Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

J. Heering, P. Klint, J. Rekers

Principles of lazy and incremental program generation

Computer Science/Department of Software Technology Report CS-R8749 November

69D12, 69D44

Principles of lazy and incremental program generation

J. Heering
Department of Software Technology, Centre for Mathematics and Computer Science

P. Klint
Department of Software Technology, Centre for Mathematics and Computer Science
Department of Computer Science, University of Amsterdam

J. Rekers
Department of Software Technology, Centre for Mathematics and Computer Science

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, fourth generation language, greedy, lazy, and incremental program generation, lazy and incremental generation of lexical scanners, lazy and incremental generation of parsers, lazy and incremental compilation, dynamic compilation.

1987 CR Categories: D.1.2 [**Programming Techniques**]: Automatic programming; D.3.4 [**Programming Languages**]: Processors.

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

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

Note: This paper will be submitted for publication elsewhere.

1. INTRODUCTION

''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. In her keynote address to the ACM History of Programming Languages Conference in June 1978 Grace Murray Hopper recalled how she became aware of the possibility of ''program writing programs'': ''*I think the first step to tell us that we could actually use a computer to write programs was Betty Holberton's 'Sort-Merge Generator.' You fed it the specifications of the files you were operating on, and the Sort-Merge Generator produced the program to do the sorting and merging, including not only carrying out the operations, but also managing all of the input and output in the various tape units, and it contained, I think, what I would define as the first version of a virtual memory in that it made use of overlays automatically without being told to by the programmer. I think that meant a great deal to me. It meant that I could do these things automatically; that you could make a computer write a program. Of course, at that time [around 1952] the Establishment promptly told us—at least they told me quite frequently—that a computer could not write a program; it was totally impossible; that all that computers could do was arithmetic, and that it couldn't write programs; that it had none of the imagination*

Report CS-R8749
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

and dexterity of a human being. I kept trying to explain that we were wrapping up the human being's dexterity in the program that he wrote, the generator, and that of course we could make a computer do these things so long as they were completely defined'' [6].

Compilers for high-level programming languages are the most successful and widely used program generators to date. The arguments for using program generators and the high-level specification languages associated with them (sometimes called ''fourth generation languages'') are exactly the same as those for using high-level programming languages and compilers. The fact that many program generators are tailored towards a specific and rather limited application area does not detract from these arguments. There is a general tradeoff between the ''depth'' of the generation process and the application area covered by the generator. Depth can be gained only at the expense of scope. The code generation phase of compilers tends to be rather ''shallow'', reflecting the fact that high-level programming languages have a general purpose character.

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 is another general tradeoff here. Generating a more efficient program takes more time.

There are basically two (more or less related) 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.

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. 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 (section 2.3).

Using the principles outlined in this paper, we have successfully converted greedy scanner and parser generators to versions that are both lazy and incremental. This is briefly discussed in section 3. Full details will be given in [4] and [5]. To avoid misunderstanding it should be emphasized that such conversions cannot be done in a purely mechanical or routine fashion.

In spite of their indisputable importance, very little has been written on the general principles underlying program generators. It is perhaps no coincidence that the terms ''program generator'' and ''program generation'' do not occur in the index to the 1987 version of the *Computing Reviews* classification system. The literature on *partial evaluation* contains some relevant theoretical considerations of a general nature. In [3] Ershov clearly explains the basic ideas involved. A recent bibliography is [8]. Although these ideas 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 yet attempted to interpret our work in that particular context.

2. GREEDY, LAZY, AND INCREMENTAL PROGRAM GENERATION

2.1. The traditional (greedy) case

Suppose a dyadic function

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

has to be implemented in a context in which for each $a \in A$ the value $F(a,b)$ is required for a relatively large number of $b \in B$. In such cases the efficiency of computing $F(a,b)$ for some fixed a can often be improved by replacing F with a higher-order function (''program generator'') of curried type

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

which, when given a particular $a \in A$, returns ("generates") a specialized function (program)

$$G_a: B \rightarrow C$$

such that

$$G_a(b) = F(a,b)$$

for all $b \in B$, and with the additional property that computing $G_a(b)$ takes (much) less time than computing $F(a,b)$. 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

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

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, F 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, F 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, such that $F(a,b)$ is the result of program a with input b. 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 the introduction, investing time T_a in generating G_a is not always justified:

(A) For some $a \in A$ the number of b for which $F(a,b)$ has to be computed may turn out to be so small that the break-even point $(1+\frac{1}{S-1})T_a$ is not reached.

