Acta Cybernetica, Vol. 12, No. 1, Szeged, 1995

Towards Computer Aided Development of Parallel Compilers Running on Transputer Architecture*

János Toczki †

Abstract

In this paper we state requirements for a software environment for computer aided development of parallel compilers executable on transputers. The structure of a compiler-compiler which generates parallel compilers from attribute grammar specifications is described. Problems of distributed attribute evaluation using dynamic load balancing are discussed.

Keywords: attribute grammars, compilers, transputers, parallel processing.

1 Introduction

Several types of parallel machines have become more and more popular recently. Various parallel algorithms have been developed to get more efficient softwares for several problems. An important application field is developing *parallel compilers.*

Attribute grammars are an efficient compiler specification method. Most of *compiler-compiler systems* are based on attribute grammars. A survey of sequential attribute evaluation methods can be found in [7] and [2]. A review of compiler-compiler systems based on attribute grammars is presented in [6].

Most of the positive experiences with developing *parallel semantic evaluators* are connected with non-distributed architectures with shared memory. Reviews of using parallel attribute evaluation and experiences developing parallel compilers can be found in [1], [4], [16] and [10]. A blueprint of a parallel compiler generator system is presented in [3].

On the other hand, there is no shared-memory available in *transputer* machines; each processor uses its own memory. Processors are connected through channels. Channels are used not only for synchronization, but also for sending data between processors. According to the practical experiences, the main problem with using transputers for parallel attribute evaluation is the large amount of *inter-processor communication* [17], [1]. It causes the inefficiency of these algorithms.

*This research is involved in the research project "Large Parallel Databases" financed by the European Communities, project number 93: 6638, and partly supported by the research found OTKA and the Ministery of Education and Culture, grant number F12852 and 434/94

†Department of Computer Science, József Attila University of Szeged

However, there are some *practical applications* efficiently implemented on transputers. For example, a database management system processing large databases implemented at Sheffield University on *IDIOMS* machine [12]. The input language of the system is the standard *SQL*/1. Users of *IDIOMS* requires some extension to *SQL*: it needs a precompiler which transforms queries to standard *SQL*. It is a natural demand that the precompiler should run on the same machine instead of the host computer.

In this paper we consider the problems of efficient evaluation of attribute grammars on distributed architectures from a practical point of view. Which evaluation methods can be used, which other facilities are needed to get a compiler-compiler or an environment for developing parallel compilers?

This paper is composed as follows. We summarise preliminaries and motivations in the next section. We repeat some of the basic definitions and notations, however we suppose that the reader is familiar with the following topics: *attribute grammars* and *evaluation strategies*, *compiler-compilers*, *transputer architectures*. The summary of requirements for a parallel compiler-compiler running on transputers is found in section 3. Our suggestions to meet these requirements and the preliminary design of a compiler-compiler system is given in section 4. A short summary of future research is found in the last section.

2 Motivations and Preliminaries

2.1 Motivations

Parallel machines are classified as *synchronous* and *asynchronous* machines. In synchronous machines all processors execute the same code at the same time. In asynchronous architectures processors may execute different code, synchronisation should be controlled directly by using semaphores and/or sending messages.

On the other hand, we can distinguish between *tightly coupled* and *loosely coupled* (distributed) architectures. In tightly coupled machines, all processors have access to the same shared memory, while in distributed machines each processor has its own memory. Processors can communicate by sending messages.

Transputers, which are asynchronous distributed machines have become more and more popular recently. Processors - nodes - of a transputer are usually configured along a more or less regular topology (a line, hypercube, polygon, etc.). Each processor has four channels for communication. Software connections can be configured in a flexible way via connecting channels. The number of nodes and the physical topology should be as irrelevant as possible from the point of view of transputer software. However the speed and efficiency of an algorithm usually strongly depends on the topology.

Usually the most expensive part of transputer software is the inter–processor communication. Consequently, in general, an "effective" algorithm should avoid large amounts of communication. There are some "typical" transputer applications. One of these areas is processing large distributed databases. If we connect a data storage device to each node and we distribute queries at an early stage, we get efficient data retrieving algorithms.

IDIOMS machine developed at NTSC, University of Sheffield is a special parallel machine for processing *large distributed databases* using parallel methods [12]. The user surface of the system is standard *SQL*/1 language.

Parsing user instructions, *evaluation* and distribution of queries is a usual formal language parsing and semantic evaluation problem. There are three possible solutions to perform this task.

- We can use a sequential compiler running on the host machine.
- We can use a sequential compiler running on a single transputer node and benefit only from the parallelism between the larger components of the software architecture.
- We can use a parallel compiler running on the transputer itself.

