ISSN 0105-8517

Joint Language Project
Working Note No. 8

SYNTAX DIRECTED PROGRAM MODULARIZATION

by

Bent Bruun Kristensen
Ole Lehrmann Madsen
Birger Møller-Pedersen
Kristen Nygaard

DAIMI PB-155
September 1982

Note. This paper is to appear in the Proceedings of the European Conference on Integrated Interactive Computing Systems, Stresa, Italy, September 1 - 3, 1982, edited by P. Degano, E. Sandewall, to be published by North-Holland Publishing Company, Amsterdam.

SYNTAX DIRECTED PROGRAM MODULARIZATION

by

Bent Bruun Kristensen
Institute of Electronic Systems, Aalborg University Center, Aalborg, Denmark

Ole Lehrmann Madsen
Computer Science Department, Aarhus University, Aarhus, Denmark

Birger Møller-Pedersen, Kristen Nygaard
Norwegian Computing Center, Oslo, Norway

The intent of this paper is to illustrate the following general ideas: —Use of the context free grammar of a programming language as an integrated part of its programming system —Reconsideration of the borderline between language and system —Systematic modularization of programs for the various translation phases. The specific ideas presented in this paper are language independent methods for handling: —Modularization of programs —Separate translation in the form of context sensitive parsing (type checking) of modules —Protection of part of a module, e.g. protection of the representation of an abstract data type. The mechanism for modularization is unusual as it is based on the context-free syntax of the language. A module may be a sentential form generated by any nonterminal of the grammar.

1. Introduction

The main goal of this paper is to describe a general approach to separate translation of programs. In addition the use of this approach for textual modularization of programs and as a tool for insuring protection of parts of a module is described. The work presented here has been developed as part of the BETA project. The aim of the BETA project has been to develop concepts, constructs and tools in connection with programming languages. The main efforts have been related to the development of a system programming language, BETA ([BETA a,b]), which in many aspects has been influenced by DELTA ([DELTA a,b]) and SIMULA 67 ([SIMULA]).

The approach to separate translation will be used to define a set of tools to be used in an integrated programming system for the BETA language. Although the tools we describe have been developed with the BETA language in mind, they may be used for any language with a well defined syntactic structure. In this paper we shall demonstrate their usage by means of SIMULA 67. Consequently the BETA language will not be described in this paper.

The mechanism for defining the modules that may be separately translated is unusual as it is based on the context free syntax of the language. A module may be a sentential form (a string of terminal and

Part of this work has been supported by The Danish Natural Science Research Council
Grant No. 11-3106.

nonterminal symbols) generated by a nonterminal of the grammar. The BETA language does thus not contain a construct for splitting a program up into modules that can be separately translated.

One may argue that the language should define how to modularize a given program. However, with the BETA approach, the programmer has the complete freedom to organise his program in whatever modules he may want. Furthermore by taking separate translation out of the language and leave it to the programming system reduces the number of constructs in the language.

One of the advantages of integrated programming systems is that the barrier between language and programming system is broken down. This means that features may be moved to the programming system when this is appropiate.

The tools of an integrated programming system should be based on a systematic principle when this is possible. This is e.g. the case with syntax directed editing (e.g. [Teitelbaum and Reps 81]).

The use of syntax directed editors will probably make it naturally for a programmer to construct libraries of abstract syntax trees (ASTs) for parts of programs. The abstract syntax trees need not be complete programs. The root of an AST may be any nonterminal of the grammar and they may contain unexpanded nonterminals.

With such libraries of ASTs there is a need for more flexible tools in order to handle additional parts of the translation process. It should not always be necessary to operate with complete programs in the remaining parts of the translation process. If some constituent AST in a program is changed it should be possible to restrict the re-translation to this AST and its immediate environment. The work on incremental attribute evaluation ([Demers et al. 81, Reps 81]) is an attempt in this direction.

