Scan Grammars: Parallel Attribute Evaluation Via Data-Parallelism

Thomas Reps

Technical Report #1120

November 1992

Scan Grammars: Parallel Attribute Evaluation Via Data-Parallelism

Thomas Reps
University of Wisconsin–Madison

Abstract

This paper concerns the problem of how to exploit parallelism during the phases of compilation concerned with syntax-directed analysis and translation. In particular, we address the problem of how to exploit parallelism during the evaluation of the attributes of a derivation tree of a non-circular attribute grammar. What distinguishes the ideas presented in this paper from earlier work on parallel attribute evaluation is the use of a *data-parallel* model: We define a new class of attribute grammars, called *scan grammars*, which are designed so that attributes can be evaluated in a data-parallel fashion. Scan grammars include a *scan-attribution* construct, which defines a *scan* over attribute values located at a derivation tree's leaves. (The scan operation is also known as *parallel-prefix*.) A scan-attribution computes the partial sums (with respect to a given associative operator) in a left-to-right or right-to-left traversal of the derivation tree's leaves. Each partial sum computed by the scan is "left behind" at the appropriate leaf, where it may be an argument to other attribute equations or used in another scan-attribution operation. (Although scan-attribution is a data-parallel construct, this does not mean that scan grammars are only suitable for use on a SIMD machine; on the contrary, there are good reasons to expect that our parallel scan-grammar evaluator will perform well on a MIMD machine.)

Scan grammars can be evaluated efficiently in parallel. Given one processor per production instance in the derivation tree, a scan-attribution can be evaluated in parallel in $2D + 1$ steps, where D is the *depth* of the derivation tree. When only $P < N$ processors are available, where N is the number of production instances in the derivation tree, each actual processor must simulate the actions that need to be carried out at some number of production instances. Using such a simulation technique, it is possible evaluate a scan-attribution in at most $2N/P + (2D + 1)$ steps.

In all previous work on parallel attribute evaluation, the maximum speedups reported for Pascal attribute grammars have been relatively modest—in the range of 4 to 7 (having tailed off rapidly beyond about 16 processors). One of the chief reasons for this is that the dependences among the attributes that control symbol-table processing form long chains, which are an obstacle to parallel processing. In this paper, we show how to surmount this difficulty: We show how to express symbol-table processing using the scan-attribution construct (*i.e.*, as a scan-attribution with respect to a certain associative operator), which makes it amenable to our parallel-evaluation techniques. Our estimates show that it may be possible to obtain speedups for Pascal programs on the order of 10-fold with 10 processors, 64-fold with 100 processors, and 100-fold with 250 processors.

1. Introduction

This paper concerns the problem of how to exploit parallelism during the phases of compilation concerned with syntax-directed analysis and translation. Among the tasks that arise during these phases are symbol-table construction, name analysis, type checking, and either code generation or translation to an intermediate representation for subsequent processing. The techniques we present in the paper are not restricted to just the programming-language translation tasks listed above. They also apply to many other problems that can be posed as the translation of a derivation tree of a context-free language, such as pretty-printing, text formatting, generation of verification conditions from assertions, and verifying the correctness of proofs

---

This work was supported in part by a David and Lucile Packard Fellowship for Science and Engineering, by the National Science Foundation under grant CCR-9100424, and by the Defense Advanced Research Projects Agency under ARPA Order No. 8856 (monitored by the Office of Naval Research under contract N00014-92-J-1937).

Author's address: Computer Sciences Department, University of Wisconsin–Madison, 1210 W. Dayton St., Madison, WI 53706.
E-mail: reps@cs.wisc.edu

(for different kinds of mathematical and programming logics). Thus, our results contribute to the development of parallelized implementations of tools in all of these domains.

The particular problem addressed is how to exploit parallelism during the evaluation of the attributes of a derivation tree of a (non-circular) attribute grammar [15] (see also [24] or [2]). Although this problem has been addressed by others [6, 13, 16, 25], what distinguishes the ideas presented in this paper from earlier work on parallel attribute evaluation is the use of *data-parallelism*. The basic operation in the data-parallel model is a *scan* over a sequence with respect to an associative operator [5]. (The scan operation is sometimes called parallel-prefix or parallel-suffix.) A scan computes a new sequence whose elements are the "partial sums" of the original sequence. In general, we are given a list $x_1, \cdots, x_n$ and an associative binary operator $\oplus$. In a left-to-right $\oplus$-scan (or $\oplus$-parallel-prefix operation), the goal is to compute the list $y_1, \cdots, y_n$, such that $y_i = x_1 \oplus \cdots \oplus x_i$, for $1 \le i \le n$. For example, the result of applying the left-to-right $+$-scan operation to the sequence (1, 2, 3, 4, 5) is the sequence (1, 3, 6, 10, 15).

This paper defines a new class of attribute grammars, called *scan grammars*, which are designed so that attributes can be evaluated in a data-parallel fashion. Scan grammars include a *scan-attribution* construct, which defines a scan over attribute values located at a derivation tree's leaves. A scan-attribution computes the partial sums (with respect to a given associative operator) in a left-to-right or right-to-left traversal of the derivation tree's leaves. Each partial sum computed by the scan is "left behind" at the appropriate leaf, where it may be an argument to other attribute equations or used in another scan-attribution operation. (Although scan-attribution is a data-parallel construct, this does not mean that scan grammars are only suitable for use on a SIMD machine; on the contrary, as discussed in Section 5.3, there are good reasons to expect that our parallel scan-grammar evaluator will perform well on a MIMD machine.)

The scan-attribution construct can be simulated with a conventional attribute grammar using attribute equations that are threaded left-to-right (or right-to-left) through the derivation tree to create the partial sums (see Section 3.1). Any of a number of sequential attribute evaluators can then be used to provide a sequential implementation of a scan-grammar evaluator. However, because scan-attributions are defined in terms of associative operators, scan grammars can be evaluated more efficiently in parallel. Instead of a simple left-to-right flow of information, the parallel-evaluation strategy uses a different pattern of information flow, which is based on the one employed in the algorithms for carry-lookahead addition [20] and the efficient parallel evaluation of scan operations [5, 17] (see Section 3.2). Given one processor per production instance in the derivation tree, a scan-attribution can be evaluated in parallel in $2D + 1$ steps, where D is the *depth* of the derivation tree. With fewer processors than one per production instance, substantial speedups can still be obtained by having each actual processor simulate the actions that need to be carried out at some number of production instances. In particular, by a theorem of Brent [7, 8], a scan-attribution can be evaluated in at most $2N/P + (2D + 1)$ steps, where N is the number of production instances in the derivation tree and $P < N$ is the number of processors.