The advantage of the first two solutions is that the theory and practice of developing sequential compilers is a well-researched topic of computer science. On the other hand, the host machine is the only connection between the transputer and the outer world. The host machine will be very busy with transfering results of queries. It is a natural demand to use the inner part of the system instead.

Another advantage of the last method is that – using parallel algorithms – we can speed up the compilation process itself. For example Gross achieved 3 to 6 typical speed-up with a hand-written compiler running on 9 independent workstations benefiting only from parallel compilation of independent functions (blocks) [8]. It does not exclude running the compiler in parallel with some other components, as well.

2.2 Parallel Compilation

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 made by Schell [22]. Schell's method – in essence – is the same as Jordan's one reported in [10].

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 speed up is the number of phases (usually 3 or 4). However, it is a hard work to balance the 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 [20]. 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 [19]. Vandevoorde [27] and Seshardi [23] used the same approach developing compilers running on different architectures. Seshardi investigated the concurrent processing of declarations as well.

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

2.3 Automatic Compiler Construction

Compiler-compiler systems generate executable compilers from formal specifications. Most recent compiler writing systems are based on attribute grammars. Experiences with compiler construction proved feasibility and efficiency of attribute grammars for compiler specification. A survey of attribute grammar based compiler generation can be found in [6].

The compiler generator system *PROF-LP* [21] developed in Szeged has been used for generating various practical compilers, for example [25], [5]. We refer to our experiences at appropriate places in the next section. We mention that these experiences and suggestions for development in the case of sequential compilation are summarized in [26].

2.4 Attribute Grammars

Various compiler-compiler systems have been developed to generate compilers from formal specifications. Most of these systems are based on attribute grammars [15].

In this section we recall some basic notions of attribute grammar theory. We give only informal definitions instead of formal ones. We feel that it is enough to make clear our concepts. More complete definitions can be found in the literature for example [2], [7].

Let $G = (N, T, S, P)$ be a context-free grammar, where N is the set of nonterminal symbols, T is the set of terminal symbols, $S \in N$ is the start symbol, and P is the set of context-free productions.

We associate a set of *attributes* to each nonterminal symbol. There are two types of attributes. The values of *inherited* attributes are evaluated top-down (from the start symbol to the terminals) and the values of *synthesized* attributes are evaluated in the opposite direction.

Attribute values are determined by semantic functions associated to the synthesized attribute occurrences of the left-hand side nonterminal and to the inherited attribute occurrences of the right-hand side nonterminals of each production. The arguments of semantic functions are the other attribute occurrences of the same production.

An abstract syntax tree of the grammar G is said to be *decorated* if all its attribute instances have their own values. We say that a decoration of a syntax tree s is *correct* iff the values of all attribute instances of s satisfy corresponding semantic rules.

Attribute instances in a syntax tree s depend on each other. We say that an attribute instance $A(N)$ *depends on* an attribute instance $b(M)$, if the value of $b(M)$ is needed to evaluate the semantic rule computing the value of $a(N)$, i. e. if it occurs as a parameter of the semantic function.

In this paper we consider only *non-circular* attribute grammars. An attribute grammar is non-circular iff there cannot exist any circular dependencies among the attribute instances in any syntax tree. In this case all attribute instances can be evaluated in a definite order constrained by the dependencies of the given syntax tree.

2.5 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 [16], [18] 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 splitted 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 splitted again, while in the case of *non-nested distribution*, the syntax tree is splitted 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 to 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 can not 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 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.

3 Parallel Compilation on Transputers

3.1 Assumptions

A transputer is a loosely-coupled parallel machine having no shared memory. Processors communicate via channels. Channels serve not only for synchronization but for data exchange, as well. Process loading and channel connections are flexible. The only physical bound is the amount of memory and the number of hardware connections (usually 4). Peripherals are handled by a host computer which is connected to the processor network via channels.

In this paper we concentrate on the *semantic* part of compilers. We suppose that the complete syntax tree is available on the host computer or on a transputer node. In the case of source fragmentation, before syntactic parsing different parts of the syntax tree may be produced by different processes running on different nodes. This situation can be handled by appropriate process level distribution, see in 4.1. The result of semantic evaluation, the decorated syntax tree, is sent back to the host. In the case of a semantic error, an error message is sent to the host and the evaluation process is stopped. In some applications, it would be better to pass the result to another application. It is only a technical point.

We suppose a *static evaluation method* driven by the dependency graphs of productions. Among others *OAG* [11] and *ASE* [9] are feasible strategies. These classes of attribute grammars are large enough in practical cases.

We suppose availability of a block structured high level algorithmic programming language - we have chosen parallel *C* - and availability of a flexible *CDL* (configuration description language) usual on transputers.

3.2 Distribution

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 a single process on the same processor, we should allocate them to as many different processors as possible to increase "real" parallelity of the compiler. *On the other hand*, attribute values have to be sent among processors. As inter–processor communication is the most expensive task on transputers, we should decrease the amount of sending data among processes allocated to different processors.

