Parallel Attribute Grammar Evaluation

Hans-Juergen Boehm
Willy Zwaenepoel

Department of Computer Science
Rice University
Houston, Texas

Abstract

This paper reports on experiments with parallel compilation of programming languages. In order to take advantage of the potential parallelism, we express the language translation process as an *attribute grammar* evaluation problem. We see three primary benefits to using attribute grammars: First, since attribute grammars provide a *functional* specification of the language translation process, they are easily amenable to parallel implementation, with relatively little synchronization overhead. Second, as a high-level specification of the language, they allow parallel translators to be produced *automatically*, relieving the compiler writer from the burden of dealing with parallelism. Third, they provide a basis for a wide variety of language translation problems, ranging from traditional programming language compilation to more ambitious problems such as proof checking, text formatting, etc.

We study the efficiency and the potential for parallelism of various attribute grammar evaluation methods and we present the design of a "combined" evaluator, which seeks to combine the potential for concurrency of dynamic evaluators and the (sequential) efficiency of static evaluators. We have used our methods to generate a parallel compiler for a large Pascal subset. Measurements on a network multiprocessor consisting of up to 6 SUN-2 workstations connected by an Ethernet network indicate that the parallel compiler outperforms its sequential counterpart by a factor of up to 3, with sequential compilation times and quality of produced code comparable to commonly available compilers.

1 Introduction

We are interested in speeding up the language translation process by exploiting parallelism. We take a fairly broad view of the phrase "language translation" to include not only traditional programming language compilation but also text formatting, proof checking [1], assembling, and various other software tools that can be viewed as implementing the translation of a context free language. We are concentrating on the "semantic" phase of the translation process, rather than on scanning and parsing, since most modern compilers (should) spend relatively little time parsing [14].

In order to take advantage of the potential parallelism, we express the language translation process as an *attribute grammar* evaluation problem (see Section 2). We see three primary benefits to using attribute grammars: First, since attribute grammars provide a *functional* specification of the language translation process, they are relatively easily amenable to parallel implementation. Second, as a high-level specification of the language, they allow parallel translators to be produced *automatically*, relieving the compiler writer from much of the burden of dealing with parallelism. Finally, they allow a wide variety of language translation problems to be specified.

Broadly speaking, traditional attribute grammar evaluation methods can be divided in two categories: dynamic and static evaluation. In essence, static evaluators are more efficient on a sequential machine, both in terms of CPU time as well as memory utilization, while dynamic evaluators have a higher potential for concurrency. We present the design of a combined static/dynamic evaluator which seeks to combine the potential for concurrency of dynamic evaluators with the sequential efficiency of static evaluators.

We have measured the performance of parallel evaluators on a network multiprocessor (both combined evaluators as well as purely dynamic ones) and compared their performance to sequential evaluators. The parallel combined evaluator outperforms the sequential evaluator by a factor of up to 3, and consistently outperforms the parallel dynamic evaluator. The sequential compilation speeds and quality of the produced code are comparable to commonly available Pascal compilers. Good sequential and parallel performance is achieved through several optimizations, including very fast memory allocation, efficient applicative symbol table updates, and the use of a string librarian for efficient distributed string handling.

The outline of the rest of the paper is as follows. In Section 2 we detail our approach, including a short introduc-

This research was supported in part by the National Science Foundation under grants DCR-8511436 and DCR-8607200 and by an IBM Faculty Development Award.

tion to attribute grammars and attribute grammar evaluation methods. Section 3 describes the experimental setting. Section 4 presents a detailed account of our current measurement results and discusses some of the efficiency techniques used in our implementation. Related work is covered in Section 5. Finally, in Section 6 we draw some conclusions and explore some avenues for further work.

2 Approach

2.1 Structure of the Parallel Compiler

Our parallel compiler consists of a sequential parser and of a number of attribute evaluators executing in parallel on different machines. The parser builds the syntax tree, divides it into subtrees, and sends them to the attribute evaluators. The attribute evaluators then proceed with the actual translation by evaluating attributes belonging to the symbols in their subtree. In the process some attribute values are communicated to other evaluators. The evaluators may have to wait to receive attribute values from other evaluators before they can proceed. We now briefly describe the nature of the attribute evaluation process.