The sequential evaluation of the left-to-right-threaded equations that implement scan-attribution requires $2N$ steps. Thus, the parallel-evaluation bounds ($2D + 1$ steps and $2N/P + (2D + 1)$ steps) suggest that the parallel evaluation of scan grammars will be substantially faster than sequential evaluation. By measuring the size of the abstract syntax trees of two (uncontrived) Pascal programs, we found that the ratio $2N/(2D + 1)$ was 119 for a 478-line Pascal program and 174 for a 776-line Pascal program. Furthermore, the ratio $2N/(2N/P + (2D + 1))$ for the same two programs suggests that as much as 9-fold speedup with 10 processors, 55-fold speedup with 100 processors, and 81-fold speedup with 250 processors could be obtained for the smaller program and 10-fold speedup with 10 processors, 64-fold speedup with 100 processors, and 103-fold speedup with 250 processors could be obtained for the larger program. (See the tables presented in Sections 3 and 4.) These quantities are likely to increase with larger programs.

In all previous work on parallel attribute evaluation, the speedups reported for Pascal attribute grammars have been relatively modest—in the range of 4 to 7 (having tailed off rapidly beyond about 16 processors). One of the chief reasons for this is that the dependences among the attributes that control symbol-table processing form long chains, which are an obstacle to parallel processing. Section 5 describes how the data-parallel approach surmounts this difficulty. In particular, we show how symbol-table processing can be expressed using the scan-attribution construct (*i.e.*, as a scan-attribution with respect to a certain associative operator) and hence—by the results of Section 3—is amenable to being carried out in parallel.

While the results of ours quoted above are suggestive of the potential of our approach to parallel syntax-directed analysis and translation, achieving these substantial speedups will require careful attention to a number of implementation issues. These issues are discussed in Sections 3, 4, and 5.

The remainder of the paper is organized as follows: Section 2 introduces scan grammars by means of an example. Section 3 discusses a parallel evaluation algorithm for scan grammars, which is based on the ideas used in the carry-lookahead addition algorithm (and other algorithms for computing scans quickly in parallel). Section 4 discusses the representation of lists derived from the "list nonterminals" that are typically used in context-tree grammars. Section 5 describes how the construction of symbol tables is expressed in a scan grammar. Section 6 discusses some extensions to our basic model of scan grammars. Section 7 describes how the ideas presented in this paper relate to previous work on parallel attribute evaluation.

2. Scan grammars

In this section, we introduce the components of scan grammars by means of a simple example—the computation of the value of a binary numeral.

Attribution rules are defined with respect to a set of grammar rules. Here we define the abstract syntax of binary numerals by giving a collection of operator/operand declarations:[1]

```
numeral : Numeral(bits);
bits    : Pair(bits bits)
        | Bit(bit)
        ;
bit     : Zero()
        | One()
        ;
```

Aside. Before giving the scan grammar's attribution rules for the binary-numeral problem, we first give rules for a conventional attribute grammar that solves the problem. In a conventional attribute grammar, two integer-valued attributes—say "position_in" and "position_out"—are used to define a right-to-left pat-

[1]This notation is a variant of context-free grammars in which the operator names (Numeral, Pair, Bit, Zero, and One) serve to identify the productions uniquely. For example, the declaration "`numeral : Numeral(bits);`" is the analogue of the production "numeral → bits".

In general, the notation used in this paper's examples is adapted from the notation used in the Synthesizer Generator [22], a widely distributed system based on attribute grammars.

tern of information flow through the derivation tree (so-called "right-to-left threading"):

```
bits, bit { inherited INT position_in;
            synthesized INT position_out;
          };
```

In this example, these attributes are used to determine a bit's position in the numeral. More precisely, the goal is for the value of bit.position_out to be the bit's position with respect to the right end of the numeral (where the rightmost bit is considered to be at position 1). This is arranged by the following attribute declarations:[2]

```
numeral : Numeral   {  bits.position_in = 0;  };
bits    : Pair      {  bits$3.position_in = bits$1.position_in;
                       bits$2.position_in = bits$3.position_out;
                       bits$1.position_out = bits$2.position_out;
                    }
        | Bit       {  bit.position_in = bits.position_in;
                       bits.position_out = bit.position_out;
                    }
        ;
bit     : Zero      {  bit.position_out = bit.position_in + 1;   }
        | One       {  bit.position_out = bit.position_in + 1;   }
        ;
```

From the position of a bit in the numeral (*i.e.*, the value of bit.position_out), the bit's contribution to the numeral's overall value can be determined:

```
bit { synthesized INT val; };
bit     : Zero      {  bit.val = 0;  }
        | One       {  bit.val = 2 ** (bit.position_out - 1);  }
        ;
```

The value of the entire numeral is determined by summing the contributions from each subtree:

```
numeral, bits { synthesized INT val; };
numeral : Numeral   {  numeral.val = bits.val;  };
bits    : Pair      {  bits$1.val = bits$2.val + bits$3.val;  }
        | Bit       {  bits.val = bit.val;  }
        ;
```

When the derivation tree is consistently attributed according to these rules, the value of the attribute "numeral.val", which occurs at the derivation tree's root, holds the value that corresponds to the numeral. **End Aside.**

We now show how the binary-numeral problem is specified using a scan grammar. The solution involves two scan-attributions, called "position" and "value", both of which are directed right-to-left.

A scan-attribution is declared by specifying a direction (*i.e.*, LR or RL), a type, an associative operation, and a "seed" value. For example, the declaration

```
scan position (RL, [INT] → [INT], +, 0);
```

defines a scan-attribution named "position" as a right-to-left scan-attribution that maps a sequence of INT's

---

[2] In the attribute equations for a production such as "`bits : Pair(bits bits);`" it is necessary to distinguish between the three different occurrences of nonterminal "`bits`". We use the notation "`bits$1`", "`bits$2`", and "`bits$3`", where "`bits$1`" denotes the left-hand-side occurrence, *etc.*

to a sequence of INT's via the addition function, with seed value 0. Such a declaration creates two attributes at each leaf, denoted by "position′input" and "position′output". At each leaf in the derivation tree, the value of position′output is the appropriate partial sum in a right-to-left scan, with respect to the addition function, of the sequence formed from the position′input attributes at the derivation tree's leaves. (In practice, the input sequence and the output sequence are both conceptual. The input sequence and the output sequence are both distributed over the leaves of the derivation tree and neither is ever materialized as an explicit data structure.)[3]

In this example, the goal is for the value of bit.position′output to be the bit's position with respect to the right end of the numeral. (The bit.position′output attributes serve the same function as the bit.position_out attributes in the conventional attribute grammar given earlier.) Thus, for each instance of a bit nonterminal, we want the value of bit.position′input to be 1, which is easily arranged with the following attribution equations:

```
bit       : Zero   {  bit.position′input = 1;   }
          | One    {  bit.position′input = 1;   }
          ;
```