Below we present a mechanism for making context sensitive syntax analysis on parts of a program (sentential forms). We concentrate on the essential (abstract) operations of this mechanism. Thus the actual integration of the mechanism in a concrete programming system will not be treated in detail.

It is not possible to perform context sensitive syntax analysis on arbitrary program fragments without having access to part of the context of the program fragment. This means that certain language dependent restrictions will limit the possibilities. We briefly discuss the possibilities for extending the approach to the remaining phases of the translation process. This will impose further restrictions upon the program fragments that may be treated.

Another consequence of syntax directed editors is that the programmer explicitly must use the context free grammar of the programming langauge since the editor is driven by the productions of the grammar.

In the BETA language, the grammar will be an integrated part of the language. Besides the mechanism described here, it will be possible to define subsets of the language, define abstractions (see [BETA b]), and define a new syntax for a package of abstractions. All of these mechanisms will be based on the BETA grammar.

The rest of this paper is organised as follows. Section 2 describes a general method for making a textually modularization of programs. In Section 3 this method is extended to cover separate analysis of program modules. Section 4 describes an application of the proposed mechanism to protect the representation of abstract data types and to specify parameterized data types.

2. Modularization

In this section we introduce a simple mechanism for textually splitting up a program text into arbitrary modules. A similar effect can be obtained by using a standard text macro generator. The mechanism presented here is based on the context free syntax of the grammar.

We recall that a context free grammar is a tuple $G = (N, \Sigma, P, S)$ where N is a (finite) set of nonterminal symbols, Σ is a (finite) set of terminal symbols, P is a (finite) set of productions of the form $A \rightarrow \alpha$ where A is in N and α is in $(N \cup \Sigma)^*$, and S is a distinguished symbol in N called the startsymbol of G.

We need a slightly more extended definition of the notion of sentential form:

> Let A be in N. Then an *A-sentential-form* (or just *A-form*) is defined as follows:
> A is an A-form.
> Let α, γ be in $(N \cup \Sigma)^*$ and B in N. If $\alpha B \gamma$ is an A-form and $B \rightarrow \beta$ is a production, then $\alpha \beta \gamma$ is an A-form.

In the standard terminology for context free grammars a sentential form is then an S-form (S is the start symbol). We shall extend the notion of sentential form (or just form) to mean an A-form where A is an arbitrary nonterminal.

The input to the translator will be a sequence of forms called a *translation unit*:

(form
  <form definition$_1$>
// *<form definition$_2$>*
  ...
// *<form definition$_n$>*
form)

A *form definition* has the following syntax:

<"name" : "syntactic category"> "sentential form"

where *"name"* is an identification of the form and *"syntactic category"* is a nonterminal of the grammar. If *"syntactic category"* is the nonterminal A, then *"sentential form"* must be an A-form. A nonterminal appearing in a sentential form is called a *form application* and it has the following syntax:

<"name" : "syntactic category">

"name" refers to a form definition of the specified syntactic category. It is required that all forms have different names.

Consider the following subset of a SIMULA 67 grammar:

```
<program> ::= <block>

<block> ::= begin <declaration> ; <statement> end

<declaration> ::= <declaration> ; <declaration>
                | <type> procedure <procedure-name>
                    <formal parameter part> ; <block>
                | <prefix> class <class-name>
                    <formal parameter part> ; <block>
                | <variable declaration>

<formal parameter part> ::= <empty>
                          | ( <formal parameters> );
                              <formal parameter specification>

<type> ::= <value-type> | <reference type>

<value type> ::= integer | real | boolean

<reference type> ::= ref ( <class-name> )

<prefix> ::= <class-name> | <empty>
```

The syntactic categories of this grammar include program, block, declaration, etc.

Note that the grammar is ambiguous (*<declaration>*). In practice it is no problem to handle such ambiguities. Alternatively one may include right hand sides with regular expressions to handle sequences.