2.2 Attribute Grammars

Attribute grammars were introduced by Knuth to specify semantics of context free languages [11]. Each node in the parse tree of a sentence has a collection of associated attribute values. *Semantic rules* associated with each production specify the values of the attributes of nonterminals in a given production in terms of the values of other attributes of symbols in the same production. Together these semantic rules define the values of the attributes of all symbols in the parse tree. [1] The process of computing all attribute values associated with a parse tree is referred to as attribute evaluation. This is normally performed by an *attribute evaluator* which can be constructed automatically from the attribute grammar specification. The appendix gives a simple attribute grammar that defines the value of arithmetic expressions augmented with constant declarations.

If an attribute grammar is used to specify a compiler, only the attribute values at the root of the parse tree are of interest. The root attributes normally include the machine language code for the program, as well as a list of any "semantic" errors encountered in the translation process. The attributes of other nodes represent intermediate results used in the computation of the root attributes. For instance, the code attribute of the root is produced by properly concatenating the code attributes of its children in the parse tree.

[1] Attributes of terminals are normally predefined by the lexical analyzer, though this is not part of Knuth's original formalism.

Several conventional (sequential) compilers have been constructed using attribute grammars [7]. Probably a more common use has been in conjunction with syntax directed editors [15]. The applicative (functional) nature of an attribute grammar specification (i.e., the fact that the semantic rules mentioned above must be pure functions with no visible side-effects) minimizes the constraints on the evaluation order of individual attributes. The remaining constraints are readily apparent to an evaluator generator. This makes it feasible to construct the incremental evaluators necessary in an editing environment. It is this same observation that makes attribute grammars particularly well suited to efficient parallel evaluation. The relatively unconstrained evaluation order keeps synchronization overhead to a minimum.

We now describe two kinds of evaluation methods, dynamic and static evaluation (for a more detailed survey, see [6]).

2.3 Dynamic and Static Attribute Evaluation

Given a parse tree, a dynamic evaluator first computes the *dependency graph* between the attributes of all symbols in the tree. This dependency graph is constructed by making the attribute appearing on the left hand side of a semantic rule dependent on the attributes on the right hand side necessary for its evaluation. As is conventional, we restrict our attention to grammars for which the resulting dependency graph is acyclic. The graph is topologically sorted, and attributes are evaluated as they become ready in the topological sort, until all attributes are evaluated (see Figure 1).

Parallelizing this scheme is rather straightforward. Each of the evaluators builds the dependency graph for its subtree, marking attributes to be computed in other evaluators as unavailable, does a topological sort, and starts evaluation. In doing so, it may have to wait for some of the remotely computed attributes. When they arrive, the dependency graph is updated as appropriate. In addition, each of the parallel evaluators must communicate some of the locally computed attributes to other evaluators. While this method achieves a high degree of concurrency, it is expensive in terms of space and CPU usage because of the time and the storage necessary to compute and store the

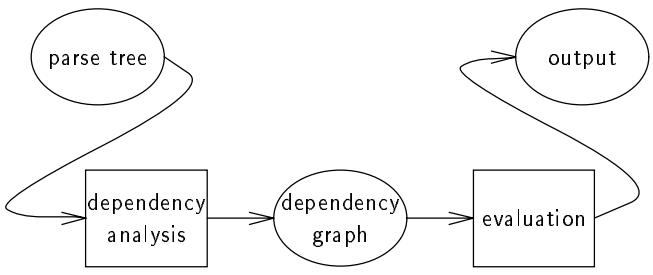

FIGURE 1: Operation of Dynamic Evaluator

dependency information.

State-of-the-art sequential evaluators normally avoid these problems by the use of "static" evaluation techniques. With these techniques a prepass is made over the grammar, whereby an order is computed in which attributes (of any parse tree) can be evaluated consistent with the dependencies of the grammar (see Figure 2). At evaluation time, attributes are evaluated in this precomputed order, without having to perform any dependency analysis at evaluation time (see Figure 3). Such an evaluator usually takes the form of a collection of mutually recursive *visit* procedures, one per production, which are used to walk the parse tree according to the precomputed order. We use Kastens' ordered evaluation method throughout this paper as the example of a static evaluator, as it is fairly efficient and capable of dealing with a large class of grammars [9].