The value of the entire numeral is defined by declaring a second right-to-left scan that forms the partial sums of the individual bits' contributions:

```
scan value (RL, [INT] → [INT], +, 0);
bit       : Zero   {  bit.value′input = 0;   }
          | One    {  bit.value′input = 2 ** (bit.position′output - 1);   }
          ;
```

As these rules illustrate, the results from one scan-attribution (*e.g.*, the values of the various bit.position′output attributes) can be used in creating the input to a second scan-attribution. The role of bit.value′output is somewhat different from that of bit.val in the conventional attribute grammar given earlier—each bit.value′output attribute defined by the rules given above represents the value the numeral would have if the left prefix of the numeral up to the leaf were discarded.[4]

Remark. It is tempting to simplify the rules we have given, and use only one (left-to-right) scan-attribution declaration instead of two, as follows:

```
scan value (LR, [INT] → [INT], λx.λy.2*x + y, 0);
bit       : Zero   {  bit.value′input = 0;   }
          | One    {  bit.value′input = 1;   }
          ;
```

However, this scan-attribution declaration uses the function $\lambda x . \lambda y . 2 * x + y$, which is not an associative operation. It will become clear in the discussion of scan operations in Section 3 that the associativity property is of crucial importance if scans are to be evaluated efficiently in parallel. Associativity provides us with the freedom to re-group operations, and it is this freedom that makes scan-attribution amenable to parallel processing.

[3]It is also possible to add *segmented scan* operations [5, 17] to the repertoire of scan-grammar primitives.

[4]It would also have been possible to define the "value scan" as a left-to-right—rather than a right-to-left—scan, in which case the value of a bit's bit.value′output attribute would be the part of the numeral's value contributed by the given bit and those bits to its left.

For scan-attributions using such non-associative operators to be well-defined, the evaluator would have to work either left-to-right or right-to-left in sequential order. This would require $2N$ steps, where N is the number of production instances in the derivation tree, rather than $2D + 1$ or $2N/P + (2D + 1)$ steps.
End Remark.

Returning to our example, closer inspection reveals that the scan grammar given above is not quite equivalent to the conventional attribute grammar that was given earlier. With the scan grammar, the numeral's overall value will be found at the leftmost leaf of the derivation tree, not at the derivation tree's root as is the case for the conventional attribute grammar. This is easily rectified by changing the declaration of the operator "Numeral" to add an additional child (say "answer") at the root of the tree, from which it is only possible to derive a leaf:

```
numeral  : Numeral(answer bits);
answer   : Answer();
```

The "answer" leaf, now the leftmost leaf of the derivation tree, can participate in the scan-attribution:

```
answer   : Answer    {  answer.position'input = 1;
                        answer.value'input = 0;
                     }
         ;
```

Finally, the value of the numeral can be passed from the leaf to the root:

```
numeral { synthesized INT val; };
numeral : Numeral    {  numeral.val = answer.value'output;   };
```

An equivalent—and probably more convenient—approach is to permit interior nodes to play a role in scan-attributions directly. The advantage of this approach is that we only have to add attribute equations; we do not have to make changes to the underlying grammar. Conceptually, each interior node of arity-k has $k+1$ leaves, one to the left of the leftmost child, one to the right of the rightmost child, and one in between every pair of consecutive children. For example, with our original definition "`numeral : Numeral(bits);`", a "numeral" node has two such conceptual leaves, denoted by "numeral[0]" for the one on the left and "numeral[1]" for the one on the right. Using this notation, the following rules specify the same result as the ones given above:

```
numeral { synthesized INT val; };
numeral : Numeral(bits)   {  numeral[0].position'input = 1;
                             numeral[0].value'input = 0;
                             numeral.val = numeral[0].value'output;
                          }
        ;
```

(We will make use of this notation in the examples in Section 5.)

3. Evaluation of scan grammars

3.1. Sequential implementation of scan-attribution

It is straightforward to compile scan-attributions into conventional attribute-grammar specifications, whereupon they can be evaluated using any of a number of sequential evaluators. Each scan-attribution construct can be translated into a left-to-right or right-to-left attribute threading that creates the partial sums. For instance, in the binary-numeral example, the "position" scan would compile into essentially the same set of right-to-left-threaded attribute equations that were given for attributes position_in and position_out in the description of how the binary-numeral example is handled in a conventional attribute grammar. The "value" scan would also be translated into a right-to-left threading, as follows:

```
bits, bit { inherited INT value_in;
            synthesized INT value_out;
          };
numeral { synthesized INT val; };
numeral : Numeral    {  bits.value_in = 0;
                        numeral.val = bits.value_out;
                     }
        ;
bits    : Pair       {  bits$3.value_in = bits$1.value_in;
                        bits$2.value_in = bits$3.value_out;
                        bits$1.value_out = bits$2.value_out;
                     }
        | Bit        {  bit.value_in = bits.value_in;
                        bits.value_out = bit.value_out;
                     }
        ;
bit     : Zero       {  bit.value_out = bit.value_in;   }
        | One        {  bit.value_out = 2 ** (bit.position_out - 1)
                                                  + bit.value_in;    }
        ;
```

3.2. Parallel implementation of scan-attribution

Because scan-attributions are defined in terms of associative operators, they can be evaluated efficiently in parallel. The pattern of information flow in the parallel evaluation algorithm is based on that employed in the algorithm for the efficient parallel evaluation of scan operations [5, 17] (of which carry-lookahead addition is one example [20]).

A $\oplus$ - scan operation with respect to an associative operator $\oplus$ can be performed on a list of length k in $2\lceil \log k \rceil + 1$ steps by decomposing the problem into subproblems and arranging the subproblems in a balanced binary tree of depth $\lceil \log k \rceil$. One processing element is assigned to each leaf and interior node of the decomposition tree. Here is Blelloch's explanation of the pattern of information flow in the parallel implementation of a left-to-right scan:

> The technique consists of two sweeps of the tree, an up sweep and a down sweep . . . The values to be scanned start at the leaves of the tree. On the up sweep, each unit executes $\oplus$ on its two children units and passes the sum to its parent. Each unit also keeps a copy of the value from the left child in its memory. On the down sweep, each unit passes to its left child the value from its parent and passes to its right child $\oplus$ applied to its parent and the value stored in the memory (this value originally came from the left child). After the down sweep, the values at the leaves are the results of a scan [5].[5] .

It is actually not necessary that the problem-decomposition tree be balanced, nor is it necessary that it be a binary tree. The $\oplus$ - scan computation can be carried out in parallel on any *fixed-arity* tree that represents a

[5]Because of a slight technical difference between the way we have defined the $\oplus$ - scan of a list and the way Blelloch defines it, we must add the following: It is also necessary for each leaf to retain a copy of its original value and to apply $\oplus$ to the value received from the parent and the original value.

decomposition of the problem in $2D + 1$ steps, where D is the *depth* of the decomposition tree.

