Specification and Implementation Problems of Programming Languages Proper for Hierarchical Data Types

Bericht Nr. 8410
Dezember 1984

A. Kreczmar
A. Salwicki
Institute of Informatics
University of Warsaw
PKiN p.o.box 1210
PL-00-901 Warsaw

M. Krause
H. Langmaack
Institut für Informatik und Praktische Mathematik
Universität Kiel
Olshausenstr. 40
D-2300 Kiel 1
Abstract

LOGLAN is a programming language proper for hierarchical data types. LOGLAN is an extension of SIMULA-67 and especially allows prefixing of modules by classes at many levels. This language construct causes semantics specification and implementation problems. In order to study these problems the programming language Mini-LOGLAN is introduced which is a smallest extension of ALGOL-like languages that allows prefixing. Based on the notion of original prefix elimination an algebraic pure static scope semantics of Mini-LOGLAN-programs is given. By means of complement modules and their unique existence a new principle of associating lists of display register numbers to modules is introduced. The number of necessary display registers is bounded by the height of the nesting tree of program modules. The proposed scheme of addressing does not cause display register reloadings while computing in one prefix chain. The designed run time system implementing pure static scoping admits a more efficient implementation of many level prefixing than the existing implementation of LOGLAN with its quasi-static scoping does.

Keywords

LOGLAN, SIMULA, class, many level prefixing, hierarchical data types, static scoping, algebraic semantics, complement modules, implementation, run time system, display registers.
Contents

0. Introduction
1. Semantics specification
   1.1 A contextfree-like grammar for Mini-LOGLAN
   1.2 Basic definitions
   1.3 Binding functions and prefix chains
   1.4 Original prefix elimination
   1.5 Discussion of original prefix elimination
   1.6 Algebraic semantics of programs with prefixing
   1.7 Prefix elimination by transformation into procedures
2. Implementation
   2.1 Complete modules
   2.2 Association of lists of display register numbers
      to modules
   2.3 Design of the run time system for programs with
      many level prefixing
   2.4 Compilation of essential program constructs
   2.5 A run time system with short linkages

Appendices A - H

Literature
0.Introduction

There are many situations in programming which need an appropriate software tool. Let us quote some cases.

1. Abstract data types. Following Hoare [Ho72] one can find his advice of factorization a convenient and useful principle. Let us recall what the principle says: Whenever possible split any reasonable "closed" piece of software into two modules: An abstract program accompanied by a module implementing the data type (i.e. representation of data and operations on them). The advantages of the factorization are easily seen. One can use the implementing module for several abstract programs and/or one can retain the abstract program and change the implementing module in order to gain better efficiency. When we think of separate compilation of modules, the principle of factorization seems to be a good advice. However, there are only a few languages supporting this style of programming.

2. Enforcing certain rules or axioms. The best example is the protocol of mutual exclusion of entry procedures of a monitor.

3. Description of families of data structures.

3.1. It is frequently so that we treat a declaration of a data type as a description of the set of objects which can potentially be constructed and memorized in a computer. In many situations there is a need to develop a hierarchy of (potential) sets of objects. E.g. in the automatization of a bank we must define a hierarchy of various types of records.

3.2. Similarly one can think of hierarchies of abstract data types. Suppose we have defined a problem oriented language as a data structure, an algebra A extended in various ways by structures B,C,... In this way one can arrive at a tree-like structure of problem oriented languages, cf. simulation class in LOGLAN.
3.3. In programming we meet often a need to define and implement
dynamic systems in which objects can also play an active role
(realized either as coroutines or processes).

4. Factorization of algorithms. Sometimes two or more algorithms
have common initial parts (cf. insert, member, delete in binary
search trees). In such situations it is natural to extract the
common part in order to avoid repetitions of text. Obviously one
can achieve the desired result with the help of procedures, but
prefixing all the procedures by a common prefix would be also
interesting.

Regarding the situations 1.-4. we see that in almost every case
we can achieve the desired goal by means of prefixing. Prefixing
which can also be explained as a rule of composition of modules
has been invented by O.J. Dahl, B. Myhrhaug and K. Nygaard [Da70]
and introduced in SIMULA-67 for the first time. In order to
understand prefixing one has to be acquainted with the notion
of class (again SIMULA-67 was the first language which incor-
porated classes).

Prefixing is a two argument operation on modules of programs.
The prefix should be a class, the prefixed module can be of
any kind: class, procedure, function or block. Roughly speaking
the result of prefixing is the module obtained by concatenation
of the declarative parts of two modules and by enclosing the
statement part of the prefixed module by the prologue and epi-
logue coming from the prefixing module. The details will be
explained later. What is more difficult to accept at a first
encountering with prefixing is that the result is not a visible
module. In some sense we operate in a free algebra of modules
with the prefixing operation, i.e. the module

   <name> : <prefix identifier> <prefixed module>

represents the result of prefixing.

This form of program construction has an unexpectedly broad
spectrum of applications. In fact, we can not say that all
possible advantages of prefixing are known already.

The reader should not be misled by a first impression: The concatenation rule can be explained in terms of textual operations, but the realization should not be done by textual operations. Let us recall the analogy between the copy rule for procedures and implementations of procedures in computers.

The history of prefixing can be traced back to SIMULA-67. This attractive software tool has been overlooked for years and the community of software engineers had poor conscience of the possibilities offered by prefixing.

Before we shall pass to further history let us mention a few drawbacks of SIMULA's concept of prefixing. In SIMULA there are two system classes which serve as problem oriented languages: SIMSET and SIMULATION which is prefixed by SIMSET. There is no tool for enlarging the set of system classes however. SIMULA has also a restriction: Both arguments of prefixing operation must be brothers or cousins in the tree of nesting structure of program modules (same level of prefixing), they cannot be in a nephew-uncle relation (multi-level prefixing). Due to this limitation there is no chance to extend the library of system classes. Also separate compilation of modules is difficult and of limited application due to the same reason. LOGLAN a programming language designed and implemented at the Institute of Informatics, University of Warsaw, abandons this limitation. It has turned out however that
1. it is not clear how to understand the prefixing operation if the restriction is abandoned,
2. it is difficult to find an efficient and correct implementation of prefixing by a computer system (compiler plus runtime system). S.Kroghdahl [Kr79] has discussed problems concerning many level prefixing and its implementation. There have been long studies and discussions in the Warsaw group. A first solution has been proposed in 1979 and realized in 1981 by a team led by A.Kreczmar. The results were interesting and of commercial value.
In 1983 H. Langmaack has observed that the implemented semantics of LOGLAN in certain situations does not behave according to the rule of static scoping and that this drawback can be overcome by a new principle of associating display register numbers to modules. His talk at the Zaborow Summer School on LOGLAN 1983 has caused broader interest. Now we present a version which contains contributions of several persons.

Part 1. begins with a presentation of Mini-LOGLAN. This is a smallest extension of the concepts of ALGOL or PASCAL-like languages which admits prefixing. Its abridged form enables to concentrate on main problems of prefixing. The important notions of prefix chain and binding between applied and defining occurrences of identifiers are introduced. Based on the notion of original prefix elimination an algebraic semantics of programs with prefixing is given.

Part 2. of the paper is divided into five chapters. Chapter 2.1 introduces the important notion of complement module and the Main Lemma on unique existence of complement modules is stated and proved. In Chapter 2.2 a new principle of associating lists of display register numbers to modules is proposed which is a crucial point in efficient pure static scope compiler construction for LOGLAN. The number of necessary display registers is bounded by the height of the module tree of a program and nevertheless the proposed scheme of addressing does not cause display register reloadings while computing in one prefix chain. These features essentially improve the efficiency of generated code and run time system as compared with the existing LOGLAN compiler which implements quasi-static scoping. Pure static scoping is not only intellectually more pleasing than other semantics proposals, it admits even a more efficient implementation. Chapters 2.3 and 2.4 contain the design of run time storage and generated code; run time system subroutines are described in Appendix G. Chapter 2.5 shows different run time systems which need less storage place.
1. Semantics specification

1.1 A context-free-like grammar for Mini-LOGLAN

In order to enable a proper treating of semantics specification and implementation of programming languages with prefixing we should like to present in Appendix A the language Mini-LOGLAN which is a smallest extension of ALGOL- or PASCAL-like languages which allows prefixing. The grammar for Mini-LOGLAN is not complete but contains all relevant parts to talk about prefixing.

Modules are blocks, procedures and classes and they can be prefixed by classes (which have no local formal parameters in Mini-LOGLAN). Procedures need not necessarily be prefixed; their prefixing can be simulated by prefixing of their bodies written as blocks. A class initialization

\[
\text{new } \ell
\]

can be simulated by a simple prefixed block

\[
\text{n}: i \text{ block begin end } n
\]

with an empty declaration and statement list. The main part of the statement list \( F \) of a class body contains exactly one simple control statement \( \text{inner} \) with its prologue \( I_1 \) and epilogue \( I_2 \):

\[
F = I_1 \text{ inner } I_2.
\]

The main part of a program piece is that part outside all inner procedure or class modules.

The grammar shows indications where non-standard, non-system identifiers occur in a defining manner. Further defining identifier occurrences are variable identifiers in variable declarations and formal parameter identifiers in formal parameter lists. All other non-standard, non-system identifier occurrences are called applied.

1.2 Basic Definitions

A syntactical Mini-LOGLAN-program (program for short) \( \psi \) is a finite string of lexical entities generated by the Mini-LOGLAN-grammar with <program> as its axiom. Lexical entities are identifiers (including system identifiers = word delimiters = word symbols and standard
identifiers), numbers, strings and delimiters. Every program \( \tau \) has a length \(|\tau| > 0\). \( s \) is called a substring of \( \tau \) iff \( \tau = xsy \). The couple \((i, s)\) is called an occurrence of a substring in \( \tau \) (or shorter: a substring in \( \tau \)) iff \( \tau = xsy \) and \( i = |x|+1 \). We write also \( \bar{s} \) and even \( s \) for \((i, s)\) as soon as no misunderstandings are possible. We shall mainly talk about structured substrings in \( s \); these are substrings in \( \tau \) generated by the (unique because the grammar is unambiguous) structure tree of \( \tau \). The generating subtrees of structured substrings in the structure tree of \( \tau \) are uniquely determined. Two structured substrings are either disjoint or contained in each other.

An occurrence \( i \) in a program \( \tau \) is an integer with \( 1 \leq i \leq |\tau| \). We are especially interested in identifier occurrences \( ^i\xi \) in \( \tau \) (or identifiers in \( \tau \)) where \( \xi \) is a non-system, non-standard identifier and \( ^i\xi \) is a substring in \( \tau \). Unless especially mentioned we shall understand the word "identifier" in this restricted sense.

Positions of defining occurrences of identifiers in a program have been indicated in the grammar, all other occurrences are applied ones. A module occurrence \( ^M \) in a program \( \tau \) starts with block, class or proc and finishes with a matching end-symbol; we speak about block, class or procedure modules respectively. Modules \( ^M \) in \( \tau \) form a tree; they have nesting levels \( \nu = 1 \); the largest module in a program is a block module \( ^1M_1 \) and has level 1. If an occurrence \( i \) or a structured substring \( s \) is in \( ^M \), \( j \leq i < j+|\xi|_1 \), then it has an environment module \( \text{env}(i) \) resp. \( \text{env}(s) \), namely the smallest module \( ^M \) in which \( i \) occurs, \( j \leq i < j+|\xi|_1 \). The associated local identifier list \( \text{locid}(\cdot) \) of a module in \( \tau \) is the ordered list of all defining identifier occurrences \( ^i\xi \) with \( \text{env}(\xi) = ^M \). Ordering in this list means that \( ^i\xi_1 \) is left of \( ^i\xi_2 \) iff \( 1 \leq i_1 < i_2 \).

1.3 Binding Function and Prefix Chains

The binding function \( \text{bdfct}(i, \xi) \) of an identifier \( \xi \) with respect to occurrence \( i \) and the binding function \( \text{bdfctpref}(i, \xi, ^M) \) of an identifier \( \xi \) with respect to occurrence \( i \) in a prefix chain starting
with class identifier occurrence \( i^n \) in \( \pi \) are mutually recursively defined:

\[
\text{bdfct}(i, \xi) =_{DF} \begin{cases} 
\text{if } i \text{ is outside the largest module } M_{1} \text{ in } \pi \\
\text{then if } \pi \text{ is a block named by an identifier equal to the argument } \xi \\
\text{then } \xi \\
\text{else undefined } fi \\
\text{else if } \xi \text{ occurs in locidl}(env(i)) \\
\text{then the rightmost entry } j^\xi \text{ in locidl}(env(i)) \\
\text{else if } env(i) =^j M \text{ has a prefix identifier } j^{-1}n \\
\text{then } bdfctpref(j-1, \xi, j^{-1}n) \\
\text{else } bdfct(j-1, \xi) fi fi fi
\end{cases}
\]

\[
\text{bdfctpref}(i, \xi, j^{-1}n) =_{DF} \begin{cases} 
\text{if } \text{bdfct}(i, \eta) \text{ is a defining class identifier occurrence } \eta \text{ with its class module } \eta M, \eta = k+3 \text{ or } k+2 \text{ depending on whether } \eta M \text{ is prefixed or not} \\
\text{then if } \xi \text{ occurs in locidl}(\eta M) \\
\text{then the rightmost entry } j^\xi \text{ in locidl}(\eta M) \\
\text{else if } \eta M \text{ has a prefix identifier } \eta^{-1} \xi \\
\text{then } bdfctpref(i, \xi, \eta^{-1} \xi) \\
\text{else } bdfct(i, \xi) fi fi fi
\end{cases}
\]

If \( \text{bdfct}(i, \xi) \) is \( j^\xi \) and if \( i \) and \( j \) are inside the largest module \( M_{1} \) in \( \pi \) then the level of \( \text{env}(j^\xi) \) is obviously \( i \) the level of \( \text{env}(i) \).