Unfortunately, since these static evaluators rely on a predetermined order of computing the different attributes, and since this order has been determined under the assumption that the entire parse tree can be visited by the evaluator, it is much less obvious how to adapt a static evaluator to a parallel environment, where only part of the parse tree is available to any evaluator. While such an adaptation is perhaps feasible, we have chosen to construct a combined static/dynamic evaluator which tries to combine the potential for concurrency in the dynamic evaluator with the lower CPU and memory usage of the static ordered evaluator.

2.4 The Combined Evaluator

The basic idea is to perform dynamic evaluation *only* for those attributes belonging to tree nodes on the path from a remotely evaluated leaf to the root of the local subtree, and to use static evaluators for all other attributes (see Figure 4). During the reconstruction of the syntax subtree from the linearized form received over the network, we determine for each node N whether it is on a path from the root to a separately processed subtree. If not, N is to be evaluated statically and no dependency information is computed. Otherwise, we inspect N's children to see if any of them should be evaluated statically. If so, we enter the (transitive) dependencies between the child's attributes (as precomputed by the static evaluator generator) into the dynamic dependency graph. We then add the dependencies generated by the semantic rules of the production at node N to the dependency graph, as for dynamic evaluation. When the tree construction is completed, evaluation starts in topological order, as for dynamic evaluators. When all predecessors for a statically evaluated attribute become available, the appropriate static visit procedure is invoked.

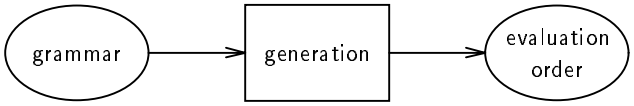

FIGURE 2: Construction of Static Evaluator

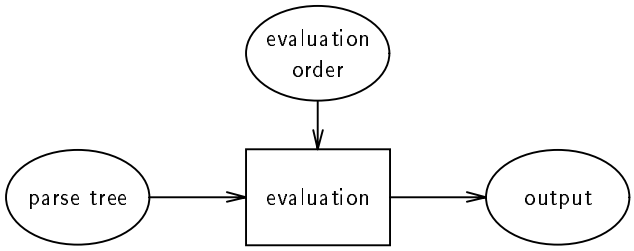

FIGURE 3: Operation of Static Evaluator

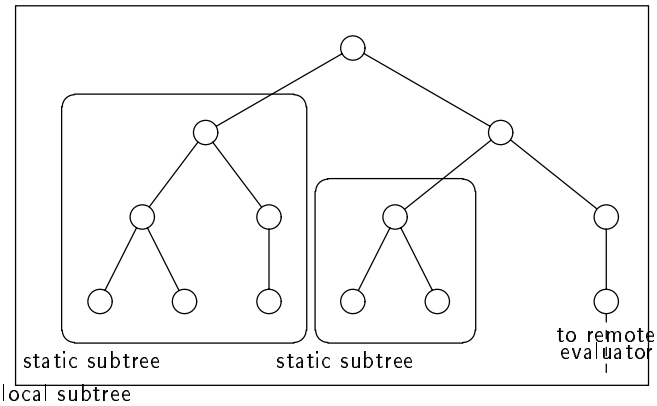

FIGURE 4: Operation of Combined Evaluator

As will be seen in the Section 4, this results in the vast majority of attributes being evaluated statically. In particular, all "bottom" subtrees are evaluated entirely statically. There is however some loss of concurrency because static evaluation, by preselecting an evaluation order, effectively introduces additional dependencies not inherent in the grammar.

2.5 The Compiler Generator

Both the parser and the parallel evaluators are generated automatically from a single attribute grammar specification. The attribute grammar is specified in a conventional manner, except that we require the following additional information (see also the appendix for an example):

1. The attribute grammar specifies at which nonterminals the syntax tree may be split, and the minimum size of the subtree to be evaluated separately. This size can be scaled by a runtime argument to the parser, to allow for easy experimentation with decompositions with different granularities.
2. For attributes of tree nodes at which the tree can conceivably be split, conversion functions must be specified. These convert between the internal representation of the attribute and a contiguous representation suitable for transmission over a network.