Because the information flow in each of the two sweeps can be expressed as an attribution of the decomposition tree according to the rules of an attribute grammar, we are able to use the scan-evaluation algorithm to evaluate a scan-attribution in parallel. The first step is to translate the scan-attribution construct to a set of attribute equations of a special form. We illustrate this translation by means of two examples.

Example. Suppose $\oplus$ is an associative operation of type $T \times T \to T$, and we are given a left-to-right scan-attribution of the form

```
scan foo (LR, [T] → [T], ⊕, v);
root : Root(a);
a    : Pair(a a)
     | Singleton(b)
     ;
b    : Leaf()          { b.foo'input =  · · · ;  };
```

This scan-attribution is translated into the following equations:

```
/* Sweep I ------------------------------------------------------ */
a, b { synthesized T foo1; };
a    : Pair(a a)      {  a$1.foo1 = a$2.foo1 ⊕ a$3.foo1;  }
     | Singleton(b)   {  a.foo1 = b.foo1;  }
b    : Leaf()         {  b.foo1 =  · · · ;  };

/* Sweep II ----------------------------------------------------- */
a, b { inherited T foo2; };
root : Root(a)        {  a.foo2 = v;  };
a    : Pair(a a)      {  a$2.foo2 = a$1.foo2;
                         a$3.foo2 = a$1.foo2 ⊕ a$2.foo1;
                      }
     | Singleton(b)   {  b.foo2 = a.foo2 ⊕ b.foo1;  }
     ;
```

We say that equations like the ones in the above example are in *scan form*. We now show what this translation produces for the two right-to-left scans of the binary-numeral example.

Example. The "position" scan is translated into the following attribute equations in scan form:

```
/* Sweep I ------------------------------------------------------ */
bits, bit { synthesized INT position1; };
bits     : Pair      {  bits$1.position1 = bits$2.position1 + bits$3.position1;  }
         | Bit       {  bits.position1 = bit.position1;  }
         ;
bit      : Zero      {  bit.position1 = 1;  }
         | One       {  bit.position1 = 1;  }
         ;

/* Sweep II ----------------------------------------------------- */
bits, bit { inherited INT position2; };
numeral  : Numeral   {  bits.position2 = 0;  };
bits     : Pair      {  bits$3.position2 = bits$1.position2;
                        bits$2.position2 = bits$3.position1 + bits$1.position2;
                     }
         | Bit       {  bit.position2 = bit.position1 + bits.position2;  }
         ;
```

The "value" scan is translated as follows:

```
/* Sweep I -------------------------------------------------------- */
bits, bit { synthesized INT value1; };
bits     : Pair      {  bits$1.value1 = bits$2.value1 + bits$3.value1;   }
         | Bit       {  bits.value1 = bit.value1;   }
         ;
bit      : Zero      {  bit.value1 = 0;   }
         | One       {  bit.value1 = 2 ** (bit.position2 - 1);   }
         ;

/* Sweep II ------------------------------------------------------- */
bits, bit { inherited INT value2; };
numeral : Numeral   {  bits.value2 = 0;   };
bits     : Pair      {  bits$3.value2 = bits$1.value2;
                        bits$2.value2 = bits$3.value1 + bits$1.value2;
                     }
         | Bit       {  bit.value2 = bit.value1 + bits.value2;   }
         ;
```

Given equations in scan form, there is an obvious parallel algorithm for evaluating them, which is to use one processor per production instance and have each processor evaluate the tree's attributes in accordance with the method quoted above. By this means, a scan-attribution can be evaluated in parallel in $2D + 1$ steps, where D is the depth of the derivation tree.

When only $P < N$ processors are available, where N is the number of production instances in the derivation tree, each actual processor must simulate the actions that need to be carried out at some number of production instances. The scheduling technique used in the simulation theorem of Brent can be used [7, 8] and consequently a scan-attribution can be evaluated in at most $2N/P + (2D + 1)$ steps.

To get a feel for what this means for abstract syntax trees of actual programs, we measured two (uncontrived) Pascal programs: format.p, a simple text-formatting program taken from Kernighan and Plauger's book [14], and gradestats.p, a grading program obtained from a colleague. Figure 1 presents figures on steps, speedup, and efficiency as a function of number of processors for the two programs. Figure 1 shows that it may be possible to obtain speedups for Pascal programs on the order of 10-fold with 10 processors, 64-fold with 100 processors, and 100-fold with 250 processors.

The measurements reported in Figure 1 were taken on trees defined by the Pascal attribute grammar that is distributed with the Synthesizer Generator system for generating language-sensitive editors [22]. The numbers in Figure 1 reflect two adjustments to the raw figures: (1) Because atomic leaves of Synthesizer Generator trees generally do not have attributes, in the formula for calculating "Steps" the number of "Atomic leaves" is subtracted from "Nodes". (2) Also in the formula calculating "Steps", the "Depth" is decreased by two, one for the level of atomic leaves and one for the root production. (A third adjustment incorporated in Figure 1 is explained in Section 4.)

It is also possible to define a different parallel scan-grammar evaluation algorithm based on the same underlying idea, but in which the evaluation of attributes during Sweep II is pipelined (and evaluation is carried out "differentially"). Suppose $\oplus$ is the operation with respect to which the scan is being performed and that e is the identity element for $\oplus$:

(1) Allocate one processor per production instance.

(2) Each attribute in the scan's scan-form equations is initially assigned the value e.

	Program	format.p		Program	gradestats.p	
	Lines	478		Lines	776	
	Nodes	4889		Nodes	8671	
	Atomic leaves	1016		Atomic leaves	1787	
	Depth	34		Depth	41	
P	Steps	Speedup	Efficiency	Steps	Speedup	Efficiency
1	7811	1	1.00	13847	1	1.00
10	840	9	0.93	1456	10	0.95
20	452	17	0.86	767	18	0.90
30	323	24	0.81	538	26	0.86
40	259	30	0.75	423	33	0.82
50	220	36	0.71	354	39	0.78
60	194	40	0.67	308	45	0.75
70	176	44	0.64	276	50	0.72
80	162	48	0.60	251	55	0.69
90	151	52	0.57	232	60	0.66
100	142	55	0.55	217	64	0.64
150	117	67	0.45	171	81	0.54
200	104	75	0.38	148	94	0.47
250	96	81	0.33	134	103	0.41
500	80	97	0.19	107	130	0.26
1000	73	107	0.11	93	149	0.15
2000	69	113	0.06	86	161	0.08
3873	67	117	0.03			
4000				82	168	0.04
6884				81	171	0.02

Steps = 2 * (Nodes – Atomic leaves) / P + (2 * (Depth – 2) + 1)
Speedup = (Steps for P = 1) / Steps
Efficiency = Speedup / P

Figure 1. Steps, speedup, and efficiency as a function of the number of processors P for two Pascal programs.

