Incremental Generation of Parsers

J. Heering[1], P. Klint[1,2], J. Rekers[1]

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

2) Programming Research Group, University of Amsterdam
P.O. Box 41882, 1009 DB Amsterdam, The Netherlands

ABSTRACT

An LR-based parser generator for arbitrary context-free grammars is described, which generates parsers by need and processes grammar modifications by updating already existing parsers. We motivate the need for these techniques in the context of interactive language definition environments, present all required algorithms, and give measurements comparing their performance with that of conventional techniques.

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 situations however, where 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 not only determined by the efficiency of the parser, but also by that of the parser generator.
  - It may happen that some parts of the grammar are not 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 corresponding modification in 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 incrementally incorporate modifications of the grammar in the parser.

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. When typical input sentences need only a small part of the grammar, a smoother response is achieved than in the conventional case, since there is no delay time due to the parser generation phase and parsing can start immediately. When 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 unaffected by the modification in the grammar are re-used. Hence, the effort put in generating them is not thrown away. This clearly has advantages for interactive language definition systems.
- The efficiency of the parsing process itself remains unaffected, in the sense that once all needed parts of the parser have been generated, the parser will be as efficient as a conventionally generated parser.
- The parsing algorithm is capable of handling arbitrary context-free grammars.

For the general principles underlying our method, see [HKR87b]. In [HKR87a] 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 syntax written in SDF, 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 on the fly from the SDF definition.

In section 2 we discuss related algorithms and show how our technique evolved from them. In section 3 we present the parsing algorithm used by us. Section 4 describes a conventional parser generation algorithm. We extend this algorithm into a lazy parser generation algorithm in section 5. In section 6 we extend it once again into an incremental parser generation algorithm. Finally, section 7 gives the results of efficiency measurements, and section 8 contains some concluding remarks.

2. THE CHOICE OF A PARSING ALGORITHM

2.1. A comparison of algorithms

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, ch. 4.7]
  These algorithms are controlled by a parse table that is constructed beforehand by a table generator. The table is constructed top-down, while 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 impossible to update an already generated parse table incrementally, if the grammar is modified.
- Recursive descent and LL(k) algorithms [ASU86, ch. 4.4]
  A recursive descent parser generator constructs a parsing program, whereas an LL generator constructs 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 (dot) 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 recursively traversed. 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, because 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, p. 60] "parsing can be expensive for complex expressions", which makes the algorithm less suitable for large input sentences.
- Pseudo-parallel LR parsing [Lan74, Tom85, Rek89]
  This is an extended LR parsing algorithm, that requires a conventional (but possibly ambiguous) LR(0) or LR(1) parse table. The parser starts as an LR parser, but when it hits a conflict in the parse table, it splits up in several LR parsers that work in parallel. The theoretical framework for parallel LR and LL parsing was introduced in [Lan74], and, independently, Tomita optimized it for LR parsing. Grammars are restricted to the class of finitely ambiguous context-free grammars. 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 parallel LR 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 the 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 controlled Tomita parser.

Fig. 1 gives an overview of the following characteristics of these algorithms: capable of handling arbitrary context-free grammars (powerful), efficient on large input sentences (fast), efficient processing of modifications of the grammar (flexible), and modular composition of parsers should be possible (modular).

	powerful	fast	flexible	modular
LR(k)	+/−	++	−−	−−
LL(k)	−	++	−−	−−
Earley	++	−−	++	++
Cigale	−−	−	++	++
OBJ	+	−−	+	+
Par. LR	++	++	−−	−−
IPG	++	++	+	+

Fig. 1: Comparison of various parsing algorithms.

2.2. Evolution of parsing algorithms

The simplest scheme for a (general) parser is given in Fig. 2.a where the parser is controlled directly by the grammar. An example of this technique is Earley's algorithm in its pure form. This kind of parser adapts easily to modifications in the grammar, but is inefficient because for each parse step all parsing information must be (re)computed from the grammar.

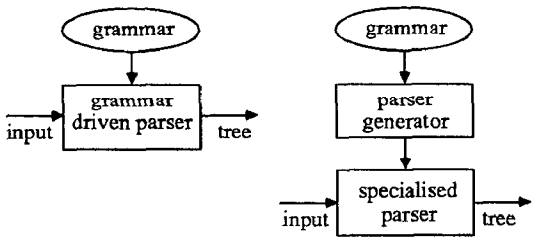

Fig. 2.a: Grammar driven parser, **2.b** specialized parser.

These general grammar driven parsing algorithms have evolved into the scheme of Fig. 2.b, where a specialized parser is generated for a given grammar. An example of this scheme is the recursive descent algorithm. These parsers are more efficient because the parsing information is computed only once in the parser generation phase.

Another frequently used organization is shown in Fig. 3.a, where the parser is split into a grammar dependent part, the parse table, and a grammar independent part, the table driven parser. The parse table and the table driven parser together form a specialized parser for the grammar, and the parse table is computed in a separate table generation phase. Examples of this technique are LL and LR parsing algorithms and Tomita's parsing algorithm. The algorithms to be presented in section 3 and 4 all fall in this category.

In the system shown in Fig. 3.b the table generation phase is made part of the parser. Here the table driven parser is controlled by a parse table that is generated by need. The parser uses the same efficient technique as that of Fig. 3.a. This is the lazy parse table generation technique that will be explained in section 5.

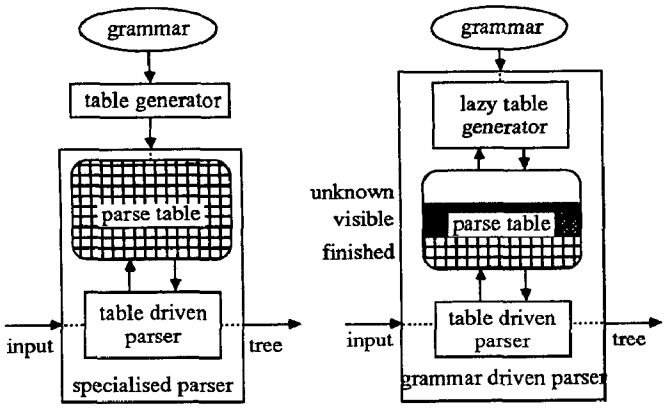

Fig. 3.a: Table driven parser, **3.b** lazy parser generation.

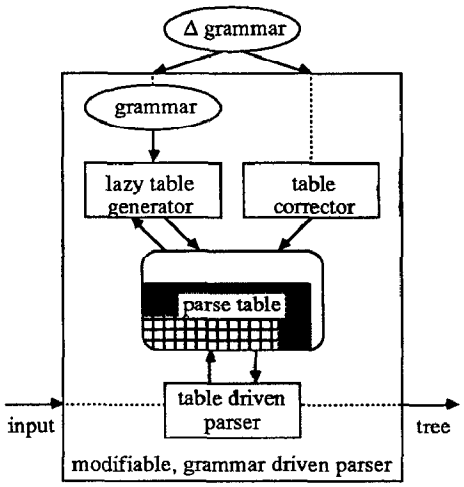

Fig. 4: Incremental parser generation

Fig. 4 shows an incremental parse table generator for modifiable grammars. It consists of the same lazy table generator and the same table driven parser, but these are extended with a table corrector. The latter can remove parts of the generated parse table that have become incorrect due to a modification in the grammar. For now, these modifications may not occur *during* the actual parsing of a sentence. This technique will be explained in section 6.

