An Attributed LL(1) Compilation of PASCAL into Lambda-Calculus

by

Erich Kaltofen

Mathematical Sciences Department

Rensselaer Polytechnic Institute

Troy, New York 12181

November 1979

TABLE OF CONTENTS

PREFACE

This thesis describes a PASCAL compiler which is rather unique because its target language is the lambda-calculus instead of some machine code. Undertaken as a Computer Science Master's Project at Rensselaer Polytechnic Institue, and actually accomplishing more than in its original proposal, this compiler is part of a program verification project. Altough the object code that it generates can be executed by means of a lambda-expression-reducer [2] (resembling a pure LISP interpreter), the intended use of the code is in proving programs correct.

The compiler is written in PASCAL itself and contains an attributed LL(1) parser [6] of the complete standard PASCAL language [4]. Its error recovery is quite elaborate and it provides substantially better error diagnostics than several existing standard PASCAL compilers [8]. It produces code for a large subset of standard PASCAL covering almost all programs that are interesting from the theoretical program-verification viewpoint. The translated features include multidimensional arrays, any PASCAL assignment statements, I/O facilities, compound, conditional and repetitive statements, and procedures also allowing recursive calls and global side effects.

This thesis divides into two parts: 1) A formal definition of the target language containing a model for PASCAL [1], and 2) a code independent description of compilation algorithms including the complete LL(1) push-down automaton. It is assumed that the reader is familiar with the basic ideas of the lambda-calculus [9] and the top-down parsing methods [6].

Acknowledgment: The author wishes to express his gratitude towards his superb project advisor Prof. Kamal Abdali.

```
PPPPPPPPPPP    AAAAAAAAAA    RRRRRRRRRRR   TTTTTTTTTTTT         11
PPPPPPPPPPPP  AAAAAAAAAAAA  RRRRRRRRRRRR   TTTTTTTTTTTT        111
PP        PP  AA        AA  RR        RR       TT             1111
PP        PP  AA        AA  RR        RR       TT               11
PP        PP  AA        AA  RR        RR       TT               11
PPPPPPPPPPPP  AAAAAAAAAAAA  RRRRRRRRRRRR       TT               11
PPPPPPPPPPP   AAAAAAAAAAAA  RRRRRRRRRRR        TT               11
PP            AA        AA  RR    RR           TT               11
PP            AA        AA  RR     RR          TT               11
PP            AA        AA  RR      RR         TT               11
PP            AA        AA  RR       RR        TT         1111111111
PP            AA        AA  RR        RR       TT         1111111111
```

1.1. Introduction

The target language of the compiler is a slightly modified form of the lambda-calculus [3]. The structure of the PASCAL source program will be partially preserved by the translation into this language. Theoretically, an object program could be converted into a single lambda-expression. However, this is undesirable since the resulting code will lack clarity and will be inefficient for later automatic evaluation. Furthermore there is a ("software") machine which will execute this language in a slightly different syntactic setting [2]. This part introduces the essential concepts of the calculus and its modelling capabilities for PASCAL programs. For a broader introduction to the lambda-calculus and the modelling of ALGOL-like programming languages in it, the reader is referred to [1].

1.2. The Lambda - Calculus

Adopting a commonly used terminology [1], the syntax of the lambda-calculus is given by the following BNF-definition:

```
(1) <indeterminate> ::= <PASCAL-identifier  including '$' in the
                         letter set>
(2) <lambda-expression> ::= <indeterminate>
(3) <lambda-expression> ::= <application>
(4) <lambda-expression> ::= <abstraction>
(5) <application> ::= (<lambda-expression> <lambda-expression>)
(6) <abstraction> ::= (λ<binding indeterminates>:<lambda-expres-
                       sion>)
(7) <binding indeterminates> ::= <indeterminate>
```