(3) Sweep I is carried out in the normal fashion.

(4) During Sweep II, each processor is in charge of evaluating two attribute instances, say "`child1.foo2`" and "`child2.foo2`". Whenever new information about the value of `parent.foo2`—in the form of a change Δ in the value of `parent.foo2`—is received from the processor of the parent node in the derivation tree, the current values of `child1.foo2` and `child2.foo2` are *updated* by making the assignments

```
child1.foo2 := Δ ⊕ child1.foo2
child2.foo2 := Δ ⊕ child2.foo2.
```

(5) The value Δ is passed down the derivation tree to the processors associated with the child nodes in the derivation tree.

With this method, information is passed to a neighboring processor whenever possible; an attribute's value steadily *accumulates* as more and more information flows down from the `parent.foo2` attribute. Because the passing of information down the derivation tree is pipelined, an attribute's final value is determined only after it has received all of the information from the collection of attributes along the path from the attribute's tree node to the root of the tree.

Remark. In the pipelined version of the evaluation algorithm, an attribute's value steadily accumulates as more and more information flows down from the Sweep II attribute of its parent node in the derivation tree. This is an unusual feature for an attribute-evaluation algorithm, but other examples of such algorithms are known.

One example of previous work in which the final value of an attribute accumulates from previous values is the "differential" algorithm of Hoover and Teitelbaum for incremental updating of aggregate-valued attributes in language-sensitive editors [10], which may consider an attribute several times during updating. This is done to compensate for the evaluator's use of imprecise information about the ordering of dependences between attributes. Because the algorithm is for a sequential processor, the multiple evaluation of attribute instances is not a desirable characteristic of the algorithm (although it does not appear to hurt the algorithm's performance in practice).

Other examples of evaluation algorithms in which the final value of an attribute is accumulated from previous values are the various algorithms proposed for evaluating circular attribute grammars [4, 9, 11, 12, 23].

End Remark.

4. Representation of lists derived from list nonterminals

In this section, we discuss the representation of lists derived from the "list nonterminals" that are typically used in context-free grammars. For example, a programming-language grammar may have a number of different list nonterminals, such as *stmtList*, *expList*, *idList*, *etc.* In the abstract syntax trees of programming languages, there are several different ways to represent lists derived from list nonterminals using nodes of fixed-arity. (Recall that the algorithm for efficient parallel evaluation of scan operations calls for a fixed-arity problem-decomposition tree.) One way is to use right-recursive productions:

$$
\begin{aligned}
&\mathit{stmtList} \rightarrow \mathit{stmt}\ \ \mathit{stmtList}\\
&\mathit{stmtList} \rightarrow \mathit{stmt}\\
&\mathit{stmt} \rightarrow \cdots
\end{aligned}
$$

However, in the case where N processors are available, the parallel scan-grammar evaluation algorithm requires $2D + 1$ steps, where D is the depth of the tree. If lists are represented with right-recursive productions, a list of length k has depth k, and consequently this representation is detrimental to the performance of the evaluation algorithm.

From this standpoint, it is better to replace the right-recursive productions with ambiguous binary productions and to represent lists as *balanced* complete binary trees:

$$
\begin{aligned}
&\mathit{stmtList} \rightarrow \mathit{stmtList}\ \ \mathit{stmtList}\\
&\mathit{stmtList} \rightarrow \mathit{stmt}\\
&\mathit{stmt} \rightarrow \cdots
\end{aligned}
$$

Using this representation of lists can only reduce the depth of the tree; a list of length k has depth $\lceil \log k \rceil$.

To determine how much would be gained from using balanced binary trees to represent lists, we gathered some statistics on the depths of Pascal abstract syntax trees. Again, the measurements were taken on trees defined by the Pascal attribute grammar that is distributed with the Synthesizer Generator. Although the Synthesizer Generator uses right-recursive productions to represent lists, the depth counts were adjusted to reflect what the depth of the tree would have been if balanced binary trees had been used. Figure 2 presents the statistics that were obtained. The column labeled "depth" gives the tree's depth before

program	description	lines	nodes	atomic leaves	depth	adjusted depth
empty.p	empty template for a program	5	15	0	4	3
primes.p	sieve of Eratosthenes	32	235	52	29	24
queens.p	eight-queens problem	56	549	116	36	28
format.p	example from Kernighan & Plauger	478	4889	1016	64	34
gradestats.p	grading program	776	8671	1787	94	41

nodes: number of production instances in the tree
atomic leaves: number of leaves consisting of one of the primitive types INT, CHAR, ID, STR, *etc.*
depth: right-recursive "combs" used to represent lists
$xList \rightarrow x \ xList \mid x$
$x \rightarrow \cdots$
adjusted depth: balanced binary trees used to represent lists
$x \rightarrow xList \ xList \mid x$
$x \rightarrow \cdots$

Figure 2. Sizes of some representative Pascal abstract syntax trees. The column labeled "adjusted depth" is (an upper bound on) the depth of the tree obtained by replacing every list by a balanced complete binary tree.

the adjustment; "adjusted depth" gives the adjusted figure. Because the algorithm for efficient parallel evaluation of scan operations requires $2D + 1$ steps, the two larger examples show that the balanced-binary-tree representation would be better by a factor of 4 to 4.5 when N processors are available. This factor is likely to increase with larger programs.

(In Figure 1, the figures already reflect the use of the balanced-binary-tree representation.)

5. Symbol-table construction via scan-attribution

One of the most common syntax-directed computations that arise in attribute-grammar specifications is *symbol-table construction.* For example, attribute grammars that specify the static semantics of programming languages usually build symbol tables that map identifiers to a descriptor of scope and binding information as the starting point for name analysis (*i.e.*, the detection of undeclared and multiply declared variables) and type checking. Each node of the derivation tree is annotated with (a pointer to) a data structure that records the identifiers known in the current scope and their properties.

In this paper, we use the term "symbol table" in a broader sense: For our purposes, a symbol table is any *finite function*—any function defined only for a finite set of argument values. For example, a programming-language grammar might make use of a "symbol table" for the labels on statements, gathered in preparation for the resolution of gotos; a text-formatting grammar might make use of a "symbol table" consisting of the current formatting properties. The ideas described below apply equally well to all of these different kinds of values.

