4: a a
R5: [S ::= a S] (T4 S6)
S6: S (R7)
R7: [S ::=] ()
R8: [S ::=] ()
```

The nodes in this graph are identified by a number prefixed by "S", "T" or "R", which stands respectively for *symbol node*, *term node* or *rule node*. Fig. A.1 depicts the same graph. If we parse "a" with as selection "(1 3)", the result is non-ambiguous, and stands for the right branch of Fig. A.1.

```
? (#:PGtool:PARSE (#:PGtool:GEN "ASA.grammar") '(1 3) '(a))
result: S1
S1: S (R2)
R2: [S ::= a S] (T3 S4)
T3: a a
S4: S (R5)
R5: [S ::=] ()
```

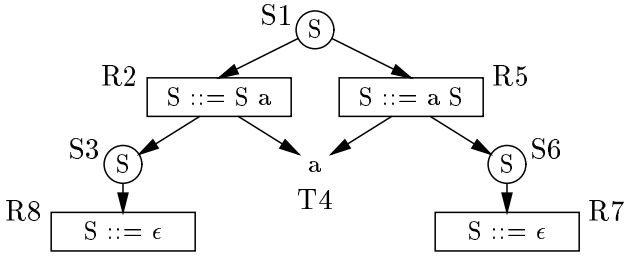


Figure A.1: The parse graph of "a"

Appendix B

An SDF definition

We list the SDF definition developed in [Chapter 5 of this thesis] once more in its entirety.

Module Tokens

```
exports
    sorts
        Id Digits Num
    lexical syntax
        [ \t\n]                              -> LAYOUT
        "{" ~[{}]* "}"                       -> LAYOUT

        [a-zA-Z] [a-zA-Z0-9]*                -> Id

        [0-9]+                               -> Digits

        Digits                               -> Num
        Digits "." Digits                    -> Num
        Digits "." Digits "E" Digits         -> Num
        Digits "." Digits "E" [+\-] Digits   -> Num
        Digits "E" Digits                    -> Num
        Digits "E" [+\-] Digits              -> Num
```

Module Expressions

```
imports
    Tokens

exports
    sorts
        Expression
    context-free syntax
        Expression "=" Expression              -> Expression {non-assoc}
        Expression "<>" Expression             -> Expression {non-assoc}
        Expression "<" Expression              -> Expression {non-assoc}
```

```
        Expression "<=" Expression              -> Expression {non-assoc}
        Expression ">=" Expression              -> Expression {non-assoc}
        Expression ">" Expression               -> Expression {non-assoc}

        "+" Expression                          -> Expression
        "-" Expression                          -> Expression
        Expression "+" Expression               -> Expression {left}
        Expression "-" Expression               -> Expression {left}
        Expression or Expression                -> Expression {left}

        Expression "*" Expression               -> Expression {left}
        Expression "/" Expression               -> Expression {left}
        Expression div Expression               -> Expression {left}
        Expression mod Expression               -> Expression {left}
        Expression and Expression               -> Expression {left}
        not Expression                          -> Expression

        Id                                      -> Expression
        Id "(" {Expression ","}+ ")"            -> Expression
        "(" Expression ")"                      -> Expression {bracket}
        Num                                     -> Expression

priorities
    {non-assoc: "=", "<>", "<", "<=", ">=", ">"} <
    { "-" Expression -> Expression,
      "+" Expression -> Expression,
      Expression "+" Expression -> Expression,
      Expression "-" Expression -> Expression,
      or } <
    {left: "*", "/", div, mod, and}
priorities
    {left: Expression "+" Expression -> Expression,
           Expression "-" Expression -> Expression,
           or }
```

Module Statements

```
imports
    Tokens Expressions

exports
    sorts
        Compound-statement Statement Variable
    context-free syntax
        begin {Statement ";"}* end              -> Compound-statement

        Compound-statement                      -> Statement
        Variable ":=" Expression                -> Statement
        if Expression then Statement
                     else Statement             -> Statement
        while Expression do Statement           -> Statement
        Id                                      -> Statement
        Id "(" {Expression ","}+ ")"            -> Statement

        Id                                      -> Variable
```

```
        Id "[" Expression "]"                   -> Variable
```

Module Declarations

