1991

J. Heering, P. Klint, J. Rekers

Lazy and incremental program generation
(revised version)

Computer Science/Department of Software Technology Report CS-R9124 April

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

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

Lazy and Incremental Program Generation
(revised version)

J. Heering
CWI
Department of Software Technology
P.O. Box 4079, 1009 AB Amsterdam,
The Netherlands

P. Klint
CWI
Department of Software Technology
P.O. Box 4079, 1009 AB Amsterdam,
The Netherlands
Department of Computer Science, University of Amsterdam

J. Rekers
CWI
Department of Software Technology
P.O. Box 4079, 1009 AB Amsterdam,
The Netherlands

Current program generators usually operate in a *greedy* manner in the sense that a program must be generated in its entirety before it can be used. If generation time is scarce, or if the input to the generator is subject to modification, it may be better to be more cautious and to generate only those parts of the program that are indispensable for processing the particular data at hand. We call this *lazy program generation*. Another, closely related, strategy is *incremental program generation*. When its input is modified, an incremental generator will try to make a corresponding modification in its output rather than generate a completely new program. It may be advantageous to use a combination of both strategies in program generators that have to operate in a highly dynamic and/or interactive environment.

Key Words & Phrases: program generator, greedy, lazy, and incremental program generation, lazy and incremental generation of lexical scanners, lazy and incremental generation of parsers, lazy and incremental compilation.

1991 CR Categories: D.1.2 **[Programming techniques]:** Automatic programming; D.3.4 **[Programming languages]:** Processors - *Compilers, Parsing, Translator writing systems and compiler generators.*

1985 Mathematics Subject Classification: 68N20 **[Software]:** Compilers and generators.

Note: Partial support received from the European Communities under ESPRIT projects 348 (Generation of Interactive Programming Environments - GIPE) and 2177 (GIPE II), and from the Netherlands Organization for Scientific Research (NWO) under the *Incremental Program Generators* project.

Note: This is an extensively revised version of J. Heering, P. Klint, and J. Rekers, Principles of Lazy and Incremental Program Generation, Report CS-R8749, Centre for Mathematics and Computer Science, Amsterdam, November 1987. In comparison with the previous version, a much less formal approach has been adopted and an in-depth discussion of several existing lazy and lazy/incremental program generators has been added.

1. INTRODUCTION

1.1. Greedy, lazy, and incremental program generation

"Automatic programming" necessarily means production of programs by means of other programs. The latter are usually called *program generators*. The use of program generators dates back almost to the beginning of the programmable electronic computer [13]. Compilers for high-level programming languages are the most successful and widely used program generators to date, and the well-known arguments for using them apply to other program generators as well. The fact that many program generators are

tailored towards a specific and rather limited application area does not detract from these arguments.

Usually, the time spent by a program generator is not that important as long as the program produced by it is efficient. If there is ample time available for the generator, the complete program can be generated before it is used. We call this *greedy* program generation (Section 2.1). There are basically three reasons why greedy program generation is not always an efficient or feasible strategy:

(A) The generated program is used so little that the time invested in generating it is largely lost.

(B) The real-time requirements (interactive response time requirements) that have to be met by the environment in which the generator runs are too stringent for the generator to finish its task completely.

(C) The generated program is too large to be run in its entirety.

So, rather than generating a program all at once, it may be better to generate only those parts of it that are indispensable for processing the particular data at hand. We call this *lazy* program generation (Section 2.2). Another, closely related, strategy is *incremental* program generation. When its input is modified, an incremental generator will try to make a corresponding modification in its output rather than generate a completely new program. The best known examples of incremental program generators are incremental compilers. It may be advantageous to use a combination of both strategies in program generators that have to operate in a highly dynamic and/or interactive environment. This is our main topic (Section 2.3).

We first came across the above-mentioned weaknesses of greedy program generation while designing a syntax-oriented editor for the ASF+SDF language definition formalism, which has very general user-definable syntax [9]. We realized that generating a completely new parser after each syntax change would lead to unacceptable response times. To solve this problem, we converted greedy scanner and parser generators to versions that are both lazy and incremental. These are discussed in Section 3.