It may be necessary to separate the lambda-expressions in production (5) by blank spaces. Blank spaces are allowed also whenever no syntactic unit is split apart. The lambda-expression in production (6) is the scope of the preceding binding indeterminates. An instance of an indeterminate within a given lambda-expression is bound if it occurs in the scope of a same binding indeterminate. However, it is bound by only the same innermost binding indeterminate. Otherwise its occurence is free in this lambda-expression. If e, f1, f2,..., fn are lambda-expressions and x1, x2,..., xn are pairwise distinct indeterminates[+], then

sub[f1,x1; f2,x2;...; fn,xn; e]

denotes the result of simultaneously substituting fi for all

[+] In the following, e, f, g,... will denote lambda-expressions and x, y,... indeterminates. Unless stated otherwise, they are always assumed universally quantified in definitions and theorems of the meta-language.

free occurences of xi ($1 \leq i \leq n$) in e.

The lambda-calculus contains the following contraction and expansion rules:

Alpha-conversion (renaming bound variables):
(λ x: e) —(alpha)—> (λ y: sub[y,x; e]) provided that y has no free occurence in e.

Beta-contraction (substitution):
((λ x: e)f) —(beta)—> sub[f,x; e] if no free indeterminates in f occur bound in e.

Eta-contraction (extensionality):
(λ x:(e x)) —(eta)—> e if x does not occur free in e.+

The converses of beta- and eta-contractions are called beta- and eta-expansions, respectively. A (possibly empty) sequence of contractions and alpha-conversions is called reduction (denoted by "—>"). Conversion ("<—>") also includes expansions. An irreducible lambda-expression cannot be beta- or eta-contracted further. If a lambda-expression e can be converted into an irreducible lambda-expression f, then f is uniquely determined up to alpha-conversions and, futhermore, e —> f. The leftmost (outermost) computation rule is save in the sense that it always leads to this irreducible lambda-expression ("normal form") provided that this expression exists. Most results may be proved using the Church-Rosser Theorem [3]: If e <—> g, then there is a conversion from e into g in which no expansion preceeds any contraction.

The following lemma suggests a useful extension of the syntax for lambda-expressions:

Lemma:
(...((λx1:(λx2:(...(λxn:e)...)))f1)...fn)—>sub[f1,x1;f2,x2;...; fn,xn; e] provided no free indeterminate in any of the fi's is bound in e.

Therefore the syntax of lambda-expressions will be extended by allowing a list of indeterminates in abstractions, viz.

(8) <binding indeterminates> ::= <binding indeterminates>,<indeterminate>

This notation can be viewed as a shorthand of nested abstrac-

+ This rule will not be applicable to the lambda-expressions generated by the compiler.

tions:

(λ x1,x2,...,xn: e) means (λ x1:(λ x2:(...(λ xn: e)...))).

However, an automatic evaluator can use the above lemma for a faster substitution algorithm for lambda-expressions in this form.

1.3. Systems of Lambda - Expression Definitions

It is a convenient practice to define a certain name[+] to be a representative of a given lambda-expression. Then this name may be used many times without rewriting its whole definition. The following productions complete the syntax of the target language:

(9) <list of definitions> ::= <definition>
(10) <list of definitions> ::= <list of definitions><definition>
(11) <definition> ::= <name>=<lambda-expression>.
(12) <lambda-expression> ::= <name>
(13) <name> ::= <PASCAL-identifier including '$' in the letter set>

An "object"-program is then a <list of definitions>. At this point a single lambda-expression may not always be recovered by merely replacing all names by their corresponding lambda-expressions because some names could be referred recursively using their own names on right-hand sides of their definitions. Before this problem can be resolved, some basic lambda-expressions shall be introduced. Since the compiler generates source-program-dependent names, special or predefined names will always be distinguished from these by a preceeding '$' character. This is the reason for including a '$' sign to the PASCAL letter set in the productions (1) and (13).

$ID = (λ X: X).
The identical ("do-nothing") function and empty list.