Some attributes – as 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 expenses.*

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

We can state now 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 [16]. 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.

3.3 Process Loading

Most of transputer operating systems include some *load balancing* mechanisms. It means that the system distributes processes among processors on the bases of their current status. On the other hand the user has the possibility to describe her/his own configuration using *CDL* (Configuration Description Language).

Automatic load balancing assumes a *farmer-workers* architecture, while the user can use (almost) any other architecture. It gives a large amount of flexibility. On the other hand, it is much easier to program an automatically balanced system.

Another important question that we should answer is: should we develop a general evaluation process which contains all the semantic functions and run it on all processors or should we develop several smaller processes? Execution time of

semantic functions and the number of attribute instances evaluated by a process may be quite different. Furthermore scheduling many small processes causes too much overhead time. It is more efficient to run a *general evaluator* on all processors and implement evaluation processes as tasks rather than real physical processes. In this approach a *task* means evaluation of all attribute instances allocated to a logical process. Each task has a set of *output attribute instances* – the attribute instances which are computed – and a set of *input attribute instances* – values of which are needed for the computation.

In this case, we cannot use the automatic load balancing mechanism of the operating system: load balancing means allocating tasks to processes, and not allocating physical processes to processors. The dynamic load balancing method described in [24] is applicable for any *decomposable problem*. Although attribute evaluation is not a decomposable problem, we can associate a home processor to each attribute instance. The value of an attribute instance is sent back to its home processor after evaluation. More detailed description of process loading can be found in the next section.

4 Developing Parallel Compilers

In this section we describe the structure of a software environment for developing parallel compilers based on the requirements stated in section 2. First we discuss the features of a metalanguage for specifying a parallel compiler and describe the general structure of the generated compiler. After that, we sketch the structure of the development environment including a generator tool. Finally we consider some technical questions.

4.1 Parallel Compiler Specification

The *specification* of a parallel compiler is an *attribute grammar* completed with evaluation instructions and with the implementation of semantic functions. We start from the metalanguage of *PROF-LP* [21]. This metalanguage has the following features.

- The set of synthesized and inherited *attributes* are declared. The domain of an attribute is given by a data type of the implementation language.
- The set of *nonterminals* with the list of their attributes is declared. The generated compiler is modular, a module is formed from a set of nonterminals.
- The set of *terminals* is declared. Some terminals, called tokens, may have input attributes. The lexical structure is described separately.
- *Productions* are listed together with semantic functions. A semantic function is given by an expression or by a subroutine written in the implementation language.
- The description is completed with one or more program *modules* written in the implementation language including attribute types, semantic functions and any other elements as constants, variables, subroutines. This makes it possible for the user to implement dynamic data structures and global program objects.

We mention here that an *augmented metalanguage* is defined in [26] containing such elements as regular right hand side productions (sometimes called as *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 levels of modularity:

  Metalanguage level. A module is a usually large part of the attribute grammar described in one input file and processed at the same time. 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, target of 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 from more modules.
- Production descriptions are applicable in their original form.
- We do not consider the lexical description here.

Formal consistency of the specification can be checked in the same way as it is usual in the case of sequential compilers. Checking correctness of semantic functions against the requirements of the implementation language is left to the compiler.

4.2 General Structure of the Generated Parallel Compiler

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*. As we use a dynamic load balancing method, the whole syntax tree should be sent to each processor first. The result – the values of synthesized attribute instances of the root symbol – are sent to the host.

  Task scheduler and load balancer. The dynamic *load balancer* given in [24] 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 *dependent* on each other. As it is very hard to check this condition, we can use another conditions instead.
  * We can use Kuiper's algorithm [16] which decides whether any two instances of two attribute occurrences may be dependent.
  * We can use Jordan's dependency based dynamic distributor [10]. 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 *more simple* 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 has become 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.

Executing a semantic function may need an 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.

4.3 Compiler Development Environment

The compiler development environment contains the following modules.

Metalanguage parser: checking the formal correctness of the specification.

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

Distribution analyser 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 [3]. As can be seen, the structure of the compiler-compiler is very similar to the structure of a sequential system.

4.4 Technical Issues

We have started the implementation of the parallel compiler development environment with developing a small *prototype* for semantic check of a simple block structured language. Using the prototype, we get a statement by statement *specification* of the generated system as well as the kernel of the compiler. We implement it in parallel C running on a network of 16 T8000 processors.

Meanwhile an *attribute grammar specification* of the metalanguage is under development. We will generate the metalanguage parser from this specification using the compiler generator *PROF-LP*. The whole system will run on IBM PC under DOS, the implementation language is Turbo Pascal. As the host computer of our transputer is a Unix machine we have to transfer the generated compiler to the host. It may cause some technical problems.