A translation unit in SIMULA 67 may be as follows:

```
(form
    <P: program>
    begin
        <Math: declaration>;
        <I/O: declaration>;

        procedure F(x,y);
        real x,y; name y;
        <FBlock: block>;

        procedure G(z);
        real z; name z;
        <GBlock: block>;

        real a, b;
        InReal(a); F(a, b); G(b); OutReal(b)
    end ;
//
    <Math: declaration>
    real procedure sin(x); real x; begin ... end ;
    real procedure cos(x); real x; begin ... end
//
```



```
    <I/O: declaration>
    procedure InReal(R); real R; begin ... end ;
    procedure OutReal(R); real R; begin ... end
//
    <FBlock: block>
    begin y: = sin(cos(x)) end
//
    <GBlock: block>
    begin b: = b + cos(b) end
form)
```

A programming system for handling translation units based on forms may then replace any form application by the corresponding form definition. Resulting program-forms may then be translated as an ordinary program. The above translation unit will then be reduced to the following program:

```
begin
    real procedure sin(x); real x; begin ... end ;
    real procedure cos(x); real x; begin ... end ;

    procedure InReal(R); real R; begin ... end ;
    procedure OutReal(R); real R; begin ... end ;

    procedure F(x,y);
    real x,y; name y;
    begin y: = sin(cos(x)) end ;

    procedure G(z);
    real z; name z;
    begin b: = b + cos(b) end ;

    real a, b;
    InReal(a); F(a, b); G(b); OutReal(b)
end ;
```

To summarise, the programming system should realise the following abstract operations:

- A *constructor* for a set of forms:
  $form : (sequence\ of\ character)^* \rightarrow set\ of\ forms$
  denoted by
  (form F_1 // F_2 // ... // F_n **form)** .

- A *substitution* operator:
  $substitute : form \times form \rightarrow form$
  defined as follows
  $substitute(F_1, F_2)$ =
  if $F_1 = \langle N : A \rangle \alpha \langle M : B \rangle \gamma$ and $F_2 = \langle M : B \rangle \beta$
  then $\langle N : A \rangle \alpha \beta \gamma$
  else F_1

— A *reduction* operator :

reduce : *set of forms* → *set of forms*

defined as follows:

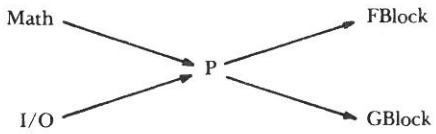
$reduce(\{F_1, F_2, \ldots F_n\}) =$
if $substitute(F_i, F_j) \neq F_i$ for some i, j in $[1, n]$
then $\{F_1, F_2, \ldots, F_{i-1}, subsitute(F_i, F_j), F_{i+1}, \ldots, F_n\}$
else $\{F_1, F_2, \ldots, F_n\}$

The forms need not appear in the same text. By integrating a suitable library, filesystem or database and extending the names appearing in form applications, the programming system may easily fecth a given form at the appropiate place.

3. Separate translation

We shall now consider the possibilities for making *separate analysis* of forms. By analysis is meant lexical analysis, context free syntax analysis (parsing), and context sensitive syntax analysis (static semantics). A form that has been processed by the analysis module will be converted to a representation called a *term*. A term may be viewed as an AST where declarative information has been collected, names have been connected to their corresponding declaration and all static semantic checks have been performed.

Consider the following graph of the forms in section 2:



The graph describes a partial ordering between the forms. This ordering defines the analysis order of the forms. There is no ordering between Math and I/O since they do not refer to declarations within one another. Math and I/O must be analysed *before* P since P refer to declarations in these forms. Finally FBlock and GBlock are independent but they can only be analysed *after* P.

In forms, like P, it is necessary to distinguish between form applications like Math and I/O and form applications like FBlock and GBlock. The analysis module must know how a form being analysed depends on the form applications which it contains.