To avoid misunderstanding, it should be emphasized that we have not *mechanized* the process of converting greedy program generators to lazy/incremental ones. The above-mentioned conversions were done by hand. Gordon [8] has raised the question whether lazy functional languages like Miranda, Ponder, or LML [4] can be used to obtain lazy program generation more or less automatically as a special case of lazy evaluation. In the same vein, one might ask whether an incremental language like INC [24] can be used to obtain the incremental behavior for free. These questions remain to be investigated.

1.2. Related work

In spite of their indisputable importance, very little has been written on the general principles underlying program generators. An interesting paper with a practical flavor on the construction of application generators is the one by Cleaveland [3]. The literature on *partial evaluation* [20] contains some relevant theoretical considerations of a general nature. In [5] Ershov clearly explains the basic ideas involved. Although these have influenced our discussion of greedy program generation in Section 2.1 to some degree, our viewpoint in this paper is not that of partial evaluation, and we have not attempted to interpret our work in that particular context.

In the way of lazy and incremental program generation we mention Brown's lazy BASIC compiler for small machines [2], the lazy LL(1) parser generator developed by Koskimies [17] (both of which are discussed in Section 2.2), Szafron and Ng's interactive incremental scanner generator LexAGen [21], Horspool's incremental LALR(1) parser generator ILALR [14] (these are discussed in Section 3), and Fritzson's incremental PASCAL compiler [6, 7]. To the best of our knowledge, program generators that are both lazy and incremental have not been discussed before, except in our papers [10] and [11] (cf. Section 3).

2. GREEDY, LAZY, AND LAZY/INCREMENTAL PROGRAM GENERATION

2.1. The traditional (greedy) case

Suppose a program with two arguments

$$P: A \times B \rightarrow C$$

is used in a context in which it has to be applied to a relatively large number of $b \in B$ for each $a \in A$. In such

cases it is often possible to gain efficiency by replacing P with a higher-order program (''program generator'') of curried type

$$G: A \rightarrow (B \rightarrow C)$$

which, when given a particular $a \in A$, yields (''generates'') a specialized program

$$G_a: B \rightarrow C$$

such that for all $b \in B$ execution of G_a with argument b yields the same result as execution of P with arguments a and b, and with the additional property that G_a is a much more efficient program than P with first argument a. If, for instance, the former is more efficient than the latter by a multiplicative speed-up factor $S>1$, the investment of time T_a in generating G_a will start paying off after G_a has run for a total time t defined by

$$t = T_a + \frac{t}{S},$$

leading to a break-even point

$$t = (1 + \frac{1}{S-1})T_a.$$

Within certain limits, the larger the investment T_a made in generating G_a, the larger the speed-up S that can be achieved. In practice, a speed-up S of 10 or more can often be achieved at acceptable cost.

For the sake of concreteness, it may be instructive to consider a parser generator from the general viewpoint. In that case, P would be a general parser, A would be a domain of grammars (probably described in BNF), B would consist of sentences to be parsed, and C would contain parse trees and some failure value. The associated parser generator would be G, and G_a would be a parser for a specific grammar a.

Compilation of a programming language L is another instructive example. In this case, P would be an interpreter for L-programs, A the collection of L-programs, B a domain of input values, and C a domain of result values. The compiler for L would be G, and G_a would be the object code for a.

2.2. Lazy program generation

As indicated in Section 1.1, investing time T_a in generating G_a is not always possible or justified. In such cases, lazy program generation may offer a solution. Rather than generating G_a all at once, a lazy generator produces for each b only those parts of G_a that are actually needed to compute the required result. Except if G_a is too large to be run in its entirety (Section 1.1, case (C)), parts generated for previous inputs b (if any) are retained indefinitely, so the lazy generation process is cumulative. Whether the complete program G_a is ever generated in this way, depends on the particular sequence of inputs involved. Parts of G_a that are not needed by any b are never generated.

