Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

J. Rekers

A parser generator for finitely ambiguous context-free grammars

Computer Science/Department of Software Technology Report CS-R8712 March

---

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

A Parser Generator for Finitely Ambiguous Context-Free Grammars

Jan Rekers
Department of Software Technology,
Centre for Mathematics and Computer Science,
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

An implementation of the syntax definition formalism SDF is described. SDF combines the specification of lexical syntax, context-free syntax, and abstract syntax in a single formalism. From an SDF definition a lexical scanner and parse tables are generated. These parse tables together with a universal parser form a parser for the defined language. The algorithm used in the universal parser allows finitely ambiguous context-free grammars.

1980 Mathematics Subject Classification: 68F25 **[Linguistics]:** *Parsing;* 68B99 **[Software]**.

1986 CR Categories: D.3.4 **[Programming Languages]:** Processors - *Parsing;* F.4.2 **[Mathematical Logic and Formal Languages]:** Grammars and other Rewriting Systems - *Parsing.*

Keywords & Phrases: parser generator, context-free grammar, ambiguity, Tomita's parsing algorithm, syntax definition formalism, user-defined syntax.

Note: Partial support received from the European Communities under ESPRIT project 348 (Generation of Interactive Programming Environments - GIPE).

1. INTRODUCTION

This paper describes the implementation of a parser generator and a universal parser for finitely ambiguous context-free grammars. The parser generator and the universal parser were implemented as part of an implementation of the syntax definition formalism SDF [SDF, SDFref].

In general, separate formalisms are used to describe the lexical syntax, the context-free syntax, and the abstract syntax of a language. In SDF these three are combined into a single formalism with the following properties:

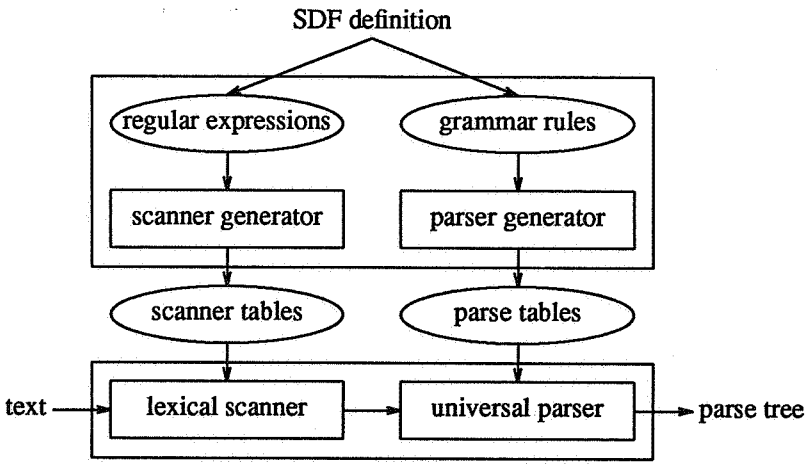
- It allows the simultaneous definition of lexical, context-free and abstract syntax in a uniform way.
- The formalism implicitly defines a translation from parse trees to abstract syntax trees.
- The need for introducing 'unnecessary' non-terminals in the context-free section of the definition is reduced.
  This is achieved by (a) using priority rules to eliminate undesirable parses, (b) using associativity attributes, (c) allowing arbitrary context-free grammars, and (d) using the syntactic iterators + and *.

For an example of the use of SDF the reader is referred to [Mini-ML], where SDF is used in combination with an algebraic specification formalism to define the type-checking of a sublanguage of ML.

The algorithm used for the universal parser was originally developed by Tomita as a parsing algorithm for natural languages [Tomita]. The algorithm allows finitely ambiguous context-free grammars and adapts its time complexity dynamically to the difficulty of the grammar and the input sentence. This results in a parsing algorithm which is as powerful as Earley's general context-free parsing algorithm [Earley], while it promises to be as fast as ordinary LR parsing algorithms on LR grammars.

Report CS-R8712
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

The structure of the SDF implementation is as follows:



The implementation of SDF consists of a preprocessing phase which extracts regular expressions and grammar rules from the SDF definition and uses them to generate scanner and parse tables. The scanner generator and the parser generator are thus in fact generators of scanner tables and parse tables. In the second phase these tables control the lexical scanner and the universal parser during the transformation of an input text to a parse tree.

This paper will not cover all parts of the SDF implementation in detail. Only SDF itself, the parsing algorithm, and a part of the parser generator are explained in depth. These explanations are followed by measurements of the efficiency of the implementation and by some concluding remarks.

2. INTRODUCTION TO THE SYNTAX DEFINITION FORMALISM SDF

The SDF definition of a language consists of two parts: lexical syntax and context-free syntax. In both parts the notions 'sort' and 'function' play an important role. They correspond, respectively, to non-terminals and to production rules as used in BNF grammars.

An example of a simple lexical syntax part of an SDF definition is:

```
lexical syntax
    sorts
        ID, DIGIT, CHAR
    layout
        SPACE
    functions
        [a-zA-Z]     -> CHAR
        CHAR+        -> ID
        [0-9]        -> DIGIT
        " "          -> SPACE
```

The sort and layout declarations introduce the sorts. The function declarations specify how strings over these sorts can be constructed. The function declarations may be composed of other lexical sorts, terminals, character classes and list expressions involving the iterators + and *. Strings belonging to the layout sorts that appear between elements of the context-free syntax, will be skipped by the lexical analyzer generated from the SDF definition.