$CAT = (λ X,Y: (λ Z: ((X Z) Y))).
The concatenation of objects. If (λ X: ((((X y1) y2) ...) yn)) represents a list of n elements $CAT will then append an element to a list of this structure.

[+] Names act like variables in the language. They are distinctly different from indeterminates in that the language does not contain any rules indicating what objects indeterminates represent or how they may "vary" throughout a calculation. By specifying theorems about the language, indeterminates often attain the property of meta-variables. Therefore "name" was chosen to avoid possible confusion.

$OMEGA = ((λ X,Y: (X X))(λ X,Y: (X X))).
The undefined value. It should be noted that ($OMEGA f) —> $OMEGA but $OMEGA does not possess a normal form.

$Y = (λ X:((λ Y:(X(Y Y)))(λ Y:(X(Y Y))))).
The recursion operator. Since ($Y g) <—> (g ($Y g)), ($Y g) is a solution to the recursive definition of F = (g F), provided that g does not contain the name F.

$Y can now be employed to model general recursion. First it is necessary to beta-expand the right-hand sides of recursive definitions into the form

((((g <name>)<name>)...)<name>)

such that no name occurs inside the lambda-expression g and all recursively referenced names are listed in a given order. The explicit solution of a system of definitions

n1 = ((((g1 n1) n2)...) nk).
n2 = ((((g2 n1) n2)...) nk).
⋮
nk = ((((gk n1) n2)...) nk).

is determined by

ni $\underline{=}$ ((((Y[i,k] g1) g2)...) gk), with

$\underline{\text{XZ-list}}$ $\underline{=}$ (λ Y:((... ((Y(X Z1))(X Z2)) ...)(X Zk))), and
$\underline{\text{Y[i,k]}}$ $\underline{=}$ (λ Z1,...,Zk:(($Y (λ X: $\underline{\text{XZ-list}}$))(λ X1,...,Xk:Xi))).

This is also the least fixed point solution in some certain ordering [7]. However, it should be noticed that an automatic evaluator will work more efficiently by replacing names recursively during execution time rather than introducing the $Y operator beforehand.

1.4. Primitives in the Model

It is possible to represent natural numbers and arithmetic operations in the lambda-calculus [3]. This representation can be extended also to (signed) integers and much of computer arithmetic [1].

But instead of defining them as lambda-expressions, we accept a number of arithmetical and logical constants and operators as primitives in our model. The reduction characteristics of these primitives reflect the algebraic properties of the corresponding objects (viz. the integers and the logical values). An evaluator program can simulate these primitives with computer internal arithmetic operations rather than using their lambda-calculus definitions, thus gaining a considerable speed-up.

The compiled program may contain the following names associated with primitives:

0, 1, 2, ...	The positive integers. These are the only names syntactically different from identifiers. n, m,... will denote lambda-expressions reducing to the integers n, m,... or their names.
$MINUSUNARY...	Integer negation.
$PLUS.........	Integer addition.
$MINUS........	Integer subtraction.
$MULT.........	Integer multiplication.
$DIV..........	Integer division.
$TRUE.........	Boolean value true. It is assumed that (($TRUE g) h) —> g.
$FALSE........	Boolean value false. It is assumed that (($FALSE g) h) —> h.
$NOT..........	Boolean negation.
$AND..........	Boolean conjunction.
$OR...........	Boolean disjunction.
$EQ...........	Integer comparison equal.
$NE...........	Integer comparison not equal.
$GT...........	Integer comparison greater.
$GE...........	Integer comparison greater than or equal to.
$LT...........	Integer comparison less.
$LE...........	Integer comparison less than or equal to.

The following three primitives are used for array handling. An n dimensional PASCAL array is treated as a vector of n-1 dimensional arrays with 0 dimensional arrays treated as scalar objects. Vectors will be treated as lists in the object language (see $CAT). It is not an easy task to obtain the lambda-expressions of these primitives [1].

