Annales Univ. Sci. Budapest., Sect. Comp. 17 (1998) 367-380

ATTRIBUTE GRAMMAR APPLICATIONS

J. Toczki and L. Schrettner (Szeged, Hungary)

Abstract. In this paper we summarize our research on an application field of attribute grammars: a compiler-compiler generating parallel compiler from formal specifications is described. A system processing Latin medical texts is also mentioned.

1. Introduction

In this paper we summarize our research on an application field of attribute grammars.

W-SQL machine developed at the University of Sheffield is a transputer based computer system which is used for parallel processing of large databases. Executing queries written in several extended versions of SQL and OQL language we can find typical formal language processing problems. The transputer architecture is used in this situation, so we considered the problem of parallel compilation on distributed systems.

Several parallel attribute evaluation methods are known from the literature. They benefit from the fact that independent attribute instances can be evaluated in parallel. Static tree and attribute based distributors [4] and dynamic dependency based distributors [7] are known.

In the case of most practical programming languages none of these distributors are efficient alone. Using static distributors, the most serious problem is that the size and evaluation time of attribute instances usually differ from

---

This research is involved in the research project "Large Parallel Databases" financed by the European Community, project number 93: 6638; partly supported by the research found OTKA, grant number F12852 and grant number F012853.

each other. On the other hand the author of the attribute grammar has some knowledge on these costs.

We have prepared the specification and detailed design of a compiler-compiler system generating parallel compilers. The distribution strategy is specified by the user, the generated compiler uses a dynamic load balancing algorithm.

Another possible application field is also summarized. It is well-known that attribute grammars can efficiently be used for specifying semantics of formal languages, while understanding of natural languages usually needs more complicated models. However there are some technical languages using only restricted structures from a natural language. For example such a field is the pure Latin medical terminology used in recipes, diagnoses and some other texts. We developed two attribute grammars specifying drugstore recipes and pathological diagnoses. The semantic parsers prepared from these attribute grammars can be used in educating medical students, in automatic coding and processing medical texts and diagnoses. These applications are under development.

In the next section we describe the structure of our parallel compiler generator. Some sentences on further research can be found in the last section.

2. A parallel compiler generator

2.1. Preliminaries

The first hand-written parallel compilers were developed in the early 1970s. Most of these compilers run on vector processors and based on parallel execution of certain phases of compilation. The first significant investigation into parallel semantic evaluation was made by Schell [11]. Schell's method – in essence – is the same as Jordan's one reported in [4].

The most natural way to build a parallel compiler is to run different compilation phases as separate processes and form a pipeline. The maximal possible speedup is the number of phases (usually 3 or 4). However, it is a hard work to balance different phases. Miller and LeBlanc compared sequential and pipeline versions of a Pascal compiler having 4 phases and they got 2.5 speed up as an average [9]. This result shows the limitations of pipelining.

Another possible way to construct a parallel compiler is to split the source program into smaller independent parts and compile these parts concurrently.

Lipkie was the first who suggested the combination of pipelining with source fragmentation [8]. Vandevoorde [17] and Seshardi [12] used the same approach developing compilers running on different architectures. Seshardi investigated the concurrent processing of declarations as well.

These experience show the importance of pipelining as well as the necessity of concurrent semantic evaluation. In this paper we concentrate on the semantic evaluation phase. Concerning lexical and syntactic parsing phase, pipelining with immediate fragmentation seems to be a proper solution. We also concentrate on more general methods which can be used in an automatic compiler development tool.

Compiler-compiler systems generate executable compilers from formal specifications. Most of recent compiler writing systems are based on attribute grammars. Experiences with compiler construction proved the feasibility and efficiency of attribute grammars for compiler specification. A survey of attribute grammar based compiler generation can be found in [3].

The compiler generator system *PROF-LP* [10] developed in Szeged has been used for generating various practical compilers, for example [14], [2]. We refer to our experiences at appropriate places in the next section. We mention that these experiences and directions of development in the case of sequential compilation are summarized in [15].