In view of the foregoing, a rough outline of the lazy counterpart L_a of G_a (expressed in some suitable language) is:

```
L_a(b): B→C
  constant a: A
  static g: B ⇀ C with initial value g_0
begin
  return g(b)
  when attempting to execute gap γ in g
  do
    g := EXPAND(a,g,γ)
    resume
  od
end.
```

For each new $b=b_n$, L_a initially tries to compute the required result by means of the incomplete program g_{n-1} generated during the previous activations of L_a, that is, by means of the value of the static function

variable $g: B \rightharpoonup C$, where $\rightharpoonup$ indicates a partial function. The fact that g is static means that its value is retained between different activations of L_a. For n=1 the value of g is its initial value g_0, which consists of nothing but a single gap and is undefined everywhere. Only if execution of g_{n-1} with argument b_n hits a gap γ in g_{n-1}, L_a generates an additional piece of program by calling procedure *EXPAND* in the body of the exception handler. This procedure, which is a suitably adapted version of the greedy generator G of the previous section, produces the required extension in some unspecified, application dependent way using a, the incomplete program g_{n-1} generated so far, and the gap descriptor γ (which identifies the gap in question). It fills the gap only to the extent necessary, so part of the gap in the form of one or more new gaps may remain. Computation is then resumed at the point where the exception occurred using the extended version of g_{n-1}. The computation may hit several gaps in succession, so the extension of g_{n-1} to g_n may require several activations of *EXPAND*. If no gap in g_{n-1} is encountered, no extension is necessary and g_n=g_{n-1}. Hence, *EXPAND* is not called and L_a runs as fast as G_a. An extreme and, from the viewpoint of lazy program generation, undesirable case is g_1=G_a. L_a has to generate the whole program G_a just to handle b_1. Obviously, the lazy character of L_a is lost in this case. Lazy program generation may alleviate the problems mentioned in Section 1.1 if *EXPAND* has to generate only relatively small extensions at each step and if the total generation time is distributed more or less evenly.

In the previous section we mentioned parser generation and compilation as examples of greedy program generation. Both can be done in a lazy manner. Koskimies [17] has developed a lazy parser generator for modular LL(1) grammars. Each module is supposed to contain the definition of a single nonterminal symbol of the grammar. It would be nice if the parser for the complete grammar could be obtained by linking parsing procedures generated separately for each individual module. For an ordinary recursive descent parser this is impossible, however, since each parsing procedure depends on the set of first symbols of the corresponding nonterminal. In general, computation of this set requires access to the definition of other nonterminals and hence to other modules. Koskimies circumvents this problem to some extent by generating a hybrid recursive descent/table-driven parser consisting of separately generated procedures incorporating a rule selection mechanism driven by so-called *start trees*. These are built by need during parsing to minimize the initial generation delay. In this case, incomplete programs g_n consist of a fixed recursive descent part and a possibly incomplete set of start trees. A gap γ is a nonterminal whose start tree has not yet been computed. If such a nonterminal is encountered during parsing, the corresponding start tree is computed by the function StartTree_γ, which is generated separately for each module. So instead of a single *EXPAND* function that handles all gaps, each gap has its own specialized version.

A lazy BASIC compiler for small machines was developed by Brown [2]. Statements are compiled by need when encountered during execution. Each compiled statement is placed in the workspace immediately after the previously compiled statement. If the workspace is full, all object code accumulated in it is thrown away. This radical strategy eliminates the problem of dangling jumps to object code that no longer exists, except for stacked return addresses. To prevent these from doing harm, they must refer to source code rather than object code. Returns are therefore effected by indirect transfers through the lazy compiler. Thus, in this case an incomplete program g_n is a series of compiled statements possibly with embedded gaps containing references to source code. If such a reference is encountered during execution, the corresponding statement is compiled and placed in the workspace. The reference that triggered the compilation is replaced with a direct jump to the compiled statement or even with the compiled statement itself if the reference was the last item of the object code. It may turn out that the statement had already been compiled. In that case, the reference is merely replaced by a direct jump to the compiled statement. Obviously, unreachable parts of the program will never be compiled.

2.3. The combination of lazy and incremental program generation

In the previous section a was kept constant. Now suppose that a is subject to modification, perhaps because it is being developed and experimented with interactively. Ordinarily, a completely new program would have to be generated for each new version of a. If modifications follow each other in quick succession, chances are that only a small part of each a is used before it is modified. This fact may be exploited by a lazy program generator. The program generated for the old version of a is still thrown away, but, as it will be incomplete most of the time, less time is wasted than before. As explained in the previous section, this is the strategy used in Brown's lazy BASIC compiler, albeit for a different reason.