The binding function of an identifier occurrence \( i^\xi \) in \( \pi \) is

\[
\text{bdfct}(i^\xi) =_{DF} \text{bdfct}(i, \xi)
\]

and the prefix module of a module \( i^M \) in \( \pi \) is

\[
\text{pref}(i^M) =_{DF} \begin{cases} 
\text{if } i^M \text{ has a prefix identifier } i^{-1}n \\
\text{then if } \text{bdfct}(i-1, n) \text{ is a defining class identifier occurrence } \eta \text{ with its class module } \eta M, \eta = k+3 \text{ or } k+2 \\
\text{then } \eta M \\
\text{else undefined } fi \\
\text{else undefined } fi
\end{cases}
\]

\[
\text{pref}(i^M) =_{DF} \begin{cases} 
\text{if } i^M \text{ has a prefix identifier } i^{-1}n \\
\text{then if } \text{bdfct}(i-1, n) \text{ is a defining class identifier occurrence } \eta \text{ with its class module } \eta M, \eta = k+3 \text{ or } k+2 \\
\text{then } \eta M \\
\text{else undefined } fi \\
\text{else undefined } fi
\end{cases}
\]
For $\text{pref}(M) = M'$ we write also $M \rightarrow M'$.

The **prefix chain** of $M$ is the sequence

\[
\ldots \; \text{pref}\,1(M) \rightarrow \text{pref}\,2(M) \rightarrow \text{pref}\,3(M) \rightarrow \ldots
\]

which is of **length** $l > 0$ iff $\text{pref}\,l^{-1}(M)$ is defined but $\text{pref}\,l^{-1}(M)$ is not defined. The module levels from right to left are not increasing, obviously.

We denote the smallest **strict environmental module** of a module $M$ by $\text{strenv}(M)$. If $M$ is named by an identifier $\xi^1$ in front of $M$ then

$$\text{strenv}(M) = \text{env}(\xi^1)$$

or both are undefined (in case $M = M_1$).

Let $M$ be a module occurrence of level $\nu_M$ and let

$$M_1 \leftarrow M_2 \leftarrow \ldots \leftarrow M_{\nu_M - 1} \leftarrow M_{\nu_M} = M$$

with

$$M_{\nu_M} = \text{strenv}(M_1)$$

be the chain of environmental (surrounding) modules of $M$. The **total prefix chain** of $M$ is the list

$$\text{pref}\,1(M_1) \rightarrow \text{pref}\,2(M_2) \rightarrow \ldots \rightarrow \text{pref}\,\nu_M(M_{\nu_M})$$

$M_1$ has no prefix module because a possible prefix identifier would not be bound to a defining class identifier occurrence. If we replace every module $N'$ in the total prefix chain by its local identifier list $\text{locidl}(N')$ and prefix the resulting list by the possible defining identifier occurrence $\xi^1$ in front of the whole program $\tau$ then we get the so called **total identifier list** $\text{totidl}(M)$.

Let us assume that all prefix chains in $\tau$ are finite. Then we may characterize the binding function $\text{bdot}(i, \xi)$ with the help of the notion total identifier list: Let $i$ be an occurrence in the largest module $M_i$ in $\tau$ with $i_{\text{size}} = |M_i|$. Then $\text{bdot}(i, \xi)$ is defined to be $h_\xi$ if and only if there is a rightmost identifier entry $\xi^h$ in $\text{totidl}(\text{env}(i))$ with $\xi = n_\xi$ and $h = j$. 
i, $\xi$ and $\text{bdfct}(i,\xi)=j_\xi$ determine especially a minimal module

$$\text{minmod}(1,\xi)$$

such that $j_\xi$ occurs in the local identifier list of a module $\text{env}(j_\xi)$
in the prefix chain of $\text{minmod}(1,\xi)$:

$$\text{minmod}(1,\xi) \rightarrow \text{env}(j_\xi) \rightarrow \text{env}(i)$$

A closed program $\pi$ is one where every applied identifier $i_\xi$ in $\pi$ is
bound to its associated defining occurrence $j_\xi=\text{bdfct}(i_\xi)$ and where
all prefix chains of modules in $\pi$ are finite. If all applications
in a closed program "make sense" then we talk about a compilable
or proper program. A closed program $\pi$ may be named by a block identifier
$1_\xi$ but $\pi$ has no prefix.

A program $\pi$ is called to be distinguished iff different defining occurrences
of identifiers $i_\xi$ in $\pi$ are denoted by different identifiers $i_\eta$ and free identifiers are different from bound ones; the latter condition is always satisfied in a closed program.

Bound renamings of identifiers in $\pi$ can always generate a so called
congruent distinguished program $\pi_d$.

Appendix $B$ shows a program example $\pi_1$ with the following environmental
and prefix structure:

```
M

1

A x

3

B x

C x

bdfct
```

There are defining occurrences of $x$ in the local identifier lists of
module $M$ and $A$. There are applied occurrences of $x$ in the main parts
of module $A$, $B$ and $C$ as shown in the diagram. The binding functions
of these three occurrences all point into module $A$, no one into $M$,
because $A$ is the prefix of module 1 and 3. Especially, minmod of $x$
in $A$ is class A, of $x$ in $B$ is block 1, and of $x$ in $C$ is block 3.
1.4 Original Prefix Elimination

We shall base the semantics of programs with prefixing on the idea of prefix elimination which makes prefix chains shorter. We call this process original prefix elimination because we shall later discuss a different elimination method.

Let \( \pi \) be a closed program. Let in \( \pi \) a class declaration

\[
(1) \quad \mathsf{class} \; \Delta \; \mathsf{begin} \; \varepsilon \; \mathsf{end} \; \eta
\]

or a block

\[
(2) \quad \mathsf{block} \; \Delta \; \mathsf{begin} \; \varepsilon \; \mathsf{end} \; \eta
\]

be given, prefixed by \( \xi \) which identifies a class

\[
(3) \quad \xi : \xi' \mathsf{class} \; \Delta' \; \mathsf{begin} \; \varepsilon_1' \; \mathsf{inner} \; \varepsilon_2' \; \mathsf{end} \; \xi'.
\]

We have assumed that this class is again prefixed by \( \xi' \) what must not necessarily be the case.

Prefix elimination replaces the class (1) or block (2) by a class (1') or block (2') in the following way:

\[
(1') \quad \mathsf{class} \; \Delta' \; \mathsf{begin} \; \varepsilon_1 \; \varepsilon_2 \; \mathsf{end} \; \eta
\]

or

\[
(2') \quad \mathsf{block} \; \Delta' \; \mathsf{begin} \; \varepsilon_1 \; \varepsilon_2 \; \mathsf{end} \; \eta.
\]

If \( \xi' \) is not existent in (3) then \( \xi' \) is simply not existent in (1') and (2').

Elimination of prefixes of procedures is done in an analogous way. We see especially that replaced modules remain modules of the same kind, namely classes, blocks or procedures.

**Lemma 1:** If \( \pi \) is a closed program then this prefix elimination yields a new closed program \( \pi' \) if

\[
\pi \frac{\text{pref elim}}{} \pi'
\]

if a) \( \pi \) is distinguished or

b) the prefixed class, block or procedure \( \eta \) is outside any prefixed class, block or procedure or

c) class \( \xi \) has no prefix.
If $\tau$ is a proper program then the prefix elimination yields a new proper program $\tau'$ if

1. $\tau$ is distinguished.

1.5 Discussion of Original Prefix Elimination

As an illustration of Lemma 1 Appendix C shows an example of a program $\tau_2'$ which is

3) not distinguished and where

$\beta)$ the prefixed block $\eta=Z$ is inside a prefixed block $A$ and where

3) class $\iota=\eta$ has a prefix $X$.

$\tau_2$ is proper, especially closed

```
   \begin{center}
   \begin{tikzpicture}
   \node (A) at (0,0) {$A$};
   \node (B) at (1,0) {$B$};
   \node (M) at (1,1) {$M$};
   \node (Z) at (-1,0) {$Z$};
   \node (X) at (0,-1) {$X$};
   \node (Y) at (1,-1) {$Y$};
   \node (X') at (2,-1) {$X'$};
   \draw (A) -- (B);
   \draw (B) -- (M);
   \draw (A) -- (Z);
   \draw (Z) -- (X);
   \draw (Y) -- (X');
   \node at (0,-2) {bdftct};
   \end{tikzpicture}
   \end{center}
```

but prefix $Y$ elimination leads to a program $\tau_2'$ which is not proper,

```
   \begin{center}
   \begin{tikzpicture}
   \node (A) at (0,0) {$A$};
   \node (B) at (1,0) {$B$};
   \node (X) at (0,-1) {$X$};
   \node (Y) at (1,-1) {$Y$};
   \node (X') at (2,-1) {$X'$};
   \node (Z) at (-1,0) {$Z$};
   \node (u) at (-1,-2) {u};
   \node at (0,-2) {bdftct undefined};
   \end{tikzpicture}
   \end{center}
```

because the applied identifier occurrence $u$ in block $Z$ has no associated defining occurrence $u$.

The example from Appendix C shows that the elimination of prefixes leading to ALGOL-like programs must be done with due care.
If we rename class \( X \) in block \( A \) in \( \tau_2 \) into class \( X' \) then after prefix \( Y \) elimination \( u \) in block \( Z \) in \( \tau_2' \) is bound to \( \text{var} \ u \) in class \( X \) in class \( B \) what is reasonable:

\[
\begin{array}{c}
\tau_2' \\
A \\
\end{array}
\begin{array}{c}
M \\
\rightarrow \\
B \\
\end{array}
\begin{array}{c}
S \\
\rightarrow \\
X' \\
\end{array}
\begin{array}{c}
Y \\
\rightarrow \\
X \\
\end{array}
\begin{array}{c}
u \\
\rightarrow \\
\text{bdft} \\
\end{array}
\]

Now influential bound renamings for the prefix elimination process are this is demonstrated by Appendix D. We apply prefix elimination to example \( \pi_1 \) of Appendix B and successfully eliminate the prefixes \( A \), \( B \) and \( C \). \( \pi_1'' \) has no prefixes and yields an output

1. \( 2.0, 4.0, 4.0 \)

If we would have made \( \pi_1 \) distinguished by a bound renaming then \( \pi_1'' \) would yield an output

2. \( 2.0, 2.0, 2.0 \)

If we make not only the starting program \( \pi_1 \) but also the intermediate results \( \pi_1' \) and \( \pi_1'' \) distinguished then \( \pi_1'' \) delivers

3. \( 2.0, 2.0, 3.0 \)

The last proceeding (3) follows the ALGOL-like or pure static scope idea whereas "no renaming" (1) is often referred to as dynamic scoping. (2) represents the rationale for the first implementation of LOGLAN \([ Lo63 ]\) which in case of ALGOL- or SIMULA 67-like programs as \( \tau_2 \) \([ Na63, Da70 ]\) works with static scoping and in case of programs with many level prefixing like \( \pi_1 \) adds on "some kind" of dynamic scoping. We call this scoping quasi-static. The new implementation of LOGLAN will follow the pure static scope strategy which is not only intellectually more pleasing but offers even a more efficient implementation as we shall see later in Chapters 2.3 and 2.4.
The original prefix elimination process cannot eliminate all prefixes. Program \( \pi_3 \) in Appendix B is an example of so-called recursive prefixing, i.e., \( \pi_3 \) has an infinite formal execution lattice \( B_3 \), see next chapter, although there are no procedures declared in \( \pi_3 \).

\[ \pi_3: \]

Bound renamings of identifiers have no influence on the identifier binding and the elimination process applied to \( \pi_3 \).

Recursive prefixing is a phenomenon not possible in SIMULA 67-like programs because any module and its prefixes are required to have the same nesting level there.

Resumée of this discussion. Original prefix elimination should only be applied to a program \( \pi \) if prior to every elimination step the programs have been made distinguished by bound renamings (a weaker notion of distinguishability would also work, but we should not like to formulate this here).
1.6 Algebraic Semantics of Programs with Prefixing

We should like to define the \textit{semantics} of Mini-LOGLAN-programs in an algebraic style [Cu81]. We view this style as an abstraction from the operational style as presented e.g. in the ALGOL 60-report [Na63].

There are no difficulties to apply the algebraic method to ALGOL-like languages even with a full procedure and function concept [La73], and now we extend this method to programs with prefixing.