The syntax of form applications is extended to express this:

A form application (*pre-application*) that refers to a pre analysed form has the following syntax

<pre *"name" : "syntactic category"*>

A form application (*post-application*) that refers to a form that will be inserted later has the following syntax:

<post *"name" : "syntactic category"*>

The form applications defined in section 2 will be called *direct-applications* to distinguish them from pre- and post-applications. Note that the form definition corresponding to a direct application must

appear within the same translation unit.

A translation unit containing the program-form P may then be as follows:

```
(form
    <P: program>
    begin
        <pre Math: declaration>;
        <pre I/O: declaration>;

        procedure F(x,y);
        real x,y; name y;
        <post FBlock: block>;

        procedure G(z);
        real z; name z;
        <post GBlock: block>;

        real a, b;

        InReal(a); F(a, b); G(b); OutReal(b)
    end ;
form)
```

We shall assume the existence of a *library file* containing pre-analysed forms represented as terms. The library file will then have the form

L = **(term** T1 // T2 // ... // Tm **term)**

where each Ti is a *term-definition* on the form

<name: "syntactic category"> "term".

Within "term", post-applications of the form **<post** *N : A*> may appear. Neither pre-applications nor direct-applications will appear since such applications have been connected to their corresponding form-definition.

The forms in a translation unit that has been submitted for analysis will be pre-analysed and appended to the library file.

Before analysing a translation unit containing P, translation units containing Math and I/O must have been analysed. After the analysis of P, translation units containing FBlock and GBlock may be analysed. Notice the difference between the relations like Math ⟶ P and relations like P ⟶ FBlock. Math is inserted into P but FBlock is also inserted into P. The ordering between two forms express that declarations in one of the forms may be used in the other form, provided that the scope rules of the language are not violated.

Let L = **(term** T1 // T2 // ... // Tm **term)** be the library file. Pre-analysis of a translation unit

T = **(form** F1 // F2 // ... // Fn **form)**

is carried out in the following way:

- T is reduced (the reduction operator) such that possible direct-applications within T are eliminated.

- Each form, F = <*N* : *A* > α, within T is *connected* to possible corresponding term definitions in the library unit:
  Pre-substitution: Any pre-application within F *must* refer to a term within the library file. I.e. if F = <*N* :*A*> δ <**pre** *M* : *B*> π then some term definition within L must have the form <*M* : *B*> ϕ. The pre-application of M in N is connected to the term definition of M in L.

  Post-substitution: *If* <*N* : *A*> identifies a post-application in the library file *then* F is connected to this applictation. I.e if some term definition in L has the form <*M* : *B*> β <**post** *N* : *A*>δ then the post-application of N in M is connected to the form definition of N in T.

- The connected forms may now be analysed and the resulting terms may be stored in the library file.

We have not given an exact definition of "connected". Among other things there is a standard call by value/call by reference problem. The essential operation is that definition and application of forms/terms should be identified. Whether one in fact make a call by value substitution or just a call by reference substitution will depend upon the actual programming system. One may interpret the difference between value and reference substitution in the following way. If you make a value substitution you forget about the source of the copying, i.e. a subsequent change in the source has no effect on the destination. In a reference substitution one remembers the source in such a way that a subsequent change in the source will be propagated to the destination. The propagation should then imply some action all depending on the programming system.

Pre-analysis of a form means:

(a) The form is parsed and converted to an AST.

(b) All declarations in the AST are collected.

(c) All applied names in the AST are connected to their corresponding declaration, which must appear either in the (local) declarations in the AST or in the declarations of one of the connected terms.

(d) All static semantic checks in the AST are performed.

The above definition of pre-analysis is made as simple as possible. The essence of pre-analysis is: For each pre-application the corresponding form definition must be available -- For each post-application it must be possible to analyze the translation unit without accesss to the corresponding form definition.

Further Restrictions

Depending on the actual programming language, it may be necessary to impose certain restrictions upon form applications that may be pre- or post-applications. Below we discuss such restrictions for SIMULA 67 and suggestions to overcome the problems are given.