Although lazy generation may certainly offer a partial solution, the above scheme is still rather crude in that it does not attempt to retain the largest possible part of the old program. This part can be characterized in terms of the greatest lower bound of two incomplete programs with respect to the subsumption order, which is the natural partial order on incomplete programs. More specifically, an incomplete program g subsumes an incomplete program h ($g \leq h$) if h can be obtained from g by partially or completely expanding the gaps in g. Except if G_a is too large to be run in its entirety, the lazy generator L_a of the previous section produces a sequence of incomplete programs $\{g_n\}_{n\geq 0}$ such that

$$g_0 \leq g_1 \leq g_2 \leq \cdots \leq G_a.$$

The greatest lower bound $g \wedge h$ of two incomplete programs g and h with respect to $\leq$ has the usual properties, namely,

$g \wedge h \leq g$,
$g \wedge h \leq h$, and
$f \leq g \wedge h$ for all f such that $f \leq g$ and $f \leq h$.

It is the most specific (least general) incomplete program that can be expanded to both g and h. In the worst case it is equal to the program that consists of nothing but a single gap.

Actually, the greatest lower bound is maximal only in a relative sense. It depends on the domain of incomplete programs in which it is interpreted. From the viewpoint of abstract syntax the simplest incomplete programs are Ω-terms, in which the special constant Ω acts as a gap. For instance,

$$program(if(\Omega,\Omega,\Omega)) \leq program(if(eq(x,0),assign(y,\Omega),assign(\Omega,\Omega))).$$

and

$$program(if(eq(x,0),assign(y,\Omega),assign(\Omega,\Omega))) \wedge program(if(lt(x,0),assign(y,1),\Omega)) = program(if(\Omega,assign(y,\Omega),\Omega).$$

A better greatest lower bound is obtained if incomplete programs are generalized Ω-terms containing n-adic gaps for any $n \geq 0$ rather than conventional Ω-terms containing only zero-adic gaps. For instance, in that case

$$program(if(eq(x,0),\Omega)) \wedge program(while(lt(x,0),\Omega))$$

would be equal to

$$program(\Omega(\Omega(x,0),\Omega))$$

rather than to

$$program(\Omega).$$

Now, suppose a is changed to a' after the lazy generator L_a has processed its kth input. The largest part of the incomplete program g_k generated so far that can be retained in the context of a' is

$$g' = g_k \wedge G_{a'}.$$

Obviously, computing $G_{a'}$ is contrary to the rationale of the lazy generation strategy, so the above characterization of the largest part g' of g_k that can be retained in the context of a' is useless from a computational viewpoint. Fortunately, in many concrete cases a reasonable approximation to g' can be computed efficiently on the basis of the incomplete program g_k generated so far, the modification Δ to be made to a, and a itself, without computing $G_{a'}$. Whether this is feasible has to be investigated separately in each specific case. This is crucial to the success of the proposed lazy/incremental strategy.

A rough outline of the lazy/incremental counterpart I_a of L_a is

```
I_a(Δ,b): (A→A)×B→C
  static α: A with initial value a
  static g: B ⇀ C with initial value g_0
begin
  if Δ≠id_A then α,g := MODIFY(α,Δ,g) fi
  return g(b)
  when attempting to execute gap γ in g
  do
    g := EXPAND(α,g,γ)
    resume
  od
end.
```

In I_a, a is no longer constant, but the initial value of static variable α which is subject to modifications Δ of type $A \rightarrow A$. *MODIFY* computes (a suitable approximation to) the largest part of the old value of g that remains valid in the modified context in terms of the old value of g itself, the modification Δ, and the old value of α (see above). It also updates the value of α by applying Δ to it.

3. LAZY/INCREMENTAL SCANNER AND PARSER GENERATION

In this section we discuss the lazy/incremental lexical scanner and parser generators ISG and IPG. As mentioned in Section 1.1, we use the combination ISG/IPG in a syntax-oriented editor for the ASF+SDF language definition formalism, which has very general user-definable syntax.

3.1. ISG - a fully lazy/incremental lexical scanner generator

ISG is a fully lazy/incremental lexical scanner generator. In this case, A is the domain of regular grammars, B contains the sentences to be scanned, and C consists of legal strings with their lexical type(s) and a failure value. For each regular grammar there is a deterministic finite automaton (DFA) recognizing the language generated by the grammar. ISG constructs this automaton by need, so the incomplete programs g produced by ISG are partial DFAs (PDFAs), which may be viewed as approximations to the complete automaton for the input grammar. Modifications Δ are additions and deletions of a single regular expression.