Let a proper program \( \pi \) be given. First we form the \textit{associated formal execution lattice} \( E_\pi \):

We define a \textit{generating relation}

\[ \pi' \vdash \pi'' \]

between certain proper programs \( \pi' \) and \( \pi'' \). Let a distinguished program \( \pi' \) be given. Then we generate a program \( \pi'' \) by looking for one of the following three types of statements in \( \pi' \):

- A correct procedure statement

\[ \text{call } \sigma(a_1, \ldots, a_n) \]

in the main part of \( \pi' \) (\textit{main program of} \( \pi' \)) outside all prefixed blocks.

Here we apply the \textit{copy rule} known already from the ALGOL 60-report. \textit{Correctness} guarantees that the generated program \( \pi'' \) is also proper \(^1\). If procedure \( \sigma \) is prefixed by \( \xi \) then the copy rule produces a so called \textit{generated block} in \( \pi'' \) also prefixed by \( \xi \).

- A class initialization statement

\[ \text{new } \eta \]

in the main program of \( \pi' \) outside all prefixed blocks.

Here the copy rule for classes is applied which acts in an analogous way as the copy rule for procedures where the control statement \textit{inner} in the main part of class \( \eta \) is to be replaced by the empty statement. If class \( \eta \) is prefixed by \( \xi \) then the copy rule produces a generated block also prefixed by \( \xi \), and the gene-

\(^1\) In a theory of copy rule application it is advantageous not to demand that in a proper program all procedure statements are correct; they have to be only "partially correct".
rated program $\pi''$ is proper.

A maximal prefixed block

$$\xi : \text{block} \Delta \text{begin } \xi \text{ end } \xi$$

in the main program of $\pi'$.

Here we apply original prefix elimination and produce a generated block $\xi$ in the newly generated program $\pi''$ which is proper due to Lemma 1.

Now we consider the equivalence classes $[\pi]$ of congruent (boundly renamed) proper programs and define the extended generating relation

$$[\pi'] \rightarrow [\pi'']$$

between classes iff there are representatives

$$\pi' \in [\pi'] \text{ and } \pi'' \in [\pi'']$$

with

$$\pi' \rightarrow \pi''$$

what implies that $\pi'$ must be distinguished.

The formal execution lattice $E_{\pi}$ of a proper program $\pi$ is defined to be

$$E_{\pi} \overset{\text{def}}{=} \{\pi']|\{\pi\} \rightarrow^* [\pi']\},$$

the set of all equivalence classes generated by $[\pi]$ and $\rightarrow^*$.

**Lemma 2:** $(E_{\pi}, \rightarrow^*)$ is a distributive lattice with $[\pi]$ as its least element.

A distributive lattice is always isomorphic with a sublattice of the power set lattice $(\mathcal{P} T, \subseteq)$ of a set $T$. In our special situation $T$ can be chosen to be the set $T_{\pi}$ of all those equivalence classes $[\pi'] \in E_{\pi}$ where the generated blocks occurring in $\pi'$ are all nested in each other.

**Lemma 3:** $(T_{\pi}, \rightarrow^*)$ is a tree $(E_{\pi}, \rightarrow^*)$ and $(E_{\pi}, \rightarrow^*)$ is isomorphic with the sublattice $(\mathcal{P} T_{\pi}, \subseteq)$ of all finite initial trees $I \subseteq T_{\pi}$. If class $[\pi'] \in E_{\pi}$ and initial tree $T' \in T_{\pi}$ are associated due to this isomorphism then the tree of all generated
blocks occurring in \( \pi \) is isomorphic with \( \Gamma' \). \( E_\pi \) is finite if and only if \( T_\pi \) is finite. \( T_\pi \) is the so called formal execution tree of \( \pi \).

Now we reduce all programs \( \pi' \) in \( E_\pi \); we erase all procedure and class declarations and replace all remaining prefixed blocks and all remaining procedure and class initialization statements by the error statement \( \text{error} \). Congruent programs remain congruent by this process.

The semantics \( \Gamma'_{\text{error}} \) is the totally undefined state transformation and the semantics \( \Gamma'_{\text{red}} \) of a reduced program \( \pi'_{\text{red}} \) is a well definable state transformation because \( \pi'_{\text{red}} \) is a block structured program without any procedures or classes. The state transformations \( \Gamma'_{\text{red}} \) of all programs \( \pi' \) in \( E_\pi \) are continuations of each other simply because two different classes \( [\pi'] \) and \( [\pi''] \) in \( E_\pi \) have a common supremum \( [\pi'''] \)

\[
\begin{align*}
\text{[\pi']} & \rightarrow \ast \\
\text{[\pi'']} & \rightarrow \ast \\
\ast & \rightarrow \ast \\
\text{[\pi]} & \rightarrow \ast \\
\end{align*}
\]

in \( E_\pi \). So the union

\[
\bigcup_{[\pi'] \in E_\pi_{\text{red}}} \Gamma'_{\text{red}}
\]

is a well defined state transformation. This one we define to be the semantics \( \Gamma_\pi \) of the proper program \( \pi \). All programs \( \pi' \) in \( E_\pi \) are semantically equivalent

\[
\Gamma'_{\pi'} = \Gamma_\pi
\]

again simply because two classes in \( E_\pi \) have a common supremum in \( E_\pi \).

\(^1\) For technical reasons it is advantageous to call the largest block of any program \( \pi' \) in \( E_\pi \) also a generated block. So program \( \pi \) in \( E_\pi \) where no generating step has been applied has exactly one occurring generated block and this one element tree is isomorphic with the one element initial tree \( [[\pi]] \subseteq T_\pi \).
1.7 Prefix Elimination by Transformation into Procedures

As long as we deal only with SIMULA 67-like programs (within MiniLOGLAN) with prefixing on the same level only then successive original prefix elimination leads to programs which have no longer any class initialization, prefixed block or prefixed procedure. So the remaining classes and their prefixes have become redundant and the resulting programs may be called ALGOL 60-like ones the semantics of which is well known. This proceeding offers another way to define the semantics of programs with prefixing, but it does not work for all Mini-LOGLAN-programs because in Appendix E we have seen an example \( \pi_3 \) with recursive prefixing.

But there is a different prefix elimination process by transforming classes and prefixed blocks into procedures. Let \( \pi \) be a proper distinguished program. We are allowed to assume that there are no class initialization statements nor prefixed procedures in \( \pi \).

Let a non-prefixed class declaration

\[
\eta : \text{class } \Delta \text{ begin } \Gamma_1 \text{ inner } \Gamma_2 \text{ end } \eta
\]

in \( \pi \) be given which defines a module \( ^3M \) in \( \pi \). Let

\[
\xi_1, \xi_2, \ldots, \xi_n
\]

be the local identifier list locidl(\( ^3M \)). inner indicates the only inner-statement in the main part of statement list \( \Gamma \). Then the module above is transformed to

\[
\eta : \text{proc}(\xi); \Delta \text{ begin } \Gamma_1 \text{ call } \xi(\xi_1, \ldots, \xi_n) \text{ end } \Gamma_2 \text{ end } \eta
\]

where \( \eta \) is a new formal procedure identifier with an appropriate specification (which we have deleted). The specification of \( \eta \) is induced by the declarations of \( \xi_1, \ldots, \xi_n \) in a well known way.

Now we consider a prefixed class declaration

\[
\xi : \xi \text{ class } \Delta \text{ begin } \Gamma_1 \text{ inner } \Gamma_2 \text{ end } \xi
\]

in \( \pi \) which defines a module \( ^3M \) in \( \pi \) with its finite prefix chain

\[
\text{pref}^{l-1}(^3M) \leftarrow \cdots \leftarrow \text{pref}^0(^3M), l > 1,
\]

and its local identifier lists
\[ i_1 \xi_1 \ldots i_n \xi_n = \text{locidl}(\text{pref}^0(3M)) \]
\[ j_1 \xi_1 \ldots j_m \xi_m = \text{locidl}(\text{pref}^{1-1}(3M)) \ldots \text{locidl}(\text{pref}^1(3M)). \]

Then the above module is transformed to

\[
\eta : \text{proc}(\eta_g);
\]
\[
\eta_g : \text{proc}(\xi_1, \ldots, \xi_m);
\]
\[
\Delta \begin{tabular}{l}
begin \\
\Sigma, \text{call } \eta_\xi(\xi_1, \ldots, \xi_m, \xi_1, \ldots, \xi_n) \xi_2 \\
end \eta_g; \\
begin \\
\text{call } \xi(\eta_g) \\
end \eta
\end{tabular}
\]

where \( \eta_\xi \) and \( \eta_g \) are new procedure identifiers with appropriate specifications (which we have deleted). The specifications of \( \eta_\xi \) and \( \xi_1, \ldots, \xi_m \) are induced by the declarations of \( \xi_1, \ldots, \xi_n \) and \( \xi_1, \ldots, \xi_n \).

A prefixed block is treated similarly: Let

\[
\eta : \text{& block } \Delta \text{ begin } \Sigma \text{ end } \eta
\]

be such a block in \( \pi \) which defines a module \( 3M \) in \( \pi \). This block is transformed to

\[
\eta : \text{block} \\
\eta_g : \text{proc}(\xi_1, \ldots, \xi_m);
\]
\[
\Delta \begin{tabular}{l}
begin \\
\Sigma \\
and \eta_g; \\
begin \\
\text{call } \xi(\eta_g) \\
end \eta.
\end{tabular}
\]

The symbols have the same meanings as before.
Now we should like to sketch a proof why the given Mini-LOGLAN-program $\pi$ and its transformed ALGOL-like program are semantically equivalent in the sense of the preceding Chapter 1.6.

Let us look at a non-prefixed class

(1) $\xi : \text{class} \; \Delta' \; \begin{array}{c} \xi \; \text{inner} \; \Delta_2 \; \text{end} \; \xi \end{array}$

which is a prefix of a block

(2) $\eta : \xi \; \text{block} \; \Delta \; \begin{array}{c} \xi \; \text{end} \; \eta \end{array}$

Let

$\iota_0, \iota_1, \ldots, \iota_n$

be the local identifier list of class $\xi$. The translated class and block look as follows

(3) $\xi : \text{proc}(\iota_0);$

$$\Delta'$$

$$\begin{array}{c} \xi \; \text{begin} \; \Gamma_1 \; \text{call} \; \xi_0(\iota_1, \ldots, \iota_n) \; \Gamma_2 \; \text{end} \; \xi \end{array}$$

(4) $\eta : \text{block}$

$\begin{array}{c} \eta_0 : \text{proc}(\iota_1, \ldots, \iota_n); \end{array}$

$$\Delta$$

$$\begin{array}{c} \xi \; \text{begin} \; \Gamma \; \text{end} \; \eta_0 \; \text{end} \; \xi \; \text{begin} \; \text{call} \; \xi(\eta_0) \; \text{end} \; \eta \end{array}$$

Now we compare original prefix elimination in (1), (2) and copy rule applications in (3), (4). Prefix elimination gives

$\eta : \text{block}$

$$\Delta'$$

$$\begin{array}{c} \xi_0 \; \text{inner} \; \Delta_2 \; \text{end} \; \xi \end{array}$$

and copy rule applications give
first step:
\[ \eta : \text{block} \]
\[ : \]
\[ \begin{align*}
\text{begin} \\
\text{block} \\
A' \\
\text{begin} \\
\xi_1' \text{ call } \eta_y(z_1, \ldots, z_n) \xi_2'
\end{align*} \]
\[ \text{end} \]
\[ \text{end } \eta \]

second step:
\[ \eta : \text{block} \]
\[ : \]
\[ \begin{align*}
\text{begin} \\
(\ast) : \text{block} \\
A' \\
\text{begin} \\
\xi_1' \\
(\ast) : \text{block} \\
A \\
\text{begin} \\
\xi \\
\text{end } (\ast) \\
\xi_2' \\
\text{end } (\ast) \\
\text{end } (\ast) \\
\text{end } \eta.
\end{align*} \]
If we assume distinctivity for \( n \) and separation for class \( \xi \) and
block \( \eta \), then both copy rule applications do not cause scope binding
errors. If we assume distinctivity for \( n \) and if block \( \eta \) is contained
in class \( \xi \), then both copy rule applications might cause binding
errors in the sense of static scoping. If we do not want them then
global parameters of block \( (*\) may not point into \( \xi' \) of block \( **\).  
Appropriate renamings must be done. Procedure \( \eta_g \) is redundant finally.

**Resumée:** In case of distinctivity of \( n \) and separation of class \( \xi \) and
block \( \eta \), both processes, prefix elimination and transforming rule
plus copy rule applications, lead to essentially equivalent
programs. All other cases

- a block \( \eta \) prefixed by a prefixed class \( \xi \)
- a class \( \eta \) prefixed by a non-prefixed class \( \xi \)
- a class \( \eta \) prefixed by a prefixed class \( \xi \)

lead to analogous results. We say "essential equivalence": We have
full equivalence if the control statement \( \text{inner} \) in (1) does not
occur inside a loop in \( \xi_1 \text{ inner } \xi_2 \). In order to cope also with
this situation the transformation process has to be a little
more complicated.