3 Experiments

We have generated sequential and parallel evaluators for a sizable Pascal subset. All control constructs except with statement and goto statements are included, as are value and reference parameters and most standard data types. Variant records, enumerated types, sets, floating point, file I/O, and functions and procedures as parameters are omitted or severely restricted. VAX assembly language is produced. A limited amount of local optimization is performed. The overall code quality is at least comparable to that produced by the Berkeley UNIX Pascal compiler. The attribute grammar currently contains 131 context-productions and 1117 semantic rules. Parse trees can be split at statement nodes, statement list nodes, procedure declaration nodes, and lists of procedure declarations.

The experiments are run on a collection of SUN-2 workstations connected by a 10 megabit Ethernet. During the experiments the machines are exclusively used by the programs involved in the experiment. The machines are running the V-System, an experimental message-based operating system developed at Stanford University [3]. Interprocess communication is by means of messages and is transparent (i.e., independent of the location in the network of the communicating processes).

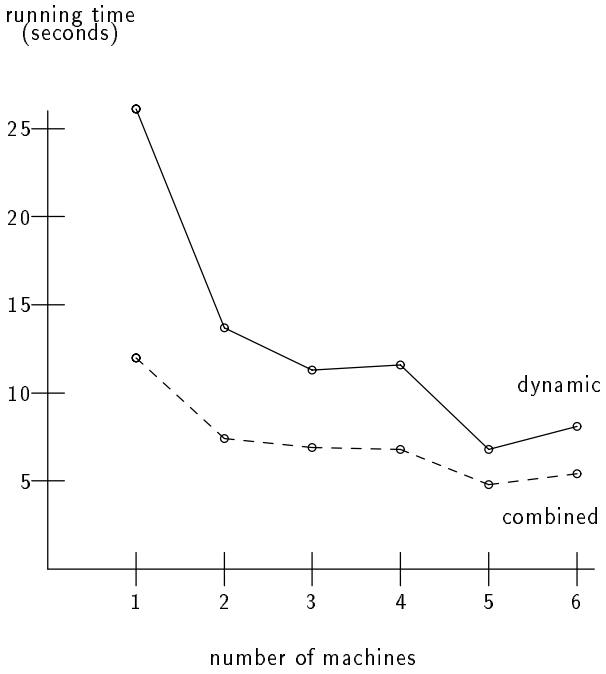

FIGURE 5: Evaluator Running Times

4 Measurements

We have compiled and measured several programs using the dynamic and the combined evaluators, both sequentially and in parallel (The sequential combined evaluator is essentially identical to a purely static sequential evaluator, since the entire tree is processed locally). Here, we give measurements for compiling a compiler and interpreter for a simple language used in our compiler course. The program is about 1,000 lines long, contains 33 procedures, 5 of which are at a nesting level deeper than 1. The assembly language program is 60 kilobytes long. These results reported here are typical for compilations of programs of that size.

4.1 Running Time

Figure 5 shows the running times of the dynamic and the combined evaluators, when using from 1 up to 6 machines. Running time is measured from the time the parser initiates evaluation until it receives back the root attributes. The parallel combined evaluator running on 5 machines is approximately 3 times faster than the sequential version. The parallel dynamic evaluator running on 5 machines is achieves a speedup factor of 4 over the sequential version. Within the bounds of these experiments, the combined evaluator performs consistently better than the purely dynamic evaluator, although the differences become less outspoken as the number of machines increases. Detailed analysis of the behavior of the combined evaluator reveals that on average less then 0.1 percent of the attributes are evaluated dynamically. Hence, the superior efficiency of static evaluation — without any need for dynamic dependency analysis — supersedes the increased potential for concurrency in dynamic evaluators, especially for small numbers of machines. A caveat needs to be added here: Although static evaluators of the type used here can accommodate most common programming language constructs, dynamic evaluators can handle a wider variety of languages. Hence, in some circumstances it might be necessary to resort to a dynamic evaluator, regardless of performance considerations.