The lexical sorts can be used in the context-free syntax part as terminals of the grammar, but terminals can also be introduced directly by the context-free part itself. An example of a simple context-free syntax part is:

```
context-free syntax
    sorts
```

```
    EXP, STAT, SERIES
functions
    ID                                          -> EXP
    EXP "-" EXP                                 -> EXP
    EXP "*" EXP                                 -> EXP
    if EXP then SERIES else SERIES endif        -> STAT
    while EXP do SERIES endwhile                -> STAT
    {STAT ";"}*                                 -> SERIES
```

In the above example of a context-free syntax part several terminals, like `"-"`, `"*"`, `if` and `then`, are introduced. Terminals in an SDF definition need not be quoted when they do not interfere with sorts or with keywords of SDF and only consist of letters and hyphens.

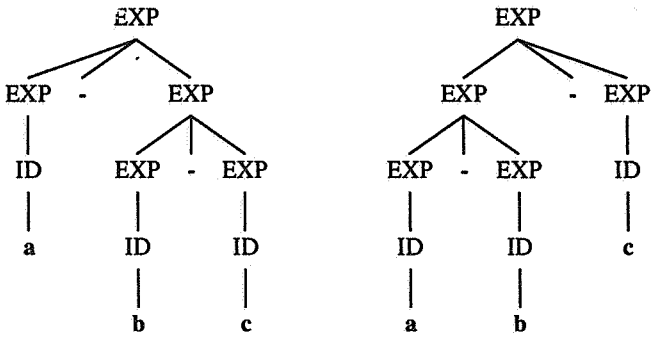
In the function declarations of the context-free syntax terminals, sorts, lexical sorts and list expressions can be used. The following list expressions are allowed:

S*	Zero or more occurrences of sort S
S+	One or more occurrences of sort S
{S t}*	Zero or more occurrences of sort S, separated by the terminal t
{S t}+	One or more occurrences of sort S, separated by the terminal t

Only the first two kinds of list expressions may be used in the lexical syntax part.

The examples of a lexical syntax part and a context-free syntax part given above together form an SDF definition of a simple language with statements and expressions. The definition of the sort `EXP` can give an ambiguous interpretation of sentences with `-` and `*`. When the user wants to give `-` and `*` the normal interpretation, two things must be changed in the context-free part of the SDF definition:

- The priority between the function declarations `EXP "-" EXP -> EXP` and `EXP "*" EXP -> EXP` is not defined. With the addition of the declaration '`priority "*" > "-"`' the normal priority rules will be followed.
- The associativity of `-` and `*` is not defined. For example, the sentence `a - b - c` can be parsed in two ways:



With the attribute `left-assoc` added to the function declaration `EXP "-" EXP -> EXP`, the second parse, with left-associative matching of the `-`, will be chosen by the generated parser.
For the normal associativity of `*`, the attribute `assoc` can be added to `EXP "*" EXP -> EXP`. This states that left-associative and right-associative matching of the `*` are considered equivalent and thus non-ambiguous.

We now give a more extensive example, in which almost all features of SDF are used.

```
module Example
begin
    lexical syntax
        sorts
```

```
        digit, integer, real, letter, id, comment-char
    layout
        white-space, comment
    functions
        [0-9]                                   -> digit
        digit+                                  -> integer
        digit* "." digit+                       -> real
        [a-zA-Z]                                -> letter
        letter+                                 -> id
        [ \t\n\r]                               -> white-space
        ~[{}]                                   -> comment-char
        "{" comment-char* "}"                   -> comment

context-free syntax
    sorts
        program, statement, exp
    priorities
        "-" exp > "*" > (exp "-" exp , "+"),
        if then else > if then
    functions
        {statement ";"}+                        -> program
        id ":=" exp                             -> statement
        print exp*                              -> statement
        if exp then statement                   -> statement
        if exp then statement
               else statement                   -> statement
        "begin" {statement ";"}+ "end"          -> statement
        id                                      -> exp
        integer                                 -> exp
        real                                    -> exp
        "-" exp                                 -> exp  {par}
        exp "-" exp                             -> exp  {left-assoc, par}
        exp "+" exp                             -> exp  {assoc, par}
        exp "*" exp                             -> exp  {assoc, par}
end Example
```

The definitions in the lexical part will be sufficiently clear, except perhaps the function `~[{}] -> comment-char`, which defines `comment-char` as all characters except '{' and '}'.

The priority declarations in the context-free syntax part define the relative priority of functions. Priority relations may not be circular. The full left-part of a function must be supplied, but the keyword skeleton can be used as an abbreviation if unambiguous. For example, `"*"` can be used in the priority declarations, instead of `exp "*" exp`, because this skeleton can only indicate the function `exp "*" exp -> exp`.

The attribute `par` can be added to a function declaration to indicate that the function may be surrounded by parentheses in order to change its priority.

3. THE PARSING ALGORITHM

How can all these features of SDF be implemented? In this section the parsing algorithm will be explained. Section 4 explains how a definition for a lexical scanner generator and a definition for a parser generator are deduced from an SDF definition. Section 5 deals with the implementation of priorities.

3.1. Requirements for the parsing algorithm

The implementation of SDF has been developed as a part of the GIPE project. The aim of this project is to develop a system to generate interactive programming environments for arbitrary programming languages. A parser generated from an SDF definition will form a part of the generated environment.