**Theorem 1:** A Mini-LOGLAN-program and its effectively transformed
ALGOL-like program (all classes and prefixes eliminated) are seman-
tically equivalent.

Appendix F shows the transformed programs \( \pi_1 \) and \( \pi_3 \) of Appendix B
and E.

2. Implementation

Designing an efficient implementation for LOGLAN with many level
prefixing and pure static scope semantics is a severe problem,
much severer than for ALGOL 60 or SIMULA 67. The starting idea is
Dijkstra's [Dik60], namely to enter activation records in a run time
stack when modules are activated, to cancel them when modules are
terminated and to use compile time determinable display (index)
registers and offsets (relative addresses) for fast access to contents
of non-formal variables. Like for SIMULA 67 incarnation (instanti-
tions) of modules in one prefix chain shall be grouped as a so called
object into one activation record and no display register reloading shall be needed when computing in and running through the main parts of modules in a prefix chain. As an illustration look at the run time stack content (pure static scoping) of program \( \tau_1 \) in Appendix B with its environmental and prefix structure in chapter 1.3 just before class C is terminated:

![Diagram of linkage blocks and fixed storage](image)

In SIMULA 67 as in ALGOL 60 or PASCAL it suffices to associate any module \( M \) of level \( \nu_M \) with a list of display registers of numbers 1, 2, \ldots, \( \nu_M \) and to associate any applied occurrence of a variable \( \xi \) with a display register numbered by the level \( \nu_{\text{env(bdfct}(\xi))} \).
But this proceeding does no longer work for many level prefixing. Krogh [Kr79] discusses this for pure static scoping; he recommends in general to make reloadings of display registers when running through a prefix chain and to look for optimizations at compile time which will often be applicable.

The first implementation of LOGLAN uses a 1-1-association of modules and display registers what ends up with a total of 6 registers for \( \pi_1 \). But the implemented semantics is not pure static scope only quasi-static scope, see Chapter 1.5 and Appendix D.

We demonstrate in this paper that pure static scope semantics can get along with a number of display registers bounded by the maximum module level in a program such that no reloadings inside a prefix chain are necessary. So pure static scope semantics is not only the most pleasing one, it admits even a more efficient implementation than other proposals do.

Before we present more formally the general solution let us look how it works for the example from Appendix B.

For \( \pi_1 \) we shall have the following lists of display register numbers:

![Diagram](image)

Please remark that these lists are not monotonous. Further remark that the contents of the applied occurrence of variable \( x \) in class B (\( x \) is defined in class A) are accessed with the help of display register 2 because \( \text{minmod}(x \text{ in } B) \) is block 1 with level 2 which
points to register 2 in the list for B and of variable x in class C
are accessed with the help of display register 4 because minmod(x in C)
is block 3 with level 3 which points to register 4 in the
list for C. We see: The machine instructions for different applied
occurrences of the same variable x in one prefix chain C\rightarrow B may
show up different index register modifications. But pay attention:
These compile time phenomena do not lead to run time inefficiencies.
Many level prefixing is as efficiently implementable as SIMULA 67
with its same level prefixing.

2.1 Complement modules

Let the following diagram in a proper program τ be given:

\[ \begin{array}{c}
    M'' \\
    \uparrow \\
    M' \\
    \longrightarrow M
\end{array} \]

where \( M' = \text{pref}(M) \) and \( M'' = \text{strenv}(M') \).

Then \( M' \) is named by an identifier \( ^{i}_{n} \) (defining occurrence) and \( M \) is
prefixed by \( ^{i}_{n} \) (applied occurrence). Especially \( ^{i}_{n} \) is outside \( M \)
and \( \text{env}^{(i}_{n}) = M'' \). So we have the minimal module \( M'' = \text{minmod}(i,n) \) with

\[ \begin{array}{c}
    M'' \\
    \uparrow \\
    \text{env}^{(i}_{n}) \\
    \uparrow \\
    M \\
    \longrightarrow M'
\end{array} \]

\( M'' \) is the smallest module fulfilling the diagram

\[ \begin{array}{c}
    M'' \\
    \uparrow \\
    \ast \\
    \uparrow \\
    M \\
    \longrightarrow M'
\end{array} \]

and we call \( M'' \) the complement module \( \text{compl}(M,M',M'') \).
Now let a diagram

\[
\begin{array}{c}
M'' \\
\downarrow \quad \ast \\
M \\ \\
\ast & \rightarrow M'
\end{array}
\]

be given. What is the complement module \( \text{comp}(M, M', M'') \) in this case?

The diagram is a \( \rightarrow \) -chain between \( M \) and \( M'' \) and we consider the family of all such chains. This family is finite because \( \ast \) is a proper program and we have no repetitions of modules in such chains. If we define that \( \rightarrow \) precedes \( \rightarrow \) then we have a lexicographical total ordering of precedence among all \( \rightarrow \) -chains.

Now let us replace any subchain

\[
\begin{array}{c}
\ast \rightarrow M' \rightarrow M'' \\
\end{array}
\]

by

\[
\begin{array}{c}
\ast \rightarrow \text{comp}(M, M', M'') \rightarrow M'' \\
\end{array}
\]

which leads to a strictly preceding \( \rightarrow \) -chain between \( M \) and \( M'' \). Finiteness of the family guarantees

Main Lemma: Successive replacing ends up with a uniquely determined least preceding chain which is of the form

\[
M \rightarrow M'' \rightarrow M'''.
\]

\( M'' \) is called the complement module \( \text{comp}(M, M', M'') \). Especially \( \text{comp}(M, M, M') = M'' \), \( \text{comp}(M, M'', M') = M \) and if \( M' \not\equiv M'' \) then \( \text{comp}(M, M', M'') \not\equiv M \).

Intuitively we may say we have paved diagram (\( \ast \)) with pavestones of the type (\( \ast \ast \)) and have ended up with a paved diagram

\[
\begin{array}{c}
\ast \rightarrow M'' \rightarrow M'''
\\
\ast \\
M \rightarrow M'
\end{array}
\]

In general \( M''' \) is not the smallest module fulfilling this diagram.
when $M$, $M'$ and $M''$ are given. Appendix M shows a program example $\pi_4$
with an environmental and prefix structure:

The complement of $E$, $C$, $A$ is $A$:

what can be found out by peaving, whereas $D$ is the minimal environment
fulfilling

when $E$, $C$ and $A$ are given.
Let module $M'$ be in the prefix chain of module $M$

$$M \rightarrow M'$$

We may consider the environmental chain

$$M'_1 \leftarrow M'_2 \leftarrow \ldots \leftarrow M'_{M'-1} \leftarrow M'_M = M'$$

with its level list

$$1, 2, \ldots, M'-1, M'$$

where $M'_1$ is the largest module in $M$. The so-called complement environmental chain of $M, M'$ is defined by

$$M'_1 = M_1 \leftarrow M_2 \leftarrow \ldots \leftarrow M_{M'-1} \leftarrow M_M = M$$

with

where the single diagrams are complemented diagrams. The level list of the complement environmental chain

$$1 = M'_1, M'_2, M'_3, \ldots, M'_M = M$$

is strictly monotonous and is called the complement level list of $M, M'$. 
2.2 Association of lists of display register numbers to modules

Every module $M$ of level $v_M^{-1}$ has to be associated with $v_M$ distinct display register numbers
\[ d_M(1), \ldots, d_M(v_M) \]
with $1 \leq d_M(j) \leq v_M$ for $j = 1, \ldots, v_M$. So $d_M$ is a permutation of the numbers $1, \ldots, v_M$. This association shall fulfill the following:

**Condition:** Let $M'$ be in the prefix chain of $M$
\[ M \rightarrow M'. \]
Then the display register numbers $d_M(j), j = 1, \ldots, v_M$, are to be the same as the display register numbers $d_M(v_M'), j = 1, \ldots, v_M'$ of the complement level list of $M, M'$. Especially $d_M(v_M) = d_M'(v_M') = d_M(v_M)$.

Is such an association $d_M$ of lists of display register numbers to modules $M$ possible? We define $d_M$ by induction over the lexicographical total ordering of couples $(v_M, l_M)$ of level $v_M$ and prefix chain length $l_M$.

**Induction beginning** $(v_M, l_M) = (1, 1)$:
\[ d_M(1) = 1 \]

is the only choice possible.

**Induction step** $(v_M', l_M') > (1, 1)$:
First case $l_M = 1$: Then $v_M > 1$ and there is an $M'$ with $M \rightarrow M'$,
v$_{M'}$ = v$_M$ - 1 and $(v_{M'}, l_{M'})$ precedes $(v_M, l_M)$ lexicographically. So $d_{M'}$ may be assumed to be defined.
\[ d_M(1) = \begin{cases} d_M(1) & \text{for } i = 1, \ldots, v_{M'} \\ v_M & \text{for } i = v_M \end{cases} \]

Second case $l_M > 1$: Then there is an $M'$ with $M \rightarrow M'$,
In case of ALGOL 60- or SIMULA 67-like programs with its same level prefixing we get the display register numbers association known from the literature. We easily prove by induction over the length of \( \rightarrow \) -chains.

**Lemma 4:** The above mentioned condition is fulfilled and the complement level list of \( M, M' \) can be described by

\[
\nu_{M} = d_{N}^{-1} \circ d_{M}(j), \quad j=1, \ldots, \nu_{M}'.
\]

where \( d_{N}^{-1} \) is the inverse permutation of the numbers \( i=1, \ldots, \nu_{M} \).

For the program \( \pi_{2} \) of the Appendices C, E and F we have the following association:

\[
\pi_{2} \text{(SIMULA 67-like)}
\]

![Diagram](image)
For program $\tau_1$ in Appendix B we have seen the association already in the introduction to part 2.

2.3 Design of the run time system for programs with many level prefixing

Let $m$ be a distinguished proper program. Every module $M$ has a certain fixed storage amount.

$$\text{fst}(M) = 0$$
determined by the declarations of all variables $^j_x$ with $\text{env}(^j_x) = M$.

$\text{fst}(M)$ is known at compile time. When $M$ is activated then an activation...
record of the whole prefix chain of \( M \)
\[ M = \text{pref}^0(M) \rightarrow \text{pref}^1(M) \rightarrow \ldots \rightarrow \text{pref}^{l-1}(M) \]
of length \( l > 0 \) is entered into the run time stack with a fixed storage amount
\( \text{fstat}(M) = x + \text{fstat}(\text{pref}^{l-1}(M)) + \ldots + \text{fstat}(\text{pref}(M)) + \text{fstat}(M) \geq 0. \)

\( x \) is a number \( > 0 \) known at compile time; \( K \) is the storage amount for the linkage of an activation record. So \( \text{fstat}(M) \) is known at compile time.

Let \( \text{pref}(x) \) be a defining occurrence of a variable with \( \text{env}(\text{pref}(x)) = M \). The compiler reserves a storage cell in the fixed storage of \( M \). This cell has a compile time known relative address or offset in the storage for the prefix chain of \( M \). So
\[ \text{fstat}(M) > \text{reladdr}(\text{pref}(x)) \Rightarrow \text{fstat}(M) - \text{fstat}(M). \]

Let \( \text{pref}(x) \) be an applied occurrence of a variable in the main part of module \( M^* = \text{env}(\text{pref}(x)) \) and let \( \text{bdstat}(\text{pref}(x)) = \text{pref}(x) \) and \( \text{minmod}(i, x) = R \):
\[
\begin{array}{c}
\text{R} \\
\text{M}^* \\
\end{array}
\]

Let e.g. \( \text{pref}(x) \) be the right hand side of an assignment statement
\[ \ldots := x; \ldots \]

Then we would like to compile this into a load instruction (assembler language)
\[ \text{LDA } i^x, \text{ reladdr}(\text{pref}(x)) \]
which is read
"load accumulator from a cell with address reladdr(\text{pref}(x)) which is modified (increased) by the content of index (display) register of number \( i^x \)."

In a higher level assembler language this instruction would look as follows
\( \mathcal{M}i^X \) is a linear array representing the main storage of memory cells and \( \mathcal{S}^i \) is a linear array representing the series of index or display registers.

What display register number \( d^i \) do we take? We take

\[
d^i = d^* \mathcal{S}^i(v^X).
\]

We demonstrate this definition for the compilation of the assignment statement

\( x := y; \)

in class B of program \( \tau \),

\[
\mathcal{M}[\mathcal{S}[2] + \text{readdr}(x)] := \mathcal{M}[\mathcal{S}[2] + \text{readdr}(y)];
\]

and of

\( y := x; \)

in class C:

\[
\mathcal{M}[\mathcal{S}[4] + \text{readdr}(y)] := \mathcal{M}[\mathcal{S}[4] + \text{readdr}(x)];
\]

Please remember: Display registers are to be loaded or reloaded only if a block \( \xi \) is entered or terminated, a procedure \( \rho \) is called or terminated or a class \( \eta \) is initialized by \texttt{new} \( \eta \) or terminated.

No reloadings shall happen when running through the main parts of the prefix chain of a module.