3. LR PARSING ALGORITHMS

In this section we discuss a simplified version of Tomita's (pseudo-)parallel LR parsing algorithm [Tom85, Rek89]. It basically consists of a (dynamically varying) number of conventional LR parsers running in parallel. We therefore recall the details of ordinary LR parsing first.

In the algorithms which follow we need some low-level functions. For objects we use the functions *new*(T) to create an object of type T, and *copy*(O) to make a copy of object O. When an object O has a field f, that field can be accessed by $O.f$. For lists we use the functions *head*(L), *tail*(L), and *length*(L), and for stacks we use *push*(e, S), *pop*(S), and *top*(S).

3.1. Conventional LR parsing

An LR parser is controlled by its parse table, which has an *ACTION* and a *GOTO* component. The *ACTION* table determines, on the basis of the current state of the parser and the current input symbol, the action for the parser to perform. An action can be either a 'shift', 'reduce', 'accept', or 'error'. After a 'reduce' action, the parser uses the *GOTO* table to determine what to do with the recognized non-terminal.

LR-PARSE: A simple LR parser.

Input: A start state *start-state* and a sentence *sentence*. *Start-state* is the state in which the parser starts, and *sentence* is a list of terminals terminated by the end-marker \$.

Output: 'true' if *sentence* is grammatically correct, 'false' otherwise.

Description: The LR parse algorithm uses a stack of states. The state on top of the stack is the current state of the parser, and the head symbol of the input sentence is its current input symbol. The parser calls *ACTION*(*state, symbol*) with the current state and current symbol. Basically, the action '(shift *state'*)' means that the parser has advanced one step in recognizing a syntax rule and must go to state *state'*, the action '(reduce *rule*)' means that the parser has recognized the syntax rule *rule* completely, the action 'accept' means that the whole input has been recognized, and the error action, which is denoted by an empty action set, means that the input read so far can never become a sentence of the language any more.

We adapted the original LR parsing algorithm, as given for instance in [ASU86, ch. 4.7], a bit to our needs later on: (1) *ACTION* returns a set of possible actions rather than a single action. *LR-PARSE* can only handle sets of at most one action correctly, but the parallel LR-parser discussed in section 3.2 can handle action sets of arbitrary length. (2) To keep things simple, we do not generate parse trees and we do not keep symbols on the parse stack, but only states. (3) We use an object of type 'LRparser' with a single field *stack* as the parse stack of the algorithm. *LR-PARSE* uses only one stack, but the parallel version requires a dynamically varying number of them.

Algorithm:

```
LR-PARSE(start-state, sentence):
  parser := new(LRparser)
  push(start-state, parser.stack)
  symbol, sentence := head(sentence), tail(sentence)
  while true do
    state := top(parser.stack)
    if ∃action ∈ ACTION(state, symbol) then
      if action = (shift state') then
        push(state', parser.stack)
        symbol, sentence := head(sentence), tail(sentence)
      elseif action = (reduce A ::= β) then
        for 1 · · · length(β) do pop(parser.stack) od
        state' := top(parser.stack)
        push(GOTO(state', A), parser.stack)
      elseif action = (accept) then
        return true
      fi
    else
      return false
    fi
  od
```

3.2. (Pseudo-)parallel LR parsing

The parallel LR parsing algorithm starts as a simple (conventional) LR parser, but splits up in multiple parsers when *ACTION*(*state, symbol*) returns multiple actions. All simple LR parsers are synchronized on their shift actions in such a way, that only when all parsers have shifted on the current input symbol, the next symbol is processed.

The parallel execution of all possible actions makes the algorithm fit for arbitrary context-free grammars, independently of the used look-ahead k of its LR(k) parse table generator. A larger k will however make parsing faster, as less

parser will then be needed.

PAR-PARSE: Parse a sentence with (pseudo-)parallel running LR parsers.

Input: A start state *start-state* and a sentence *sentence*. *Start-state* is the state in which the first simple LR parser starts, and *sentence* is a list of terminals terminated by the end-marker $.

Output: 'true' if *sentence* is grammatically correct, 'false' otherwise.

Description: The synchronization on shift actions is expressed in the algorithm by using two pools of parsers, *this-sweep* and *next-sweep*. The parsers in *this-sweep* still have to shift on the current input symbol, the parsers in *next-sweep* are waiting for the next symbol. Only when *this-sweep* is empty (and if there are parsers left in *next-sweep*), the next symbol is read from the input sentence. When both pools of parsers are empty, this means that all parsers died in an accepting or rejecting configuration. *PAR-PARSE* accepts its input if at least one simple parser accepts it.

For each input symbol parsers are taken from *this-sweep*, until *this-sweep* is empty. The parser that is taken from *this-sweep* is removed from it, because for each action a copy of the parser is made and the action is performed on this copy. So, when there are no actions to be performed, the parser just disappears. Shift actions place the copy in the *next-sweep* pool, reduce actions put the copy back in *this-sweep*. Accept actions only set the variable *accepted* to indicate that a simple parser has accepted the input.

It is important for the lazy parser generator that the implementation of the *copy* operation for parsers is such that the parse stacks become different objects which share the states on them.

Algorithm:

```
PAR-PARSE(start-state, sentence):
  accepted := false
  start-parser := new(LRparser)
  push(start-state, start-parser.stack)
  next-sweep := {start-parser}
  while next-sweep ≠ ∅ do
    symbol, sentence := head(sentence), tail(sentence)
    this-sweep, next-sweep := next-sweep, ∅
    while ∃parser ∈ this-sweep do
      this-sweep := this-sweep − {parser}
      state := top(parser.stack)
      actions := ACTION(state, symbol)
      for ∀action ∈ actions do
        parser' := copy(parser)
        if action = (shift state') then
          push(state', parser'.stack)
          next-sweep := next-sweep ∪ {parser'}
        elseif action = (reduce A ::= β) then
          for 1 ··· length(β) do pop(parser'.stack) od
          state' := top(parser'.stack)
          push(GOTO(state', A), parser'.stack)
          this-sweep := this-sweep ∪ {parser'}
        elseif action = (accept) then
          accepted := true
        fi
      od
    od
  od
  return accepted
```

The optimization of Tomita on this parallel parsing algorithm consist of using a parse graph, instead of a number of parse stacks. Stacks in this graph are split when more actions than one are possible, and joined again whenever the same state appears on top of them. This maximally avoids double work [Tom85].

PAR-PARSE is controlled by *start-state* and the results of *ACTION* and *GOTO*. *Start-state* is part of a larger parser control structure from which *ACTION* and *GOTO* receive their information too. The next section describes how this control structure can be generated by a parser generator.

4. CONVENTIONAL PARSER GENERATION

The parse table generation algorithm we describe in this section is the conventional LR(0) algorithm, of which the lazy parse table generation algorithm discussed in section 5 will turn out to be a straightforward extension. As far as the parsing algorithm itself is concerned, there is no difference between the two generators.