$TUPINIT......	Initialization of an array. Its reduction property is (...(($TUPINIT n) m1)... mn) —> "list of m1 lists of m2 lists of ... mn $OMEGAs".
$RETRIEVE.....	Indexing of vector elements. Its reduction property is (f (($RETRIEVE i) k)) —> "i-th element of f if f is a list of k items (lambda-expressions)".
$REPLACE......	Assigning a vector element. Its reduction property is (f((($REPLACE i) k) g) —> "list of k items provided that f is a list of k elements where all but the i-th element are copied from f and the i-th position is g".

In the lambda-calculus, characters may be modelled by their corresponding numerical code. In order to distinguish the codes from numbers a primitive equivalent to the standard PASCAL function "CHR" is introduced:

CHR........... Code character. In the pure lambda-calculus CHR = $ID.

1.5. Functional Semantics of Variables and Statements

Each statement of a PASCAL program operates on two different entities: A set of variables (global and local) addressable at the time the statement is executed (its "environment"), and some sort of register indicating which place in the program is currently executed. It is not hard to imagine that this register may contain an eventually recursive description of the entire portion of the program not executed so far (the "continuation" or the "program remainder"). With this view a statement acts more like a functional since one of its arguments, the continuation, itself turns out to be a function. A statement can be then translated into an abstraction with respect to the continuation, denoted by the indeterminate "$PHI", and the environment variables, denoted by their PASCAL identifiers whenever possible. If imported and local identifiers coincide, conflicts will be resolved by appending "$" and the proper block level number to these identifiers. If all continuations and current values of variables are arranged in a certain list form, this abstraction will not become too complex to construct.

In the following, a representation rule [1] of the form

{S}/(v1, v2,..., vn) $\underline{=}$ abstraction.

where S is a statement and (v1, v2,..., vn) is its environment will be used to describe which kind of abstractions model these statements, and to give a more concise expression to the underlying ideas. As the compiler defines each abstraction of the statement i by the name "$STMi", representation rules can also be seen as patterns for these definitions.

1.6. Compound Statements and Blocks

Compound statements are compositions of functions. This suggests the following representation rule:

{<u>begin</u>; S1;S2;...; Sn <u>end</u>}/E $\underline{=}$
    (λ $PHI:({S1}/E ({S2}/E (...({Sn}/E $PHI)...)))).

It should be noticed that the program remainder of each Si is the first operand applied to the statement. Therefore -- and this is true for all representations -- a statement representation merely has to substitute this first operand into a place where it will become applicable after the statement's reduction is finished. In the following, the environments will not be explicitly specified if they stay the same throughout a representation rule.

Blocks introduce new (local) variables, initialize them to an undefined value and delete them from the environment after execution of their body. Let E be the global and F=(u1,..., um, v1,..., vn) be the local environment (identifier conflicts already resolved):

```
{var u1:<type1>;...; um:<typem>; begin S1;...; Sk end}/E =
   (λ $PHI:(...(((S1}/F ({S2}/F (...({Sk}/F (λ u1,...,um:$PHI))
   ...))){init u1}) {init u2})... {init um}));
where {init u} is $OMEGA if u is a scalar variable,
               is (($TUPINIT 1) p) if u is a vector of p items,
               is ((($TUPINIT 2) p1) p2) if u is a p1*p2 matrix,
                                 •
                                 •
                                 •
```

As a statement is translated into an abstraction of $PHI and the environment variables, all current values of these variables must follow the continuation before and also after the abstraction was reduced. It should be noticed that an attempt to reference an undefined value will result in an infinite reduction sequence due to a property of $OMEGA.

1.7. Expressions and Assignments

So far it has been specified how to compile only integers and Boolean constants. But since scalar identifiers and characters can be treated as their ordinal numbers, all scalar constants suitable for lambda-calculus representation can also be compiled. Entire variables are indeterminates of their own identifiers with ambiguities removed. As a slight restriction only unary and binary operators on scalar operands are accepted. Due to the list-like translation of arrays, records can be viewed as a special form of arrays and their field identifiers as indices.