```
imports
    Tokens

exports
    sorts
        Declaration Type Standard-type
    context-free syntax
        var {Id ","}+ ":" Type ";"                      -> Declaration

        Standard-type                                   -> Type
        array "[" Num ".." Num "]" of Standard-type     -> Type

        integer                                         -> Standard-type
        real                                            -> Standard-type
```

Module Subprograms

```
imports
    Declarations Statements

exports
    sorts
        Subprogram-declaration Arguments Parameter
    context-free syntax
        function Id Arguments ":" Standard-type ";"
          Declaration*
          Compound-statement ";"                    -> Subprogram-declaration
        procedure Id Arguments ";"
          Declaration*
          Compound-statement ";"                    -> Subprogram-declaration

        "(" {Parameter ";"}+ ")"                    -> Arguments
                                                    -> Arguments
        {Id ","}+ ":" Type                          -> Parameter
```

Module Program

```
imports
    Declarations Subprograms Statements

exports
    sorts
        Program
    context-free syntax
        program Id "(" {Id ","}+ ")" ";"
          Declaration*
          Subprogram-declaration*
          Compound-statement "."                    -> Program
```

Bibliography

[AD83] R. Agrawal and K.D. Detro. An efficient incremental LR parser for grammars with epsilon productions. *Acta Informatica*, 19:369–376, 1983.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers. Principles, Techniques and Tools.* Addison-Wesley, 1986.

[BCD+88] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. In *Third Annual Symposium on Software Development Environments (SIGSOFT'88)*, Boston, 1988.

[BHK89] J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification.* ACM Press Frontier Series. The ACM Press in cooperation with Addison-Wesley, 1989.

[BL89] 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.

[Bur90a] B. Burshteyn. Generation and recognition of formal languages by modifiable grammars. *SIGPLAN Notices*, 25(12):45–53, 1990.

[Bur90b] B. Burshteyn. On the modification of the formal grammar at parse time. *SIGPLAN Notices*, 25(5):117–123, 1990.

[Cel78] A. Celentano. Incremental LR parsers. *Acta Informatica*, 10:307–321, 1978.

[Chr90] H. Christiansen. A survey of adaptable grammars. *SIGPLAN Notices*, 25(11):35–44, 1990.

[CIL89] D. Clément, J. Incerpi, and B. Lang. *The virtual tree processor.* INRIA, Sophia-Antipolis, 1989.

[Cor89] G.V. Cormack. An LR substring parser for noncorrecting syntax error recovery. *SIGPLAN Notices*, 24(7):179–191, 1989.

[Deu91] A. van Deursen. An algebraic specification for the static semantics of Pascal. Report CS-R9129, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1991. Extended abstract in: *Conference Proceedings of Computing Science in the Netherlands CSN'91*, pages 150-164.

[Dyb87] K.R. Dybvig. *The Scheme Programming Language.* Prentice Hall, 1987.

[Ear70] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[FGJM85] K. Futatsugi, J.A. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In B. Reid, editor, *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 52–66. ACM, 1985.

[Flo64] R.W. Floyd. Bounded context syntactic analysis. *Communications of the ACM*, 7(2):62–67, 1964.

[GJ90] D. Grune and C.J.H. Jacobs. *Parsing Techniques – A Practical Guide.* Ellis Horwood, 1990.

[GKP89] R.L. Graham, D.E. Knuth, and O. Patashnik. *Concrete Mathematics.* Addison-Wesley, 1989.

[GM79] C. Ghezzi and D. Mandrioli. Incremental parsing. *ACM Transactions on Programming Languages and Systems*, 1(1):58–70, 1979.

[GM80] C. Ghezzi and D. Mandrioli. Augmenting parsers to support incrementality. *Journal of the ACM*, 27(3):564–579, 1980.

[Gro91] J.F. Groote. *Proces Algebra and Structured Operational Semantics.* PhD thesis, University of Amsterdam, 1991.