An activation record begins with \( K \) cells for linking with the following relative addresses and contents:

Relative address 0, mnemonic technically denoted = RA: \texttt{Return address} in the compiled program, where control has to go after regular termination.

Relative address 1 = DLD: \texttt{Dynamic level} of the dynamic predecessor of this activation record, i.e. the address of the return address cell of the immediately preceding activation record of this activation record.
Relative address $2 \equiv \text{ID: Identifier}$ of the activated block or procedure or initialized class.

Relative address $3 \equiv \text{DS: N cells for the dynamic levels of the static chain of this activation record. Nz1 is the maximal nesting level $v_M$ of all modules M in a program \pi.}$

If the activated module M has a level $v_M \geq 1$ then the first $v_M$ cells have relevant contents; the display registers $\mathcal{D}[1], \ldots, \mathcal{D}[v_M]$ are loaded resp. reloaded when this activation record is resp. becomes again the topmost entry of the run time stack and module M is activated resp. reactivated. The contents of the other $N-v_M$ cells are undefined.

Relative address $N+3 \equiv \text{LG: Length of this activation record (relevant only for programs with global jumps).}$

So $K$ is the number $N+4$ which is known at compile time.

The compiled program acts upon a run time stack in the main storage which is considered as an array

```plaintext
var \mathcal{M} : \text{array} \{0:=\} \text{ of something};
```

The series of display registers forms an array

```plaintext
var \mathcal{D} : \text{array} \{1:=\} \text{ of \{0:=\}};
```

The momentary dynamic level which shows to the return address cell of the momentary topmost activation record entry in the run time stack is held in a simple variable

```plaintext
var \text{MDL} : \{0:=\};
```

The momentary free storage level is held in a simple variable

```plaintext
var \text{FSL} : \{0:=\};
```

Further auxiliary variables

```plaintext
var \text{AUX, AUX1 : \{0:=\}};
```

are used for procedure calls.
2.4 Compilation of essential program constructs

Let a distinguished proper program $\Gamma$ be given such that w.r.o.g. every block has a block identifier.

I. A program $\Gamma$ is a block
   $\eta$ : block
   $\Delta$
   begin
   $\Sigma$
   end $\eta$

and is compiled this way:
   call initialization;
   compiled $\Delta$
   compiled $\Sigma$
   call finish program;

II. A non-prefixed block different from the whole program $\Gamma$
   $\eta$ : block
   $\Delta$
   begin
   $\Sigma$
   end $\eta$

with block module $M_\eta$ is compiled this way$^1$:
   call blockentering($\eta$);
   Start $\Gamma$ of $M_\eta$.
   compiled $\Delta$
   compiled $\Sigma$
   call finish;
   End of $M_\eta$;

---

1) More efficient code will be generated if non-prefixed blocks $\eta$ are treated like statements. They can be treated like special modules $M_\eta$ without associated nesting level $\nu_\eta$. Compilation is simply:
   compiled $\Delta$
   compiled $\Sigma$. 
III. A prefixed block

\[ \delta \]

begin
  \[ \Sigma \]
end \[ \delta \]

with block module \( M_\eta \) is compiled this way: Let \( \xi \) denote a class with class module \( M_\xi \). The translator reserves a variable \( x_\xi \) in the fixed storage of \( M_\xi \):

\textbf{call} prefixed block entering(\( \eta \));

Start 1 of \( M_\eta \):

\[ \mathcal{M}[\mathcal{N}[d_{M_\eta}(v_{M_\eta})]+\text{reladdr}(x_\xi)] := \text{Start 2 of } M_\eta; \]

\textbf{goto} Start 1 of \( M_\xi \);

Start 2 of \( M_\eta \); compiled \( \delta \)

Start 3 of \( M_\eta \):

\[ \mathcal{M}[\mathcal{N}[d_{M_\eta}(v_{M_\eta})]+\text{reladdr}(x_\xi)] := \text{Start 4 of } M_\eta; \]

\textbf{goto} Start 3 of \( M_\xi \);

Start 4 of \( M_\eta \); compiled \( \Sigma \)

\textbf{goto} After inner of \( M_\xi \);

End of \( M_\eta \);

Remember: \( x_\xi \) in block \( \eta \) is to be treated as an applied occurrence with its defining occurrence in class \( \xi \). So \( \text{minmod}(x_\xi)=M_\eta \) and the display register to be compiled is \( d_{M_\eta}(v_{M_\eta}) \) which is equal to \( d_{M_\xi}(v_{M_\xi}) \) due to Lemma 4. So it makes no difference whether we write \( d_{M_\eta}(v_{M_\eta}) \) or \( d_{M_\xi}(v_{M_\xi}) \).
IV. A non-prefixed class
\[ \eta : \text{class} \]
\[ \Delta \]
begin
\[ \Sigma_1 \text{ inner } \Sigma_2 \]
end \[ \eta \]
with class module \( M_\eta \) is compiled this way:

The translator reserves a variable \( x_\eta \) in the fixed storage of \( M_\eta \):
Start 1 of \( M_\eta \):
Start 2 of \( M_\eta \):
compiled \( \Delta \)
\[ \text{goto} \ [\mathcal{M}[\mathcal{V}[d_{M_\eta}(v_{M_\eta})]+\text{reladdr}(x_\eta)] \}; \]
Start 3 of \( M_\eta \):
Start 4 of \( M_\eta \):
compiled \( \Sigma_1 \)
\[ \text{goto} \ [\mathcal{M}[\mathcal{V}[d_{M_\eta}(v_{M_\eta})]+\text{reladdr}(x_\eta)] \}; \]
After inner of \( M_\eta \):
compiled \( \Sigma_2 \)
call \text{finish};
End of \( M_\eta \);

V. A prefixed class
\[ \eta : \epsilon \text{ class} \]
\[ \Delta \]
begin
\[ \Sigma_1 \text{ inner } \Sigma_2 \]
end \[ \eta \]
with class module \( M_\eta \) is compiled this way:
Start 1 of \( M_\eta \):
\[ \mathcal{M}[\mathcal{V}[d_{M_\eta}(v_{M_\eta})]+\text{reladdr}(x_\eta)] \] := Start 2 of \( M_\eta \);
\[ \text{goto} \] Start 1 of \( M_\epsilon \);
Start 2 of \( M_\eta \):
compiled \( \Delta \)
\[ \text{goto} \ [\mathcal{M}[\mathcal{V}[d_{M_\eta}(v_{M_\eta})]+\text{reladdr}(x_\eta)] \}; \]
Start 3 of \( M_\eta \):
\[ \mathcal{M}[\mathcal{V}[d_{M_\eta}(v_{M_\eta})]+\text{reladdr}(x_\eta)] \] := Start 4 of \( M_\eta \);
\[ \text{goto} \] Start 3 of \( M_\epsilon \);
Start 4 of $M_5$:
    compiled $I_3$
    goto $\mathcal{M}(\alpha^{\uparrow}[d_{M_5}(v_{M_5})] + \text{readdr}(x_5))$;

After inner of $M_5$:
    compiled $I_2$
    goto After inner of $M_5$;
End of $M_5$;

VI. A non-prefix procedure declaration

$\phi : \text{proc} (\xi_1, \ldots, \xi_n)$;

\begin{align*}
&\Delta \\
&\begin{array}{l}
\text{begin} \\
\Sigma \\
\text{end } \phi
\end{array}
\end{align*}

with its procedure module $M_\phi$ is compiled this way:

Starting address of procedure $\phi$:
    compiled $\Delta$
    compiled $\Sigma$
    call finish;
End of $M_\phi$;

VII. A prefix procedure declaration

$\phi : \xi \text{ proc} (\xi_1, \ldots, \xi_n)$;

\begin{align*}
&\Delta \\
&\begin{array}{l}
\text{begin} \\
\Sigma \\
\text{end } \phi
\end{array}
\end{align*}

with its procedure module $M_\phi$ is compiled this way:

Starting address of procedure $\phi$:
    $\mathcal{M}(\alpha^{\uparrow}[d_{M_\phi}(v_{M_\phi})] + \text{readdr}(x_5)) := \text{Start 2 of } M_\phi$;
    goto Start 1 of $M_5$;
Start 2 of $M_\phi$:
    compiled $\Delta$
Start 3 of $M_\phi$:
    $\mathcal{M}(\alpha^{\uparrow}[d_{M_\phi}(v_{M_\phi})] + \text{readdr}(x_5)) := \text{Start 4 of } M_\phi$;
    goto Start 3 of $M_5$;
Start 4 of \( M_o \):
compiled \( E \)
\textbf{goto} after inner of \( M_o \):
End of \( M_o \):

VIII. A non-formal procedure statement
\\[ \text{call } \phi(a_1, \ldots, a_n) ; \]

occurring in the main part of module \( M^* \) with module identifier \( \chi \)
is compiled this way (we assume that no actual parameter \( a_1 \) invokes
an implicit module activation):

\begin{itemize}
  \item compilation of \( a_1 \)
    \[ \mathcal{M}[\text{FSL+reladdr}(\xi_1)] := \text{actual information about } a_1 ; \]
  \item \vdots
  \item compilation of \( a_n \)
    \[ \mathcal{M}[\text{FSL+reladdr}(\xi_n)] := \text{actual information about } a_n ; \]
\end{itemize}

\( \xi_1, \ldots, \xi_n \) are the formal parameters of procedure \( \phi \) corresponding
to \( a_1, \ldots, a_n \). Their storage cells are in the fixed storage of \( \phi \)
and their relative addresses are known at compile time.

Let \( i_\phi \) be the applied occurrence of \( \phi \) immediately behind \text{call}.
Let \( \mathcal{M} = \text{minmod}(i, \phi) \) be the minimal module with \( \mathcal{M} \rightarrow M^* \).
If \( d_{M^*} = d_{\mathcal{M} \# [1, v_M]} \) and the module \( \mathcal{M} \# M^* \) to be entered is not prefixed
then compilation of \text{call } \phi \) is
\begin{itemize}
  \item call simple non-formal procedure(\( i_\phi \));
  \item Return address of procedure call;
\end{itemize}
otherwise
\begin{itemize}
  \item call non-formal procedure(\( \chi, \phi \));
  \item Return address of procedure call;
\end{itemize}

If case the non-formal procedure statement
\begin{itemize}
  \item call \( \phi(a_1, \ldots, a_n) \)
\end{itemize}
is only partially correct, but not correct then the statement
is compiled into
\begin{itemize}
  \item \textbf{error};
\end{itemize}
See the discussion about proper programs in Chapter 1.5.
IX. Let $a_i$ be an actual parameter occurring in the main part of
module $M^*$ and let $\xi_i$ be the corresponding formal parameter.
What compiled code does compute the actual information about $a_i$?
We have to differ between five cases IX.a to IX.e.

IX.a. Let $a_i$ be a non-formal procedure identifier. Let $v_R$ be
\text{minmod}(a_i)$ with $R \leftarrow M^*$. The actual information about $a_i$ is a couple
$(\star) \ (a_i, \text{content of } \nu[d_{M^*}(v_R)])$.
This information is computed by the code
\[ a_i \circ d_{M^*}(v_R) \]
where $\circ$ is a "machine operation" which couples the procedure identifier $a_i$
with the content of display register $\nu[d_{M^*}(v_R)]$ numbered
by $d_{M^*}(v_R)$.

IX.b. Let $a_i$ be a formal procedure identifier. The actual information about a couple like that above $(\star)$ is the content of
\[ \mathcal{M}(\nu[d_{M^*}(v_R)]+\text{reladdr}(a_i)), \]
and this is the code which computes the information (a load instruction with index register modification).

IX.c. Let $a_i$ be an expression (e.g. of type real) and $\xi_i$ be
a formal input parameter (e.g. also of type real in order to
avoid type transfers). The actual information about $a_i$ is a
real number computed by the compiled code of $a_i$.

IX.d. Let $a_i$ be a non-formal simple variable (e.g. of type real)
and $\xi_i$ be a formal output variable (necessarily also of type
real due to partial correctness). The actual information about
$a_i$ is an absolute address, namely the sum
\[ \nu[d_{M^*}(v_R)]+\text{reladdr}(a_i), \]
and this is the code which computes the information (a load and
an add instruction).
IX.e. Let $a_1$ be a formal output variable (e.g. of type real) and $\delta_1$ be a formal output variable (necessarily also of type real). The actual information about $a_1$ is an absolute address, namely the content of

$$\Psi[\sigma_1 \text{ M}_1 + \text{reladdr}(a_1)]$$

and this is the code which computes the information (a load instruction with index register modification, compare IX.b.).