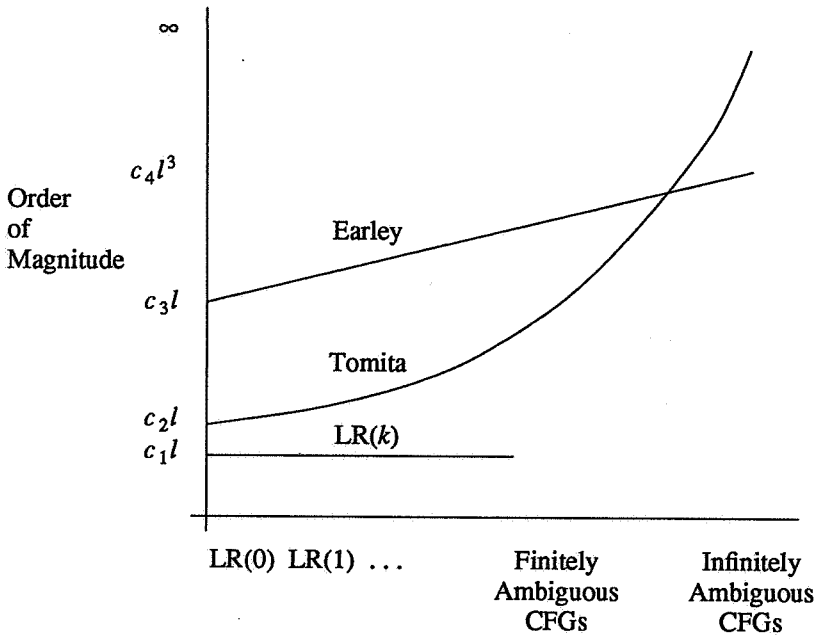
This future use imposes the following requirements on the parser generator and the parsers generated by it:

- No restriction on the class of grammars:
  A user should not be restricted to a certain class of grammars during the design of an SDF definition of a language. This has the advantage that the definition need only contain sorts and functions which are needed to describe the language, but no 'unnecessary' sorts and functions which are only needed to let the definition fit the class of acceptable grammars.
  The intention is to put no restrictions on the class of accepted grammars, except that they must be context-free.
- Ambiguous parses:
  When an input sentence is ambiguous with respect to the given grammar, all its different parses must be produced by the parser. This is the only way to notify the environment in which the parser operates that a grammar is ambiguous, without introducing restrictions. It is up to the environment to decide whether the ambiguities are acceptable, to choose one of the parses, or even to disambiguate the grammar.
- Fast parser generation:
  The parser generation algorithm must be fast enough to watch the effect of changes in a grammar on the generated parser interactively.
- Separate parser generation:
  In the future the user will be able to distribute the SDF definition of a language over several modules. It should be possible to generate a parser for each module separately and to combine them into a parser for the total definition. Then, when the user changes one module, a new parser can be generated for it which can be joined with the parsers of the other modules without regenerating the latter.
- Efficient parsers:
  Because most programming languages are LR(1) or almost LR(1), the generated parser should be as efficient as a normal LR(1) parser for LR(1) grammars.

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

- 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, ambiguous grammars are still impossible and parser generation (table construction) time increases exponentially with k.
- Earley's universal context-free parsing algorithm:
  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:
  This algorithm can be placed between LR(k) algorithms and Earley's algorithm. The class of accepted grammars is restricted to finitely ambiguous grammars and the time complexity of the algorithm depends on the complexity of the grammar and on the particular sentence being parsed. Tomita's algorithm uses an LR(0) or LR(1) parse table constructor as a parser generator.

The following figure gives a rough sketch of the performance of Tomita's algorithm in relation to LR(k) parsers and to Earley's algorithm. It only compares *parse* times and does not compare *parser generation* times.



As can be seen from the graph, 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 infinitely ambiguous grammars. Tomita's algorithm can handle more 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. Tomita's 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.

Tomita's algorithm meets nearly all requirements, so this algorithm has been used as a basis for the SDF implementation. It is not yet possible with the normal LR parse table construction algorithm to generate parsers separately. This issue is currently under investigation.

Tomita uses an extended LR(1) algorithm. We shall explain the conventional LR(1) parsing algorithm first, and then Tomita's algorithm. The LR(1) parser generation (or parse table construction) algorithm is not explained here because it is too complex and unnecessary for the understanding of Tomita's parsing algorithm. Interested readers are referred to [Aho&Ullman].

3.2. Conventional LR parsing

An LR parser is a push down automaton controlled by parse tables. It reads an input sentence $a_1 a_2 \cdots a_n \$$ consisting of a list of terminals, followed by the end-marker $\$$. It uses a parse stack of the form $s_0 X_1 s_1 X_2 s_2 \cdots X_m s_m$, where the s_i are states and the X_i are terminals or non-terminals of the grammar. The state s_m on the top of the stack is called the *current state*.

The parse tables consist of an action table and a goto table. These define the next step for the parser to take in the current state s_m upon the current input symbol a_i or $\$$. The entries of the action table can have one of the values: shift, reduce, accept and error, which instruct the parser to do the following:

shift s:
  push the current input symbol on the stack;
  remove it from the head of the input stream;
  push state s on the stack (s becomes the current state).
reduce r:
  with rule[r] = $A ::= \beta$;
  generate a part of the parse tree with root A and the components of β as sons;
  pop 2 * *length*(β) symbols of the stack;
  s = Goto[*TopOfStack*, A];

push nonterminal *A* on the stack;
push state *s* on the stack (*s* becomes the current state).
(So a reduce action does not affect the current input symbol.)
accept:
the input sentence is accepted;
return the top-nonterminal.
error (or empty):
the input sentence is rejected.

The parser starts with state 0 on top of the stack and repeats the actions as defined by its parse tables until the input sentence is accepted or rejected.