We often speak of a *parser* generator, while in the LR case, we actually ought to speak of a *parse table* generator. But, as one can argue that the generated parse tables are interpreted by a hard-wired LR-parsing algorithm, a parse table can be seen as a program running on an LR-parsing machine.

The table generated by an LR parse table generator is a tabular representation of an internal structure built by the generator. This internal structure is a 'directed graph of item sets'. Each row in the parse table represents a state of the parser, and each state is equivalent to a set of items. The graph of these item sets is thus the notion underlying both the parse table and the parsing states. Fig. 5, 6 and 7 give an example of an ambiguous grammar, its parse table and its graph of item sets.

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

Fig. 5: Grammar of the Booleans.

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	

Fig. 6: Parse table of the Booleans.

Each box in the graph of Fig. 7 is a set of items. The arrows between sets of items are the labeled edges of the graph. During parsing, the parser moves through the graph: a shift action causes a move along an edge labeled with a terminal symbol; a reduce action first causes a move back according to the stacked path of states, and next a move forward along an edge labeled with a non-terminal symbol. Fig. 8 shows the moves of a parser when parsing the sentence 'true or false'.

A set of items is an object with the following fields:

- *kernel*

  The *kernel* field of a set of items contains the rules that are potentially being recognized by the parser in that state/set of items. The dots '•' indicate how far the parser has progressed in each rule.

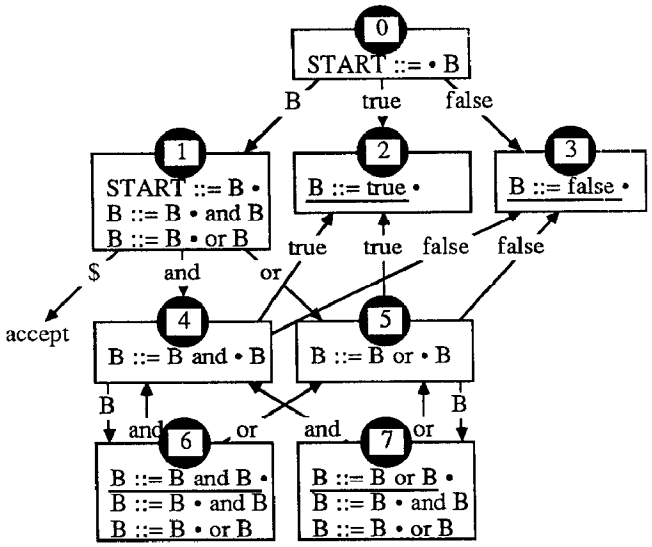

Fig. 7: Graph of item sets of the Booleans.

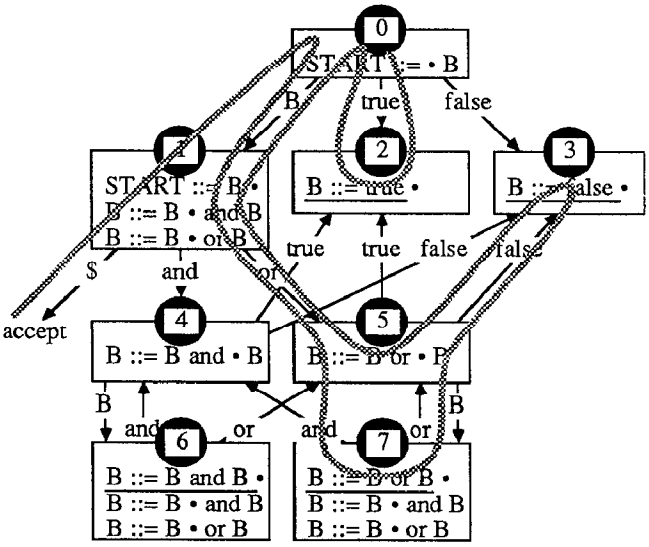

Fig. 8: The parsing of 'true or false'.

- *transitions*
  Each transition in the *transitions* field of a set of items contains an edge to another set of items labeled with a symbol. Transitions have the form (*symbol itemset*), with *itemset* a set of items. If *symbol* is a terminal the transition is a shift action; if it is a non-terminal the transition is a *GOTO* transition. The transition ($ accept) is a special case, the accept action.
- *reductions*
  The *reductions* field of a set of items contains the syntax rules that have been recognized completely in this state/set of items. These rules may be reduced by the parser. In diagrams we indicate reductions by underlining a rule in the corresponding *kernel* field (reductions of rules which are not also in the *kernel* field can not be represented in these diagrams).
- *type*
  The value of the *type* field of a set of items may be 'initial' or 'complete'. If it is 'initial', the fields *transitions* and *reductions* have not yet been computed. In diagrams we indicate 'complete' sets of items with a black circle and 'initial' sets of items with an open circle. The number in the circle serves as a unique reference to each set of items.

The parse table in Fig. 6 is a tabular representation of the graph of item sets of Fig. 7. This representation is normally used for an LR parse table. It contains the results of the functions *ACTION(state, symbol)* and *GOTO(state, symbol)* with a row for each state and a column for each symbol. We shall not use these parse tables further, because the lazy parser generator also needs the *kernel* field of each set of items during parsing. How *ACTION* and *GOTO* derive their information from the *transitions* and *reductions* fields is described by the following algorithms.

ACTION:
Input: A state *state* and a terminal symbol *symbol*.
Output: The actions the parser can perform in *state*.
Description: The argument *state* is a complete set of items, and the return value is deduced from its *transitions* and *reductions* fields.
Algorithm:
ACTION(state, symbol):
  result :=
    {(reduce $A ::= \beta$) | $A ::= \beta \in$ *state.reductions*} ∪
    {(shift *state'*) | (*symbol state'*) ∈ *state.transitions*} ∪
    {(accept) | (*symbol* accept) ∈ *state.transitions*}
  return *result*

GOTO:
Input: A state *state* and a non-terminal *symbol*.
Output: The new state for the parser after reducing a rule that delivered *symbol* in state *state*.
Description: The argument *state* is a complete set of items, and the return value is deduced from its *transitions* field. Because we assume the graph of item sets to have been generated correctly, we can be sure that there is exactly one transition for *symbol* in *state.transitions*.
Algorithm:
GOTO(state, symbol):
  return *state'*: (*symbol state'*) ∈ *state.transitions*

The graph of item sets from which *ACTION* and *GOTO* obtain their information is generated from the grammar by the routine *GENERATE-PARSER*:

GENERATE-PARSER: Build a graph of item sets for a grammar.
Input: A grammar *Grammar*, which is a set of syntax rules $A ::= \alpha$, with A a non-terminal and α a list of zero or more terminals and/or non-terminals. The non-terminal START is the start symbol of the grammar. START may not be used in the right-hand side of a syntax rule.
Output: The state in which parsing must start.
Description: This routine generates a graph of item sets for the given grammar. The set of items *start-itemset* it returns is the state in which parsing starts and is the root of the graph of item sets for *Grammar*. *ACTION* and *GOTO* can access other sets of items in this graph. The *kernel* field of *start-itemset* is composed of all rules in *Grammar* with START as left-hand side, with the dot placed before the first symbol of the right-hand side.
Itemsets contains all sets of items created during the generation process. It is used when sets of items are expanded, as well as for searching for sets of items that are not yet complete. Routine *EXPAND* transforms an initial set of items into a complete one. While expanding a set of items, *EXPAND* may add initial sets of items to *Itemsets*.
Algorithm:
GENERATE-PARSER(Grammar):
  start-itemset := *new*(itemset)
  start-itemset.type := initial

```
start-itemset.kernel :=
    {START ::= •β | START ::= β ∈ Grammar}
Itemsets := {start-itemset}
while ∃itemset ∈ Itemsets: itemset.type = initial do
  EXPAND(itemset)
od
return start-itemset
```

EXPAND: Transform an initial set of items into a complete set of items.