With conventional attribute grammars, symbol-table creation is typically specified using attribute equations that are either threaded left-to-right or top-down through the derivation tree. With the top-down attribution pattern, information for the declarations in a local scope is synthesized up from the declaration subtree and merged with the global information coming down from above. In cases where the top-down attribution pattern is used, it is usually possible to recast the specification as an equivalent left-to-right threading. As will become clear, the latter pattern of attribution form can in turn be recast as an equivalent left-to-right scan-attribution.

In this section, we show how symbol-table processing can be expressed using the scan-attribution construct (*i.e.*, as a scan-attribution with respect an associative operator). The key observation is as follows: *The basic operation used for building up symbol tables—regardless of their particular implementation—is* ***associative****. Associativity provides us with the freedom to re-group the symbol-table construction operations to make symbol-table construction more amenable to parallel processing.* This observation, in conjunction with the attribution pattern discussed in Section 3, allows us to overcome the sequential-processing bottleneck reported for symbol-table construction in previous work on parallel attribute evaluation [6, 16, 25].

In this section, our examples deal with a simple imperative language whose abstract syntax is defined as follows:

```
program   : Program(declList stmtList);
declList  : DeclListPair(declList declList)
          | DeclListSingleton(decl)
          ;
decl      : Decl(ID type);
type      : Undefined()
          | Boolean()
          | Integer()
          ;
stmtList  : StmtListPair(stmtList stmtList)
          | StmtListSingleton(stmt)
          ;
stmt      : Assignment(ID exp)
          | Conditional(exp stmtList stmtList)
          | WhileLoop(exp stmtList)
          ;
exp       : IdExp(ID)
          |  . . .
          ;
```

The symbol tables for this language will be finite functions from identifiers to types:

$$env =_{df} \text{ID} \rightarrow \text{type}.$$

The domain of types contains a distinguished value "Undefined"; if st is an env and $st(i) = \text{Undefined}$, then st is undefined on i.

In the presentation below, we initially address the case where there is only a single global scope. We then extend our approach to cover the case of nested scopes. Finally, we address an important performance issue: Because symbol-table attributes are not unit-sized objects, we cannot afford the overhead of shipping copies of their values from processor to processor; methods that avoid this potential overhead are presented in Section 5.3.

5.1. Scan-attribution for a single global scope

If there is only a single global scope, it is straightforward to use scan-attribution to specify the annotation of the derivation tree with appropriate symbol-table values.

In this section, rather than fix on a particular representation for symbol tables (*i.e.*, the domain env), we treat them at an abstract level. We define two operations on env: one, denoted by $[x : v]$, creates a symbol table defined at a single point; the other, denoted by $a \otimes b$, combines two symbol tables a and b. More precisely, the two operations behave as follows:

$$[x:v](i) = \begin{cases} v & \text{if } x = i \\ \text{Undefined} & \text{otherwise} \end{cases} \qquad (a \otimes b)(i) = \begin{cases} b(i) & \text{if } b(i) \neq \text{Undefined} \\ a(i) & \text{otherwise} \end{cases}$$

We use e to denote the everywhere-undefined environment: $e(i)$ = Undefined, for all i.

It is easy to see that $\otimes$ is associative and that e is the identity element for $\otimes$. We state this as the following proposition:

Proposition 5.1. *For all a, b, c ∈ env, the following properties hold:*

(1) $(a \otimes b) \otimes c = a \otimes (b \otimes c)$
(2) $e \otimes a = a \otimes e = a$

With this notation, we can now express symbol-table construction using the scan-attribution construct.

Example. Symbol-table construction for our simple programming language can be expressed with a left-to-right scan, with each declaration contributing a symbol table defined at a single point:

```
scan symbol_table (LR, [env] → [env], ⊗, e);
decl : Decl        {  decl[2].symbol_table′input = [ID:type];   };
stmt : Assignment  {  stmt[0].symbol_table′input = e;   };
exp  : IdExp       {  exp[0].symbol_table′input = e;   };
```

Note that each use of an identifier in a statement or expression contributes a symbol table as well, but always with value e. As a consequence, symbol_table′output is available at that point in the tree; this is employed to access the symbol-table information for the identifier used at that point in the program:

```
stmt : Assignment  {  stmt.lhs_info = stmt[0].symbol_table′output(ID);   };
exp  : IdExp       {  exp.id_info = exp[0].symbol_table′output(ID);   };
```

Similarly, if we wish to detect duplicate declarations, then we would change the specification so that Decl contributes two symbol tables, first e and then [ID : type]:

```
decl : Decl   {  decl[0].symbol_table′input = e;
                 decl[2].symbol_table′input = [ID:type];
                 decl.id_info = decl[0].symbol_table′output(ID);
              }
        ;
```

If decl[0].symbol_table′output(ID) is not equal to Undefined, then this occurrence of ID is a duplicate declaration.

5.2. Scan-attribution for nested scopes

With nested scopes, it is still possible to use scan-attribution to specify the annotation of the derivation tree with appropriate symbol-table values. In this section, it is convenient to fix on a concrete representation for symbol tables; we will use LISP-like lists whose elements are of the form CONS(ID, type) (*e.g.*, CONS(CONS(x, Integer), CONS(CONS(y, Boolean), NIL)) is an example of a symbol table in this representation). With this concrete representation, the operation $\otimes$ is $\lambda x . \lambda y$. APPEND(y, x) (where APPEND is the function that appends two lists), and its identity element e is NIL. Note that the way we express symbol-table construction in the case of a single global scope changes only slightly, as follows:

```
scan symbol_table (LR, [env] → [env], λx.λy.APPEND(y,x), NIL);
decl : Decl          {  decl[2].symbol_table'input = CONS(CONS(ID, type), NIL);  };
stmt : Assignment    {  stmt[0].symbol_table'input = NIL;   };
exp  : IdExp         {  exp[0].symbol_table'input = NIL;   };
```

To model nested scopes, we extend the abstract-syntax declarations for our example language with a block construct:

```
        stmt : Block(declList stmtList);
```

We also introduce two markers, denoted by BLOCK_ENTRY and BLOCK_EXIT, which will be used in symbol-table lists to bracket scopes that have been exited. In particular, the lookup function ignores all symbol-table entries when a BLOCK_ENTRY is encountered until the matching BLOCK_EXIT is found. BLOCK_ENTRY and BLOCK_EXIT are used in the scan-attribution equations for Block, as follows:

```
  stmt : Block   {  stmt[0].symbol_table'input = CONS(BLOCK_EXIT, NIL);
                    stmt[2].symbol_table'input = CONS(BLOCK_ENTRY, NIL);
                 }
        ;
```

There is one drawback to this approach: The symbol table associated with a given node in the derivation tree contains symbol-table entries for all declarations to the left of the node—*including declarations from scopes that are not active*. This will cause an application of the lookup function on identifier i to be much less efficient whenever it does not find a symbol-table entry for i in the local scope.