Take for example the following rules

rule no.	rule
0	BOOL ::= true
1	BOOL ::= false
2	BOOL ::= BOOL or BOOL
3	BOOL ::= BOOL and BOOL
4	start ::= BOOL

and the corresponding parse tables

State	Action					Goto
	true	false	or	and	$	BOOL
0	s2	s3				1
1			s5	s4	acc	
2	r0	r0	r0	r0	r0	
3	r1	r1	r1	r1	r1	
4	s2	s3				6
5	s2	s3				7
6	r3	r3	r3	r3	r3	
7	r2	r2	r2	s4	r2	

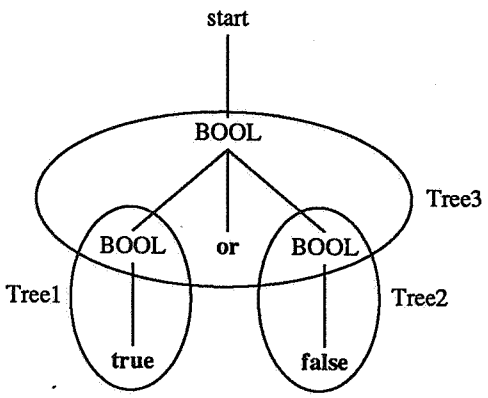
The entries in the Action table may be s*i* (shift), r*i* (reduce), acc (accept) or empty (error). Some examples illustrate this:

- The action table entry *s3*, at Action[4,false], stands for: when the symbol 'false' is the input symbol and the current state is 4, the action to take is *shift 3*. This means: push 'false' on the stack, remove 'false' from the input and push state 3 on the stack. This action makes the current state 3.
- The action table entry *r2* stands for *reduce 2* and rule[2] = 'BOOL ::= BOOL or BOOL'. When the parser encounters the action *reduce 2* the top part of the stack will be: $\cdots s_{m-3}$ BOOL s_{m-2} or s_{m-1} BOOL s_m. The six top symbols: 'BOOL s_{m-2} or s_{m-1} BOOL s_m' are now popped from the stack and they are replaced by the symbol 'BOOL'. The Goto table entry: Goto[s_{m-3} , BOOL] is used to determine the new state that must be pushed on the stack.
- The action table entry *acc* stands for 'accept the input sentence'
- An empty action table entry indicates that the input symbol was not expected in the current state. This is an error and the parser rejects the input sentence.

The sentence 'true or false' is parsed as follows:

Parse stack Action	Input stream	Parse Tree
0	true or false $	
Action[0,true] = s2		
0 true 2	or false $	
Action[2,or] = r0		Tree1
0 BOOL 1	or false $	
Action[1,or] = s5		
0 BOOL 1 or 5	false $	
Action[5,false] = s3		
0 BOOL 1 or 5 false 3	$	
Action[3,$] = r1		Tree2
0 BOOL 1 or 5 BOOL 7	$	
Action[7,$] = r2		Tree3
0 BOOL 1	$	
Action[1,$] = accept		

This parse generates the following tree:



3.3. Tomita's parsing algorithm

The rules and parse tables used in the previous section were generated from the SDF definition:

```
module LRexample
begin
    lexical syntax
        layout SPACE
        functions
            [ \t\n\r]           -> SPACE
    context-free syntax
        sorts BOOL
        priorities
            and > or
        functions
            true                -> BOOL
            false               -> BOOL
            BOOL or BOOL        -> BOOL { left-assoc }
            BOOL and BOOL       -> BOOL { left-assoc }
end LRexample
```

This SDF definition, with the priority and associativity declarations removed from it, would generate the

same rules but the following ambiguous parse tables:

State	Action					Goto
	true	false	or	and	$	BOOL
0	s2	s3				1
1			s5	s4	acc	
2	r0	r0	r0	r0	r0	
3	r1	r1	r1	r1	r1	
4	s2	s3				6
5	s2	s3				7
6	r3	r3	s5/r3	s4/r3	r3	
7	r2	r2	s5/r2	s4/r2	r2	

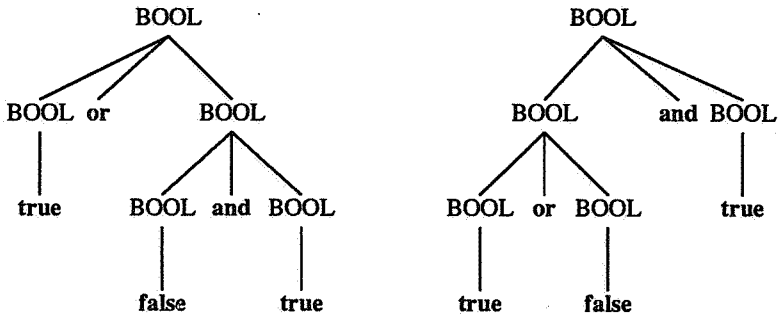
The ambiguities in the grammar are reflected in the shift/reduce conflicts in the action table. For example, Action [6,or] = s5/r3 is a multiple entry and two actions are possible: (1) shift the 'or' and go to state 5 or (2) reduce 'BOOL and BOOL' to 'BOOL'.

The normal LR parsing algorithm cannot handle such multiple entries in its parse tables, but Tomita's algorithm can. It starts as a normal LR parser, but when it encounters a multiple entry in its action table as many new LR parsers are created as there are multiple entries. Each of these parsers maintains its own stack. Their shift actions are synchronized: only when all parsers are ready to do so, the next input symbol is pushed on their stacks simultaneously. For example, the ambiguous sentence 'true or false and true' is parsed as follows:

Parse stack Action	Input stream
0 s2	true or false and true $
0 true 2 r0	or false and true $
0 BOOL 1 s5	or false and true $
0 BOOL 1 or 5 s3	false and true $
0 BOOL 1 or 5 false 3 r1	and true $
0 BOOL 1 or 5 BOOL 7 s4/r2, so split up	and true $