The sequential running time of both evaluators compares favorably to the running time of commonly available Pascal compilers running on identical hardware, with comparable code quality being produced. Compilation of the example program on a SUN-2 using the vendor supplied compiler takes 34 seconds without assembly and 57 seconds including assembly. For proper comparison, parsing time must be added to the running time of our evaluators. Our parser takes about 3 seconds for the above program. A more efficient implementation could reduce parsing time significantly, for instance by using the techniques described in [14]. It must also be taken into consideration that our compiler only implements a subset of Pascal, although only two currently unimplemented language constructs appear to contribute to compilation time, namely operator overloading and proper treatment of *write, writeln,* and the like, which are currently treated as keywords.

The current attribute grammar specifies translation from Pascal to VAX assembly language. Assembly can be specified as a separate attribute grammar, which can be run as a separate parallel pass after compilation. Alternatively, assembly can be integrated into the current

grammar, with the assembly process being decomposed in the same way as compilation and with a linking phase at the end. This approach has the additional advantage that machine language is much more compact than assembly language, resulting in smaller attributes being transmitted over the network. Given the relative importance of assembly in the aforementioned compilation and assembly times for the SUN Unix Pascal compiler, it seems highly desirable to include assembly into a parallel compiler.

The running time of the parallel evaluator does not decrease monotonically with increasing number of machines. The "best" performance is obtained by using five machines. The decomposition obtained for five machines results in subtrees of about equal size being passed to the evaluators. This intuitively results in good concurrent behavior since all evaluators run (in parallel) for approximately the same amount of time. Using six machines results in a more uneven decomposition with little increased concurrency, but with additional overhead involved in using the sixth machine. We now study the behavior of the parallel combined evaluator in more detail.

4.2 Behavior of the Parallel Evaluator

Figure 6 shows the behavior of the parallel combined evaluator when running on 5 machines. The source program is decomposed into subtrees for separate evaluation as shown in Figure 7. In Figure 6 horizontal lines represent the activity of the individual evaluators and the string librarian (see Section 4.3), with thin lines indicating idle periods and thick lines indicating active periods. The time axis runs from left to right, and the arrows indicate communication of attributes between the corresponding evaluators. As can be seen clearly from Figure 6, symbol table generation and propagation is essentially sequential, while good concurrency is achieved during the code generation phase. The final phase, result propagation, during which the evaluators propagate their result attributes back to the parser, is discussed in more detail in the next section.

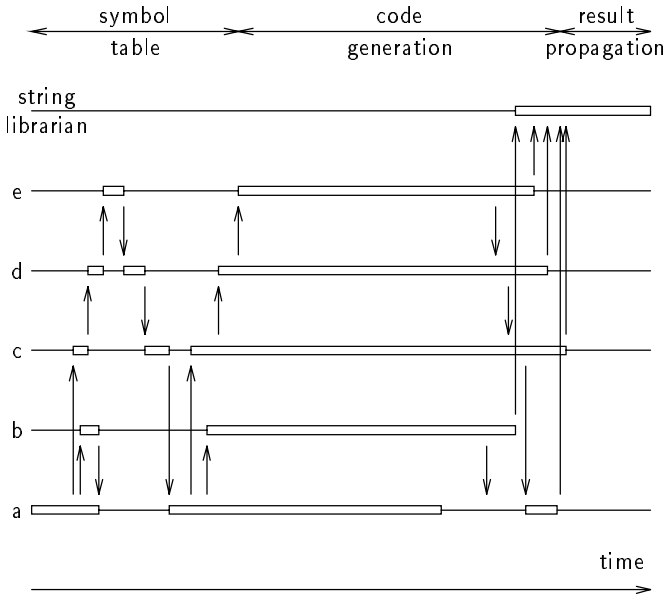

FIGURE 6: Behavior of Combined Evaluator

4.3 Efficiency Techniques

Major concern has been devoted to the efficiency of the sequential code. Symbol tables are implemented as binary search trees, making *applicative* updates simple and fast. Symbol table entries map the hash table index of an identifier to the information associated with that identifier. This insures that key values are essentially uniformly distributed, and thus symbol table trees stay balanced (see also [5] for an alternative technique). Strings are implemented as binary trees, with the actual text residing in the leaves. Thus, string concatenation is a constant time operation. Storage allocation is extremely fast throughout, since we make no provision for reusing memory.