X. A class initialization statement

```plaintext
new n;
```

occurring in the main part of module $M^*$ with module identifier $\chi$ is compiled this way:

```
compilation of init $\eta$
Start 1 of new:
$$\Psi[\sigma_1 \text{ M}_1 + \text{reladdr}(x_\eta)] := \text{Start 2 of new;}$$
goto Start 1 of M_1;
Start 2 of new:
Start 3 of new:
$$\Psi[\sigma_1 \text{ M}_1 + \text{reladdr}(x_\eta)] := \text{After inner of M}_1;$$
goto Start 3 of M_1;
Start 4 of new:
Return address of class initialization:
The compiler reserves a variable $x_\eta$ in the fixed storage of the class module M_1.
```

Compilation of init $\eta$ is done similar to call $\gamma$ in VIII. and gives
call simple class($\eta$);
respectively
call class($\chi, \eta$);.
XI. A formal procedure statement

\texttt{call \psi(a_1, \ldots, a_n);} \\

occurring in the main part of module \( M^* \) with its module identifier \( \chi \) is compiled this way (we assume that no actual parameter \( a_i \) invokes an implicit module activation):

\begin{verbatim}
 call prepare formal procedure(\chi, \psi);
 compilation of \( a_1 \\
 M[\mathrm{AUX}+\mathrm{reladdr}(\ell_1)] := \text{actual information about } a_1;
 call check actual parameter(1);

 \vdots

 compilation of \( a_n \\
 M[\mathrm{AUX}+\mathrm{reladdr}(\ell_n)] := \text{actual information about } a_n;
 call check actual parameter(n);
 call formal procedure;
\end{verbatim}

Return address of formal procedure call:

\( \ell_1, \ldots, \ell_n \) are the \textit{fictitious formal parameters} of the formal procedure \( \psi \) corresponding to \( a_1, \ldots, a_n \). Their relative addresses are \( k^{-1}+1, \ldots, k^{-1}+n \), i.e. \( \psi \) is treated as if \( \psi \) had no prefix.

Only in case \( \ell_1 \) is a formal procedure

\begin{verbatim}
 call check actual parameter(1);
\end{verbatim}

needs to be compiled.

After this specification of code generation in section I. to XI. the subroutines

\begin{verbatim}
 initialization,
 blockentering(n),
 finish,
 prefixed blockentering(n),
 simple non-formal procedure(e),
 non-formal procedure(\chi, \psi),
 simple class(n),
 class(\chi, n),
 prepare formal procedure(\chi, \psi),
 check actual parameter(i),
 formal procedure
\end{verbatim}

must be described. This will be done in Appendix G. A detailed proof of the correctness of this implementation will be given in a further publication.
Theorem 2: A proper Mini-LOGLAN-program \( \tau \) and its compiled program have the same state transformation resp. input/output function as their semantics.

2.5. A runtime system with short linkages

The runtime system presented in chapter 2.3 and Appendix G is time efficient but space consuming because each linkage demands \( N^2 \) cells to store dynamic levels of static chains. We want to make the linkages shorter. We reserve only one cell with relative address \( 3 \times \text{DLS} \) for the immediate static predecessor which is the content of the old \( M (= \text{dynamic level} + \text{DLS} - 1 + v_0 \cdot - 1) \) in case of the activated module \( M \) is \( \# M_1 \). The new relative address for the activation record length is \( 4 \times \text{LOG} \) and the new linkage length is \( K = 5 \). If we successively go down the immediate static predecessors then we get the pseudo static chain of an activation record which the static chain is a part of.

The main problem for the reorganized runtime system is to determine the static chain and the proper display registers loading when an activation record of a certain dynamic level \( d \) is created or reactivated:

```plaintext
display registers loading: subroutine (d: dynamic level);
begin
  if d=0 then \( M[1] := 0 \)
  else call display registers loading(\( M[d + \text{DLS}] \));
  let be the module identifier in cell \( M[d + \text{ID}] \).
  let \( M' = \text{strenv}(M_0) \) with \( M_0 \rightarrow M' \) and \( v_0 = v_{0,1} \).
  We do the simultaneous assignment
    \[
    \begin{pmatrix}
    M[d_0(1)] \\
    \vdots \\
    M[d_0(v_0-1)] \\
    M[d_0(v_0)]
    \end{pmatrix} :=
    \begin{pmatrix}
    M'[d_0(1)] \\
    \vdots \\
    M'[d_0(v_0)] \\
    d_l
    \end{pmatrix}
    \]
end display registers reloading
```
Let us explain what the simultaneous assignment actually does:

Let n be the module identifier in \( \mathcal{M}(\mathcal{M}(d_1+DLS)+\mathcal{M}) \). The preceding display registers reloading has given us the static chain of \( \mathcal{M}(d_1+DLS) \) in the form

\[
\begin{pmatrix}
\forall [d_n(1)] \\
\vdots \\
\forall [d_n(\nu_m-1)] \\
\forall [d_n(\nu_m)]
\end{pmatrix}
\]

\( M' \) is in the prefix chain of \( \mathcal{M}_n : M_n \xrightarrow{\mathcal{M}} M' \) with \( \nu_m \leq \nu_m \).

\( d_{M_n}^{-1} : d_{M_n}([1:\nu_m]) \) represents the complement level list of \( M_n, M' \).

The static chain of \( M' \) is a subchain of the static chain above, namely

\[
\forall [d_n(\nu_m)] = \forall [d_n(\nu_m)]
\]

for \( i=1, \ldots, \nu_m \). If we add dl then we have the static chain of \( \mathcal{M}_o \) resp. dl which is stored in

\[
\begin{pmatrix}
\forall [d_o(1)] \\
\vdots \\
\forall [d_o(\nu_m-1)] \\
\forall [d_o(\nu_m)]
\end{pmatrix}
\]

Essential changes for the reorganized run time system are necessary only for the subroutines finish and formal procedure. The simultaneous assignment in finish is replaced by display registers reloading (MDL),

in formal procedure by

\[
\mathcal{M}(FSL+DLS) := AUX; \\
display registers reloading (FSL);
\]

It is easy to transform the recursive subroutine display registers reloading into a more efficient iterative one. Activation records created by

- call blockentering,
- call simple non-formal procedure or
- call simple class
should be specially marked. When such activation of a module \( M \) is finished and module \( M' = \text{strenv}(M) \) is not in the prefix chain of any module \( \tilde{M} \) with \( \text{vtot}(\tilde{M}) \neq \text{vtot}(M) \), then display registers reloading (MDL) needs not to go down to dynamic level 0 but only to \( \tilde{M}(\text{FSL+DLS}) \).

Subroutine display register reloading in some sense passes the genesis of an activation record. We now present a subroutine which helps to reconstruct the history of an activation record step by step. The subroutine is written as a function DLSP in a LOGLAN-like style which computes the dynamic level of the immediate static predecessor of an activation record with dynamic level \( dl \) and with \( \tilde{M}(dl + ID) = \phi \) with respect to a module identifier \( \xi \) with \( \text{vtot}(\tilde{M}) \neq \text{vtot}(M) \). If \( \xi = \phi \) then DLSP \((dl, \xi) = \tilde{M}(dl+DLS)\).

**DLSP** function \((dl: \text{dynamic level}, \xi: \text{module identifier}): \text{dynamic level};\)

\begin{verbatim}
var \( \xi': \text{module identifier}; i: \text{integer}; \)
begin
result := \( \tilde{M}(\text{addr+DLS}) \);
\( \xi' := \text{id} \) where \( \text{vtot}(\tilde{M}(dl+ID)) = \text{strenv}(\text{vtot}(M_{\xi})) \);
for \( i := 2 \) to \( \text{vtot}(\tilde{M}(dl+ID)) \) do
result := DLSP(result, \( \xi' \));
\( \xi' := \text{id} \) where \( \text{vtot}(\tilde{M}(dl+ID)) = \text{strenv}(\text{vtot}(M_{\xi})) \)
end DLSP
\end{verbatim}

**Lemma:** Let \( dl \) be the dynamic level of an activation record with \( = \tilde{M}(dl+ID) \) and \( \text{vtot}(\tilde{M}) > 1 \). Let \( \xi \) be a module identifier such that \( \tilde{M}(dl+ID) = \text{strenv}(\text{vtot}(M_{\xi})) \). Then DLSP \((dl, \xi) = dl'\) such that there exists a \( \text{KNOE} \) with \( \text{vtot}(\tilde{M}(dl'+ID)) = \text{compl}(\text{vtot}(M_{\xi}), \tilde{M}(dl+ID)) \), i.e., \( dl' \) is the immediate static predecessor of \( dl \) w.r.t. \( \xi \).

Let \( |dl| \) denote the length of the pseudo static chain of the activation record with dynamic level \( dl \). If \( |dl| = 2 \) then for every \( \xi \) with \( \text{vtot}(\tilde{M}) = 2 \) holds. \( \tilde{M}(dl+DLS) \) is the dynamic level of the activation record of module \( M_{\xi} \) with \( \text{vtot}(\tilde{M}) = 1 \) and is returned as value of DLSP \((dl, \xi) \) since \( M_{\xi} = \text{compl}(\tilde{M}(dl'+ID), \tilde{M}(dl+ID)) \).
Let $|dl|>2$ and assume that for every $dl''$ with $|dl''|<|dl|$ in the pseudo static chain of $dl$ the lemma already holds.

If $\psi = \xi$, then $h=0$ and $\mathcal{M}(\mathcal{M}(\mathcal{M}(dl+DLS)+ID)=\text{compl}(M_\psi, M_\xi, \text{strenv}(M_\xi))=\text{strenv}(M_\xi)$.

$\mathcal{M}(dl+DLS)$ is returned as value of DLSP($dl, \xi$) because the for-loop will not be executed.

If $\check{\psi} = \text{Df}$, $\check{\psi} = \text{compl}(M_\psi, M_\xi, \text{strenv}(M_\xi))$ then $\text{strenv}(M_\psi)=\text{compl}(M_\psi, M_\xi, \text{strenv}(M_\xi))$, i.e. there is a $k=0$ with $\text{strenv}(M_\psi)=\text{pref}^k(\text{strenv}(M_\psi))$. Thus $\mathcal{M}(dl+DLS)$ is the dynamic level of the immediate static predecessor of $dl$ w.r.t. $\xi$ as well as w.r.t. $\psi$ and is returned as value of DLSP($dl, \xi$) because the for-loop will not be executed.

\check{\psi}2: Considering the nesting tree of modules we have the following situation:

\[
\text{compl}(M_\psi, M_\xi, \text{strenv}(M_\xi)) \rightsquigarrow \text{strenv}(M_\xi)
\]

\[
\text{strenv}(M_\xi) \uparrow
\]

\[
M_\xi \overset{\check{\psi}}{\rightarrow} \text{compl}(M_\psi, M_\xi, \text{strenv}(M_\xi))
\]

We have to compute the $\check{\psi}$ static predecessor of $dl$ w.r.t. "the path from $M_\psi$ to $\text{compl}(M_\psi, M_\xi, \text{strenv}(M_\xi))"$.

The first one is the immediate static predecessor of $dl$ w.r.t. $\psi$, thus it has the dynamic level $\mathcal{M}(dl+DLS)$. Now for every of the $\check{\psi}-1$ iterations of the for-loop we call DLSP with a dynamic level $dl''$ of an activation record such that $|dl''|<|dl|$. The $\check{\psi}$-th static predecessor is also the immediate static predecessor of $dl$ w.r.t. $\xi$ and is returned as value of DLSP($dl, \xi$).
Again essential changes for the runtime system are necessary only for the subroutine finish and formal procedure.

The simultaneous assignment in finish is replaced by

\[ \forall \{d_{M_n}(v_{M_n})\} := \text{MDL}; \]

\[ \xi := \#(\text{MDL} + \text{ID}); \]

for \( i = v_{M_n} - 1 \) downto 2

\[ \forall \{d_{M_n(i)}\} := \text{DLSP}(\forall \{d_{M_n(i+1)}\}, \xi); \]

\[ \xi := \xi' \text{ where } M_{\xi'}, = \text{strenv}(M_{\xi}) \]

od;

in formal procedure by

\[ \forall \{FSL + DLS\} := \text{Aux}; \]

\[ \forall \{d_{M_0}(v_{M_0})\} := \text{FSL}; \]

\[ \xi := \#(\text{FSL} + \text{ID}); \]

for \( i = v_{M_0} - 1 \) downto 2

\[ \forall \{d_{M_0(i)}\} := \text{DLSP}(\forall \{d_{M_0(i+1)}\}, \xi); \]

\[ \xi := \xi' \text{ where } M_{\xi'}, = \text{strenv}(M_{\xi}) \]

od;

To load the required display registers only one call of subroutine display register loading is necessary whereas the user of DLSP must know how often it must be called. In both cases one has to go to the end of the pseudo static chain. Subroutine display register loading does operations on the display registers, i.e. intermediate results are held in them, when going back to the beginning of the pseudo static chain. DLSP holds its intermediate results in local variables. Operations on display registers must be done outside of DLSP when going to the end of the pseudo static chain.
Appendix A: A context-free-like grammar for Mini LOGLAN.