Parse stack Action	Parse stack Action	Input stream
0 BOOL 1 or 5 BOOL 7 s4, but has to wait	0 BOOL 1 or 5 BOOL 7 r2	and true $
0 BOOL 1 or 5 BOOL 7 s4	0 BOOL 1 s4	and true $
0 BOOL 1 or 5 BOOL 7 and 4 s2	0 BOOL 1 and 4 s2	true $
0 BOOL 1 or 5 BOOL 7 and 4 true 2 r0	0 BOOL 1 and 4 true 2 r0	$
0 BOOL 1 or 5 BOOL 7 and 4 BOOL 6 r3	0 BOOL 1 and 4 BOOL 6 r3	$
0 BOOL 1 or 5 BOOL 7 r2	0 BOOL 1 acc	$
0 BOOL 1 acc		$

This parse generates the following trees:



3.4. Optimizations in Tomita's algorithm

Straightforward application of the method described in the previous section leads to a parser working in exponential time. Several optimizations are therefore necessary: combination of stacks, subtree sharing and local ambiguity packing.

Combination of stacks

In the parsing algorithm described thus far the number of computations can grow exponentially when ambiguities are encountered, because there is no way in which a parser can use the results of other parsers.

This can be avoided by joining the parse stacks of different parsers. If parsers are in a common state, that is, if their stacks have the same state on top, they will behave in exactly the same manner until that state is popped off again. So the tops of their stacks can be combined and the parsers will work as one parser for a while. In the previous example the parse stacks can combined after the shift of the 'and'. The current state of both parsers becomes '4' and they will remain in the same states until the 'and' is popped off their stacks again for the reduction of 'BOOL → BOOL and BOOL'.

This method significantly reduces the number of computations, but the number of stacks that must be maintained may still grow exponentially when ambiguities are encountered. This number can be reduced by not copying the whole stack when a multiple entry is encountered but only the necessary portion of it. If both methods are applied, all parse stacks become part of a single graph-structured stack.

The parsing algorithm must also produce parse trees. Even with the parsing method described, the parser would take exponential time to print out all the parse trees in case of ambiguous sentences. So an efficient representation for the various parse trees is needed. This is achieved by subtree sharing and local ambiguity packing.

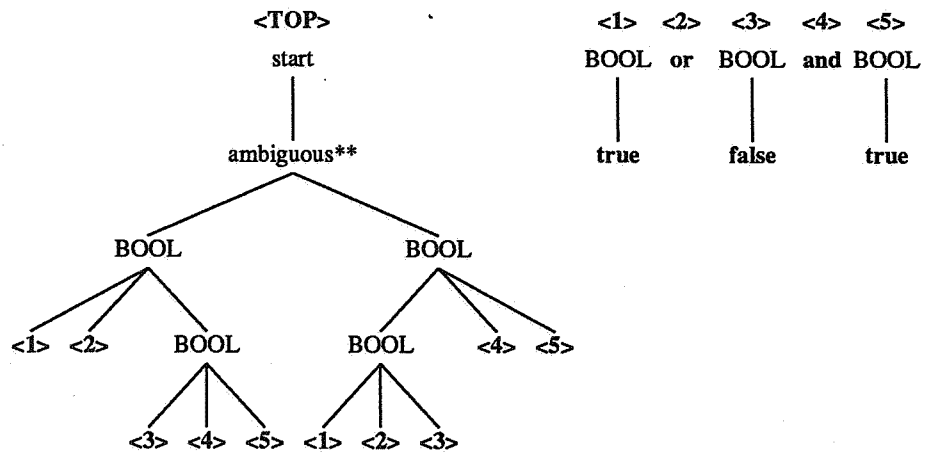
Subtree sharing

If two or more parse trees have a common subtree, that subtree will be represented only once in the parse forest. This is easy to implement, because when parses have a common subtree, the corresponding parsers were already working together.

Local ambiguity packing

A sentence is called locally ambiguous if a fragment of it can be reduced to a the same nonterminal 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. This packing can be implemented in a rather straightforward manner with the parsing techniques used.

With these techniques the parse tree for the sentence 'true or false and true' of the previous example becomes:



In this parse tree all multiply used subtrees are shared by the different parses and the ambiguous subtrees are packed into one 'ambiguous' node.

Tomita's parsing algorithm can work also with ambiguous (i.e. multiple) lexical interpretations of an input text. For example, in the SDF definition given at the end of section 2, the string `then` can be interpreted both as the literal defined in `if exp then statement -> statement`, as well as the lexical sort `id` as defined in `letter+ -> id`.

When a string can have the lexical interpretations $l_1, \cdots, l_n$ the parser merges the entries Action[s,l_1], $\cdots$, Action[s,l_n] into a single set, which is handled in the same way as a multiple entry in the parse tables. Tomita's algorithm cannot handle lexical ambiguities with different lengths, because the parsers have to work simultaneously on the same symbol.

The time efficiency of Tomita's algorithm will be discussed further in section 6.

4. THE PREPROCESSING PHASE IN THE SDF IMPLEMENTATION

Before generating lexical scanners and parsers from an SDF definition, the definition must be checked for statically detectable errors such as: improperly used sorts, recursive definitions in the lexical part, faulty priority declarations, etc.

When an SDF definition turns out to be statically correct, separate lexical rules (regular expressions) and BNF rules are generated from it.