We would like the symbol-table list to include entries only for declarations in enclosing scopes. This can be arranged by filtering symbol-table lists, removing all symbol-table information between a matching BLOCK_ENTRY/BLOCK_EXIT pair. This is permissible because all information between a matching BLOCK_ENTRY/BLOCK_EXIT pair is ignored by the lookup function. In the attribute grammar for the language given above, such filtering can be done in the Sweep I equations for the symbol-table scan.

Remark. Technically what we have done is to change the meaning of the scan operation slightly: We are given a list $x_1, \cdots, x_n$ and an associative binary operator $\oplus$. The goal is to compute the list $y_1, \cdots, y_n$, such that $y_i = x_1 \oplus \cdots \oplus x_i$, for $1 \le i \le n$, where "$a = b$" now means "equal according to what is observable in a and b". For example, in the case of symbol-table lists, "$a = b$" means "equal according to what is observable in a and b via the lookup function".
End Remark.

5.3. Pragmatic performance considerations

Because symbol-table attributes are not unit-sized objects, the performance of the parallel scan-grammar evaluator will degrade seriously from the theoretical speedup of $2N/(2N/P + (2D + 1))$ if substantial amounts of symbol-table information must be passed from processor to processor. Consequently, the amount of symbol-table information that has to be passed between processors is an important pragmatic issue.

To solve this problem, it is necessary to consider the manner in which symbol-table data is accessed. In particular, note that symbol-table construction is carried out during the evaluation of the "symbol_table" scan-attribution; only later are the symbol-table attributes accessed in order to perform lookups. Furthermore, because of the value semantics of attribute grammars (*i.e.*, no side effects are permitted in attribute equations), the symbol-table construction phase involves the use of memory in a "write-once" fashion. When lookup operations are carried out, each (virtual) processor at a node of the derivation tree accesses the symbol table in a "read-only" fashion.

Thus, there are three key aspects to reducing the cost of communicating symbol-table values from processor to processor:

(1) The multiprocessor should provide a *shared-memory* abstraction.[6]

(2) The "handle" to a symbol table needs to be represented by a small value (such as a pointer to a location in shared memory).

(3) It must be possible to access a symbol-table entry with only a few accesses to shared memory.

For the case of a single global scope, one way to implement a symbol table that satisfies these conditions is to use a balanced tree (such as a 2-3 tree, B-tree, or AVL tree) that is updated applicatively (*i.e.*, the spine of the tree is copied on each insertion) [19,21]. With such structures, the only information that needs to be passed explicitly from processor to processor during a symbol-table construction scan is a pointer to the root of the tree. A symbol-table entry in a symbol table of size k can be accessed with $O(\log k)$ references to shared memory.

To handle nested scopes, we can use a list of balanced trees. Again, the only information that needs to be passed explicitly from processor to processor during a symbol-table construction scan is a pointer, in this case to the root of the list.

6. Combining scan-attribution with other *O(D)* processing patterns

While scan-attributions are convenient for specifying some attribution computations, it is clearly desirable to be able to combine them with attribute equations of a more standard sort (as we did in the examples in Section 2). There are three dependence patterns that mesh particularly well with the $2D + 1$ bounded behavior of scan-attribution:[7]

(1) Purely local attribution phases, during which some attribute instances of each production instance are given values.

(2) Purely bottom-up attribution phases, during which all information flows from the leaves toward the root.

(3) Purely top-down attribution phases, during which all information flows from the root to the leaves.

All three kinds of patterns can be evaluated in at most D steps (assuming N processors, where N is the number of production instances in the derivation tree); when only $P < N$ processors are available, both kinds of passes can be evaluated in at most $N/P + D$ steps.

Because we represent lists derived from list nonterminals as balanced complete binary trees, it is necessary to impose certain restrictions on the right-hand-side functions used in the attribute equations of these productions. In particular, in the rules for a bottom-up pass, the right-hand-side functions must be associative; in the rules for a top-down pass, the right-hand-side functions must be commutative.

---

[6]There are many approaches that have been used to provide shared memory on multiprocessors, either in software (*e.g.*, shared virtual memory on loosely coupled multiprocessors [18]) or in hardware (via snooping caches [3] or directory protocols [1]).

[7]By combining our evaluation techniques with previously developed ones for parallel attribute evaluation, it may be possible to create parallel attribute evaluators in which scan-attribution is combined with conventional attribute equations other than sets of equations that fall into the three patterns enumerated here.

The reasons why these restrictions need to be imposed can be seen with the following example. Suppose the attribute grammar contains the following definitions:

```
xList { synthesized T foo;
        inherited T bar;
      };
xList : Op(xList  xList)   {
            xList$1.foo = xList$2.foo ⊕ xList$3.foo;
            xList$2.bar = F(xList$1.bar);
            xList$3.bar = G(xList$1.bar);
        }
      ;
```

Only when operation $\oplus$ is associative can we guarantee that the xList.foo attribute at the root of Op(a, Op(b, c)) is equal to the xList.foo attribute of Op(Op(a, b), c). Without associativity, the attribution of lists represented as balanced complete binary trees would not be uniquely defined. Similarly, only when functions F and G commute (*i.e.* F(G(x)) = G(F(x)), for all x) can we guarantee that the various xList.bar attributes at the leaves of Op(a, Op(b, c)) are equal to the corresponding xList.bar attributes at the leaves of Op(Op(a, b), c).

7. Relation to previous work

Other work on parallel attribute evaluation includes [6, 13, 16, 25]. Kaplan and Kaiser present a distributed evaluator for the problem of incremental attribute updating in language-sensitive editors [13]. Boehm and Zwaenepoel describe an implemented parallel attribute-grammar evaluator that runs on a network multiprocessor of six SUN-2 workstations connected by an Ethernet network [6]. Zaring presents parallel algorithms for ordered attribute grammars [25]; he gives algorithms for both tightly coupled and loosely coupled multiprocessor architectures. The approach presented in this paper is much different from these previous approaches in two respects: (1) we focus on attribute specifications that employ a data-parallel construct; (2) the essence of our technique is to exploit associativity, which provides us with the freedom to re-group the computation, thereby making it more amenable to parallel processing.

Kuiper's work is somewhat closer to ours in spirit in that he makes use of an attribute-grammar transformation to restructure computations [16]. When his transformation is applicable, the attribute equations of the transformed attribute grammar performs a computation equivalent to the original attribute equations, but the derivation tree's dependency chains are shorter, which increases the amount of potential parallelism. However, Kuiper's transformation modifies the attribute equations only, leaving the underlying context-free grammar (and derivation trees) unchanged.