Consider the program-form using Math and I/O. The same name may be declared in both forms making it impossible to insert both forms without introducing double declarations in the block.

This may be treated in various ways. The analysis module may check such pre-applications for double declarations and announce an error if double declarations appear. Names declared in these forms may be qualified by the name of the form. I.e. Math'sin, I/O'InReal, etc. In this case the resulting program obtained by textually inserting all form applications is no longer a correct SIMULA 67 program unless the syntax of SIMULA 67 identifiers is extended to include identifiers such as Math'sin.

Inclusion of missing form applications should not be able to change the binding of names in the AST or in any other way violate the performed static semantic analysis.

This rule also impose restrictions upon the mechanism. Consider the following form:

```
<Q: block>
begin
        class A; begin procedure pip; <post Pip: block> end ;
        A class B; <post B: block>;
        ref (B) RB;
        RB:- new B;
        RB . pip
end
```

During analysis of Q, the remote identifier RB . pip may be bound to the declaration of pip in class A. However the following B-form implies that this binding is wrong:

<*B: block*> **begin procedure** *pip;* **begin** ... **end end**

RB . pip must now be bound to the declaration of pip in the main part of class B.

In order to overcome this problem, the analysis module may forbid post-applications of forms that contain declarations, that may change the binding of names. This is easy to check, since during search for the declaration of a name one may easily check if an attempt is made to search in a missing post-application.

4. Protection and parameterized data types.

The proposed mechanism makes it is possible to make separate analysis of program fragments as can be done with the constructs of SIMULA 67, PASCAL 3 (the CDC implementation, [PASCAL 3]) and ADA ([ADA]). In SIMULA 67 and PASCAL 3 it is only possible to make separate analysis corresponding to pre-applications. In ADA it is also possible to make separate analysis corresponding to post-applications. In all the languages it is language defined which program fragments that may be separately analysed. In ADA it is also possible to make separate analysis of a "specification" of a package.

We shall give an example of how the proposed mechanism may be used to separately analyse a specification part and an implementation part of an abstract data type to obtain an effect similar to ADA packages. The example is a stack of integers.

The following is a "specification" of the stack:

```
(form
      <Stack: declaration>
      class stack;
      begin  procedure push(e);integer e;
                        <post Push: block>;
             integer procedure pop;
                        <post Pop: block>;
             boolean procedure empty;
                        <post Empty: block>;

             <post Local: declaration>;
             <post Initialize: statement>
      end
form)
```

The implementation of the stack may be as follows:

```
(form
      <Push: block>
      begin a[top]:=e; top:=top+1 end
 //
      <Pop: block>
      begin top:=top-1; pop:=a[top]end
 //
      <Empty: block>
      begin empty:=top=1 end
 //
      <Local: declaration>
      integer array a[1:100]; integer top
 //
      <Initialize: statement>
      top:=1

form)
```

The above example demonstrates another possibility of the mechanism. In a form using a stack, it is only possible to access the push, pop, an empty attributes of stack objects since the declarations in Local are not available. This means that the representation of the stack is protected from illegal access.

It is however necessary to strengthen the definition of connecting a post-application to its corresponding form definition. If connection is interpreted as a textual insertion then the text in the form will be visible after the connection. In the stack example this will mean that forms using the stack after analysis of the stack implementation will know the declarations in Local. The programming system should handle post-applications in such a way that the text in the corresponding form will never be made visible.

The above specification of a stack has fixed the elements of the stack to be of type integer. By letting the type of the stack elements be specified by a post-application it is possible to obtain an effect similar to generics in Ada.