Extracting regular expressions from an SDF definition

The lexical scanner generator uses regular expressions as its input formalism plus information on what to do when a regular expression matches a string of characters in the input sentence. There are four kinds of lexical definitions in SDF definitions:

- Lexical functions describing layout sorts. Input strings matched by the corresponding regular expression must be skipped.
- Lexical functions describing sorts that are used by the context-free part. When the corresponding regular expressions match a string, the name of the sort and the string itself must be passed on to the parser.
- Lexical functions describing sorts that are not used by the context-free part. These sorts are defined for internal use in the lexical grammar, so they may only be used as part of other regular expressions.
- Terminals introduced by the context-free part of the SDF definition. For these terminals unique token names must be generated, which will also be used in the BNF rules of the parser. When such a terminals is recognized by the lexical scanner, the token name and the terminals must be given to the parser.

Because of the restrictions imposed on the lexical part of an SDF definition, all lexical functions can easily be translated to regular expressions. The terminals in the context-free part are already regular expressions.

Extracting BNF rules from an SDF definition

The context-free part of an SDF definition must be transformed into a BNF grammar for use by the parse table generator. BNF grammars consist of a set of rules of the form A ::= β. A is a non-terminal and β stands for zero or more terminals and non-terminals. This rule means that A can be derived from β.

This transformation from the context-free part into BNF rules implies that:

- All sorts are made derivable from the start symbol of the grammar.
- The priority and associativity declarations are passed to the parser generator unchanged. The parser generator processes them separately from the BNF rules.
- In principle, each function declaration β → A generates the rule A ::= β. There are exceptions: When the function declaration has a 'par' attribute, the rule A ::= (β) must also be generated. When the function declaration contains list expressions, the transformation is done according to the following schema:

function declaration	generated rules
α B* γ → A	A ::= α B1 γ B1 ::= B1 ::= B B1
α B+ γ → A	A ::= α B1 γ B1 ::= B B1 ::= B B1
α {B t}* γ → A	A ::= α B1 γ B1 ::= B1 ::= B2 B2 ::= B B2 ::= B t B2
α {B t}+ γ → A	A ::= α B1 γ B1 ::= B B1 ::= B t B1

In reality this schema is more complicated, because α and γ can contain list expressions also.

The generation of lexical tables and parse tables

The technique used to generate a lexical scanner from regular expressions is described in [Alex]. The lexical scanner that is generated from the regular expression works as a finite automaton. It always tries to find the longest string that matches a regular expression. When more than one expression matches that longest string, all interpretations are passed to the parser.

The grammar rules are used by the parser generator to generate parse tables. At the moment the table generation algorithm used is LR(0), but in the future this will be changed to an LALR(1) algorithm. Both algorithms are described in [Aho&Ullman]. After the parse table generation phase, the priorities and associativities still have to be implemented. This implementation is described in the next section.

5. THE IMPLEMENTATION OF THE PRIORITY AND ASSOCIATIVITY DECLARATIONS

Giving one function declaration a higher priority than another one amounts to putting a partial order on parse trees. Hence, in a straightforward implementation of priorities, the parser must (1) generate all possible parse trees for a sentence, (2) sort them according to the priority declarations, and (3) select the parse tree with the 'highest priority'. Needless to say such an implementation is quite inefficient.

A more efficient implementation of priorities was proposed in [Aho,Johnson&Ullman]. It implements priority relations in the parser generation phase and works as a postprocessor on (ambiguous) parse

tables. This technique only covers priority relations that can be decided with a look-ahead of one symbol.

We take the rules and parse tables from section 3.3 as an example:

rule no.	rule
0	BOOL ::= true
1	BOOL ::= false
2	BOOL ::= BOOL or BOOL
3	BOOL ::= BOOL and BOOL
4	start ::= BOOL

State	Action					Goto
	true	false	or	and	$	BOOL
0	s2	s3				1
1			s5	s4	acc	
2	r0	r0	r0	r0	r0	
3	r1	r1	r1	r1	r1	
4	s2	s3				6
5	s2	s3				7
6	r3	r3	s5/r3	s4/r3	r3	
7	r2	r2	s5/r2	s4/r2	r2	

We want to implement the following priorities and associativity attributes in the parse tables:

- BOOL ::= BOOL and BOOL > BOOL ::= BOOL or BOOL
- left-associative: BOOL ::= BOOL or BOOL
- left-associative: BOOL ::= BOOL and BOOL

When the parser encounters the entry Action[6,or], the top part of the stack must be 'BOOL and BOOL' (with the states left out). The entry at that point is s5/r3, a conflict between shifting the symbol 'or' and reducing 'BOOL and BOOL' to 'BOOL'. What are the consequences of making a particular choice in this shift/reduce conflict?

- When the 'or' is shifted, the top part of the stack will eventually become 'BOOL and BOOL or BOOL', which only leaves the possibility to first reduce 'BOOL or BOOL' to 'BOOL' and then 'BOOL and BOOL' to 'BOOL'. So when the 'or' is shifted, a higher priority is given 'BOOL or BOOL' than 'BOOL and BOOL'.
- When the reduce action is preferred, the reduction of 'BOOL and BOOL' to 'BOOL' is made immediately. After this the 'or' can be shifted, which will later make the reduction of 'BOOL or BOOL' possible. So in this case a higher priority is given to 'BOOL and BOOL' than to 'BOOL or BOOL'.

Thus, according to the priority declarations, the reduction is the desired choice in the conflict and the shift action can be removed from the entry Action[6,or].

The associativity attributes can be implemented in the same manner: the entry Action[7,or] = s5/r2 is a conflict between the shift of the 'or' symbol and the reduction of the already existing 'BOOL or BOOL' to 'BOOL'. When the shift action is chosen, the reduction is postponed and the result is a right-associative matching of the 'or'. When the reduce action is chosen, the reduction is made immediately and the 'or' matches left-associatively. Thus, with the given associativity attributes, preference must again be given to the reduce action, and the entry Action[7,or] becomes r2. In general, the attribute 'left-assoc' gives preference to the reduce action and 'right-assoc' to the shift action. The attribute 'assoc' has been implemented as 'left-assoc'.