Rational and real numbers are not considered in this thesis [1]. Literals and sets are part of PASCAL because they allow a very efficient implementation on a binary computer, but could be simulated in the lambda-calculus only by a rather clumsy representation. Theoretical problems going far beyond the scope of this paper caused the omission of pointer variables. Files will be treated in section 1.8.

The representation rules for expressions (without function calls) heavily involve recursion. The most significant are sketched below. In some instances a nesting level number is used as a superscript on matching pairs of parentheses to make the rules more readable:

Since our model requires prefix operators, the following representation rules are essentially infix to prefix translations:

{<expr.1> <binary operator> <expr.2>} $\underline{=}$
(("primitive of binary oper." {<epr.1>}) {<expr.2}).

{<unary operator> <expression>} $\underline{=}$
("primitive of unary oper." {<expression>}).

Arrays in list form always assume their index origin at 1. Therefore the compiler has to translate all index references explicitly to this origin. Let v be an <u>array</u> [lb..hb], b an <u>array</u> [BOOLEAN] and d be an <u>array</u> [lb1..hb1, lb2..hb2]:

{v [<expression>] } $\underline{=}$
(v (^{1}(^{2}\$RETRIEVE (^{3}(\$MINUS{<expression>}) $\underline{\text{lb-1}}$ ^{3}) ^{2})$\underline{\text{hb-lb+1}}$ ^{1})).

{b [<expression>] } $\underline{=}$
(b (^{1}(^{2}\$RETRIEVE (^{3}({<expression>} 1) 2 ^{3}) ^{2}) 2 ^{1})).

{d [<expr.1>, <expr.2>] } $\underline{=}$
((^{1}d (^{2}(^{3}\$RETRIEVE (^{4}(\$MINUS {<expr.1>}) $\underline{\text{lb1-1}}$ ^{4}) ^{3})
$\underline{\text{hb1-lb1+1}}$ ^{2}) ^{1})(^{5}(^{6}\$RETRIEVE (^{7}(\$MINUS {<expr.2>}) $\underline{\text{lb2-1}}$ ^{7}) ^{6})
$\underline{\text{hb2-lb2+1}}$ ^{5})).

•
•
•

In the following, CHR and ORD are the standard PASCAL functions on scalars:

{'<character>'} $\underline{=}$ (CHR "order of this character").
{CHR(<expression>)} $\underline{=}$ (CHR {<expression>}).
{ORD(<expression>)} $\underline{=}$ {<expression>}.

{-n} $\underline{=}$ (\$MINUSUNARY <u>n</u>).

An assignment statement will be translated into a substitution of the right-hand expression for the left-hand variable position in the list of corresponding indeterminates. Assignments to elements of an array complicate this process somewhat:

{vi:=<expression>}/E $\underline{=}$
(λ \$PHI, v1,..., vn: (...(((...(((\$PHI v1) v2)... vi-1)
{<expression>}/E) vi+1)... vn).

{v [<expr.1>] := <expr.2>}/E $\underline{=}$
(λ \$PHI, v1,..., vn: (...((...((\$PHI v1) v2)... vj-1)
(^{1}v (^{2}(^{3}(^{4}\$REPLACE (^{5}(\$MINUS{<expr.1>}/E}) $\underline{\text{lb-1}}$ ^{5}) ^{4})$\underline{\text{hb-lb+1}}$ ^{3})
{<expr.2>}/E} ^{2}) ^{1})) vj+1)... vn).

```
{d [<expr.1>, <expr.2>] := <expr.3>}/E =
   (λ $PHI, v1,..., vn: (...(((...(($PHI v1) v2)... vk-1)
   assign) vk+1)... vn);
where assign is the lambda-expression:
   (d(^0(^1(^2$REPLACE (^3($MINUS {<expr.1>}/E) lb1-1 ^3) ^2) hb1-lb+1 ^1)
   (^4(^5d (^6(^7$RETRIEVE (^8($MINUS {<expr.1>}/E) lb1-1 ^8) ^7)
   hb1-lb1+1 ^6) ^5) (^9(^10(^11$REPLACE (^12($MINUS {<expr.2>}/E)
   lb2-1 ^12) ^11) hb2-lb2+1 ^10) {<expr.3>}/E ^9) ^4) ^0)).
```