Input: A set of items *itemset* with type 'initial'.

Description: This routine computes the *transitions* and *reductions* fields of *itemset*. It starts by using *CLOSURE* to generate an extension of the kernel containing all rules that may become applicable in state *itemset*. This extended kernel is 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 *kernel'* by moving the dot over the S. When a set of items *itemset'* with kernel *kernel'* does not yet exist, it is generated as an initial set of items. The transition (S *itemset'*) is now added to *itemset.transitions*. A rule in the extented kernel having its dot at the end has been recognized completely. It depends on the left-hand side of the rule if this means an accept or a reduce action.

Algorithm:

```
EXPAND(itemset):
  closure := CLOSURE(itemset.kernel)
  itemset.transitions, itemset.reductions := ∅, ∅
  for ∀S ∈ {S | A ::= α•Sβ ∈ closure} do
    kernel' := {A ::= αS•β | A ::= α•Sβ ∈ closure}
    if ∃itemset' ∈ Itemsets: itemset'.kernel = kernel' then
      itemset.transitions :=
          itemset.transitions ∪ {(S itemset')}
    else
      itemset' := new(itemset)
      itemset'.type, itemset'.kernel := initial, kernel'
      Itemsets := Itemsets ∪ {itemset'}
      itemset.transitions :=
          itemset.transitions ∪ {(S itemset')}
    fi
  od
  for ∀A ::= β• ∈ closure do
    if A = START then
      itemset.transitions := itemset.transitions ∪ {($ accept)}
    else
      itemset.reductions := itemset.reductions ∪ {A ::= β}
    fi
  od
  itemset.type := complete
```

CLOSURE:

Input: A set of dotted rules *kernel*.

Output: The same set of dotted rules, extended with all rules that can also become applicable.

Description: *CLOSURE* computes a set *closure*, which is initialized at the incoming *kernel*. It then extends *closure* to all rules that may become applicable. If there is a rule $A ::= \alpha \bullet B\beta$ in *closure* it means that non-terminal B may become applicable. Hence, *closure* can be extended with all rules $B ::= \bullet\gamma$, because when a B can be recognized, all rules that derive B can also be recognized.

Algorithm:

```
CLOSURE(kernel):
  closure := kernel
  while ∃A, B, α, β, γ :
        A ::= α•Bβ ∈ closure ∧
        B ::= γ ∈ Grammar ∧
        B ::= •γ ∉ closure do
    closure := closure ∪ {B ::= •γ}
  od
  return closure
```

5. LAZY PARSER GENERATION

The parser generation algorithm described in the previous section generates the parser completely before it is used. This method is based on the assumption that a parser is only generated 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 smoother response time. Lazy parser generation has this property, and it has the further advantage that only those parts of the parser are generated that are really needed to parse the sentences given to it. Both these arguments in favour of lazy parser generation are only valid when typical input sentences need a relatively small part of the parser. See [HKR87b] for an in-depth discussion of the advantages and disadvantages of lazy program generation. In our specific application, we mainly use the lazy parser generation algorithm as a step towards *incremental* parser generation (section 6).

5.1. An algorithm for lazy parser generation

We only have to adjust the LR(0) parser generator of the previous section a little to transform it to a lazy parser generator. We move the parser generation phase into the parsing phase by moving the expansion of initial sets of items from *GENERATE-PARSER* to *ACTION*. This means that the state with which *ACTION* is called, cannot only be a complete set of items but also an initial one. When it is still initial, it is expanded first by *EXPAND*. *GENERATE-PARSER* now only generates *start-itemset* as an initial set of items, the rest of the parser generation will be taken care of by *ACTION*.

GENERATE-PARSER: Build the first part of a graph of item sets from a grammar.

Input: A grammar *Grammar*, which is a set of syntax rules $A ::= \alpha$.

Output: The state in which parsing must start.

Description: This routine constructs the set of items *start-itemset*: the root of the graph of item sets for the given grammar. *ACTION* and *GOTO* can access other sets of items in this graph. The *kernel* field of *start-itemset* is composed of all rules in *Grammar* with START as left-hand side, with the dot placed before the first symbol of the right-hand side.

Algorithm:

```
GENERATE-PARSER(Grammar):
  start-itemset := new(itemset)
  start-itemset.type := initial
  start-itemset.kernel :=
      {START ::= •β | START ::= β ∈ Grammar}
  Itemsets := {start-itemset}
  return start-itemset
```

ACTION:

Input: A state *state* and a terminal *symbol*.

Output: The actions the parser can perform in *state*.

Description: When *state* is an initial set of items it must be expanded first. The complete set of items is then used to

deduce the return value from the *transitions* and *reductions* fields.

Algorithm:

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

Like *ACTION*, *GOTO* uses information that is only available in complete sets of items, so one might be inclined to think that the same test for initial sets of items has to be added to *GOTO* as well. However, due to the the particular way in which the parsing algorithm works, *GOTO* will only be called with sets of items 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 was previously visited, implies that *ACTION* was already called for it. During that call the state will have been completed.

The implementation of the lazy parser generator has to treat variables *Itemsets* and *Grammar* of *GENERATE-PARSER* as global variables, because they are needed during the expansion of sets of items. Furthermore, several complete sets of items can point to the same initial set of items. When expanding an initial set of items, the implementation has to take care that all sets of items that originally pointed to the initial set of items now point to the completed one.

5.2. An example of lazy parser generation

Consider the grammar of the Booleans of Fig. 5. The graph of item sets generated by the lazy parser generator initially consists only of the start set of items (with type initial) shown in Fig. 9.a.

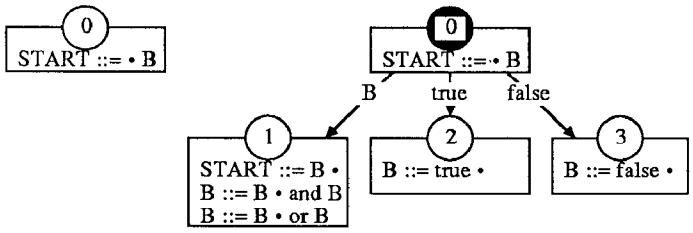

Fig. 9: The graph of item sets after **9.a** generation, **9.b** the first call to *ACTION*.

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 set of items *start-state* which is expanded first to the graph shown in Fig. 9.b. Fig. 10 shows the graph of item sets after the sentence 'true and true' has been parsed.