The proposed method becomes more complicated when more than one shift-reduce conflict appears in an entry, but this can be solved using the same technique: an effort is made to solve each shift-reduce conflict of the entry and the entry is replaced with the union of the results of each shift-reduce conflict.

When all multiple entries are scanned using the given priority and associativity declarations, the ambiguous parse tables shown above are transformed into the parse tables given in section 3.2.

In the parse table generator this implementation of priorities has been used. It proved to work for all

ordinary priorities in mathematical expressions. In the future this implementation will have to be extended with a technique that postprocesses parse trees to implement priorities in general.

6. EFFICIENCY

One of the requirements for the implementation of SDF, formulated in section 2, is that the parsing algorithm must be efficient. Tomita's algorithm appeared to be the most promising, but how did this work out in practice? Three measurements will be presented here:

- A comparison between the SDF implementation and the LALR(1) parser generator Cxyacc.
- A comparison between LR(0) and LR(1) parse table generation algorithms in the SDF implementation.
- A measurement of time efficiency for some grammars that need a large look-ahead.

SDF has been implemented in LeLisp (a Lisp dialect, developed at INRIA [LeLisp]) and all measurements apply to the compiled form of this implementation.

6.1. SDF versus Cxyacc

Being a version of Yacc [YACC], Cxyacc is an LALR(1) parser generator written in C. Cxyacc generates Lisp parse tables which are used to control a parser written in LeLisp. This parser and its associated generator were also developed at INRIA and are described in [Cxyacc].

The efficiency of the SDF implementation and that of Cxyacc were compared on the grammar whose SDF definition was given at the end of section 2. To make this grammar LALR(1), the function declaration

```
print exp*        -> statement
```

was replaced by

```
print {exp ","}* -> statement.
```

The time in seconds used by both systems was:

	Cxyacc	SDF implementation
Construction time	7.2	13.4
Parse times		
sentence 40 words	1.1	1.6
sentence 80 words	2.2	3.6
sentence 120 words	3.5	7.1
sentence 160 words	4.5	9.0
sentence 200 words	5.4	12.5
sentence 240 words	7.1	18.6

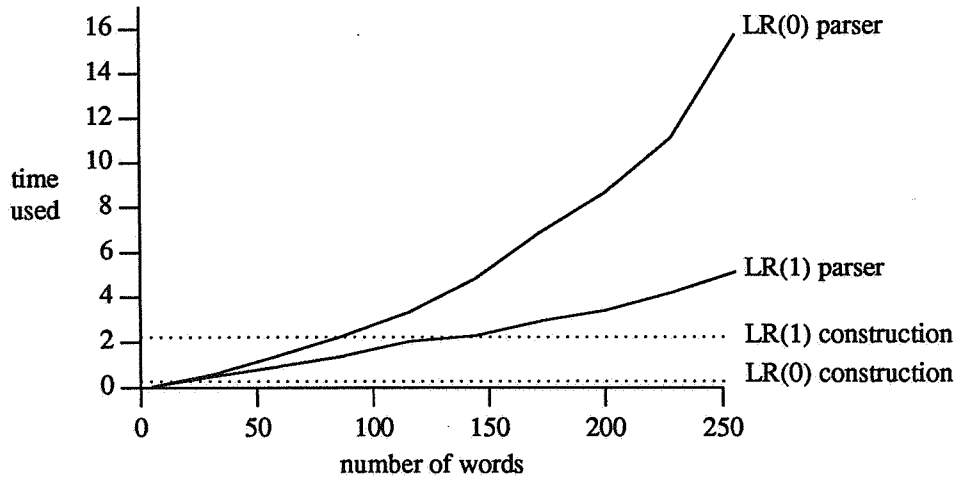
As can be seen the parser generated by the SDF implementation was about twice as slow as the Cxyacc parser. Most of the difference in time consumed by the generated parsers can be explained by the fact that the SDF implementation uses an LR(0) table construction algorithm, while that of Cxyacc is LALR(1). The effect of changing the parse table construction algorithm is shown in the next section.

6.2. LR(0) versus LR(1)

Because Tomita's parser simulates look-ahead by using several parsers in parallel, it can work with LR(0), LR(1) or LALR(1) parse table construction algorithms without any changes.

An LR(0) parse table constructor puts reductions in the tables it generates wherever possible, without checking if the look-ahead is right for a reduction. LR(1) parse table constructors, on the other hand, do put restrictions on the look-ahead when they issue a reduce action. Because an LR(0) algorithm puts more reductions in its tables than an LR(1) algorithm, Tomita's algorithm will start more parsers when controlled by LR(0) tables then when controlled by LR(1) tables. So it will be less efficient with LR(0) tables on non-LR(0) grammars.

The following figure shows the construction and parse times used for a certain LR(1) grammar:



Construction of the LR(1) parse table takes more time than that of the LR(0) table, but the resulting parser is much more efficient and works in linear time.

The grammar used in this measurement is a special case in that both the LR(0) and LR(1) constructors need the same number of states in the parse tables they generate. In general, an LR(1) table constructor will need more states than an LR(0) constructor. For example, for the grammar presented at the end of section 2 the LR(1) algorithm needs four times as many states as the LR(0) algorithm and it uses ten times as much time. The extra work for the parser to manage these larger tables makes the parser as slow with LR(1) tables as with LR(0) tables.

We expect to replace the LR(0) algorithm in the implementation of SDF with an LALR(1) algorithm, because LALR(1) parse table constructors generate parse tables with as many states as LR(0) constructors would generate, but the LALR(1) technique is more selective in putting reduce actions in its tables.