[Har69] F. Harary. *Graph theory.* Addison-Wesley, 1969.

[Har78] M.A. Harrison. *Introduction to Formal Language Theory.* Addison-Wesley, 1978.

[Hen91] P.R.H. Hendriks. *Implementation of Modular Algebraic Specifications.* PhD thesis, University of Amsterdam, 1991.

[HHKR89] 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.

[HKR87] J. Heering, P. Klint, and J. Rekers. Incremental generation of lexical scanners. Report CS-R8761, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1987. To appear in *ACM Transactions on Programming Languages and Systems.*

[HKR89] J. Heering, P. Klint, and J. Rekers. Incremental generation of parsers. *SIGPLAN Notices*, 24(7):179–191, 1989.

[HKR90] J. Heering, P. Klint, and J. Rekers. Incremental generation of parsers. *IEEE Transactions on Software Engineering*, 16(12):1344–1351, 1990. Also in: *SIGPLAN Notices*, 24(7):179-191, 1989.

[HKR91] J. Heering, P. Klint, and J. Rekers. Principles of lazy and incremental program generation. Report CS-R9124, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1991. To appear in *ACM Transactions on Programming Languages and Systems.*

[Hor89] R.N. Horspool. ILALR: an incremental generator of LALR(1) parsers. In D. Hammer, editor, *Compiler Compilers and High Speed Compilation*, number 371 in Lecture Notes in Computer Science, pages 128–136, 1989.

[Hor90] R.N. Horspool. Incremental generation of LR parsers. *Computer Languages*, 15(4):205–233, 1990.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[JJWW90a] P. de Jager, W. Jonker, A.T. Wammes, and J.J. Wester. An interactive programming environment for LOTOS. In *Third International Conference on Formal Description Techniques (FORTE'90)*, Madrid, 1990.

[JJWW90b] P. de Jager, W. Jonker, A.T. Wammes, and J.J. Wester. On the generation of interactive programming environments – a LOTOS case study. In *Third international workshop on Software Engineering and its applications*, Toulouse, 1990.

[Joh86] S.C. Johnson. *YACC: yet another compiler-compiler*. Bell Laboratories, 1986. UNIX Programmer's Supplementary Documents, Volume 1 (PS1).

[KA88] F.E.J. Kruseman Aretz. On a recursive ascent parser. *Information Processing Letters*, 29:201–206, 1988.

[Kli91a] P. Klint. Lazy scanner generation for modular regular grammars. Report CS-R9158, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1991.

[Kli91b] P. Klint. A meta-environment for generating programming environments. In J.A. Bergstra and L.M.G. Feijs, editors, *Proceedings of the METEOR workshop on Methods Based on Formal Specification*, volume 490 of *Lecture Notes in Computer Science*, pages 105–124. Springer-Verlag, 1991.

[Koo] J.W.C. Koorn. GSE: A generic text and structure editor. Programming Research Group, University of Amsterdam, to appear.

[Kos90] K. Koskimies. Lazy recursive descent parsing for modular language implementation. *Software-Practice and Experience*, 20(8):749–772, 1990.

[KP90] K. Koskimies and J. Paakki. *Automating Language Implementation - a pragmatic approach*. Ellis Horwood books in information technology. Ellis Horwood, 1990.

[Lan74] B. Lang. Deterministic techniques for efficient nondeterministic 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.

[Lan88] B. Lang. Parsing incomplete sentences. In *Proceedings of the Twelfth International Conference on Computational Linguistics*, pages 365–371, Budapest, 1988. Association for Computational Linguistics.

[Lan91] 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.

[Lee91] R. Leermakers. Non-deterministic recursive ascent parsing. In *Fifth Conference of the European Chapter of the Association for Computational Linguistics*, pages 63–68, Berlin, 1991. Association for Computational Linguistics.

[LeL87] INRIA, Rocquencourt. *LeLisp, Version 15.21, le manuel de référence*, 1987.

[Mad91] William Maddox. Personal communication, June 1991.