All sentences that only contain 'and' and 'true', will now be parsed without further expansion of the graph of item sets. Only for sentences containing 'false' or 'or', the graph of item sets has to be expanded again. The advantage of the lazy technique is rather small for the grammar of the Booleans, but for a larger grammar like that of SDF (given in appendix A) only 60 percent of the parse table had to be generated to parse the SDF definition of SDF itself (see section 7 for all measurements). In this case the lazy parser generation technique clearly has advantages.

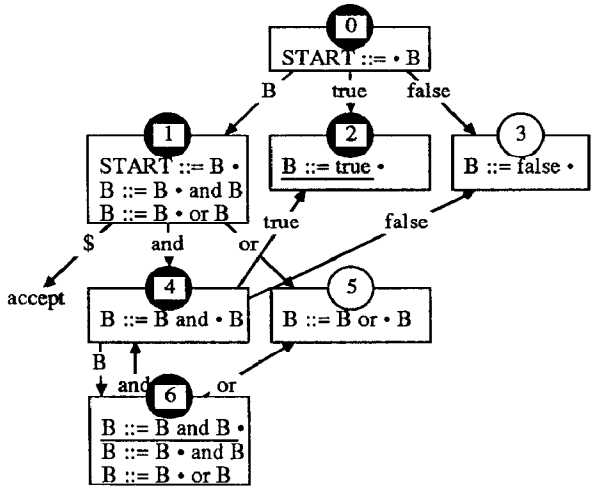

Fig. 10: The graph of item sets after the sentence 'true and true' has been parsed.

5.3. The cost of lazy parser generation

The overhead in time introduced by this lazy technique is small. The total generation time, which is now distributed over parsing, will not increase, since even in the worst case exactly the same amount of work has to be done as before. Only the test in *ACTION* which determines the type of a given set of items takes some extra time.

In contrast to the conventional technique, where only the *ACTION* and *GOTO* information was needed during parsing, the lazy parser generator also needs the *kernel* fields of the sets of items. So the lazy parser generator uses more memory than a comparable conventional one. One could decide to remove the kernels when all sets of items have been expanded, but the incremental parser generator will need them again when the grammar is modified.

We considered making the lazy parser generator even more lazy than it already is: it is unnecessary to expand an entire set of items at once, since only that part has to be expanded that is needed to deduce the actions for the specific symbol with which *ACTION* was called. However, the additional administrative overhead incurred (For what symbols has the set of items already been expanded? What was the closure of the kernel?) turned out to be so large that no net gain in efficiency was to be expected.

6. INCREMENTAL PARSER GENERATION

The lazy parser generator can only react to modifications of the grammar by throwing away the parser it has already generated and by restarting from scratch. Although the lazy technique is an improvement over the conventional method, it is still rather wasteful. We would like to exploit the fact that when a grammar is modified, it is likely that a relatively large part of it stays the same, and that the graphs of item sets for both grammars will have large parts in common.

In this section we describe an incremental parser generator that retains those parts of the old graph of item sets that can still be used in the graph for the modified grammar. How much has to be thrown away depends 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 item sets is already highly specialized towards the old grammar, chances are that larger parts of it have to be removed.

We first show that extension of a grammar does not correspond in a straightforward way to extension of its graph

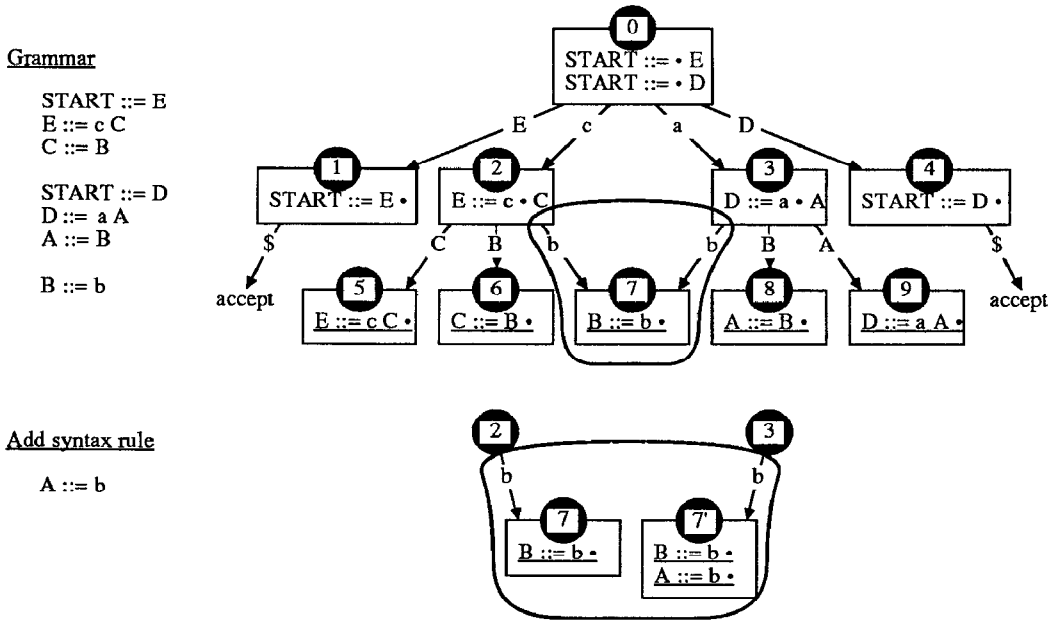

Fig. 11: A more complicated update.

of item sets. Suppose a rule is added to a grammar. Clearly, the old grammar is a subgrammar of the new one. Is the old graph of item sets in some sense a subgraph of the new one as well? There are reasons to believe so. Consider again the grammar of the Booleans of Fig. 5, and extend this grammar with the rule 'B ::= nil'. The graph of item sets for the Booleans has now to be updated, as is shown in Fig 12.

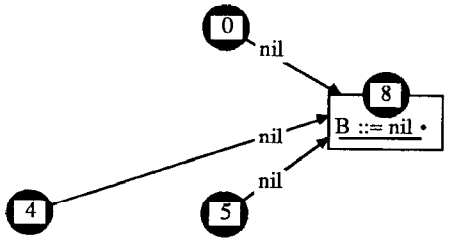

Fig. 12: The needed update of the graph of the Booleans.

Transitions have been added to some sets of items, but existing transitions or kernels did not have to be changed. Unfortunately, this is not always the case.

Consider, for instance, the grammar and the graph of item sets of Fig. 11. This grammar is a complicated way to describe a language with only two sentences '*a b*' and '*c b*', but it is the smallest grammar we could think of for which an update has more severe implications than in the previous case. Fig. 11 shows the modifications of the graph after an addition of the rule $A ::= b$ to the grammar. It illustrates that even if rules are only *added* to the grammar, there is no guarantee that the original graph is a subgraph of the graph for the modified grammar.

6.1. 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 item sets for the new grammar. It does this by making all sets of items in the graph initial, that were (from the viewpoint of the new grammar) expanded incorrectly. The lazy parser generator will then, when needed, re-expand these sets of items according to the new grammar.