Consider the following specification of a parameterized stack

```
(form
      <Stack: declaration>
      class Stack;
      begin  procedure push(e); <post Elm: value type> e;
                        <post Push: block>;

             <post Elm: value type> procedure pop;
                        <post Pop: block>;
             ....
      end
form)
```

An instantiation of the stack as an integer stack may then be specified in the following way:

```
(form
      <IntStack: declaration> <pre Stack: declaration>
 //   <Elm: value type> integer
form)
```

An instantion of a real stack may be specified by:

```
(form
      <RealStack: declaration> <pre Stack: declaration>
 //   <Elm: value type> real
form)
```

For this use of the mechanism to work, it is necessary to have a more elaborate libraray system than the simple library file assumed in section 3. With the simple library file it is not possible to make more than one instantiation. After the instantiation of IntStack, the post-application Elm has been bound to integer and the subsequent instantiation of RealStack is not possible. A more advanced library system should be able to handle a kind of local bindings. In addition the combination of parameterized data types and the protection mechanism described in the beginning of this section should be properly treated by the library system. Consider the following implementation of the general stack:

```
(form
      <Push: block>
      begin a[top]:=e; top:=top+1 end
 //   ...
 //   <Local: declaration>
      <post Elm: value type> array a[1:100]; integer top
 //   ...
form)
```

The above implementation of Stack is general. It only uses the necessary properties of the variables *a* and *e*. The variable *a* is known to be an array and the array elements and the variable *e* are known to be of the same type and assignable. The library system should make it possible to specify that the above implementation is valid for all instantiations of the stack.

5. Structure of the BETA programming System

The above mentioned tools are designed to be used in an integrated programming system for the BETA language. In the design of such an integrated programming system, it is necessary that the translator and runtime system are organised for this. This means that the translator must be split into modules that operate on well defined intermediate forms of the program. These modules and intermediate forms should then be elements of the programming system.

A BETA programming system is assumed to contain the following modules:

Parse :	Text	→	Abstract Syntax Tree (AST)
Check :	AST	→	Term
Compile :	Term	→	Template
Generate :	Template	→	System

In addition the Run module will start and control the execution of a given system.

The Parse module is an ordinary context free parser that constructs an abstract syntax tree.

The Check module collects declarative information in the AST, connects applied names to their corresponding declaration and performs context sensitive syntax analysis.

The Parse and Check modules together constitute the analysis module treated in section 3.

The Compile module converts terms into loadable code (template) for a given computer.

The Generate module loads a template on a given computer system such that it is ready for execution.

The above modules should be able to handle an arbitrary form when this is possible. The Parse module should be able to construct an AST from any form. The Check module should be able to check an AST when the context of the AST is available as described in section 3.

In order to handle forms in the remaining phases it is necessary to impose further restrictions upon the forms that can be compiled, generated and run. Since these restrictions are language dependent there is little to say about it in general. For the Compile module it should be possible to generate templates such that missing form applications may be inserted by a linking mechanism just as it is done with procedure libraries. The "private" part of an ADA package specification is an example of declarations that must be available in order to make efficient use of a pre-compiled package specification.

In some programming systems ([LISP], [SMALLTALK]) it is possible to execute incomplete programs. This means that the run module at least should be able to handle incomplete program-forms and give some meaningful reaction when an attemp is made to execute a missing form-application. Considerations about observation of BETA program executions are made in [Dahle 81].

The programming system should contain a library or data base where the programmer may store his forms represented as text, AST, term, template or system. The data base should ensure the consistency

between the different representations of a form and should be able to handle different versions of the same form.

The programming system should probably also handle an automatic call of the translator modules in the case where a given form is not available in the required representation. E.g. a call on generate may imply a call of check and compile if the form is only available as AST.

The programming system should also contain a syntax directed editor such that the programmer is able to construct an AST directly. Consequently the editor should work directly on the AST format. The editor and Parse module should be integrated in such a way that for any unexpanded nonterminal in an AST the programmer may choose to expand this nonterminal using a production of the grammar or insert a text using the Parse module. The two reverse grammar operations derivation and parsing are thus available to the programmer.

Acknowledgement. Useful contributions have been made by Dag Belsnes, Norwegian Computing Center.