(B) The real-time requirements (interactive response time requirements) that have to be met may be so stringent that there simply is no time T_a available between two successive responses of the system.

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. Parts generated for previous inputs b (if any) are retained, 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 not generated. Clearly, lazy program generation only makes sense if computing $F(a,b)$ always or nearly always involves only a relatively small part a_b of a, and if G_a can be approximated incrementally from below.

More precisely, let A be a domain with a (reflexive) partial order $\subseteq$ ("part of"). Let $a \in A$ be fixed and let U be a subset of B. Computing $F(a,U) = \{F(a,b) \mid b \in U\}$ is said to *involve* a part $a_U \subseteq a$ if the computation of $F(a,U)$ does not use parts of a outside a_U. (Note that $F(a,U)$ is just an abbreviated notation. It does not mean that F is given argument U. $F(a,U)$ is computed by computing $F(a,b)$ separately for each $b \in U$.) The notion of what it means for the computation of $F(a,U)$ to involve a_U is hard to characterize precisely, but a_U may reasonably be expected to have the following properties:

(a) $F(a_U,b) = F(a,b)$ for all $b \in U$;

(b) $a_U \subseteq a_V$ if $U \subseteq V$;

(c) $a'_U=a_U$ for all a' with $a_U \subseteq a' \subseteq a$;

(d) $a'_U \subseteq a_U$ if $a' \subseteq a$.

Whether the *involve*-relation actually has these properties has to be verified in each particular instance. Property (d) is not satisfied in the case of lazy/incremental compilation (section 3.3).

An immediate consequence of (a) and (c) is that $F(a',b)=F(a,b)$ for all $b \in U$ and all a' with $a_U \subseteq a' \subseteq a$. However, a_U need not be the smallest part of a having this property, because a_U is not primarily determined by the *functional* behavior of F, but by its *algorithmic* behavior. It may happen, for instance, that for some $b \in U$ the algorithm explores parts of a that are subsequently found to be dead ends. Although superfluous from an extensional viewpoint, these parts do in fact belong to a_U.

In practice, a_B will usually be equal to a, but it may happen that some parts of a are never needed at all. For fixed a, each $U \subseteq B$ gives rise to a completion $\bar{U}$ which is the largest superset of U such that $a_{\bar{U}} = a_U$. $\bar{U}$ contains all $b \in B$ such that computing $F(a,b)$ involves a_U.

In what follows it is assumed *that the* involve-*relation defined by F stays the same if F is implemented by means of a greedy or lazy program generator.*

Now consider a fixed $a \in A$ and a sequence $b_1, b_2, \ldots$ of elements of B. Let $U_n=\{b_i \mid 1 \le i \le n\} \subseteq B$ and consider the part $a_n \subseteq a$ involved in computing $F(a,U_n)$. Like $\{U_n\}_{n\ge1}$, the sequences $\{a_n\}_{n\ge1}$ and $\{\bar{U}_n\}_{n\ge1}$ are non-decreasing. The greedy generator G of the previous section produces a program $G_n=G_{a_n}$ for each a_n. G_n is a function of type $B \to C$ which behaves like G_a on $\bar{U}_n$:

$$G_n(b) = G_{a_n}(b) = F(a_n,b) = F(a,b) = G_a(b). \quad (b \in \bar{U}_n)$$

Outside $\bar{U}_n$ the behavior of G_n is different from that of G_a, but G_n itself is unable to detect this. This inability makes it unsuitable as an approximation to G_a in the lazy case. So, rather than G_n, consider the *incomplete* program (*partial* function) g_n: $B \to C$ which is similar to G_n, except that it contains *gaps* corresponding to parts of the program involved in computing $F(a,b)$ for b outside $\bar{U}_n$. These parts have not yet been generated. Any attempt to execute a gap causes an appropriate exception to be raised requesting the lazy generator H to "fill" the gap in question. Having done this to the extent necessary, H restarts execution of the expanded version of g_n at the point where the exception occurred.