Now we summarize some preliminary results on parallel attribute evaluation.

Usually there are some *independent* attribute instances in a syntax tree. In the case of sequential evaluation a linear order is constructed, evaluating independent attribute instances in a more or less *ad hoc* order. In general, it is possible to evaluate independent attribute instances in parallel.

Kuiper [7] defined the concept of *distributor* as an algorithm to distribute attribute instances among evaluation processes. He defines two basic types of distribution:

- A *tree based* distributor allocates all attribute instances of a subtree of the syntax tree to the same evaluation process. The syntax tree is splited at selected nodes. Selected nodes determined by the production applied at the node – *production based distribution* – or by the left hand side nonterminal of that production – *nonterminal based distribution*. The distribution can be either nested or non-nested. In the case of *nested distribution* subtrees containing selected nodes are splited again, while in the case of *non-nested distribution*, the syntax tree is splited only at the selected nodes closest to the root.

A typical application of tree-based distribution is the fragmentation of a block-structured programming language. Disjoint blocks are usually independent of each other. We can allocate attribute instances of different blocks to different processes using a nested nonterminal based distributor.

- An *attribute based distributor* allocates all instances of an attribute to the same process. The distributor cannot distinguish between different instances of an attribute. It means a strict limit on potential parallelism. If we combine it with a tree based distributor, we get a *combined distributor*.

A typical application of attribute based distribution is to allocate independent tables of a compiler to different processes. For example, symbol tables and label tables are usually independent.

Jordan introduced a third kind of distributors.

- A *dependency based distributor* allocates all attribute instances of a connected part of the dependency graph to the same process. The allocation is not predefined. An evaluation order containing parallel execution of new processes is generated from the dependency graph. In this sense this method is more "dynamic" than Kuiper's distributors.

Dependency based distributors are capable of handling more complicated situations, when neither tree based nor attribute based distributions are inefficient.

2.2. Parallel compilation on transputers

The most important feature of transputers from the point of view of attribute distribution is that there is no shared memory available. Although it is possible to run more than one process on the same processor, we should allocate them to as many different processors as possible to increase "real" parallelism of the compiler. As inter-processor communication is the most expensive task on transputers, we should decrease the amount of data sent between processes allocated to different processors.

Some attributes – e.g. symbol tables – are extremely large, while others are very small. Some attributes have the same or similar meaning. For example most of the tables of compilers are represented with a pair of a synthesized and an inherited attribute. Usually tables are stored in dynamic data structures, i.e. lists, trees, stacks and the attribute values are only pointers to these tables. It means that the basic operations "*send* the value of an attribute to a process" or "*compute* a semantic function" may have *quite different costs.*

Only the *author* of a compiler knows the size of attributes, the complexity of semantic functions. The author has enough information on which nonterminals can potentially be selected - in the case of tree based distribution -, and on "logically" independent attributes - in the case of attribute based distribution.

Now we can state the following basic *requirements*:

- The user should choose between tree based and attribute based distribution. Probably he/she will choose a combined strategy.
- The user should declare selected and non-selected nonterminals in the case of tree based distribution and declare the set of attributes evaluated by the same process in the case of attribute based distribution.

On the other hand there are efficient *algorithms* to find independent attribute instances of an attribute grammar. For example, see Kuiper's algorithm [7]. An intelligent system can help the user's decision and *check* its correctness using these algorithms.

- The system should help and check the user's decision on distribution using dependency analysis.

2.3. Parallel compiler generator

The *specification* of a parallel compiler is an *attribute grammar* completed with evaluation instructions and with the algorithms of semantic functions. We start from the metalanguage of *PROF-LP* [10].

We mention here that an *augmented metalanguage* is defined in [15] containing such elements as regular right hand side productions (sometimes called *extended cf grammar*), augmented semantic functions for such productions, global *table definitions*, structured dynamic *data type* declarations embedded in a *block structured modular* metalanguage.