In order to achieve good performance during the result propagation phase of parallel compilation, we introduce the concept of a *string librarian* process. When an evaluator computes its final code attribute, it sends the code

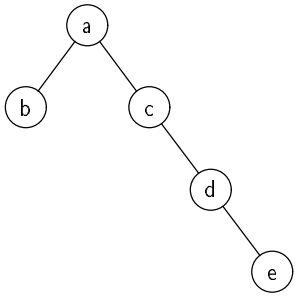

FIGURE 7: Source Program Decomposition

string to the string librarian process, and a string descriptor to its ancestor. The descriptors are combined appropriately by every process in the process tree and finally passed up from the root evaluator to the string librarian, which combines the code attributes according to the information in the descriptors. This technique results in a *single* network transmission of the code attribute resulting from each evaluator. Additionally, these transmissions proceed largely in parallel, thereby reducing their effect on running time even further. A naive implementation, whereby each evaluator passes up the entire code string to its ancestor, leads to major inefficiency. When the "bottom" evaluator computes its final code attribute, it is transmitted to its ancestor process, where it is concatenated with the code produced there, transmitted again to that process' ancestor, and the same scenario is repeated until the code attribute finally reaches the root evaluator. This results in (large) code attributes being transmitted over the network as many times as the depth of the process tree. Additionally, since at every stage the locally generated code is concatenated with the code received from "below", this process is strictly sequential. The use of a string librarian process results in approximately 1 second improvement in running time (or approximately 20 percent). Note that this optimization can be done without changing the grammar or the evaluator generator. All that needs to be changed is the implementation of the "standard" string data type used for code attributes or, more precisely, the conversion function for the root node's code attribute (see Section 2.5).

A similar but less substantial inefficiency exists with the propagation of the global symbol table at the end of the first phase of execution. Construction of the symbol table currently involves sequential propagation of several versions of the symbol table up and down the tree. This could be improved substantially by tuning the grammar.

We allow certain attributes can be marked as "priority" attributes (such as the global symbol table). These attributes are evaluated as soon as they are available. This guarantees that these attributes become available quickly and are propagated immediately to other evaluators. Without priority attribute specifications, pathological situations can occur whereby local attributes are computed ahead of attributes that are required globally.

It is often necessary to generate unique identifiers, for instance for use as labels in a program. In a sequential attribute grammar, this is often done by propagating a single attribute throughout the tree whose value is then incremented each time a new unique identifier is required. If this technique were used in the parallel evaluator, it would require virtually all evaluators to wait for the value of this attribute to be propagated. Instead, a unique value is communicated by the parser to each evaluator, and unique identifiers within that evaluator are then generated relative to this base value.

5 Related Work

Some work has been done on parallel compilation (cf. [12]). However, many approaches suffer from the lack of a solid formal underpinning as provided by attribute grammars. Also, much work has concentrated on parallel parsing (cf. [4]). We believe that in most environments, the cost of parsing is less significant than the cost of the latter phases of compilation. Presumably, some of the parallel parsing methods can be used in conjunction with the techniques presented here, if parsing time does become a significant factor.

An alternative approach to parallelizing compilation consists of pipelining the compilation process [2, 8, 13]. While this approach is appealing in that many existing compilers are written as a pipeline of processes, the speedup that can be achieved by executing different stages in parallel is limited by the number of stages in the pipeline (which is usually rather small) and by dependencies between the data produced by the different stages. Our attempt at parallelizing the portable C compiler in this way shows speedups limited to 1.3 [8]. Baer and Ellis, and Miller and Leblanc report projected or measured speedups in the range of 2 to 3 for a more fine-grained pipeline. Parallelizing several compilations can be done by using a parallel version of the Unix make facility. If the compilations are sufficiently independent, this can potentially lead to significant speedups. However, the approach suffers from differences in size between compilations, and from a sequential linking phase at the end.

A distributed *incremental* parallel attribute grammar evaluator is proposed by Kaplan and Kaiser [10]. In essence, they propose to evaluate in parallel all attributes that become "ready" at any time during the evaluation. No implementation or performance results are reported. We believe their approach is more appropriate in an environment where communication is very cheap and individual attribute evaluations are very expensive. This is not the case for our prototype system.