H can best be described as a higher-order function of type

$$A \times B \times (B \to C) \to C \times (B \to C),$$

such that

$$H(a,b_n,g_{n-1}) = (g_n(b_n),g_n) = (G_n(b_n),g_n) = (G_a(b_n),g_n) = (F(a,b_n),g_n). \quad (n \ge 1)$$

In view of the foregoing the corresponding program (expressed in some suitable language) is:

```
H(a,b,g)
begin
  return(g(b),g);
    when attempting to execute gap γ in g
      do
        g:=expand(γ,g,a)
        resume
      od
end.
```

For each new $b=b_n$ H initially tries to compute the required result by means of the incomplete program g_{n-1} generated during the previous steps. For $n=1$ this is the program g_0 which consists of nothing but a single gap. Only if execution of $g_{n-1}(b_n)$ hits a gap γ in g_{n-1}, H generates an additional piece of program by calling procedure *expand* in the body of the exception handler. This procedure produces the required extension using the gap descriptor γ, the original program g_{n-1}, and the original a. 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 $b_n \in \bar{U}_{n-1}$, no extension is necessary and $g_n = g_{n-1}$. Hence, *expand* is not called and $H(a, b_n, g_{n-1})$ runs as fast as $G_a(b_n)$. (Needless to say, the language used to implement a lazy generator need not have user-defined exceptions. These can be simulated in any programming language.)

There is a natural partial order (the *subsumption* order) on incomplete programs. An incomplete program g subsumes g' ($g \leq g'$) if g' can be obtained from g by partially or completely filling the gaps in g. The lazy generator H produces a sequence $\{g_n\}_{n \geq 0}$ such that

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

with corresponding domains

$$\varnothing \subseteq \bar{U}_1 \subseteq \bar{U}_2 \subseteq \cdots \subseteq B.$$

An extreme and, from the viewpoint of lazy program generation, undesirable case is $a_1 = a$. This means that $g_1 = G_a$. H has to generate the whole program G_a just to handle b_1. Obviously, the lazy character of H is lost in this case. If, on the other hand, the sequence $\{b_n\}_{n \geq 1}$ is such that a_n remains small, g_n may be expected to remain relatively small too, resulting in less generation overhead and a more favorable break-even point in comparison with the greedy case.

Lazy program generation may alleviate the two problems (A) and (B) mentioned in the beginning of this section if the following conditions are met:

(1) Computing $F(a,b)$ always or nearly always involves only a small part a_b of a. In that case the sequence $\{a_n\}_{n \geq 1}$ does not increase too fast, *expand* has to generate only relatively small extensions at each step n, and the total generation time is distributed evenly.

(2) The *additional* overhead in comparison with the overhead of greedy program generation is relatively small, i.e., there is a relatively small δ_a such that for every sequence $\{b_n\}_{n \geq 1}$

$$\sum_{n \geq 1} T_n \leq T_a + \delta_a,$$

where T_n is the time (≥ 0) used by *expand* to extend g_{n-1} to g_n, and T_a the time used by the greedy generator G to generate G_a (section 2.1).

2.3. The combination of lazy and incremental program generation

In the previous section a was kept fixed. 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.

Although lazy generation may certainly offer a partial solution, the above scheme is still rather crude in that it does not attempt to retain as much of the old program as possible. Suppose a is modified to a' after input b_k has been processed. The old program g_k can be retained if $g_k \leq G_{a'}$, where $\leq$ is the subsumption order introduced in the previous section. For instance, if a part of a that has not yet been involved in any computation is deleted, then $a_k \subseteq a' \subseteq a$ and property (c) of the *involve*-relation (section 2.2) guarantees that the incomplete program g_k remains valid in the context of a', except that some of its gaps may now correspond to parts of a that no longer exist. These gaps can simply be removed in advance or, alternatively, on the fly by *expand*. Hence, g_k need not be thrown away, and $F(a', b_{k+1})$ is correctly computed by $H(a', b_{k+1}, g_k)$.