- The metalanguage of *PROF-LP* augmented with modularity and block structure is applicable.
- We introduce four level of modularity:

  Metalanguage level. A module is usually a large part of the attribute grammar described in one input file and processed as a whole. A module is formed from a set of nonterminals.

  Process level. A module is a – possibly different – part of the generated compiler implemented in one process. The user may develop some other processes containing the same elements as it is usual in *PROF-LP*. Configuration of these physical processes are up to the user.

Tree level. A tree module is a connected part of the syntax tree determined by selected nonterminals. It is the basis of tree based distribution. Tree level modularity should be compatible with source fragmentation.

Task level. A task is an elementary part of evaluation, subject to automatic load balancing. A task is a set of attribute instances defined by the user.

The first two levels are applicable only in large systems. These two levels are incomparable, either a metalanguage module may contain more processes or a process may be composed of more modules.

- Production descriptions are applicable in their original form.
- We do not consider the lexical description here.

The input attribute grammar is specified in one or more metalanguage level module. These metalanguage module sources are parsed separately. However, most of the test routines check global properties of the grammar. The start symbol is specified in the main module. It has to be declared as a nonterminal symbol.

```
<Parallel Attribute Grammar> = <Main Attribute Grammar Modul>
               {<Attribute Grammar Modul>} .

<Main Attribute Grammar Modul> = <Attribute Grammar Head>
               <Start Symbol Declaration>
               <Attribute Grammar Body> .

<Attribute Grammar Modul> = <Modul Head> <Attribute Grammar
               Body> .

<Attribute Grammar Head> = "Attribute" "Grammar" <Grammar
               Name> .

<Modul Head> = "Grammar" "Modul" <Modul Name> .

<Start Symbol Declaration> = "Start" "Symbol"
               <Nonterminal Name> ";" .
```

Global objects declared in different modules can be referred in other modules. To decrease the size of global symbol tables of the metalanguage parser we introduce the "USES" clause as usual in modular programming languages.

```
<Attribute Grammar Body> = [ <Use Clause> ] {<Compiler
               Modul>} .
<Use Clause> = "Use" <Modul Name> { "," <Modul Name> } ";" .
```

The generated compiler is modular. A compiler module is formed from a set of nonterminals. Each compiler module is identified with a name, parts of a compiler module can be specified in different metalanguage modules. These parts are copied together by the code generator.

Compiler modules do not form blocks in the metalanguage source. All declarations are visible from other modules. If the compiler is not modular, i.e. there is only one compiler module, the module name can be omitted.

```
<Compiler Modul> = [ "Compiler" "Modul" <Modul Name> ]
               <Declarations> .

<Declarations> = {<Declaration List>} .
```

The user has to declare all objects used in the productions: terminal and nonterminal symbols and attributes. There are two kinds of terminal symbols. A "terminal" has only one form and cannot have attributes. Terminals are usually keywords and special symbols. A "token" possibly has different forms and it may have attributes. Identifiers and several constants are tokens in programming languages. The name of an identifier or a value of a constant is a synthesized attribute.

The tokens are defined in a separate lexical description. The values of synthesized attributes of tokens are also defined in the lexical description.

```
<Declaration List> = <Type Declaration List> ;
           <Attribute Declaration List> ;
           <Attribute Set Declaration List> ;
           <Nonterminal Declaration List> ;
           <Token Declaration List> ;
           <Terminal Declaration List> ;
           <Production List> .
```

The domains of attributes are defined with a data type declared in user-supplied modules written in the host language.

```
<Type Declaration List> = "Types"
               <Type Name> { "," <Type Name>} ";" .

<Attribute Declaration List>=("Synthesized"/"Inherited")
               "Attributes"
               <Attribute Declaration>
               { "," <Attribute Declaration> } ";" .

<Attribute Declaration>=<Attribute Name> ":" <Type Name> .
```