Unlike much of the recent work in attribute grammars, we have chosen to look initially at complete evaluation of all attribute instances in a tree, as opposed to incremental reevaluation of a few attributes after a change to the tree. There are three reasons for this. First, we hope the tradeoffs will be clearer by first focusing on the simpler problem. Second, the incremental algorithms are easily applicable only in the context of a structure editor, and it is not yet clear that this will be the preferred editing environment for a parallel compiler. Finally, experience with structure editors tends to indicate that fully attributed parse trees are too big to store over long periods. Unfortunately, secondary storage costs have not decreased sufficiently to expect this problem to disappear. Thus, even a structure editor based, highly incremental environment is likely to require a fast "batch" evaluator.

6 Conclusion and Avenues for Further Work

Attribute grammars are an appealing paradigm for specifying languages in a way that is amenable to parallel translation. They allow efficient parallel evaluators to be generated automatically, for a variety of languages, thereby relieving the compiler writer from the burden of dealing with parallelism. The functional nature of attribute grammars intuitively lends itself well to parallel implementation, since it reduces the amount of synchronization overhead.

We have detailed the design of a combined evaluator which seems to combine sequential efficiency with a high potential for concurrency. In our experiments so far, such a combined evaluator has proven to have performance superior to purely dynamic parallel evaluators as well as sequential static evaluators. We have also pointed some of the possible pitfalls resulting from a straightforward parallel implementation of attribute grammars. In particular, optimizations are needed to prevent long chains of transmissions of large attributes, such as the generated code.

Besides further experimentation with the current language, we have two longer term goals. First, we intend to study the integration of substantial (global) optimization techniques in the compiler, since they tend to be computationally expensive. The challenge is to specify those techniques in an attribute grammar framework without performing all computation at the root of the tree (and hence not gaining any concurrency). Second, we intend to port some of the attribute grammars available as part of the Cornell Program Synthesizer to our system. We are particularly interested in grammars in which the evaluation of individual attributes is very expensive relative to the cost of communicating attribute values between machines (such as for instance the proof checker described in [1]). Such grammars should derive most benefit from parallel evaluation.

A Appendix

The following attribute grammar specifies the value of expressions involving addition and multiplication. An identifier can be introduced and bound to a constant by means of the *let* construct. The expression

$$\text{let x} = 3 \text{ in } 2 + 4 * \text{x ni}$$

can be read as "the sum of 2 and 4 times x, where x = 3". The value of the expression is 14.

The syntax used for the grammar below is exactly the one used by our evaluator generator. (The syntax is based on that of YACC. The approach to semantic specification however is completely different. We do use YACC to produce the parser for our system.)

```
%name IDENTIFIER NUMBER      # terminals
%keyword LET IN NI           # keywords
%nosplit [value] [stab] expr # nonterminals
%nosplit [value] [] main_expr
%split 10 [value] [stab(st_put st_get)] block
%start printn main_expr      # start symbol
%left '+'                    # associativity
%left '*'
%%
main_expr: expr;
        $$.value = $1.value,
        $1.stab = st_create().

expr: expr '+' expr;
        $$.value = $1.value + $3.value,
        $1.stab = $$.stab,
        $3.stab = $$.stab.

expr: expr '*' expr;
        $$.value = $1.value * $3.value,
        $1.stab = $$.stab,
        $3.stab = $$.stab.

expr: IDENTIFIER;
        $$.value = st_lookup($$.stab, $1.string).

expr: block;
        $$.value = $1.value,
        $1.stab = $$.stab.

block: LET IDENTIFIER '=' expr IN expr NI;
        $$.value = $6.value,
        $4.stab = $$.stab,
        $6.stab
          = st_add($$.stab, $2.string, $4.value).

expr: NUMBER;
        $$.value = $1.string.
```

Declarations precede the "%%". We distinguish between 2 kinds of tokens, and 2 kinds of nonterminals. A "%keyword" declaration declares tokens with no further associated information. Tokens declared using "%name" have an associated attribute value that is calculated by the scanner.

The nonterminal "block" is declared to have a single synthesized attribute "value" and a single inherited attribute "stab". Instances of the "stab" attribute can be flattened to a sequential representation (for transmission over the network) by the "st_put" function. The function "st_get" performs the inverse operation. Subtrees of the parse tree headed by a "block" nonterminal may be split off and processed separately if their representation is at least 10 bytes long.