One might be inclined to think that the same would apply to all modifications such that $a_k \subseteq a'$. Unfortunately, this is not true as g_k does not contain gaps corresponding to new parts of a' that may be needed for computing $H(a', b_{k+1}, g_k)$, i.e., $a_k \subseteq a'$ does not imply $g_k \leq G_{a'}$.

To see this more clearly, it may be helpful to consider a lazy/incremental generator for context-free parsers (section 3.2). In this case, F is a universal context-free parser, A the domain of context-free grammars, B consists of sentences to be parsed, and C consists of *sets* of parse trees (parses can be ambiguous!) and a failure value. Let a be a grammar and b a sentence such that parsing b with grammar a involves a subgrammar $a_b \subseteq a$. What happens if syntax rules are added to a? There are basically the following possibilities:

(1) Parsing b with the expanded grammar a' involves the same subgrammar as before, i.e., $a'_b = a_b$. As $F(a',b) = F(a'_b,b) = F(a_b,b) = F(a,b)$, this means that b has the same parse as before or that parsing fails in both cases.

(2) Parsing b with a' involves a subgrammar a'_b which is larger than a_b, but b has the same parse as before. This may happen if the parser explores syntax rules of a' outside a_b that do not affect the final parse (see section 2.2).

(3) Parsing b with a' involves a subgrammar a'_b which is larger than a_b, and either the parse succeeds for a' while it failed for a, or the degree of ambiguity of the parse is larger for a' than it was for a.

Note that properties (c) and (d) of the *involve*-relation (section 2.2) are satisfied in this case.

Now define the *greatest common ancestor* (greatest lower bound) $g \wedge h$ of two incomplete programs g and h as the most specific (least general) incomplete program that can still "become" both g and h, i.e.,

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

It is assumed that the greatest common ancestor of two incomplete programs always exists. In the worst case it is equal to the program that consists of nothing but a single gap. Two useful properties of $\wedge$ are:

$g \wedge g = g$,
$g \wedge h = g$ if $g \leq h$.

Consider a domain D of *modifications* with a function *apply*: $D \times A \rightarrow A$ such that $apply(\delta,a)$ is the element of A obtained by applying modification δ to a. Not every δ need be applicable to every a, so *apply* is a partial function. Let $0 \in D$ be the null modification with $apply(0,a)=a$ for every $a \in A$. A lazy/incremental program generator I is a higher-order function of type

$D \times A \times B \times (B \rightarrow C) \ \rightarrow \ (C \times (B \rightarrow C)) \times A,$

with corresponding program

```
$I(\delta,a,b,g)$
begin
  $(a',g') := modify(\delta,a,g)$
  return $(H(a',b,g'),a')$
end,
```

where H is the lazy generator discussed in the previous section and procedure *modify* computes

$a' = apply(\delta,a),$

and

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

Obviously, g' should *not* be computed by first computing $G_{a'}$. It turns out that in many concrete cases g' can be efficiently computed on the basis of δ, a, and g *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.

If $\delta=0$, *modify* has no effect, and I reduces to H:

$a' = apply(0,a) = a,$

and, assuming $g \leq G_a$ (which is true if I is used properly),

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

When a is modified after input k, $I(\delta,a,b_{k+1},g_k)$ retains in g' the large possible part of g_k that is still valid in the new context. Subsequently, H expands g' to g_{k+1} by need. Actually, the part of g_k that is retained is maximal only in a relative sense. It depends on the domain of incomplete programs in which the greatest common ancestor is interpreted. From the viewpoint of abstract syntax, the simplest incomplete programs are Ω-terms in the sense of [7, section 6.5.2]. In Ω-terms the special constant Ω acts as a gap. The subsumption order $\leq$ is also called *prefix* order in this case. For instance,

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

If incomplete programs are Ω-terms, the greatest common ancestor always exists. For instance,

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

A larger part of g_k may be retained 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

$$program(\Omega).$$

In the lazy/incremental lexical scanner and context-free parser generators discussed in the next section incomplete programs are graphs with gaps rather than trees with gaps.