Suppose a rule $A ::= \beta$ is added to the grammar. We then have to find the states (sets of items) in which recognition of the new rule should start. In the new graph the *closure* of the kernel of these sets of items would contain $A ::= \bullet\beta$. How can we find these sets of items in the existing graph without recomputing the closure of every kernel? Initial sets of items 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 only be added to the closure when that closure contains at least one dotted rule with its dot before an A. But when 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* field of the set of items in question. So we can recognize all complete sets of items that should have $A ::= \bullet\beta$ in the closure of their kernel by the presence of a transition (*A itemset'*) in their *transitions* field.

Similarly when we *delete* a rule $A ::= \beta$ from the grammar, we have to find the states (sets of items) in the existing graph in which recognition of this rule started. These are the sets of items that had $A ::= \bullet\beta$ in the closure of their kernel. These are, similar to addition, the complete sets of items with a transition (*A itemset'*) in their *transitions* field.

These sets of items, which are the first ones affected by the modification of the grammar, have to be re-expanded. This can be achieved simply by making them initial and let the lazy parser generator re-expand them when needed. Because addition and deletion of a rule are so similar, *ADD-RULE* and *DELETE-RULE* use the same routine *MODIFY* to update the graph of item sets.

ADD-RULE: Add a rule to the grammar and update the corresponding graph of item sets.
Input: The rule *rule* that has to be added.
Description: *MODIFY* is called with operator '∪' to perform the actual update.
Algorithm:
ADD-RULE(*rule*):
  MODIFY(*rule*, ∪)

DELETE-RULE: Delete a rule from the grammar and update the corresponding graph of item sets.
Input: The rule *rule* that has to be deleted.
Description: *MODIFY* is called with operator '−' to perform the actual update.
DELETE-RULE(*rule*):
  MODIFY(*rule*, −)
Algorithm:

MODIFY: Modify a grammar and update the corresponding graph of item sets.

Input: A rule $A ::= \beta$ and a modification operator $\square$ (which may be '$\cup$' or '$-$').

Description: *MODIFY* uses global variables *Grammar*, *Itemsets*, and *start-itemset*. *Grammar* is updated according to the modification and the graph of item sets is reduced to a graph that is correct for the modified grammar. This is done by making all incorrectly expanded sets of items in *Itemsets* initial again.

When A is the start-symbol START of the grammar, we know that only *start-itemset* can contain $A ::= \bullet\beta$ in its kernel. When it is not, we search *Itemsets* for all complete sets of items with a transition (*A itemset'*) in their *transitions* field. These sets of items are made initial to let the lazy parser generator re-expand them when needed by the parser.

Algorithm:

MODIFY($A ::= \beta, \square$):
  Grammar := *Grammar* $\square$ $\{A ::= \beta\}$
  if A = START **then**
    start-itemset.kernel := *start-itemset.kernel* $\square$ $\{A ::= \bullet\beta\}$
    start-itemset.type := initial
  else
    for $\forall$*itemset* $\in$ *Itemsets*:
        itemset.type = complete $\wedge$
        (*A itemset'*) $\in$ *itemset.transitions* **do**
      itemset.type := initial
    od
  fi

When, for example, the rule 'B ::= nil' is added to the grammar of the Booleans, and the graph of item sets for the grammar of Fig. 7 is updated by *MODIFY*, the sets of items 0, 4, and 5 are made initial, because they had a transition for 'B' in their *transitions* field. the graph of item sets is thus transformed into the unconnected graph of Fig. 13.

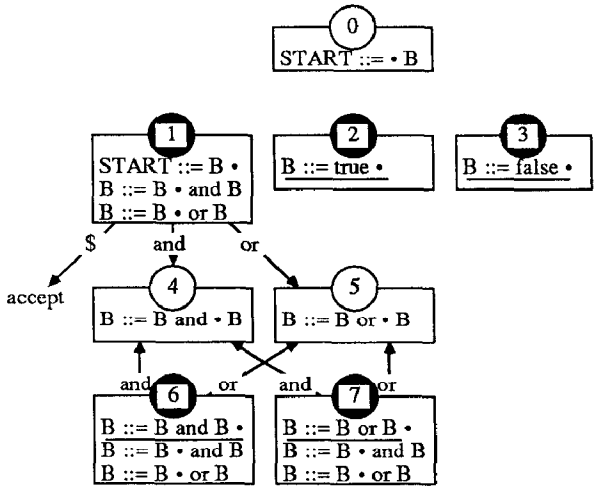

Fig. 13: Graph for the Booleans after addition of 'B ::= nil'.

When the lazy parser generator now expands set 0 again, its former connections with 1, 2, and 3 are re-established, and the initial set of items 8 is generated with kernel 'B ::= nil •'. The resulting graph is shown in Fig. 14.

The example of Fig. 11 is processed correctly by *MODIFY*. When the rule $A ::= b$ is added to the grammar of Fig. 11, set of items 3 is made initial, because it has a transition for A. When set 3 is re-expanded, the transitions to 8 and 9 will be reinstated, but the transition to 7 on b is replaced by a transition to an initial set of items with kernel $\{B ::= b\bullet\ ,\ A ::= b\bullet\}$. Set of items 7 and the transition of

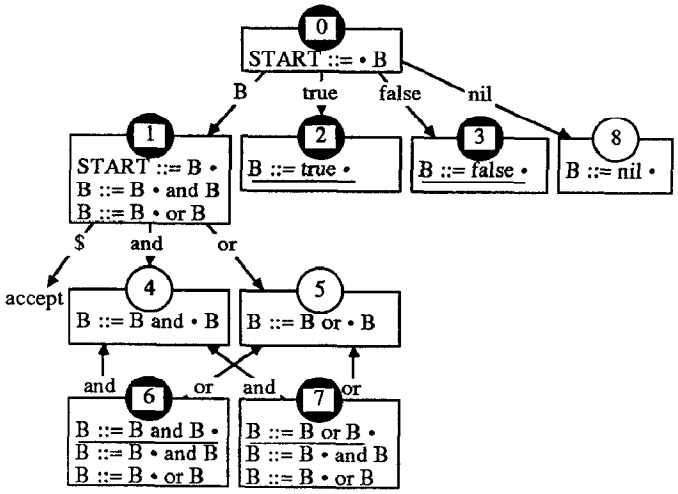

Fig. 14: The graph of Fig. 13 after re-expansion of set 0.

2 to 7 are not affected by this modification.

MODIFY is not the best possible algorithm in the sense that it does not always retain the largest possible part of the graph of item sets. In particular, *partial re-expansion* of sets of items is avoided, because it takes time, while there is no guarantee that the re-expanded sets of items will ever be used again by the parser. This would be in contradiction with the lazy philosophy. Also, in *MODIFY* the functions for generating and correcting a graph of item sets are clearly separated. This makes the algorithm easier to understand and more robust.

6.2. Garbage collection

There is yet another problem to be solved in the incremental parser generator, namely garbage collection. When *MODIFY* makes a set of items initial, the transitions of that set of items to others disappear (because, by definition, initial sets of items do not have a *transitions* field). When the set of items is re-expanded for the modified grammar, new references to these sets may (but need not) be created. On the one hand, retaining unused sets of items in *Itemsets* is essential, otherwise major parts of the graph of item sets would have to be regenerated (this would occur in the example of Fig. 13). On the other hand, there are also sets of items that will almost certainly never be used again. For example, when the rule 'B ::= B xor B' is added to the grammar of the Booleans, sets of items 1, 6, and 7 will never be re-used (unless, of course, the new rule is discarded again). Frequent modification of a grammar can cause many useless sets of items to stay forever in *Itemsets*.