⋮

1.8. Files and Input-Output

As the I/O facilities in the lambda-calculus model are rather simple, at this state of development only the two standard files INPUT and OUTPUT are supported during compilation. These files are unlike standard PASCAL files of INTEGER. Their representations in lambda-calculus are naturally lists denoted by the indeterminate $SCARDS for INPUT and $SPRINT for OUTPUT. Their initial values are all input items coded as lambda-expressions for $SCARDS and $ID (the empty list) for $SPRINT. The current file pointers INPUT@ and OUTPUT@ will be treated as integer indeterminates of the same name ("@" omitted). Furthermore, only the two predefined I/O routines GET and PUT are accepted by the code generation routines of the compiler. Upon call, data is transferred between the files and their associated file pointers. Contrary to standard PASCAL the input file is not automatically reset which means that INPUT@ contains $OMEGA at the beginning of a program and not the first data integer of $SCARDS.

In the following representation rules, one should notice that INPUT and OUTPUT are automatically adjoined to the environment G=(v1,..., vn, INPUT, OUTPUT) of a statement if their corresponding files appear in the program head:

```
{ GET }/G =
   (λ $PHI, v1,..., vn, OUTPUT, INPUT:(λ $SCARDS, $SPRINT:
   ((((...(($PHI v1) v2)... vn) OUTPUT) $SCARDS) $SPRINT))).

{ PUT }/G =
   (λ $PHI, v1,..., vn, OUTPUT, INPUT:(λ $SPRINT:((((...(($PHI
   v1) v2)... vn) OUTPUT) INPUT) (($CAT $SPRINT) OUTPUT)))).
```

These representations only require $SPRINT to be arranged in list format (see definition of $CAT) whereas the input elements merely have to follow this output list. The representation of a complete program which the compiler names $PROGRAM follows:

{$\underline{\text{program}}$...; Si.}/() $\underline{=}$ (({Si}/() \$ID)\$ID);
where Si is the outermost $\underline{\text{begin}}$-$\underline{\text{end}}$ pair and () the empty list.

The first \$ID is the final program remainder and the second the empty \$SPRINT. \$PROGRAM has the property that

$$((\$PROGRAM\ \underline{i1})\ldots\ \underline{ip}) \longrightarrow (\lambda\ X:((X\ \underline{o1})\ldots\ \underline{oq}))$$

where o1,..., oq are the output numbers which would be obtained by executing the program on the input numbers i1,..., ip.

1.9. Example#1

The following sample program illustrates all concepts described so far. The statement numbers listed will be referred within the generated code later on:

```
Stmnr     Source code:
-----     ------------
          (*$U+,X- superscripts, no cross reference *)
          PROGRAM EXAMPLE1(INPUT, OUTPUT);
          CONST
              LB=2; HB=5; (* bounds for V *)
              LB1=-3; HB1=0;
              LB2=0; HB2=5; (* bounds for D *)
          TYPE
              SC=(ONE,TWO,THREE);
              LET='A'..'Z';
          VAR
              I: INTEGER; C: LET; S: SC;
              V: ARRAY[LB..HB] OF INTEGER;
              B: ARRAY[BOOLEAN] OF TWO..THREE;
              D: ARRAY[LB1..HB1, LB2..HB2] OF CHAR;
              (* 3 dimensions! *)
              P: ARRAY[LET, SC] OF ARRAY[2..7] OF TRUE..FALSE;
          BEGIN
             (* The following statements make no sense *)
             (* but illustrate the compilation         *)
  3          GET; I:=INPUT@;
  4          V[I+1]:=-(I+1);
  5          I:=I+V[ (LB+HB) DIV 2 ] * I;
  7          OUTPUT@:=V[4]; PUT;
  8          B[FALSE]:=THREE;
  9          S:=B[ NOT(I<>0) AND (V[I]<I+1) ];
 10          BEGIN
 11             D[-2, I*2]:= 'Q';
 12             C:=D[HB1-2] [V[3]]
 12          END;
 12          (* some difficult assignments *)
 13          P[C,S,I]:=(I<=2) OR (C='B');
 14          B[P[C,S,I]]:=TWO
 14       END.
```