Attribute based distribution is specified with attribute sets. Two attributes can be allocated to the same task level module only if they are put into the same attribute set by the user.

```
<Attribute Set Declaration List> = "Attribute" "Sets"
              <Attribute Set Declaration> {
              "," <Attribute Set Declaration>}
              ";" .

<Attribute Set Declaration> = "[" <Attribute Name>
              { "," <Attribute Name>} "]" ";" .

<Token Declaration List> = "Tokens" <Token Declaration>
              { "," <Token Declaration>} ";" .
```

Token and terminal names and the list of attributes of tokens are listed in token and terminal declarations.

```
<Token Declaration> = <Token Name> [ "Has"
          <Attribute Name>
          { "," <Attribute Name> } ] ";" .
<Terminal Declaration List> = "Terminals" <Terminal Name>
          { "," <Terminal Name> } ";" .
```

The metalanguage blocks are connected to the nonterminals. Local declarations are listed in a begin-end block. It is suggested that the productions with the same left hand side nonterminal X should be listed in the declaration block of X. However, it may be uncomfortable in the case of small languages, so this style of specification is not compulsory.

Tree based distribution is described with selected nonterminals. Both nested and non-nested distribution are supported.

```
<Nonterminal Declaration List> = "Nonterminals"
              <Nonterminal Declaration> { ","
              <Nonterminal Declaration> } ";" .

<Nonterminal Declaration> = <Nonterminal Name> [ "Has"
              <Attribute Name>
              { "," <Attribute Name> } ]
              [ <Selection> ]
              ";" [ <Block> ] .

<Selection> = ( "Selected" [ "Nested" / "Non-nested" ] )
              / "Non-selected" .
```

```
<Block> = "Begin" <Declarations> "End" ";" .
```

Semantic functions are defined by expressions or functions written in the host language. We do not specify the exact syntax of expressions and function calls here, it depends on the properties of the host language. In the case of ambiguity, attribute occurrences are identified by occurrence indices.

```
<Production List>="Productions" <Production> { <Production> }.

<Production>=<Syntax> [ <Semantics> ] .

<Syntax>=<Nonterminal Name> "=" { <Symbol> } ";" .

<Symbol>=<Nonterminal Name> / <Token Name> / <Terminal Name>.

<Semantics>="Do" <Semantic Function> "End" ";" .

<Semantic Function>=<Attribute Occurrence> ":="
                ( <Expression> / <Function Call> ) ";" .

<Attribute Occurrence>=<Nonterminal Name> ["[" <Index>"]"]"."
                <Attribute Name> .
```

The generated compiler consists of three parts: A static *kernel* contains basic routines, the *attribute evaluator* is generated from the specification, *user supplied parts* are copied into the system without any change.

- The *kernel* contains the following routines.

  Input-output and distribution. In this paper we do not deal with the syntactic parser part of the compiler, so we suppose that the syntax tree is *available.*

  Task scheduler and load balancer. The dynamic *load balancer* given in [13] can be applied as follows. A task means evaluation of a set of attribute instances. Two attribute instances $N.a$ and $M.b$ are in the same set if and only if the following conditions hold:

  - The nodes M and N are in the *same tree module*, that is, there are not selected nonterminals along the path between N and M in the syntax tree.
  - Attributes a and b are in the *same attribute set* declared by the user.
  - The attribute instances $N.a$ and $M.b$ are *not independent* of each other. As it is very hard to check this condition, we can use another conditions instead.
    - We can use Kuiper's algorithm [7] which decides that any two instances of two attribute occurrences can be dependent in any syntax tree.

- We can use Jordan's dependency based dynamic distributor [4]. In its original form it is based on local dependencies of a single production. It is easy to extend it to check dependencies of a subtree (tree module).