The dilemma is thus: when all unreachable sets of items are removed immediately, it is likely that too much is thrown away, but when everything is retained, we end up with too much garbage in *Itemsets*. A compromise solution is to attach to each set of items a *refcount* field, telling how many sets of items refer to it. Routine *EXPAND* sets and increments the *refount* fields of the sets of items it creates transitions to. Furthermore, *MODIFY* should make sets of items 'dirty' rather than initial. A dirty set of items is an initial set of items with a history (its old *transitions* field). It is expanded in the same way as an initial set. After it has been expanded the *refcount* field will have been decreased of those sets of items to which it no longer refers. When the reference count of a set of items becomes zero, it is removed from *Itemsets*. In other words, the removal of unused sets of items is postponed until the chance is better that they will never be used again.

RE-EXPAND: Expand a dirty set of items
Input: A set of items *itemset* with *type* 'dirty'.
Description: *itemset* is expanded in the same way as an initial set of items, only the reference count of each *itemset'* to which *itemset* referred is decreased by one after the expansion.
Algorithm:

```
RE-EXPAND(itemset):
  old-transitions := itemset.transitions
  EXPAND(itemset)
  for ∀itemset'[(symbol itemset') ∈ old-transitions] do
    DECR-REFCOUNT(itemset')
  od
```

DECR-REFCOUNT:
Input: A set of items *itemset* whose *refcount* field has to be decreased by one.
Description: The reference count of *itemset* is decreased. When it becomes zero, *itemset* is removed from *Itemsets*. All reference counts of the set of items it refers to, have to be decreased as well.
Algorithm:

```
DECR-REFCOUNT(itemset):
  itemset.refcount := itemset.refcount − 1
  if itemset.refcount = 0 then
    Itemsets := Itemsets − {itemset}
    if itemset.type ≠ initial then
      for ∀itemset'[(symbol itemset') ∈ itemset.transitions] do
        DECR-REFCOUNT(itemset')
      od
    fi
  fi
```

The introduction of dirty sets of items and reference counting more or less affects all routines of the parser generator, but as the modifications in the routines are quite small, we will not show them. Our implementation of garbage collection cannot yet handle circular references properly. A straightforward solution for this problem would be to use a conventional mark-and-sweep garbage collector when the percentage of dirty sets of items becomes to high.

7. PERFORMANCE AND EFFICIENCY

We have compared the efficiency of the lazy and incremental parser generator IPG with that of the non-incremental version described in section 4 (which we will call 'PG'). We also compared IPG and PG with the LALR(1) parser generator Yacc [Joh79]. 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*. As we wanted to test all algorithms on the same grammar and input, the test grammar had to be LR(1), since these are the only grammars accepted by Yacc. The test grammar we used is an LR(1) version of the grammar of the syntax definition formalism SDF. 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. The SDF definition of SDF is given in appendix A, to give both an example of an SDF definition and an idea of the size of the test grammar.

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 has 15 lines and the largest 142 lines. The syntax of SDF was modified in each case by adding the grammar rule

```
<CF-ELEM> ::= "(" <CF-ELEM>+ ")?"
(or in SDF: "(" CF-ELEM+ ")?" -> CF-ELEM ),
```

which adds an element in priority and function declarations. 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 under 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' [LL87]. LeLisp garbage collections were only allowed between measurements.

The results of the measurements are given in Fig. 15. They show the following:

- 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 generation 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, while 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 out of the grammar. LR(0) tables prescribe a reduction whenever a rule has been recognized, while LALR(1) tables only do that when the look-ahead is right for the reduction. Tomita's parsing algorithm can work with both, but emits more failing parsers with LR(0) tables as with LALR(1) tables.

* We used a more efficient style of Lisp programming than Tomita did in his book [Tom85], and, after a suggestion of B. Lang, we improved the sharing of parse trees.

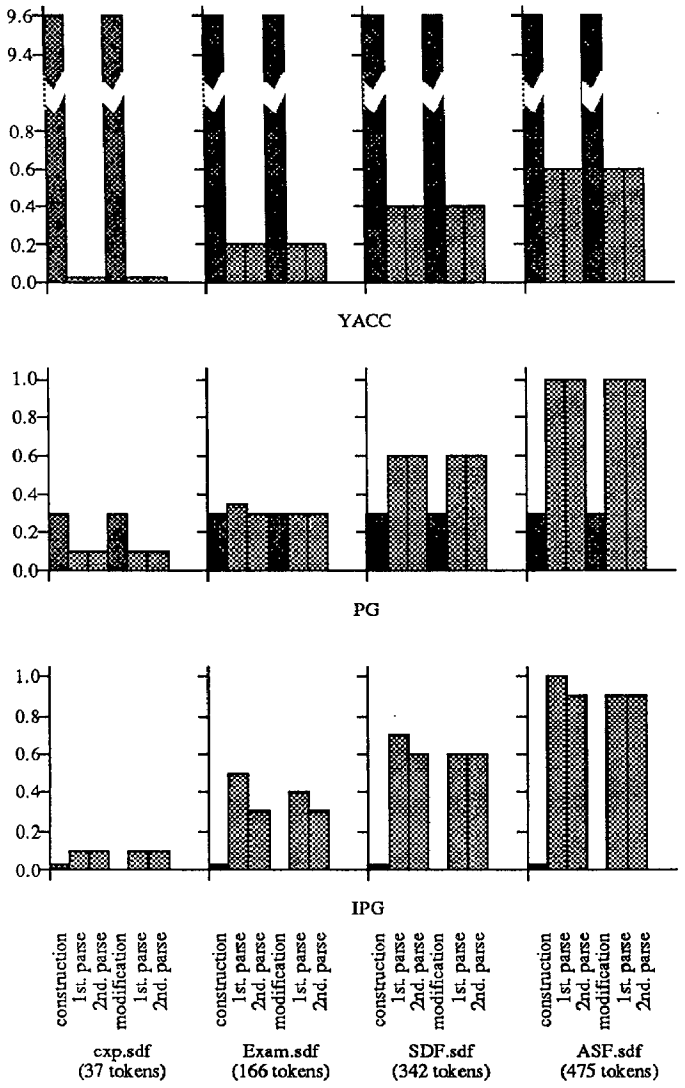

Fig. 15: Efficiency measurements on Yacc, PG and IPG.

- IPG:

  In this case the time needed for constructing the parse table is almost zero. The lazy parser generator generates the needed 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 from within a syntax-directed editor), we consider IPG to be an excellent choice for interactive language definition systems and other highly dynamic applications.

8. CONCLUSIONS AND FUTURE WORK

Although incremental generation of LR parse tables may seem 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 7, 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. However, we need incrementality in order to be able to handle 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.

  An extreme case of simultaneously editing and using syntax definitions occurs when a language can modify its own syntax. In this case, modification and use of the syntax occur in the same textual object to be edited. Limited forms of user-defined syntax appear in various disguises such as operator declarations, macro's and user-defined function notation. Clearly, the modification capability of IPG can be used to implement these changes in syntax.