We now summarize the operation of ISG in relation to the general scheme outlined in sections 2.2 and 2.3. More details can be found in [10]. Let a be a regular grammar with alphabet Σ. A PDFA g for a consists of a set of states and a binary transition function mapping state-symbol pairs to states. A state is a set of positions p in a indicating to which points in the grammar the scanning process has progressed. The start state is the set of initial positions. A gap γ in g is a state whose transitions have not yet been computed. Whereas a greedy scanner generator can throw away the positions making up a state after the full DFA has been constructed, ISG has to retain the structure of states for the purpose of further expansion and incremental modification of the PDFA.

If the PDFA g generated so far for a hits a gap γ while scanning its input string, g is expanded by an instance of *EXPAND* (Section 2.2) which, apart from error handling, looks as follows:

```
EXPAND(a,g,γ)
begin
  assertion expanded(γ)=false
  for all s ∈ Σ such that symbol(p)=s for some p ∈ γ
  do
     δ := ∪{p ∈ γ| symbol(p)=s} followpos(p,a)
     if δ ∉ g.States
        then add δ to g.States; expanded(δ) := false
     fi
     g.Transition(γ,s) := δ
  od
  expanded(γ) := true
  return g
end.
```

EXPAND computes all legal transitions $\gamma \xrightarrow{s} \delta$ of γ, where s is a symbol and δ the corresponding successor state of γ. For any s there is at most one such state since we are dealing with deterministic automata. It consists of the positions that may follow the positions p in γ at which the symbol s occurs (if any). These are computed by *followpos*. Accepting states need not have any legal transitions.

It might seem as if *EXPAND* could be made even lazier by adding only the legal transition for the current symbol rather than all legal transitions. In that case, gaps in the automaton would correspond to unexpanded transitions rather than to unexpanded states. Unfortunately, this approach would require the introduction of error transitions or something equivalent, which is not very attractive.

ISG uses the following instance of *MODIFY* (Section 2.3):

```
MODIFY(a,Δ,g)
begin
  assertion Δ≠id_A
  anew := Δ(a)
  σ := firstpos(anew)
  gtmp.Start := σ
  gtmp.States := g.States
  if σ ∉ gtmp.States
    then add σ to gtmp.States; expanded(σ) := false
  fi
  gtmp.Transition := g.Transition
  if expanded(σ)=false
    then gtmp := EXPAND(anew,gtmp,σ)
  fi
  gnew.Start := σ
  gnew.States := {δ ∈ gtmp.States | δ=gtmp.Transition( ... gtmp.Transition(σ,s_1), ... ,s_k)}
  gnew.Transition := gtmp.Transition |gnew.States
 return anew,gnew
end.
```

After applying Δ to a, *MODIFY* computes the new start state σ, which consists of the new set of initial positions, by means of *firstpos* and constructs an intermediate partial automaton *gtmp* by adding σ and its legal transitions to the old PDFA g. Since *gtmp* may be partially obsolete, *MODIFY* performs a garbage collection on it by retaining only states δ that are reachable from σ by $k \geq 0$ applications of the transition function *gtmp.Transition*, and by restricting the transition function to these states. This yields the new PDFA *gnew*.

Rather than the greatest lower bound in the sense of Section 2.3 (which is empty since the new automaton always has a different start state), *MODIFY* computes

$$EXPAND(\Delta(a), \{\sigma\} \cup g, \sigma) \wedge G_{\Delta(a)},$$

where $G_{\Delta(a)}$ is the complete DFA for the regular grammar $\Delta(a)$. Actually, it may compute somewhat less since all states of the old automaton that are not reachable from the new start state are removed. Such states may become reachable after further expansion, however, in which case they could have been retained. This can be achieved at the expense of an increase in space complexity by postponing garbage collection and allowing further lazy expansion of $\{\sigma\} \cup g$ in the context of $\Delta(a)$. This is the approach taken in the lazy/incremental parser generator IPG discussed in the next section. The ISG approach has the advantage of simplicity. The number of states retained after a modification is generally quite close to the theoretical maximum.

ISG has been implemented in LISP. In so far as a meaningful comparison can be made, the *total* generation time used by ISG is typically 2.5 times less than that used by the greedy lexical scanner generator FLEX [18], while the lexical scanners produced by it (in LISP) are typically 5 times slower than those generated by FLEX (in C). The total generation time includes compilation of the generated scanners in both cases. ISG uses negligible time updating the generated scanner after a modification to the regular grammar.