We suggest a *simpler* method instead. We can use Jordan's method to form elementary tasks. The problem is that only attribute instances evaluated in the same production are allocated to the same task. After that we can form the unions of these - small - tasks using tree based distribution.

The same universal evaluator algorithm is running on each processor. The load balancer distributes tasks among processors. The evaluation starts on a single processor with tasks belonging to the *root* of the parsing tree. When a task is executable - that is all its input attribute instances are *available* - the processor sends this task to the one of its neighboring nodes. The node is selected on the numbers of other tasks waiting for execution. Leaving a tree module means that virtually all tasks evaluating attribute instances of the module just entered are sent away.

Execution some semantic functions may need extremely long time, others may be divided into smaller parts. Rutins handling tasks - insert a new task to the waiting list, declaring input and output parameters, etc. - are available for the user.

Error handling routines. All error messages are sent to the host computer.

- The *evaluator* contains a branch for each task containing semantic functions evaluating the set of attribute instances belonging to this task. It may start other tasks, as well. An evaluator is generated from a process level module. The evaluator is called by the load balancer whenever a task is started.
- The routines containing user written semantic functions are simply copied into the system. They may send tasks for the load balancer for execution.

The compiler development environment should contain the following moduls.

Metalanguage parser: checking the formal correctness of the specification.

Dependency analyzer: computing attribute dependencies and checking its properties against the requirements of the evaluation strategy.

Distribution analyzer: checking dependencies among tree modules, attribute sets and tasks. It can help the user choosing a proper distribution strategy.

Code generator: generating the evaluator.

Developer utilities: helping the user developing semantic functions.

Execution utilities: helping the user configuring and executing the generated system.

The development process can be run on the host compiler. We mention that some suggestions to develop parallel compiler-compilers can be found in [1]. As it can be seen, the structure of the compiler-compiler is very similar to the structure of a sequential system.

2.4. An application – processing Latin medical texts

While attribute grammars can efficiently be used for specifying semantics of formal languages, the more complicated structure of natural languages usually needs enhanced formalisms. However there are some well-defined subsets of natural languages used by specialists which use only simpler grammar structures. One of these terminologies is the pure Latin medical language.

Most important medical texts as recipes and diagnoses are always written in Latin. These texts have a very strict form, only some simple grammar structures are used. Although medical students study the Latin language as a foreign language including even some classical literature, they will use only this very simple structures in practice. It would be very useful to have a software system to memorize the real "medical" Latin.

Efficient computer systems are used in most hospitals for administrative purposes. International coding systems - as SNOMED or WHO-code - are used for coding features of various diseases. This coding are done by the doctor causing a very boring and inefficient task. It would be very useful if we could use the computer for coding the text of diagnoses.

We developed an attribute grammar for specifying syntax and semantics of recipes and diagnoses. This grammar is usable for both tasks mentioned earlier. A complete system integrating these applications is under development.

As an illustration we give a part of the syntax of recipes:

```
<Recipe> = <Materials> <Instructions> .
<Materials> = <Material Description> + .

<Material Description> = <Material> + <Quantity> .

<Material> = <noun> [ <Genititive> ] <Attribute> * .

<Attribute> = <adjective> <participium> .
<Genititive> = <Material> .
<Quantity> = <Text> <Num> / <Asneeded> / <Asyoulike> .

<Text> = [ <preposition> ] <Unit> <Q> .
```

```
<Unit> = <noun> .
<Num> = "(" <preposition> <Unit> <Q> ")" .
<Q> = <numeral> .

<Asneeded> = "quantum" "satis" / "qu" "sat" .
<Asyoulike> ="ad" "libitum" / "ad" "lib" .

<Instructions> = ...          ...
```

Two grammars together consist of more than 300 productions. The most important attributes contain grammatical properties of words and phrases. The vocabulary is divided into smaller pieces according to semantic features of words. These features are stored in synthesized attributes served by the vocabulary handler. More detailed description of the system can be found in [16].