[Meu88] E.A. van der Meulen. Algebraic specification of a compiler for a language with pointers. Report CS-R8848, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1988.

[NF89] 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.

[Nij91] 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.

[NT90] 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.

[Pet90] M. Petterson. DML a meta-language for compiler generation from denotational specifications. Technical Report LiTH-IDA-R-90-43, Linköping University, Department of Computer and Information Science, Linköping University, S-581 83 Linköping, Sweden, 1990.

[Rek89] J. Rekers. Modular parser generation. Report CS-R8933, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1989.

[Ric85] H. Richter. Noncorrecting syntax error recovery. *ACM Transactions on Programming Languages and Systems*, 7(3):478–489, 1985.

[RK91a] J. Rekers and W. Koorn. Substring parsing for arbitrary context-free grammars. In *Proceedings of the Second International Workshop on Parsing Technologies, IWPT'91*, pages 218–224. Association for Computational Linguistics, 1991. Also in: *SIGPLAN Notices*, 26(5):59-66,1991.

[RK91b] J. Rekers and W. Koorn. Substring parsing for arbitrary context-free grammars. *SIGPLAN Notices*, 26(5):59–66, 1991.

[San82] D. Sandberg. LITHE: A language combining a flexible syntax and classes. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 142–145. ACM, 1982.

[Sik90] 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.

[Sik91] 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.

[Sne90] G. Snelting. How to build LR parsers which accept incomplete input. *SIGPLAN Notices*, 25(4):51–58, 1990.

[TN89] 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.

[Tom85] M. Tomita. *Efficient Parsing for Natural Languages.* Kluwer Academic Publishers, 1985.

[Voi86] F. Voisin. CIGALE: a tool for interactive grammar construction and expression parsing. *Science of Computer Programming*, 7:61–86, 1986.

[Wal91] H.R. Walters. *On Equal Terms, Implementing Algebraic Specifications.* PhD thesis, University of Amsterdam, 1991.

[Yeh83] D. Yeh. On incremental shift-reduce parsing. *BIT*, 23(1):36–48, 1983.

Samenvatting in het Nederlands

Wat zijn parsers en parser generatoren?

Een taal wordt bepaald door een verzameling correcte zinnen. Een zin bestaat uit een rijtje woorden en heeft een interne structuur. Een *parser* is in staat om te beslissen of een zin tot een taal behoort en om uit te vinden wat de interne structuur van die zin is.

Ik neem als voorbeeld de taal *Expressies*, waar onder andere de volgende zinnen toe behoren:

a × b		a b
a	maar niet:	a × × b
a × b + b		×
a × b × a		a ×
a × − b		a | b
a × a × a × a × a		

De vraag is hoe je kunt uitdrukken welke zinnen wèl tot deze taal behoren en welke niet. Alle goede (of alle foute) zinnen opsommen gaat niet omdat dat er oneindig veel kunnen zijn. Je kunt deze taal wel beschrijven met een *grammatica*:

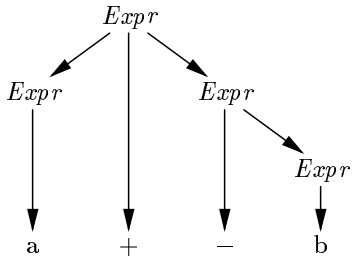
START ::= *Expr*	(*Expressies*)
Expr ::= *Expr* × *Expr*	
Expr ::= *Expr* + *Expr*	
Expr ::= − *Expr*	
Expr ::= a	
Expr ::= b	

Deze grammatica beschrijft welke zinnen tot de taal behoren via afleidingen: hierbij moet elke stap via een regel uit de grammatica lopen. Een voorbeeld van een afleiding is:

START	leidt via de regel "START ::= *Expr*" tot
Expr	leidt via "*Expr* ::= *Expr* + *Expr*" tot
Expr + *Expr*	leidt via "*Expr* ::= a" tot
a + *Expr*	leidt via "*Expr* ::= − *Expr*" tot
a + − *Expr*	leidt via "*Expr* ::= b" tot
a + − b	

Omdat er een afleiding bestaat van "START" naar "a + − b", waarbij elke stap in de afleiding via een regel uit de grammatica loopt, behoort de zin "a + − b" tot de taal. (Merk op dat er volgens deze grammatica twee verschillende afleidingen zijn die beide leiden tot de zin "a × b + c". De grammatica is dan ook ambigue.)