3. APPLICATIONS

In this section we briefly discuss three concrete applications of the general principles outlined in the previous sections.

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

ISG is a fully lazy/incremental lexical scanner generator developed by us. 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 recognizing the language generated by the grammar. ISG constructs this automaton "by need", so the incomplete programs g_n produced by ISG may be viewed as approximations to the automaton for the input grammar. The domain D of modifications consists of additions and deletions of a single regular expression. ISG has been implemented in LISP and is currently operational. Full details will be given in [4]. Suffice it to say that, in so far as a meaningful comparison can be made, the generation time used by ISG compares very favorably with that of the greedy lexical scanner generator LEX [1, section 3.8], while the lexical scanners produced by it (in LISP) have about the same performance as those generated by LEX (in C). ISG is used in conjunction with the lazy/incremental parser generator described in the next section.

3.2. IPG - a fully lazy/incremental parser generator for finitely ambiguous context-free grammars

Taking Tomita's context-free parsing algorithm [9] as our point of departure, we developed the fully lazy/incremental parser generator IPG. In this case, A is the domain of finitely ambiguous context-free grammars, B contains the sentences to be parsed, and C consists of sets of parse trees and a failure value (see also section 2.3). The programs g_n generated by IPG are incomplete (possibly ambiguous) LR parse tables which are constructed "by need". These tables may be viewed as incomplete programs for Tomita's algorithm, which is a pseudo-parallel table driven parser. The domain D of modifications consists of additions and deletions of a single production rule. IPG has been implemented in LISP and is currently operational. A detailed description of it will be published shortly [5]. The combination ISG/IPG will be used in an environment for the interactive development of formal language definitions.

3.3. Lazy/incremental compilation

In this case A is a domain of programs in some language L, B is a domain of input values for L-programs, and C is a domain of result values. The programs g_n generated by a lazy/incremental compiler I for some program a consist of pieces of object code interspersed with gaps containing source code. The initial program g_0 consists of a single gap containing the entire program a. Each time a gap is encountered during the execution of $g_{n-1}(b_n)$, the corresponding source code is (partially) compiled by *expand*, the resulting object code (with gaps) is put into the gap, and execution is resumed. Hence, in addition to the object code of g_{n-1}, g_n contains the object code needed by b_n. Unreachable parts of a will never be compiled, but an already compiled part of a may become unreachable when a is modified.

When statements are added to a program a, the new version a' may behave rather differently than a. In particular, the part a_b needed by b need not be a part of a'_b. So property (d) of the *involve*-relation (section 2.2) is not satisfied in this case.

Also known as *dynamic compilation*, lazy compilation is used in many BASIC systems. We have been unable to ascertain whether a fully lazy/incremental compiler has been implemented for some language, but something very close to it is bound to exist somewhere. See also an interesting paper by Brown [2].

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. Ershov, A.P. On the partial computation principle. *Information Processing Letters 6*, 2 (April 1977), 38-41.
4. Heering, J., Klint, P., and Rekers, J. Incremental generation of lexical scanners. Department of Software Technology, Centre for Mathematics and Computer Science, Amsterdam, in preparation.
5. Heering, J., Klint, P., and Rekers, J. Incremental generation of parsers. Department of Software Technology, Centre for Mathematics and Computer Science, Amsterdam, in preparation.
6. 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.
7. Huet, G. Formal structures for computation and deduction. CMU Course Notes, INRIA, Rocquencourt, May 1986.
8. Sestoft, P. Partial evaluation bibliography. Preliminary version, *Bulletin of the European Association for Theoretical Computer Science*, No. 31 (February 1987), 50-54.
9. Tomita, M. *Efficient Parsing for Natural Language.* Kluwer, 1985.