3. Summary

Questions of parallel evaluation of attribute grammars are discussed. Some programming work is needed to implement our ideas. However we see some open problems now, so we can draw up some directions of future research.

- How can our generated semantic analyzer be combined with *parsing*? The results of Klein and Koskimies [5], [6] also may help solving this problem.
- Which methods and *algorithms* can be used in parallel compilers? For example what kind of *symbol table handling* methods are suitable? Do these methods have any consequence to the structure of the compiler?
- The basic motivation of our research was to contribute in developing softwares for *IDIOMS* machine. We should go on in this direction as well.
- It is also important to find other *application fields*, where a compiler running on transputer is suitable and efficient.
- We can find other special natural-like languages for attribute grammar specification, as whether forcasts, questions to an information system or texts of application forms.

References

[1] **Alblas H.,** *A blueprint for a parallel parser generator,* Technical Report, Dept. of Computer Science, Univ. of Twente, 92-65, 1992.

[2] **Almási J., Horváth T., Medvey M. and Toczki J.,** On the implementation of cellular software development system, *Proc. of PARCELLA 88*, Berlin, 1988.

[3] **Deransart P., Jourdan M. and Lorho B.,** *Attribute grammars, systems and bibliography, LNCS* **323**, 1988.

[4] **Jourdan M.,** A survey of parallel attribute evaluation methods, *Proc. of SAGA, Prague, 1991, LNCS* **545**, 234-254.

[5] **Klein K. and Koskimeies K.,** Parallel one pass compilation, *Proc of WAGA, Paris, 1990, LNCS* **461**, 76-90.

[6] **Klein K. and Koskimeies K.,** How to pipeline parsing with parallel semantic analysis, *Structured Programming,* **13** (1992), 99-107.

[7] **Kuiper M.F.,** *Parallel attribute evaluation,* Ph.D. Thesis, Fac. of Informatics, Univ. of Utrecht, 1989.

[8] **Lipkie D.E.,** *A compiler design for multiple independent processor computers,* Ph.D. Thesis, Dept. of Computer Science, Univ. of Washington, Seattle, 1979.

[9] **Miller J.A. and LeBlanc R.J.,** Distributed compilation: a case study, *Proc. of the Third Int. Conf. on Distributed Computing Systems 1982*, 548-553.

[10] *PROF-LP User's Guide,* Research Group on Theory of Automata, Szeged, 1987.

[11] **Schell R.M.,** *Methods for constructing parallel compilers for use in a multi-processor environment,* Ph.D. Th., Rep. 958, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, 1979.

[12] **Seshardi V. and Wortman D.B.,** An investigation into concurrent semantic analysis, *Software, Practice and Experience,* **21** (12) (1991), 1323-1348.

[13] **Schrettner L. and Toczki J.,** Dynamic load balancing for decomposable problems, *Proc. of Workshop on Parallel Processing in Education*, Impact Tempus JEP/Hungarian Transputer Users Group, Miskolc, 1993.

[14] **Toczki J., Gyimóthy T. and Jahni G.,** Implementation of a LOTOS precompiler, *Proc. of PD 88*, Budapest, 1988.

[15] **Toczki J.,** *Attribute grammars and their applications,* Dr.Univ. Thesis, Dept. of Informatics, József Attila Univ., Szeged, 1991. (in Hungarian)

[16] **Vágvölgyi E. and Toczki J.**, Computer aided processing of Latin medical texts, *Proc. of VI. National Congress of NJSZT, Siófok, 1995.* (in Hungarian)

[17] **Vandervoorde M.T.**,*Parallel compilation on a tightly coupled multiprocessor, SRC Reports* **26**, Digital Systems Research Center, 1988.

J. Toczki and L. Schrettner
Department of Computer Science
József Attila University
Árpád tér 2.
H-6720 Szeged, Hungary
schrettner@inf.u-szeged.hu