Szafron and Ng's interactive incremental scanner generator LexAGen [21] is an interactive development environment for lexical grammars. It maintains a DFA incrementally on the basis of a regular grammar entered by the user. The DFA can be tested interactively after each change to the grammar or it can be compiled non-incrementally to high-quality C-code with approximately the same performance as that generated by FLEX. The test mode of LexAGen is maintained incrementally, but no performance figures for it are given in [21], so it cannot be compared to ISG.

ISG is used in conjunction with the lazy/incremental parser generator IPG described in the next section.

3.2. IPG - a fully lazy/incremental context-free parser generator

Taking Tomita's general context-free parsing algorithm [22] as our point of departure, we developed the fully lazy/incremental parser generator IPG. In this case, A is the domain of context-free grammars, B contains the sentences to be parsed, and C consists of sets of parse trees and a failure value. Modifications Δ are additions and deletions of a single production rule. The programs g generated by IPG are incomplete LR(0) parse tables which are constructed by need. These may be viewed as incomplete programs for Tomita's general bottom-up parsing algorithm. Since the grammars involved need not be LR(0), the corresponding LR(0) parse tables need not be deterministic, but may contain shift-reduce and reduce-reduce conflicts. Tomita's algorithm interprets non-deterministic table entries by starting as many LR(0) parsers in parallel as required. To keep the number of different LR(0) parsers to a minimum, they are joined at the earliest possible moment.

We now summarize the operation of IPG in relation to the general scheme outlined in sections 2.2 and 2.3. A more detailed description can be found in [11]. We assume the reader to be familiar with conventional LR parsing [1]. In IPG, the LR(0) parse table figures as a set of parse states, a binary transition function, and a unary reduction function. States are sets of dotted rules. A dotted rule $n ::= u.v$ is a BNF rule whose right-hand side contains a dot indicating to which point in the rule parsing has progressed. A state consists of all dotted rules that may be valid simultaneously at some time during parsing and in which some progress has already been made. The latter condition, which does not apply to the start state, means that the right-hand sides of the dotted rules involved do not start with a dot. Gaps γ are states whose transitions and reductions have not yet been computed. Unlike greedy parser generators, IPG cannot throw away the dotted rules making up states, but has to retain them for the purpose of further expansion and incremental modification of the parse table.

The transition function consists of $\gamma \xrightarrow{s} \delta$ triples, where γ and δ are states, and s is either a terminal or a nonterminal symbol of the grammar. If it is a terminal, the transition corresponds to a *shift* action, otherwise it corresponds to a *goto*.

When a gap γ in the incomplete parse table g is encountered during parsing, IPG calls the following version of *EXPAND*:

```
EXPAND(a,g,γ)
begin
  assertion expanded(γ)=false ∨ expanded(γ)=obsolete
  -------
  if expanded(γ)=obsolete
    then for all γ -n->δ ∈ g.Transition
         do
           delete γ -n->δ from g.Transition
           decrease referencecount(δ)
         od
  fi
  -------
  Γ := closure(γ,a)
  for all s such that n::=u.sv ∈ Γ
  do
     δ := { n::=us.v | n::=u.sv ∈ Γ }
     if δ ∉ g.States
       then add δ to g.States; expanded(δ) := false; referencecount(δ) := 0
     fi
     add γ -s->δ to g.Transition
     increase referencecount(δ)
  od
  for all n::=u. ∈ Γ
  do
     if n ≠ start
     then add n::=u to g.Reduction(γ)
     else add γ -$->accept to g.Transition
     fi
  od
  -------
  if expanded(γ)=obsolete
    then while referencecount(δ)=0 for some δ ∈ g.States
         do assertion δ≠γ
           for all δ -n->δ' ∈ g.Transition
           do
             delete δ -n->δ' from g.Transition
             decrease referencecount(δ')
           od
           delete δ from g.States
         od
  fi
  -------
  expanded(γ) := true
  return g
end.
```

For the moment we assume that the incomplete state γ is not *obsolete*, so we skip the first and last if-statement of *EXPAND*. The remaining part is quite similar to ISG's version of *EXPAND* (Section 3.1). It first uses *closure* to enrich γ with all dotted rules in which no progress has yet been made, but which may become applicable, and assigns the result to Γ. Using Γ, it then computes all transitions and reductions for γ.