Een *parser* is in staat om zo'n afleiding in tegengestelde richting uit te voeren: een parser leidt voor zijn invoerzin af welke grammaticaregels in een afleiding gebruikt moeten worden om van START naar die zin te gaan en representeert dit in een boomstructuur. Een parser zal de volgende boom opleveren voor de zin "a + − b":



De parse boom van "a + − b" volgens *Expressies*

Een parser is altijd maar voor één grammatica geschikt en moet voor elke grammatica opnieuw geschreven worden. Het is daarom voordelig om parsers automatisch te kunnen *genereren* voor een gegeven grammatica. Parsers en parser generatoren vormen het centrale thema van dit proefschrift.

Parsers genereren voor interactieve omgevingen

Mijn onderzoek maakt deel uit van het Esprit project GIPE, dat het "*automatisch genereren van interactieve programmeeromgevingen uit formele*

specificaties van programmeertalen" als doel heeft. In een interactieve programmeeromgeving kun je programma's intypen, wijzigen en testen in een omgeving die speciaal voor die taal geschikt is. De teksteditor bijvoorbeeld, controleert steeds of het programma syntactisch correct is en kan allerlei andere hulp bieden.

In dit project willen wij een dergelijke omgeving uit de beschrijving van een taal genereren en hebben daarvoor, onder andere, een formalisme ontworpen om syntax te definiëren. Dit syntax definitie formalisme (SDF) laat toe om de lexicale syntax, de grammatica en de vorm van de bomen in één definitie op te schrijven en laat alle context-vrije grammatica's toe. Uit een SDF definitie kan automatisch een syntax gestuurde editor worden afgeleid. Het is grotendeels mijn taak geweest om SDF te implementeren en mijn onderzoek heeft zich dan ook toegespitst op het oplossen van de vragen die door dit formalisme worden opgeworpen.

Het parse algoritme

De eerste stap was het kiezen van het algoritme voor de parser. Deze parser moet alle grammatica's aan kunnen (wat hem aardig ingewikkeld maakt) maar ook zeer efficient zijn. Ik heb hiervoor het Tomita algoritme gekozen. Dit is een goed compromis tussen beide eisen omdat het zijn efficientie dynamisch aanpast. Op "makkelijke" invoer is het behoorlijk snel, op lastiger stukken wat langzamer, maar hij komt er wel uit. Het werk aan dit algoritme wordt beschreven in hoofdstuk 1 – *Generalized LR parsing*. Het is voornamelijk werk van Tomita met enkele kleine uitbreidingen. Mijn belangrijkste bijdrage is hier dat ik dit algoritme uit de wereld van de *natuurlijke* taalverwerking naar die van de programmeertalen gehaald heb.

De parser generator

De volgende stap is de parser *generator*. Het parse algoritme heeft namelijk een parse tabel nodig waarin alle informatie over de grammatica is opgeslagen. Het construeren van zulke tabellen is op zich niet zo lastig want het Tomita algoritme werkt al heel goed met de zeer eenvoudige LR(0) parse tabellen. Alleen bleek al snel dat we meer wilden, we wilden niet alleen interactieve programmeeromgevingen genereren maar ook grammatica's interactief definiëren en testen.

Hiervoor heb je een *incrementele* generator nodig. Zo'n generator brengt bij een kleine wijziging van de grammatica ook maar een kleine wijziging aan in de al gegenereerde parser. Dit in tegenstelling tot conventionele methoden, die na elke wijziging de oude parser weggooien en van vooraf aan beginnen. Jan Heering, Paul Klint en ik hebben een aantal verschillende alternatieven voor incrementele generatie onderzocht en uiteindelijk hebben we een incrementele LR(0) parse tabel generator bedacht.

Deze incrementele generator is beschreven in hoofdstuk 2 – *Incremental parser generation* – en dit is eigenlijk het meest innovatieve gedeelte van mijn proefschrift. Ik heb dit werk op de *SIGPLAN* conferentie in Portland gepresenteerd en het is geplaatst in het tijdschrift *IEEE – Transactions on Software Engeneering*.

Modulaire grammatica's