The other two nonterminals "expr" and "main_expr" are declared using "%nosplit" indicating that they should not serve as the root of a separately processed subtree. The "%start" declaration specifies the start symbol of the context free grammar, as well as a function to be called with the final attribute values of the root node. (In this case the function "printn" would presumably just print its ar-

gument, namely the value of the entire expression). The "%left" declarations are passed through to the parser generator to indicate the precedence and associativity of "+" and "*".

The main part of the specification follows the "%%". Each group of lines consists of a context-free production and some associated semantic rules. The left side of a context-free production is separated from the right side by a ":". The notation "$*i*.x" denotes the attribute x of the *i*th symbol on the right side of the production. To refer to an attribute of the nonterminal on the left, we use "$$.x". For example, the rule

$6.stab = st_add($$.stab, $2.string, $4.value)

following the production

block: LET IDENTIFIER '=' expr IN expr NI;

states that the symbol table attribute of the second "expr" nonterminal (the body of the block) is the value obtained by applying the "st_add" function to the symbol table associated with the left side (i.e. the parent in the parse tree), the "string" attribute of the identifier (computed by the scanner), and the value attribute of the first "expr" nonterminal.

The "st_add" function is expected to return a symbol table identical to its first argument, except that the identifier specified as the second argument is bound to the value specified by the third argument. The function "st_lookup" returns the binding of an identifier in a symbol table. "St_create" returns an empty symbol table. These functions might be supplied by a standard library of symbol table routines, as might "st_put" and "st_get". They are written in a standard programming language and trusted not to produce any visible side effects. Aside from these function definitions, the attribute grammar is complete, and has been used to generate parallel expression evaluators.

References

[1] B. Alpern and T. Reps, "Interactive Proof Checking", *Proceedings of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, 1984.

[2] J.-L. Baer and C. S. Ellis, "Model, Design, and Evaluation of a Compiler for a Parallel Processing Environment", *IEEE Transactions on Software Engineering SE-3,* 6, pp. 394-405 (1977)

[3] D. R. Cheriton and W. Zwaenepoel, "The Distributed V Kernel and its Performance on Diskless Workstations", *Proceedings of the Ninth ACM Symposium on Operating System Principles*, pp. 128-140 (1983).

[4] J. Cohen and S. Kolodner, "Estimating the Speedup of Parallel Parsing", *IEEE Transactions on Software Engineering, SE-11*, 1, pp. 114-124 (1985).

[5] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R.E. Tarjan. "Making Data Structures Persistent", *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing* (1986).

[6] J. Engelfriet, "Attribute evaluation methods", *Methods and Tools for Compiler Construction*, B. Lorho, Cambridge University Press, pp. 103-138 (1984).

[7] R. Farrow, "Experience with an Attribute Grammar Based Compiler", *Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pp. 95-107 (1982).

[8] D. B. Johnson and W. Zwaenepoel, "Macropipelines on a Network of Workstations", unpublished.

[9] U. Kastens, "Ordered Attribute Grammars", *Acta Informatica 13*, pp. 257-268 (1980).

[10] S. M. Kaplan and G. E. Kaiser, "Incremental Attribute Evaluation in Distributed Language-Based Environments", *Proceedings of the Fifth Annual ACM Symposium on the Principles of Distributed Computing*, pp. 121-131 (1986).

[11] D. E. Knuth, "Semantics of Context-Free Languages", *Mathematical Systems Theory 2,* pp. 127-145 (1968).

[12] D. E. Lipkie, "A Compiler Design for a Multiple Independent Processor Computer", Ph. D. Dissertation, University of Washington, Seattle (1979).

[13] J. A. Miller and R. J. Leblanc, "Distributed Compilation: A Case Study", *Proceedings of the Third International Conference on Distributed Computing Systems,* pp. 548-553 (1982).

[14] T. J. Pennello, "Very Fast LR Parsing", *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pp. 145-151 (1986).

[15] T. Reps, T. Teitelbaum, and A. Demers, "Incremental Context Dependent Analysis for Language Based Editors", *TOPLAS 5*, 3, pp. 449-477 (1983).