Kuiper's transformation applies to certain types of threadings over lists defined as right-recursive "combs" (*i.e.*, with a production of the form $xList \rightarrow x\ xList$). For a list of length k, it transforms the dependency chain of the threading from length $2k$ into two chains each of length k. In the case of the construction of a symbol-table attribute via a thread running through a list of declarations, Kuiper's transformation applies due to the fact that the list-concatenation operation he uses to specify the construction of a symbol table is an associative operator. A key difference between our work and Kuiper's is that for this important kind of attribute computation, Kuiper's transformation does not exploit the associativity property to the fullest. His transformation is less powerful than the idea introduced in this paper, which calls for transforming both the derivation tree and the attribute equations. In particular, with our technique a right-recursive comb is transformed into a balanced binary tree; the original dependency chain of length $2k$ is replaced by a dependency graph in which the length of the longest path is $2\log k + 1$.

In all previous work on parallel attribute evaluation, the speedups reported for Pascal attribute grammars have been relatively modest—in the range of 4 to 7 (having tailed off rapidly beyond about 16 processors).[8] One of the chief reasons for this is that the dependences among the attributes that control symbol-table processing form long chains, which are an obstacle to parallel processing. As described in Section 5, the data-parallel approach surmounts this difficulty because the creation of symbol tables can be expressed using the scan-attribution construct. As a consequence, symbol-table processing can be carried out in parallel in $2D + 1$ steps, where D is the depth of the derivation tree, assuming one processing element per production instance in the derivation tree. When only $P < N$ processors are available, where N is the number of production instances in the derivation tree, a scan-attribution can be evaluated in at most $2N/P + (2D + 1)$ steps. Figure 1 shows that, using our techniques, it may be possible to obtain speedups for Pascal programs on the order of 10-fold with 10 processors, 64-fold with 100 processors, and 100-fold with 250 processors.

References

1. Agarwal, A., Simoni, R., Horowitz, M., and Hennessy, J., "An evaluation of directory schemes for cache coherence," in *Proceedings of the Fifteenth Annual International Symposium on Computer Architecture,* (Honolulu, Hawaii, June 1988), (June 1988).
2. Aho, A.V., Sethi, R., and Ullman, J.D., *Compilers: Principles, Techniques, and Tools,* Addison-Wesley, Reading, MA (1986).
3. Archibald, J. and Baer, J.-L., "Cache coherence protocols: Evaluation using a multiprocessor simulation model," *ACM Trans. on Computer Systems* **4**(4) pp. 273-298 (November 1986).
4. Babich, W.A. and Jazayeri, M., "The method of attributes for data flow analysis: Part I. Exhaustive analysis," *Acta Informatica* **10**(3) pp. 245-264 (October 1978).
5. Blelloch, G.E., *Vector Models for Data-Parallel Computing,* The M.I.T. Press, Cambridge, MA (1990).
6. Boehm, H.-J. and Zwaenepoel, W., "Parallel attribute grammar evaluation," pp. 347-354 in *Proceedings of the Seventh International Conference on Distributed Computing Systems,* (Berlin, W. Germany), ed. R. Popescu-Zeletin, G. LeLann, and K.H. Kim,IEEE Computer Society, Washington, DC (September 1987).
7. Brent, R.P., "The parallel evaluation of general arithmetic expressions," *J. ACM* **21**(2) pp. 201-206 (1974).
8. Cormen, T.H., Leiserson, C.E., and Rivest, R.L., *Introduction to Algorithms,* The M.I.T. Press, Cambridge, MA (1990).
9. Farrow, R., "Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars," *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction,* (Palo Alto, CA, June 25-27, 1986), *ACM SIGPLAN Notices* **21**(7) pp. 85-98 (July 1986).
10. Hoover, R. and Teitelbaum, T., "Efficient incremental evaluation of aggregate values in attribute grammars," *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction,* (Palo Alto, CA, June 25-27, 1986), *ACM SIGPLAN Notices* **21**(7) pp. 39-50 (July 1986).
11. Jones, L. and Simon, J., "Hierarchical VLSI design systems based on attribute grammars," pp. 58-69 in *Conference Record of the Thirteenth ACM Symposium on Principles of Programming Languages,* (St. Petersburg, FL, Jan. 13-15, 1986), ACM, New York, NY (1986).
12. Jones, L.G., "Efficient evaluation of circular attribute grammars," *ACM Trans. Program. Lang. Syst.* **12**(3) pp. 429-462 (July 1989).
13. Kaplan, S. and Kaiser, G., "Incremental attribute evaluation in distributed language-based editors," pp. 121-130 in *Proceedings of the Fifth ACM Symposium on Principles of Distributed Computing,* (1986).
14. Kernighan, B. and Plauger, P., *Software Tools in Pascal,* Addison-Wesley, Reading, MA (1981).
15. Knuth, D.E., "Semantics of context-free languages," *Math. Syst. Theory* **2**(2) pp. 127-145 (June 1968).
16. Kuiper, M.F., "Parallel attribute evaluation," Ph.D. dissertation, Dept. of Computer Science, Utrecht University, Utrecht, The Netherlands (1989).
17. Leighton, F.T., *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes,* Morgan Kaufmann Publishers, Inc., San Mateo, CA (1992).
18. Li, K. and Hudak, P., "Memory coherence in shared virtual memory systems," *ACM Transactions on Computer Systems* **7**(4) pp. 321-359 (November 1989).

---

[8] Some of the speedup figures were established by measuring an implemented system [6]; others come from simulations [16,25].

19. Myers, E.W., "Efficient applicative data types," pp. 66-75 in *Conference Record of the Eleventh ACM Symposium on Principles of Programming Languages,* (Salt Lake City, UT, Jan. 15-18, 1984), ACM, New York, NY (1984).
20. Ofman, Y., "On the algorithmic complexity of discrete functions," *Soviet Physics Doklady* 7(7) pp. 589-591 (1963). English translation.
21. Reps, T., Teitelbaum, T., and Demers, A., "Incremental context-dependent analysis for language-based editors," *ACM Trans. Program. Lang. Syst.* 5(3) pp. 449-477 (July 1983).
22. Reps, T. and Teitelbaum, T., *The Synthesizer Generator: A System for Constructing Language-Based Editors,* Springer-Verlag, New York, NY (1988).
23. Skedzeleski, S.K., "Definition and use of attribute reevaluation in attributed grammars," Ph.D. dissertation and Tech. Rep. TR-340, Computer Sciences Department, University of Wisconsin, Madison, WI (October 1978).
24. Waite, W.M. and Goos, G., *Compiler Construction,* Springer-Verlag, New York, NY (1983).
25. Zaring, A., "Parallel evaluation in attribute grammar based systems," Ph.D. dissertation and Tech. Rep. 90-1149, Dept. of Computer Science, Cornell University, Ithaca, NY (August 1990).