The compiler generated the following code. Optionally, matching parentheses are identified by superscripts. An asteriks in column one signals a comment line and this line should be ignored by automatic evaluators.

```
* LAMBDA CODE FOR EXAMPLE1

$STM2=(^{11}λ $PHI,P,D,B,V,S,C,I,OUTPUT,INPUT:(^{10}λ $SPRINT,$SCARDS:
(^{9}(^{8}(^{7}(^{6}(^{5}(^{4}(^{3}(^{2}(^{1}(^{0}$PHI P^{0})D^{1})B^{2})V^{3})S^{4})C^{5})I^{6})OUTPUT^{7})$SCARDS^{8})$
SPRINT^{9})^{10})^{11}).

$STM3=(^{9}λ $PHI,P,D,B,V,S,C,I,OUTPUT,INPUT:(^{8}(^{7}(^{6}(^{5}(^{4}(^{3}(^{2}(^{1}(^{0}$PHI
 P^{0})D^{1})B^{2})V^{3})S^{4})C^{5})INPUT^{6})OUTPUT^{7})INPUT^{8})^{9}).

$STM4=(^{14}λ $PHI,P,D,B,V,S,C,I,OUTPUT,INPUT:(^{13}(^{12}(^{11}(^{10}(^{9}(^{8}(^{2}(^{1}(
^{0}$PHI P^{0})D^{1})B^{2})(^{7}V(^{6}(^{5}(^{4}$REPLACE (^{3}(^{2}$MINUS (^{1}(^{0}$PLUS I^{0})1^{1})^{2})1^{3}
)^{4})4^{5})(^{2}$MINUSUNARY (^{1}(^{0}$PLUS I^{0})1^{1})^{2})^{6})^{7})^{8})S^{9})C^{10})I^{11})OUTPUT^{12})
INPUT^{13})^{14}).

$STM5=(^{15}λ $PHI,P,D,B,V,S,C,I,OUTPUT,INPUT:(^{14}(^{13}(^{12}(^{5}(^{4}(^{3}(^{2}(^{1}(^{0}
$PHI P^{0})D^{1})B^{2})V^{3})S^{4})C^{5})(^{11}(^{0}$PLUS I^{0})(^{10}(^{9}$MULT (^{8}V(^{7}(^{6}$RETRIEVE
 (^{5}(^{4}$MINUS (^{3}(^{2}$DIV (^{1}(^{0}$PLUS 2^{0})5^{1})^{2})2^{3})^{4})1^{5})^{6})4^{7})^{8})^{9})I^{10})^{11})^{1}
^{2})OUTPUT^{13})INPUT^{14})^{15}).

$STM6=(^{9}λ $PHI,P,D,B,V,S,C,I,OUTPUT,INPUT:(^{8}(^{7}(^{6}(^{5}(^{4}(^{3}(^{2}(^{1}(^{0}$PHI
 P^{0})D^{1})B^{2})V^{3})S^{4})C^{5})I^{6})(^{4}V(^{3}(^{2}$RETRIEVE (^{1}(^{0}$MINUS 4^{0})1^{1})^{2})4^{3})^{4})^{7}
)INPUT^{8})^{9}).

$STM7=(^{11}λ $PHI,P,D,B,V,S,C,I,OUTPUT,INPUT:(^{10}λ $SPRINT:(^{9}(^{8}(^{7}(^{6}