The implementation of the whole system needs a lot of time and manpower. Practical *experiences* will be available after the completion of the implementation.

5 Final Remarks

In this paper we considered the problem of developing *parallel compilers* running on *transputer* architecture. Our most important conclusion is that we should give a lot of *freedom* to the user during developing such compilers. Only the user has enough knowledge to make basic decisions on attribute distribution. However some steps of development can be done *automatically*. Moreover we can help the user's work with the results of some *test algorithms*.

We stated the most important requirements for a compiler-compiler for developing parallel compilers. The basic structure of the compiler and the compiler-compiler has been described.

The first version of the system is under implementation now. Moreover we should consider the following questions in the future.

- How our generated semantic analyser can be combined with *parsing*? The results of Klein and Koskimies [13], [14] 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? Have these methods any consequence for 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.

Finally we thanks to *Lajos Schrettner* for his valuable remarks and suggestions.

References

[1] Akker, R., H. Alblas, A. Nijholt, P. O. Luttighuis, K. Sikkel: An annotated bibliography on parallel parsing, updated version, Technical Report, Dept. of Computer Science, Univ. of Twente, 92-84, 1992.

[2] Alblas, H.: Attribute evaluation methods, in Proc. of SAGA, Prague, 1991. LNCS 545., pp 48-113.

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

[4] Alblas, H., R. Akker, P. O. Luttinghuis, K. Sikkel: A bibliography on parallel parsing, in ACM Sigplan Notices, Vol. 29, No. 1., 1994. pp 54-65

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

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

[7] Engelfriet, J.: Attribute grammars: Attribute evaluation methods, in Methods and tools for compiler construction, Cambridge Univ. Press, 1984, pp. 103-138.

[8] Gross, T., A. Zobel, M. Zolg: Parallel Compilation for a Parallel Machine, in Proc. of SIGPLAN '89, SIGPLAN Notices 24, 7 (1989), pp 91-100.

[9] Jazayeri, M., K. G. Walter: Alternating semantic evaluator, In Proc. of ACM 1975 Annual Conf., 1975., pp. 230-234.

[10] Jourdan, M.: A survey of parallel attribute evaluation methods, in Proc of SAGA, Prague, 1991., LNCS 545., pp 234-254.

[11] Kastens, U.: Ordered attribute grammars, Acta Informatica 13, 1980, pp. 229-256.

[12] Kerridge, J. M.: The design of the IDIOMS parallel database machine, in Proc. of British National Conf. on Databases 9, 1991.

[13] Klein, K., K. Koskimeies: Parallel one pass compilation, in Proc of WAGA, Paris 1990, LNCS 461., pp 76-90.

[14] Klein, K., K. Koskimies: How to pipeline parsing with parallel semantic analysis, Structured Programming 13, 1992., pp 99-107.

[15] Knuth, D. E.: Semantics of context-free languages, Math. Systems Theory 2, 1968., pp. 127-145, correction Math. Systems Theory 5, 1971. pp 95-96.

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

[17] Kuiper, M. F., A. Dijkstra: Attribute evaluation on a network of transputers, in Wexler (ed.): Developing transputer applications, Amsterdam, 1989., pp 142-149.

[18] Kuiper, M. F., D. Swierstra: Parallel Attribute Evaluation: Structure of Evaluators and Detection of Parallelism, in Proc. of WAGA 90, Paris, 1990., LNCS 461, pp. 61-75.

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

[20] Miller, J. A., R. J. LeBlanc: Distributed compilation: a case study, in Proc. of the Third Int. Conf. on Distributed Computing Systems, (1982), pp. 548-553.

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

[22] Schell, R. M.: Methods for constructing parallel compilers for use in a multi-processor environment, Ph.D. Th., Rep. No. 958, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, 1979.

[23] Seshardi, V., D. B. Wortman: An investigation into Concurrent Semantic Analysis, Software, Practice and Experience Vol. 21. No. 12. (1991) pp. 1323-1348.

[24] Schrettner, L., J. Toczki: Dynamic Load Balancing for Decomposable Problems, Proc. of Workshop on Parallel Processing in Education, Impact Tempus JEP/Hungarian Transputer Users Group, Miskolc, 1993.

[25] Toczki, J., T. Gyimothy, G. Jahni: Implementation of a LOTOS precompiler, in Proc. of PD 88, Budapest, 1988.

[26] Toczki, J.: Attribute grammars and their applications, (in Hugarian), Dr. Univ. Thisis, Depts. of Informatics, József Attila Univ. of Szeged, 1991.

[27] Vandervoorde, M. T.: Parallel Compilation on a Tightly Coupled Multiprocessor, SRC Reports No. 26, Digital Systems Research Center, 1988.

Received October, 1994