Obsolete states γ are produced by *MODIFY*. The way they are treated by *EXPAND* becomes easier to understand if *MODIFY* is discussed first:

```
MODIFY(a,Δ,g)
begin
  assertion Δ ≠ id_A
  anew := Δ(a)
  n := nonterminal(Δ)
  for all δ -n-> δ' ∈ g.Transition
  do
     expanded(δ) := obsolete
  od
  return anew,g
end.
```

After applying Δ to a, *MODIFY* extracts the nonterminal on the left-hand side of the syntax rule involved in the modification and assigns it to n. It then sets all states δ that have one or more transitions for n to *obsolete*. This means they are treated as gaps, but their old transitions are retained until they are re-expanded (if ever—see below). To increase the chance that parts of the graph of states that are no longer reachable from the start state are reused, *MODIFY* does not immediately perform garbage collection like ISG, but allows lazy expansion of the partially obsolete graph *gnew* and leaves garbage collection to *EXPAND*. We therefore resume our discussion of *EXPAND*.

When given an obsolete state γ, *EXPAND* first deletes its old transitions. As a result, some of the states to which γ had transitions may end up having reference count 0, that is, completely disconnected from the start state. Rather than immediately deleting these, *EXPAND* first expands γ in the way explained earlier. As a consequence, some of the disconnected states may become connected again. Only then does *EXPAND* remove any remaining disconnected states. A basic shortcoming of the reference counting method is that direct or indirect self-references may lead to garbage that is never collected. On the other hand, there is still no guarantee that states that are deleted might not have become connected after further expansion, so the moment at which garbage collection is done might have been postponed still further.

Like ISG, IPG has been implemented in LISP. The parsers produced by it are typically 2 times slower than those generated by the LALR(1) parser generator Yacc [1, 15]. Of course, IPG is not limited to LALR(1) grammars, but can handle all context-free grammars. Its *initial* generation time is very small and its *total* generation time is typically 30 times smaller than that of Yacc. This large factor is primarily due to the fact that IPG generates a parser in the same LISP work space in which it runs. A secondary reason is that the LR(0) tables generated by IPG require less effort than the LALR(1) tables produced by Yacc. IPG uses negligible time updating the generated parser after a modification to the corresponding context-free grammar.

Horspool has developed an incremental LALR(1) parser generator ILALR [14]. As is to be expected, it has a less efficient generation phase than IPG, but generates better parsers for LALR(1) grammars that are not LR(0). The incremental maintenance of LALR(1) parse tables turns out to be problematic. Adding a syntax rule to the grammar does not present new problems in comparison with the LR(0) case, but deleting a rule leads to a complete recomputation of the LALR(1) look-ahead sets.

4. FURTHER WORK

Both ISG (Section 3.1) and IPG (Section 3.2) have been extended with a subgrammar selection feature, which allows lazy restriction of the incomplete scanner or parser generated so far to a subgrammar corresponding to one of the modules making up a modular regular or context-free grammar [16, 19]. If a gap is encountered in subgrammar mode, expansion takes place with respect to the full grammar. Restriction of new parts to the currently selected subgrammar is done by need as well. Although expansion with respect to the currently selected subgrammar would almost always be faster and would be in better agreement with the lazy approach, expansion with respect to the full grammar facilitates selection of a different subgrammar and requires less modification of ISG and IPG.

Walters [23] has tailored ISG towards the efficient implementation of term rewriting systems that are subject to frequent modification. The first phase of term rewriting consists of term matching. In [12] Hoffmann and O'Donnell give a top-down matching algorithm that reduces term matching to string matching.

Basically, the set of tree patterns (left-hand sides of rewrite rules) is transformed into a finite automaton similar to the one produced by ISG. Walters first derives a set of regular expressions from the set of tree patterns and then uses his extended version of ISG to produce the automaton. In this way, term rewriting systems that are subject to modification can be handled smoothly and efficiently. No restrictions are imposed on the set of tree patterns.

As pointed out in Section 1, lazy/incremental program generation remains to be investigated from the perspective of lazy functional languages, incremental languages, and partial evaluation.