<program>::=<block>
<block>::=  
  [ <block idf.>::01 <prefix class idf.>::01 block <body>::<block idf.>::01 ]
  redundant applied redundant applied
  defining occurrence occurrence, equal to the matching defining identifier

<body>::=<declaration list> begin <statement list> end
<declaration>::= <variable declaration>
  | <class declaration>
  | <procedure declaration>
<variable declaration>::= var <specification list>
<class declaration>::=
  <class idf.>:: <prefix class idf.>::01 class <body>:: <class idf.>::01
  defining applied redundant applied occurrence occurrence, equal to the matching defining identifier

<procedure declaration>::=
  <procedure idf.>:: <prefix class idf.>::01 proc<formal parameter list>;
  defining applied redundant applied occurrence, occurrence equal to the matching defining identifier

<statement>::= <empty statement> identifier
  | <error statement>
  | <assignment statement>
  | call <procedure idf> <actual parameter list>::01 applied occurrence
  | new <class idf.>
  | inner
  | <block>
  | <compound statement>

The superscript\(^{01}\) is an indication that the superscripted entity may be there or not.
Appendix B: Program example π₁

M: block
  var x: real;
A: class
  var y: real;
  begin
    x:=3;
    inner
    end A;
  begin
    1: A block
      var y: real;
      B: class
      begin
        x:=y; print(x);
        inner
        end B;
      begin
        y:=2;
        2: new B;
        3: A block
          var y: real;
          C:3 class
          begin
            y:=x; print(y);
            inner
            end C;
          begin
            y:=4;
            4: new C
          end 3
        end 1
      end M
Appendix C: Program Example 2:

M: block
  B: class
    Y: X class
    begin
    inner
      end Y;
    end X;
    begin
    inner
      end X;
    begin
    inner
      end B;
    begin
    A: X block
      X: class
      begin
      end X;
    begin
    Z: Y block
    begin
    prefix Y
      begin
      u := j;
    elimination
    end Z
      end A
    end M

u is a free identifier occurrence. If we would have renamed class X in block A into class X', then u would be bound to var u in class X in class B which is reasonable.
Appendix D:
Elimination of prefixes A in \( \gamma_1 \) yields \( \gamma'_1 \):

\[
\begin{align*}
M & : \text{block} \\
& \quad \text{var } x : \text{real;} \\
& \quad \text{[\{ class A deleted \}]}
\end{align*}
\]

\[
\begin{align*}
\begin{block}
1 & : \text{block} \\
\quad & \text{var } x : \text{real;} \\
\quad & \text{var } y : \text{real;} \\
\quad & \text{class B} \\
\quad & \text{begin} \\
\quad & \quad x := y; \text{ print}(x); \\
\quad & \quad \text{inner} \\
\quad & \text{end B;} \\
\quad & \text{begin} \\
\quad & \quad x := 3; \\
\quad & \quad y := 2; \\
\quad & \quad \text{new B;} \\
\quad & \text{block} \\
\quad & \quad \text{var } x : \text{real;} \\
\quad & \quad \text{var } y : \text{real;} \\
\quad & \text{class C:B} \\
\quad & \text{begin} \\
\quad & \quad y := x; \text{ print}(y); \\
\quad & \quad \text{inner} \\
\quad & \text{end C;} \\
\quad & \text{begin} \\
\quad & \quad x := 3; \quad y := 4; \\
\quad & \quad \text{new C} \\
\quad & \text{end 3} \\
\end{block}
\end{align*}
\]

end M

Although in \( \gamma_1 \) all applied occurrences of \( x \) have the same defining occurrence they have different ones in \( \gamma'_1 \). This is so because they have different minimal modules (minmod) in \( \gamma_1 \), and this has an influence when prefixes A in \( \gamma_1 \) are eliminated.
Elimination of prefix B and new B in \( \pi'_1 \) yields \( \pi'_2 \):

\[
\text{M: block}
\begin{align*}
\text{var } x : \text{real}; \\
\text{begin} \\
\text{1: block} \\
\text{var } x : \text{real}; \\
\text{var } y : \text{real}; \\
\text{begin} \\
x := 3; \ y := 2; \\
\text{2: block} \\
\text{begin} \\
x := y; \ \text{print}(x); \\
\text{end 2:} \\
\text{3: block} \\
\text{var } x : \text{real}; \\
\text{var } y : \text{real}; \\
\text{C: class} \\
\text{begin} \\
x := y; \ \text{print}(x); \\
\text{y:my; print(y);} \\
\text{inner} \\
\text{end C;} \\
\text{begin} \\
x := 3; \ y := 4; \\
\text{4: new C} \\
\text{end 3} \\
\text{end 1} \\
\text{end M}
\]

Binding in case of dynamic scoping (without any renaming) is shown by arrows like \( \rightarrow \). Binding when only the source program \( \pi'_1 \) is made distinguished (x in main part of block M and y in block 3 are renamed to \( \tilde{x} \) and \( \tilde{y} \)) is shown by a correcting arrow \( \rightarrow \). Binding in case of pure static scoping (all programs are made distinguished, especially all x in block 3 in \( \pi'_1 \) are renamed to \( x' \)) is shown by a correcting arrow \( \cdots \rightarrow \).
Elimination of new C in \( \pi_1 \) yields \( \pi'' \):

\[
\begin{align*}
M & : \text{block} \\
& \quad \text{var } x : \text{ real}; \text{ begin} \\
& \quad \quad 1: \text{ block} \\
& \quad \quad \quad \quad \text{var } x : \text{ real}; \text{ begin} \\
& \quad \quad \quad \quad \quad \text{var } y : \text{ real}; \\
& \quad \quad \quad \quad \quad \text{begin} \\
& \quad \quad \quad \quad \quad \quad x := -3; \\
& \quad \quad \quad \quad \quad \quad y := 2; \\
& \quad \quad \quad \quad \quad 2: \text{ block} \\
& \quad \quad \quad \quad \quad \text{begin} \\
& \quad \quad \quad \quad \quad \quad x := y; \text{ print}(x); \\
& \quad \quad \quad \quad \quad \text{end} 2; \\
& \quad \quad \quad 3: \text{ block} \\
& \quad \quad \quad \quad \text{var } x : \text{ real}; \\
& \quad \quad \quad \quad \quad \text{var } y : \text{ real}; \\
& \quad \quad \quad \quad \text{begin} \\
& \quad \quad \quad \quad \quad \quad x := 3; y := 4; \\
& \quad \quad \quad \quad \quad \quad \text{class C deleted} \\
& \quad \quad \quad \quad \quad 4: \text{ block} \\
& \quad \quad \quad \quad \quad \text{begin} \\
& \quad \quad \quad \quad \quad \quad x := y; \text{ print}(x); \\
& \quad \quad \quad \quad \quad \quad y := x; \text{ print}(y); \\
& \quad \quad \quad \quad \quad \text{end} 4 \\
& \quad \quad \quad \quad \text{end} 3 \\
& \quad \quad \quad \text{end} 1 \\
& \quad \text{end } M \\
\end{align*}
\]

Output of the program:

- Dynamic scoping (\( \to \)): 2.0, 4.0, 4.0
- Quasi-static scoping (\( \longrightarrow \)): 2.0, 2.0, 2.0
- Pure static scoping (\( \leftrightarrow \)): 2.0, 2.0, 3.0
Appendix E: Program example $m_3$:

```plaintext
M: block
   var y: real;
   A: class
      var x: real;
      B: class
      begin
         x := y;
      inner
      end B;
      begin
         1: new B;
      inner;
      N: A block
         var y: real;
         C: B class
         begin
            y := x;
         inner
         end C;
         begin
            2: new C
         end N
      end A;
      begin
         3: new A
      end N
```
Elimination of `new A` in $\eta_3$ yields $\eta'_3$:

```
M: block
   \delta_M
begin
3: block
   \begin{align*}
   \text{var} & \ x: \text{real;} \\
   \text{B: class} & \\
   \text{begin} & \\
   x := y; & \\
   \text{inner} & \\
   \text{end B;} & \\
   \text{begin} & \\
1: \text{new B; } & \\
   \text{N: A block} & \\
   \text{var} & \ y: \text{real;} \\
   \text{C: B class} & \\
   \text{begin} & \\
   y := x; & \\
   \text{inner} & \\
   \text{end C;} & \\
   \text{begin} & \\
2: \text{new C} & \\
   \text{end N} & \\
   \text{end 3} & \\
\end{align*}
end M
```
Elimination of new B and prefix λ in $\pi_3^N$ yields $\pi_3^*$. 

\[ E_3 \]

\[ \begin{array}{l}
\Delta_M \\
\text{begin} \\
\Delta_N \\
\text{end} \\
\end{array} \]

\[ \begin{array}{l}
\Delta_3 \\
\text{begin} \\
\text{1: block} \\
\text{begin} \\
x = y; \\
\text{end 1: ;} \\
\text{end} \\
\text{var x: real;} \\
\text{B: class} \\
\text{begin} \\
x = y; \text{ inner} \\
\text{end B;} \\
\text{var y: real;} \\
\text{C: B class} \\
\text{begin} \\
y = x; \text{ inner} \\
\text{end C;} \\
\text{begin} \\
1: \text{ new B;} \\
2: \text{ new C;} \\
\end{array} \]

$\Sigma_N$ 

Repetition of $\Sigma_N$ in $\pi_3^*$ and $\pi_3^*$ shows that prefix elimination will never come to an end, the formal execution lattice $E_3^*$ is infinite.
Appendix F: Transformation of $\tau_1$ yields $\tau_1^T$:

$\tau_1:
M : block
   \begin{align*}
   &\text{var } x : \text{real}; \\
   &A : \text{proc } (A_f; \text{proc(output real)}); \\
   &\quad \text{var } x : \text{real}; \\
   &\quad \text{begin} \\
   &\quad \quad x := 3; \\
   &\quad \quad \text{call } A_f(x) \\
   &\quad \text{end } A; \\
   &\text{begin} \\
   &\quad 1 : \text{block} \\
   &\quad \quad 1_g : \text{proc } (\text{output } x; \text{real}) ; \\
   &\quad \quad \quad \text{var } y : \text{real}; \\
   &\quad \quad \quad B : \text{proc } (B_f; \text{proc}); \\
   &\quad \quad \quad \quad \text{begin} \\
   &\quad \quad \quad \quad \quad x := y; \text{ print(x)}; \\
   &\quad \quad \quad \quad \quad \text{call } B_f \\
   &\quad \quad \quad \text{end } B; \\
   &\quad \quad \text{begin} \\
   &\quad \quad \quad y := 2; \\
   &\quad \quad \quad 2 : \text{block} \\
   &\quad \quad \quad \quad 2_g : \text{proc}; \\
   &\quad \quad \quad \quad \quad \text{begin} \text{ end } 2_g; \\
   &\quad \quad \quad \quad \quad \text{call } B(2_g) \\
   &\quad \quad \quad \text{end } 2; \\
   &\quad \quad \text{begin} \\
   &\quad \quad \quad 3 : \text{block} \\
   &\quad \quad \quad \quad 1_g : \text{proc}; \\
   &\quad \quad \quad \quad \quad \text{begin} \text{ end } 1_g; \\
   &\quad \quad \quad \quad \quad \text{call } A(1_g) \\
   &\quad \quad \quad \text{end } 3 \\
   &\quad \text{end } 1; \\
   &\text{end } M
\end{align*}

where $\tau_3$ is the following block:
3 : block
   3_g : proc (output x:real);
       var y : real;
       C : proc (C_g:proc);
       C_g : proc;
       begin
         y := x; print(y);
         call C_f;
       end C_g;
       begin
         call B(C_g)
       end C;
   begin
     y := 4;
   4 : block
     4_g : proc;
     begin end 4_g;
     begin
       call C(4_g)
     end 4
   end 3_g;
   begin
     call A(3_g)
   end 3
Transformation of \( \tau_3 \) yields \( \tau_3^T \):

\[ M : \text{block} \]
\[ \begin{align*}
\text{var} & \quad y : \text{real}; \\
A & \quad \text{proc} (A_f; \text{proc} (\text{output} \quad \text{real}; \text{proc})); \\
\text{var} & \quad x : \text{real}; \\
b & \quad \text{proc} (b_f; \text{proc}); \\
\text{begin} & \quad \begin{align*}
& x := y; \\
& \text{call } b_f \\
\end{align*} \\
\text{end } b; \\
\text{begin} & \quad \begin{align*}
1 & \quad \text{block} \\
1_g & \quad \text{proc}; \\
\text{begin} & \quad \text{end } 1_g; \\
\text{begin} & \quad \text{call } b_1(1_g) \\
& \text{end } 1; \\
& \text{call } A_f(x, b); \\
& \text{end } A; \\
\text{begin} & \quad \begin{align*}
3 & \quad \text{block} \\
3_g & \quad \text{proc} (\text{output} \quad x : \text{real}; \text{proc} \_B (\text{proc})); \\
\text{begin} & \quad \text{end } 3_g; \\
\text{begin} & \quad \text{call } A_3(3_g) \\
& \text{end } 3 \\
\text{end } M
\end{align*}
\end{align*} \]

where \( \tau_3 \) is the following block:
N : block

Ng : proc (output x : real; B : proc (proc));
  var y : real;
  C : proc (C_g : proc);
  C_g : proc;
  begin
    y := x;
    call C_f
  end C_g;
  begin
    call B(C_g)
  end C;
  begin
    2 : block
    2_g : proc;
    begin end 2_g;
    begin
      call C(2_g)
    end 2
  end Ng;
  begin
    call A(N_g)
  end N
Appendix C: Run time system subroutines

initialization : subroutine;

begin
  MDL := 0;
  $\mathcal{M}[\text{MDL+RA}] := \text{undefined};$
  $\mathcal{M}[\text{MDL+DLD}] := \text{undefined};$
  $\mathcal{M}[\text{MDL+ID}] := \text{identifier of the largest block of the program which also identifies the program};$
  $\mathcal{M}[\text{MDL+DLS-1+1}] := a[i] := 0;$
  FSL := $\mathcal{M}[\text{MDL+LG}] := \text{fstable(largest module } M_i \text{ of the program)}$
end initialization

blockentering : subroutine ($\eta$:blockidentifier);

begin
  $\mathcal{M}[\text{FSL+RA}] := \text{program address for continuation when block } \eta \text{ has regularly terminated, the address End of } M_\eta \text{ is determined over the actual blockidentifier } \eta;$
  $\mathcal{M}[\text{FSL+DLD}] := \text{MDL};$
  $\mathcal{M}[\text{FSL+ID}] := \eta;$
  We do the simultaneous assignment

  \[
  \begin{pmatrix}
  \mathcal{M}[\text{FSL+DLS-1+1}] \\
  \vdots \\
  \mathcal{M}[\text{FSL+DLS-1+}v_{M_\eta} - 1]
  \end{pmatrix}
  :=
  \begin{pmatrix}
  a[i] \\
  \vdots \\
  a[i_{v_{M_\eta} - 1}] 
  \end{pmatrix};
  \]