In een formele definitie van een programmeertaal hoort naast syntax ook *semantiek* thuis. In het semantische gedeelte van een definitie beschrijft men de *betekenis* van de taal. Dat kan zijn of een programma correct is, hoe programma's in die taal uitgevoerd moeten worden of wat een compiler moet doen. Om semantiek te definiëren hebben we, los van SDF, het algebraïsch specificatie formalisme ASF ontwikkeld. ASF valt buiten het onderwerp van mijn proefschrift en ik wil er niet meer over zeggen dan dat een ASF definitie uit modules bestaat. ASF en SDF zijn later gekoppeld tot één formalisme (met de prozaïsche naam ASF+SDF), waarin alle eigenschappen van een programmeertaal gespecificeerd kunnen worden.

Het feit dat ASF modularisering ondersteunt, heeft bij het combineren van beide formalismen gevolgen voor SDF. Een modulaire definitie bestaat uit modules die elkaar kunnen *importeren*. Een SDF module beschrijft een grammatica die bestaat uit een eigen definitie plus die van alle modules die hij importeert.

Een recht toe, recht aan implementatie hiervan zou zijn om een aparte parser voor de volledige grammatica van elke module te genereren. Hiermee zou je echter veel dubbel werk doen, omdat er dan voor een module die vaak geimporteerd wordt ook steeds opnieuw een parser gegenereerd wordt. Daarnaast moet je bij een wijziging in een module de parsers aanpassen van alle modules die hem importeren.

Een andere oplossing zou zijn om alleen voor het eigen gedeelte van elke module een incomplete parser te genereren en deze parsers aan elkaar te

koppelen. Echter, gegeven de in hoofdstuk 1 en 2 gekozen technieken is dit onmogelijk.

Het is daarentegen wel mogelijk om één grote parser voor alle modules tezamen te genereren en hier kleinere parsers uit af te leiden. Wat dan ontstaat is weliswaar geen echte modulaire parser generator maar maakt het wel mogelijk om binnen het ASF+SDF formalisme met modulaire definities te werken. Deze nieuwe techniek is beschreven in hoofdstuk 3 – *Restricting a parser to a subgrammar*.

Delen van zinnen herkennen

Hoofdstuk 4 – *Substring parsing* – is eigenlijk een zijstapje. Hier ontwikkelden Wilco Koorn en ik een *substring* parser. Deze parser herkent of zijn invoer een *deel* van een volledige zin zou kunnen zijn hetgeen een slag complexer is dan gewoon parsen. Deze techniek zou gebruikt kunnen worden om *incrementeel parsen* te ondersteunen. Ook kan een substring parser ervoor zorgen dat een parser niet op de eerste fout in de invoer stokt maar ook verderop nog fouten vindt. De methode is echter (nog) te inefficiënt om toe te passen.

Uit de literatuur zijn andere oplossingen bekend voor het substring parsen. Die werken echter alleen voor een beperkte klasse van grammatica's. Op basis van het parse algoritme uit hoofdstuk 1 hebben wij een algemene en bovendien elegante methode ontwikkeld. Ik heb dit werk gepresenteerd op de *IWPT* conferentie in Mexico en gepubliceerd in het tijdschrift *SIGPLAN Notices*.

SDF

De technieken uit de eerste drie hoofdstukken lossen de fundamentele problemen van de implementatie van SDF op. Om nu naar SDF zelf over te stappen geef ik in hoofdstuk 5 – *From BNF to SDF* – een inleiding in het gebruik van SDF aan de hand van een SDF definitie van een programmeertaal. Hoofdstuk 6 – *An implementation of SDF* – beschrijft het interface van de implementatie van SDF zodanig dat deze code bruikbaar wordt voor andere programmeurs.

Tenslotte

Terugkijkend op de 6 jaar dat ik bij het GIPE project gewerkt heb, durf ik te zeggen dat we erin geslaagd zijn om een groot aantal doelen te verwezelijken. We hebben een aantal nieuwe technieken ontwikkeld en we zijn erin geslaagd om met het ASF+SDF formalisme het definiëren van een programmeertaal eenvoudiger te maken. Het enige minpunt van het ASF+SDF systeem is dat al die geavanceerdheid het wat traag maakt in vergelijking met conventionele systemen. Het aanpakken van dit probleem is echter eerder een klus voor de industrie dan voor de wetenschap.