6.3. Dynamical adjustment of look-ahead

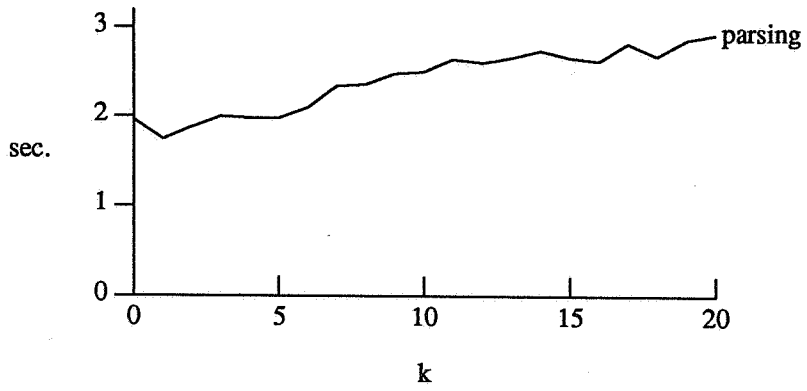
As mentioned earlier, Tomita's algorithm can handle more grammars than any LR(k) algorithm because it dynamically adjust its look-ahead by using the different parse stacks as a look-ahead mechanism.

How the time used to generate parsers and to parse a sentence varies with the required look-ahead will be shown by means of a set of grammars to recognize the structures: $a*b$, $a*ab$, $a*aab$, $a*aaab$, etc. These structures can be denoted as: $a*a^kb$ and we take k from 0 to 20. The SDF definition of the grammar with $k=4$ is:

```
module ak4b
begin
    lexical syntax
        sorts A, B
        layout SPACE
        functions
            a                   -> A
            b                   -> B
            [ \n\r\t]           -> SPACE
    context-free syntax
        sorts sentence, head, tail
        functions
            head tail           -> sentence
            A*                  -> head
```

```
                A A A A B          -> tail
end ak4b
```

The parser for such a grammar has to look k+1 symbols ahead, to be able to decide when to stop recognizing the `head` and to start recognizing the `tail`. The time needed to parse the sentence $a^{20}b$ was:



Time used to recognize the structure $a* a^k b$, $k=1,\ldots,20$

The larger the required look-ahead, the more time is consumed, but the increase remains moderate. The conclusion is that parsers generated by the SDF implementation are well suited for grammars for which a large look-ahead is needed.

7. CONCLUSIONS

The implementation of the syntax definition formalism SDF can be called successful: a lexical scanner and a parser are generated from the SDF definition of a language. These can transform sentences of that language into parse trees. These parse trees can be transformed into abstract syntax trees with little effort. It is not to be expected that this SDF implementation will exactly follow the formal definition of SDF which will appear in [SDFref], because of the implementation of the priorities and the 'assoc' attribute. Still, it is a good framework on which that implementation can be built.

In section 3.1 some requirements for the parsing algorithm were given: no restriction on the class of grammars, handling of ambiguous sentences, fast parser generation, separate parser generation and efficient parsing. Tomita's algorithm satisfied most of these requirements satisfactorily. As mentioned in section 3.1, separate parser generation is a difficult problem with the parse table constructor used for Tomita's algorithm. The expectation that Tomita's algorithm would be as fast as LR(1) parsing algorithms on LR(1) grammars was not fulfilled, improvements are still necessary there.

In the near future we expect to change the following in the implementation of SDF:

- The parse table construction algorithm (use of an LALR(1) table generator).
- The space used for the storage of the parse tables in the parser generator and the parser must be reduced.
- Issues related to detection and recovery of syntax errors have been ignored in this study. It will be necessary to investigate how existing techniques for error recovery in LR-parsers can be adjusted to general context-free parsing as used in the SDF implementation.

ACKNOWLEDGEMENTS

I would like to thank Paul Klint for his guidance in developing the implementation of SDF and our many discussions about it. Also I would like to thank Jan Heering for encouraging and guiding me while I was writing this article. Their experience in these subjects helped me a great deal.

REFERENCES

[Aho&Ullman] A.V. Aho & J.D. Ullman, *Principles of Compiler Design*, Addison-Wesley, 1977.

[Aho,Johnson&Ullman] A.V. Aho, S.C. Johnson & J.D. Ullman, *Deterministic Parsing of Ambiguous Grammars*, Comm. ACM **18**, 8 (August 1975), 441-452.

[Alex] P. Klint, *Alex - A Lexical Scanner Generator*, Centre for Mathematics and Computer Science, to appear, 1987.

[Cxyacc] G. Berry & B. Serlet, *Cxyacc et Lex-kit*, INRIA, 1984.

[Earley] J. Earley, *An Efficient Context-Free Parsing Algorithm*, Comm. ACM **13**, 2 (February 1970), 94-102.

[LeLisp] J. Chailloux, *Le_Lisp, Le Manual de référence*, INRIA, 1984.

[Mini-ML] P.R.H. Hendriks, *Type-checking Mini-ML: an Algebraic Specification with User-Defined Syntax*, Centre for Mathematics and Computer Science, to appear, 1987.

[SDF] J. Heering & P. Klint, *A syntax definition formalism*, Centre for Mathematics and Computer Science, Report CS-R8633, 1986.

[SDFref] J. Heering, P.R.H. Hendriks, P. Klint & J. Rekers, *SDF Reference Manual*, Centre for Mathematics and Computer Science, to appear, 1987.

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

[YACC] S.C Johnson, YACC: yet another compiler-compiler, in: *UNIX Programmer's Manual*, Vol. 2B, Bell Laboratories, 1979.