We have been unable to ascertain whether a lazy/incremental compiler has been implemented for some language, but something very close to it is bound to exist somewhere.

5. CONCLUSIONS

Our experience with ISG and IPG has taught us that lazy/incremental program generation is an implementation technique that merits serious consideration in highly dynamic applications in which both program generation time and program execution time are scarce. Whether it can actually be applied depends on several factors, which have to be investigated separately in each particular case. Obviously, for lazy program generation to make sense, most expansion steps should be relatively small so that the total generation time can be distributed more or less evenly over many computations. In addition to this, lazy/incremental program generation requires an efficient way of establishing which part of the already generated program remains valid in a modified context.

REFERENCES

1. Aho, A.V., Sethi, R., and Ullman, J.D. *Compilers—Principles, Techniques, and Tools.* Addison-Wesley, 1986.
2. Brown, P.J. Throw-away compiling. *Software—Practice and Experience 6* (1976), 423-434.
3. Cleaveland, J.C. Building application generators. *IEEE Software* (July 1988), 25-33.
4. Special issue on lazy functional languages. *The Computer Journal 32*, 2 (April 1989).
5. Ershov, A.P. On the partial computation principle. *Information Processing Letters 6*, 2 (April 1977), 38-41.
6. Fritzson, P. Fine-grained incremental compilation for Pascal-like languages. Report LiTH-MAT-R-82-15, Software Systems Research Center, Linköping University, 1982.
7. Fritzson, P. Symbolic debugging through incremental compilation in an integrated environment. *The Journal of Systems and Software 3* (1983), 285-294.
8. Gordon, M.J.C. Personal communication. March 1988.
9. Heering, J., Hendriks, P.R.H., Klint, P., and Rekers, J. The syntax definition formalism SDF—reference manual. *SIGPLAN Notices 24*, 11 (November 1989), 43-75.
10. Heering, J., Klint, P., and Rekers, J. Incremental generation of lexical scanners. Report CS-R8761, Department of Software Technology, Centre for Mathematics and Computer Science, Amsterdam, 1987.
11. Heering, J., Klint, P., and Rekers, J. Incremental generation of parsers. *IEEE Transactions on Software Engineering SE-16*, 12 (December 1990), 1344-1351. Also in *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation, SIGPLAN Notices 24*, 7 (July 1989), 179-191.
12. Hoffmann, C.M. and O'Donnell, M.J. Pattern matching in trees. *Journal of the ACM 29* (1982), 68-95.
13. Hopper, G.M. Keynote address to the ACM History of Programming Languages Conference. In Wexelblat, R.L., Ed. *History of Programming Languages.* Academic Press, 1981, 7-20.
14. Horspool, R.N. ILALR: an incremental generator of LALR(1) parsers. In Hammer, D., Ed. *Compiler Compilers and High Speed Compilation.* Lecture Notes in Computer Science, Vol. 371, Springer-Verlag, 1989, 128-136.

15. Johnson, S.C. Yacc—yet another compiler-compiler. Computing Science Technical Report No. 32, Bell Laboratories, Murray Hill, N.J., 1975.

16. Klint, P. Scanner generation for modular regular grammars. Report, Department of Software Technology, Centre for Mathematics and Computer Science, Amsterdam, The Netherlands, to appear.

17. Koskimies, K. Lazy recursive descent parsing for modular language implementation. *Software—Practice and Experience 20* (1990), 749-772.

18. Paxson, V. FLEX. Public domain software developed at Lawrence Berkeley Laboratory (1989).

19. Rekers, J. Modular parser generation. Report CS-R8933, Department of Software Technology, Centre for Mathematics and Computer Science, Amsterdam, The Netherlands, 1989.

20. Sestoft, P., and Zamulin, A.V. Annotated bibliography on partial evaluation and mixed computation. *New Generation Computing 6* (1988), 309-354.

21. Szafron, D., and Ng, R. LexAGen: an interactive incremental scanner generator. *Software—Practice and Experience 20* (1990), 459-483.

22. Tomita, M. *Efficient Parsing for Natural Languages*. Kluwer, 1985.

23. Walters, H.R. *On Equal Terms*. PhD Thesis, University of Amsterdam, 1991, Chapter 3.

24. Yellin, D., and Strom, R. INC: a language for incremental computations. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation, SIGPLAN Notices 2*, 7 (July 1988), 115-124.