- Modular composition of parsers.

  IPG does not yet support composition of parsers that are generated for different modules. Although it would be possible to use the incremental modification capability of IPG by adding the grammar of one module to the grammar of the other, this is an asymmetrical operation, which, we believe, is not satisfactory. How IPG can be extended to become a modular parser generator is described in [Rek].

POSTSCRIPT

While we were finishing this paper, R.N. Horspool sent us his recent report on incremental generation of LR parsers [Hor88]. As his overall goals are very similar to ours, we briefly summarize his approach.

Horspool's 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 as look-ahead sets have to be taken into account, whose incremental generation and modification turns 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).

Another relevant report, about lazy generation of LL(1) parsers, is [Kos89].

REFERENCES

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

[BHK89] J.A. Bergstra, J. Heering, and P. Klint (eds.), *Algebraic Specification*, ACM Press Frontier Series, The ACM Press in co-operation with Addison-Wesley (1989).

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

[FGJM85] K. Futatsugi, J.A. Goguen, J.-P. Jouannaud, and J. Meseguer, "Principles of OBJ2," pp. 52-66 in *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, ACM (1985).

[HKR87a] J. Heering, P. Klint, and J. Rekers, "Incremental generation of lexical scanners," Report CS-R8761, Centre for Mathematics and Computer Science, Amsterdam (1987).

[HKR87b] J. Heering, P. Klint, and J. Rekers, "Principles of lazy and incremental program generation," Report CS-R8749, Centre for Mathematics and Computer Science, Amsterdam (1987).

[Hor88] R.N. Horspool, "Incremental generation of LR parsers," Report DCS-79-IR, University of Victoria, Victoria, B.C., Canada (1988).

[Joh79] S.C. Johnson, "YACC: yet another compiler-compiler," in *UNIX Programmer's Manual 2B*, Bell Laboratories (1979).

[Kos89] K. Koskimies, *Lazy recursive descent parsing for modular language implementation, Draft*, GMD Forschungstelle an der Universitat Karlsruhe (1989).

[Lan74] B. Lang, "Deterministic techniques for efficient non-deterministic parsers," pp. 255-269 in *Proceedings of the Second Colloquium on Automata, Languages and Programming*, ed. J. Loeckx, Lecture Notes in Computer Science **14**, Springer-Verlag (1974).

[LL87] *LeLisp, Version 15.21, le manuel de référence*, INRIA, Rocquencourt (1987).

[Rek89] J. Rekers, *A parser generator for finitely ambiguous context-free grammars* (1989), Chapter 8 in [BHK89].

[Rek] J. Rekers, "Modular Parser Generation," Centre for Mathematics and Computer Science, Amsterdam, to appear.

[San82] D. Sandberg, "LITHE: A language combining a flexible syntax and classes," pp. 142-145 in *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, ACM (1982).

[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**, pp. 61-86 (1986).

APPENDIX A. The SDF definition of SDF

SDF is the language in which grammar definitions for IPG are written. SDF stands for 'Syntax Definition Formalism' and is described in [BHK89, ch. 6]. An SDF definition consists of two parts, the lexical syntax and the context-free syntax. For the measurements described in section 7 the lexical syntax part is of no importance, because we did not use the lexical scanner in the measurements.

In the context-free syntax section the non-terminals used are declared first in the 'sorts' declaration part, followed by the declaration of the syntax rules in the 'functions' declaration part. An SDF function $\beta \rightarrow A$ is equivalent to a BNF syntax rule $A ::= \beta$.

```
module SDF   -- The SDF definition of SDF
begin
  lexical syntax
    sorts
      Letter, IdTail, Id, Iterator,
      OrdChar, C-Char, CharRange, CharClass,
      L-Char, Literal, ComChar, ComEnd

    layout
      WhiteSpace, Comment

    functions
      [a-zA-Z]                      -> Letter
      [a-zA-Z0-9\- ]                -> IdTail
      Letter IdTail*                -> Id
      "+"                           -> Iterator
      "*"                           -> Iterator

      [0-9A-Za-z !#$%&'()*+,./:;<=>?@\^ '{|}~]
                                    -> OrdChar
      "\\" ~ []                     -> OrdChar

      OrdChar                       -> C-Char
      "\""                          -> C-Char
      C-Char                        -> CharRange
      C-Char "-" C-Char             -> CharRange
      "[" CharRange* "]"            -> CharClass

      OrdChar                       -> L-Char
      [\-\[\]]                      -> L-Char
      "\"" L-Char* "\""             -> Literal

      [ \t\n\r]                     -> WhiteSpace
      ~ [\n\-]                      -> ComChar
      "-" ~ [\n\-]                  -> ComChar
      "--"                          -> ComEnd
      "-\n"                         -> ComEnd
      "\n"                          -> ComEnd
      "--" ComChar* ComEnd          -> Comment

  context-free syntax
    sorts
      SDF-Definition, LexicalSyntax,
      ContextFreeSyntax, Sorts, Sort, Layout,
      LexicalFunctions, LexicalFunDef, LexElem,
      Priorities, PrioDef, Abbrev-F-List,
      Functions, FunctionDef, CfElem, Lit-or-Id,
      Attributes, Attribute

    functions

      "module" Id
      "begin"
        LexicalSyntax
        ContextFreeSyntax
      "end" Id                      -> SDF-Definition

      "lexical" "syntax"
        Sorts
        Layout
        LexicalFunctions            -> LexicalSyntax
      -- empty --                   -> LexicalSyntax

      "sorts" {Sort ","}+           -> Sorts
      -- empty --                   -> Sorts
      Id                            -> Sort

      "layout" {Sort ","}+          -> Layout
      -- empty --                   -> Layout

      "functions"
        LexicalFunDef+              -> LexicalFunctions

      LexElem+ "->" Sort            -> LexicalFunDef
```

```
    Sort                        -> LexElem
    Sort Iterator               -> LexElem
    Literal                     -> LexElem
    CharClass                   -> LexElem
    "~" CharClass               -> LexElem

    "context-free" "syntax"
      Sorts
      Priorities
      Functions                 -> ContextFreeSyntax

    "priorities"
      {PrioDef ","}+            -> Priorities
    -- empty --                 -> Priorities
    {Abbrev-F-List ">"}+        -> PrioDef
    {Abbrev-F-List "<"}+        -> PrioDef
    FunctionDef                 -> Abbrev-F-List
    "(" {FunctionDef ","}+ ")" -> Abbrev-F-List

    "functions" FunctionDef+ -> Functions

    CfElem* "->" Sort Attributes
                                -> FunctionDef

    Sort                        -> CfElem
    Literal                     -> CfElem
    Sort Iterator               -> CfElem
    "{" Sort Lit-or-Id "}" Iterator
                                -> CfElem
    Literal                     -> Lit-or-Id
    Id                          -> Lit-or-Id

    "{" {Attribute ","}+ "}" -> Attributes
    -- empty --                 -> Attributes
    "bracket"                   -> Attribute
    "assoc"                     -> Attribute
    "left-assoc"                -> Attribute
    "right-assoc"               -> Attribute
    "non-assoc"                 -> Attribute

end SDF
```