6. References

[ADA] *ADA Reference Manual.* Proposed Standard Document, United States Department of Defense, July 1980.

[BETA a] B.B. Kristensen, O.L. Madsen. B. Møller-Pedersen, K. Nygaard: *A Survey of the BETA Programming Language.* Norweigan Computing Center, Oslo, 1981.

[BETA b] B.B Kristensen, O.L Madsen, B. Møller-Pedersen, K. Nygaard: *Abstraction Mechanisms in the BETA Programming Language.* In preparation

[Dahle 81] H.P. Dahle: *Observation of BETA-systems.* Norweigan Computing Center, Oslo, 1981 (in Norweigan).

[DELTA a] E. Holbaek Hansen, P. Haandlykken, K. Nygaard: *System Description and the DELTA Language.* Norweigan Computing Center, Oslo, 1975.

[DELTA b] P. Haandlykken, K. Nygaard: *The DELTA System Description Language: Motivation, Main Concepts and Expeirence from use.* In: Software Engineering Environments (ed. H. Hunke), GMD, North-Holland, 1981.

[Demers et al. 81] A. Demers, T. Reps, T. Teitelbaum: *Incremental Evaluation for Attribute Grammars with Application to Syntax-Directed Editors.* Eight Annual ACM Symposium on Principles of Programming Languages, Williamsburg VA, January 1981 (105-116).

[LISP] E. Sandewall: *Programming in the Interactive Environment: The LISP Experience.* ACM, Computing Surveys, vol. 10, no. 1, March 1978 (35-71).

[PASCAL 3] J. Staunstrup, E.S. Jensen: *RECAU PASCAL Manual.* The Regional EDP-Center at The University of Aarhus, 1980.

[Reps 81] T. Reps: *Optimal-time Incremental Semantic Analysis for Syntax-Directed Editors.* Dep. of Computer Science, Cornell University, March 1981.

[SIMULA] O.-J. Dahl, B. Myrhaug, K. Nygaard: *SIMULA 67, Common Base Language.* Norweigan Computing Center, Oslo, 1970.

[SMALLTALK] The Xerox Learning Research Group: *Collection of articles on SMALLTALK-80.* BYTE, August 1981.

[Teitelbaum and Reps 81] T. Teitelbaum, T. Reps: *The CORNELL Program Synthesizer: A Syntax Directed Programming Environment.* Commm. ACM, vol. 24, no. 9, 1981 (563-573).

DAIMI PB-147: Ole Eriksen and Jørgen Staunstrup: *Concurrent Algorithms for Root Searching.* — July 1982. 26 pp. — Dkr. 3.00.

DAIMI PB-148: Henning Christiansen and Neil D. Jones: *Mathematical Foundation of A Semantics Directed Compiler Generator.* — July 1982. 18 pp. — Dkr. 2.25.

DAIMI PB-149: Peter Kornerup and R.T. Gregory: *Mapping Integers and Hensel Codes onto Farey Fractions.* — July 1982. 16 pp. — Dkr. 2.25.

DAIMI PB-150: Kurt Jensen and Morten Kyng: *EPSILON, A System Description Language.* — October 1982.

DAIMI PB-151: Kurt Jensen: *High-level Petri Nets.* — September 1982. 15 pp. — Dkr. 2.25.

DAIMI PB-152: Niels Damgaard Hansen and Kim Halskov Madsen: *Formal Semantics by a Combination of Denotational Semantics and High-level Petri Nets.* — September 1982. 17 pp. — Dkr. 2.25.

DAIMI PB-153: Morten Kyng: *Specification and Verification of Networks in a Petri Net based Language.* — October 1982.

DAIMI PB-154: Sven Skyum: *A measure in which Boolean negation is exponentially powerful.* — September 1982. 8 pp. — Dkr. 2.00.

DAIMI PB-155: Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, Kristen Nygaard: *Syntax Directed Program Modularization.* — September 1982. 13 pp.