  $\mathcal{M}[\text{FSL+DLS-1+}v_{M_\eta}] := a[i_{v_{M_\eta}}] := \text{FSL};$
  $\mathcal{M}[\text{FSL+LG}] := \text{fstable}(M_\eta);$
  MDL := FSL;
  FSL := FSL+\text{fstable}(M_\eta)$
end blockentering
finish : subroutining;
begin
\( FSL := MDL; \)
\( MDL := \text{MDL+DLD}; \)
The module identifier \( n \) in cell \( MDL+ID \) determines the prefix chain into which we return.
We do the simultaneous assignment
\[
\begin{pmatrix}
\mathcal{M}[\text{MDL+DLB}1]

\vdots

\mathcal{M}[\text{MDL+DLB}1+V_{n}]
\end{pmatrix}
:=
\begin{pmatrix}
\mathcal{M}[\text{MDL+DLB}1+1]

\vdots

\mathcal{M}[\text{MDL+DLB}1+V_{n}]
\end{pmatrix};
\]
goto \( FSL+RA \)
end finish

prefixed blockentering : subroutine \((n:\text{blockidentifier})\);
begin
\( FSL+RA := \text{program address for continuation when block } n \)
has regularly terminated, the address \( \text{End of block } M_{n} \)
is determined over the actual blockidentifier \( n \);
\( FSL+DLB := MDL; \)
\( FSL+ID := n; \)
Let \( \text{srenv}(M_{n}) = M' \) with \( M_{n} \rightarrow M' \) and \( V_{M'} = V_{n} - 1 \).
We do the simultaneous assignment
\[
\begin{pmatrix}
\mathcal{M}[FSL+DLB 1+1]

\vdots

\mathcal{M}[FSL+DLB 1+V_{n}]
\end{pmatrix}
:=
\begin{pmatrix}
\mathcal{M}[\text{MDL+DLB}1]

\vdots

\mathcal{M}[\text{MDL+DLB}1+V_{n}]
\end{pmatrix};
\]
\( FSL+LG := \text{fstat(M}_{n}); \)
\( MDL := FSL; \)
\( FSL := FSL+\text{fstat(M}_{n}); \)
end prefixed blockentering

The subroutine blockentering can be replaced by the subroutine prefixed block entering; but blockentering is more efficient because in case of a non-prefixed block \( n \) we have
\( d_{M'} = d_{M_{n}} \mid [1 : V_{M'}]. \)
simple non-formal procedure: subroutine (v: procedure identifier);
begin
\( M[\text{FSL+RA}] := \text{Return address of procedure call which is transmitted by the subroutine call;} \)
\( M[\text{FSL+DLD}] := \text{MDL;} \)
\( M[\text{FSL+ID}] := v; \)
Let \( \text{strenv}(M_v) = M' \) with \( M_0 \rightarrow M \) and \( v_{M_0} \rightarrow v_{M} -1 \).
We do the simultaneous assignment
\[
\begin{pmatrix}
M[\text{FSL+DLS-1+1}] \\
\vdots \\
M[\text{FSL+DLS-1} + v_{M_0} -1]
\end{pmatrix}
:=
\begin{pmatrix}
\mathcal{N}[d_{M_0} (1)] \\
\vdots \\
\mathcal{N}[d_{M_0} (v_{M_0})]
\end{pmatrix};
\]
\( M[\text{FSL+DLS-1} + v_{M_0}] := \mathcal{N}[d_{M_0} (v_{M_0})] := \text{FSL;} \)
\( M[\text{FSL+LG}] := \text{fstat}(M_0); \)
\( \text{MDL := FSL;} \)
\( \text{FSL := FSL+stat}(M_0); \)
goto Starting address of procedure \( v \)
end simple non-formal procedure

non-formal procedure: subroutine (x: module identifier, 
v: procedure identifier);
begin
\( M[\text{FSL+RA}] := \text{Return address of procedure call which is transmitted by the subroutine call;} \)
\( M[\text{FSL+DLD}] := \text{MDL;} \)
\( M[\text{FSL+ID}] := v; \)
Let \( \text{strenv}(M_v) = M' \) with \( M_0 \rightarrow M \) and \( v_{M_0} \rightarrow v_{M} -1 \).
We do the simultaneous assignment
\[
\begin{pmatrix}
\mathcal{N}[d_{M_0} (1)] \\
\vdots \\
\mathcal{N}[d_{M_0} (v_{M_0})]
\end{pmatrix}
:=
\begin{pmatrix}
M[\text{FSL+DLS-1+1}] \\
\vdots \\
M[\text{FSL+DLS-1} + v_{M_0} -1]
\end{pmatrix}
:=
\begin{pmatrix}
\mathcal{N}[d_{M_0} \cdot d_{M_0}^{-1} \cdot d_{M_0} (1)] \\
\vdots \\
\mathcal{N}[d_{M_0} \cdot d_{M_0}^{-1} \cdot d_{M_0} (v_{M_0})]
\end{pmatrix};
\]
\[ \mathcal{M}[\text{FSL+LG}] := \text{fstat}(M_{p}) \; ;
\]
\[ \text{MDL} := \text{FSL} ;
\]
\[ \text{FSL} := \text{FSL+fstat}(M_{g}) ;
\]
\begin{verbatim}
goto Starting address of procedure \phi and non-formal procedure
\end{verbatim}

M* and \( \mathcal{R} \) are defined as in section 2.4.VIII, where non-formal procedure statements are compiled. The subroutine simple non-formal procedure can be replaced by non-formal procedure; but the first one is more efficient because we have \( d_{\mathcal{R}} = d_{M^{*}}[1:v_{M^{*}}] \) and \( d_{M^{*}}[1:v_{M^{*}}] \) since \( M_{p} \) is not prefixed.

simple class : subroutine (\( \pi \): class identifier);
begin
\[ \mathcal{M}[\text{FSL+RA}] := \text{Return address of class initialization which is transmitted by the subroutine call;}
\]
\[ \mathcal{M}[\text{FSL+DL}] := \text{MDL;}
\]
\[ \mathcal{M}[\text{FSL+ID}] := \pi ;
\]
Let \( \text{strenv}(M_{n}) \rightarrow M^{*} \) with \( M_{n}^{*} \rightarrow M^{*} \) and \( v_{M^{*}} = v_{M_{n}} - 1 \).

We do the simultaneous assignment

\[
\begin{pmatrix}
\mathcal{M}[\text{FSL+DLs-1+1}] \\
\vdots \\
\mathcal{M}[\text{FSL+DLs-1+} v_{M_{n}} - 1]
\end{pmatrix}
= 
\begin{pmatrix}
\mathcal{N}[d_{M^{*}}(1)] \\
\vdots \\
\mathcal{N}[d_{M^{*}}(v_{M^{*}})]
\end{pmatrix}
\]

\[ \mathcal{M}[\text{FSL+DLs-1+} v_{M_{n}}] := \mathcal{N}[d_{M_{n}}(v_{M_{n}})] := \text{FSL;}
\]
\[ \mathcal{M}[\text{FSL+LG}] := \text{fstat}(M_{n}) ;
\]
\[ \text{MDL} := \text{FSL;}
\]
\[ \text{FSL} := \text{FSL+fstat}(M_{n})
\]
end simple class
class : subroutine (γ:module identifier, n: class identifier);
begin
\( \mathcal{M}[FSL+RA] := \) Return address of class initialization which is transmitted by the subroutine call;
\( \mathcal{M}[FSL+DDO] := MDD; \)
\( \mathcal{M}[FSL+ID] := n; \)
We do the simultaneous assignment
\[
\begin{align*}
\mathcal{M}[d_M(1)] &= \mathcal{M}[FSL+DLS-1+1] = \mathcal{M}[d_M(1)] \\
\vdots &= \vdots \\
\mathcal{M}[d_M(v_M - 1)] &= \mathcal{M}[FSL+DLS-1+v_M - 1] = \mathcal{M}[d_M(v_M - 1)] \\
\mathcal{M}[d_M(v_M)] &= \mathcal{M}[FSL+DLS-1+v_M] = \mathcal{M}[d_M(v_M)] \\
\end{align*}
\]
\( \mathcal{M}[FSL+LG] := \text{fstact}(M_n); \)
MDD := FSL;
FSL := FSL+fstact(M_n)
end class

prepare formal procedure : subroutine (χ:module identifier, Ψ: formal procedure identifier);
begin
\( \mathcal{M}[FSL+DLD] := MDD; \)
Let χ be a formal parameter of a procedure with its module \( \bar{M} \).
Let Ψ be the identifier of module \( \bar{M} \).
\( \mathcal{M}[FSL+ID] := \text{first component} (\mathcal{M}[d_{\bar{M}*[v_{\bar{M}}]}+\text{reladdr}(\Psi)]); \)
AUX := second component (\( \mathcal{M}[d_{\bar{M}*[v_{\bar{M}}]}+\text{reladdr}(\Psi)] \));
Let \( \psi \) be the non-formal procedure identifier in \( \mathcal{M}[FSL+ID] \).
\( \mathcal{M}[FSL+LG] := \text{fstact}(M_\psi); \)
AUX1 := FSL+fstact(M_\psi)+fst(M_\psi)+K
end prepare formal procedure
check actual parameter : subroutine (i:parameter number);
begin
The specification of the actual procedure identifier in the 
first component of \( M' [AUX1+X-1+i] \) is checked against the 
specification of the 1-th formal parameter of that procedure 
the identifier of which is in \( M' [FSL+ID] \). In case of incor-
rectness computation is erroneously aborted 
end check actual parameter

formal procedure : subroutine;
begin
\( M' [FSL+RA] := \) Return address of formal procedure call which is 
transmitted by the subroutine call;
The non-formal procedure identifier \( \varphi \) in \( M' [FSL+ID] \) has a 
defining occurrence \( ^{2}\varphi \) with \( M' = \text{env}(^{2}\varphi) \) and \( M' = \text{strenv}(M_{\varphi}) \).
\( M'_{\varphi} \rightarrow M' \), \( v_{M} = v_{M_{\varphi}}^{-1} \).
The module identifier \( \eta \) in \( M' [AUX+ID] \) identifies a module \( M_{\eta} \)
with \( v_{M_{\eta}} = v_{M} \), \( 2v_{M_{\varphi}} \).

We do the simultaneous assignment
\[
\begin{align*}
&d[M_{\varphi} (1)] = [M' [FSL+DLS-1+1]] \\
&\vdots \\
&d[M_{\varphi} (v_{M_{\varphi}} - 1)] = [M' [FSL+DLS-1+v_{M_{\varphi}}]] \\
&d[M_{\varphi} (v_{M_{\varphi}})] = [M' [FSL+DLS-1+v_{M_{\varphi}}]] \\
&\vdots \\
&[M' [AUX+DLS-1+1]] = [M' [AUX+DLS-1+1]] \\
&\vdots \\
&[M' [AUX+DLS-1+v_{M_{\varphi}}]] = [M' [AUX+DLS-1+v_{M_{\varphi}}]] \\
&FSL := FSL;
\end{align*}
\]
MDL := FSL;
FSL := FSL + \( [MDL+LG] \);
goto Starting address of procedure \( \varphi \)
end formal procedure
Appendix H: Program Example 5.4

M: block
   A: class
       var x: real;
       begin
           B: block
               C: class
                   begin
                       x := 0;
                       inner
                       end C;
                   begin
                       D: A block
                           begin
                               E: C block
                                   begin
                                       X := 0
                                   end E
                               end D
                           end B;
                           inner
                           end A;
               begin
                   1: A block
                       begin
                           end 1
           end M
       end A;
   end B;
end M
[Lo83] LOGLAN-82 Report, Polish Scientific Publisher, Warsaw 1983