(^{5}(^{4}(^{3}(^{2}(^{1}(^{0}$PHI P^{0})D^{1})B^{2})V^{3})S^{4})C^{5})I^{6})OUTPUT^{7})INPUT^{8})(($CAT $SPR
INT)OUTPUT)^{9})^{10})^{11}).

$STM8=(^{13}λ $PHI,P,D,B,V,S,C,I,OUTPUT,INPUT:(^{12}(^{11}(^{10}(^{9}(^{8}(^{7}(^{6}(^{1}(^{0}
$PHI P^{0})D^{1})(^{5}B(^{4}(^{3}(^{2}$REPLACE (^{1}(^{0}$FALSE 1^{0})2^{1})^{2})2^{3})2^{4})^{5})^{6})V^{7})S^{8})
C^{9})I^{10})OUTPUT^{11})INPUT^{12})^{13}).

$STM9=(^{18}λ $PHI,P,D,B,V,S,C,I,OUTPUT,INPUT:(^{17}(^{16}(^{15}(^{14}(^{13}(^{3}(^{2}(^{1}
(^{0}$PHI P^{0})D^{1})B^{2})V^{3})(^{12}B(^{11}(^{10}$RETRIEVE (^{9}(^{8}(^{7}(^{3}$AND (^{2}$NOT (^{1}(^{0}$
NE I^{0})0^{1})^{2})^{3})(^{6}(^{5}$LT (^{4}V(^{3}(^{2}$RETRIEVE (^{1}(^{0}$MINUS I^{0})1^{1})^{2})4^{3})^{4})^{5})
(^{1}(^{0}$PLUS I^{0})1^{1})^{6})^{7}) 1^{8})2^{9})^{10})2^{11})^{12})^{13})C^{14})I^{15})OUTPUT^{16})INPUT^{17}
)^{18}).

$STM11=(^{18}λ $PHI,P,D,B,V,S,C,I,OUTPUT,INPUT:(^{17}(^{16}(^{15}(^{14}(^{13}(^{12}(^{1}
^{1}(^{10}(^{0}$PHI P^{0})(^{9}D(^{8}(^{4}(^{3}$REPLACE (^{2}(^{1}$MINUS (^{0}$MINUSUNARY 2^{0})^{1})(^{0}
$MINUSUNARY 4^{0})^{2})^{3})4^{4})(^{7}(^{5}D(^{4}(^{3}$RETRIEVE (^{2}(^{1}$MINUS (^{0}$MINUSUNAR
Y 2^{0})^{1})(^{0}$MINUSUNARY 4^{0})^{2})^{3})4^{4})^{5})(^{6}(^{5}(^{4}$REPLACE (^{3}(^{2}$MINUS (^{1}(^{0}$
MULT I^{0})2^{1})^{2})(^{0}$MINUSUNARY 1^{0})^{3})^{4})6^{5})(^{0}CHR 216^{0})^{6})^{7})^{8})^{9})^{10})B^{11})V
^{12})S^{13})C^{14})I^{15})OUTPUT^{16})INPUT^{17})^{18}).
```

```
$STM12=(^14λ $PHI,P,D,B,V,S,C,I,OUTPUT,INPUT:(^13(^12(^11(^10(^4(^3(^2(^1
(^0$PHI P^0)D^1)B^2)V^3)S^4)(^9(^6D(^5(^4$RETRIEVE (^3(^2$MINUS (^1(^0$MINUS 0
^0)2^1)^2)(^0$MINUSUNARY 4^0)^3)^4)4^5)^6)(^8(^7$RETRIEVE (^6(^5$MINUS (^4V(^3(
^2$RETRIEVE (^1(^0$MINUS 3^0)1^1)^2)4^3)^4)^5)(^0$MINUSUNARY 1^0)^6)^7)6^8)^9)^1
^0)I^11)OUTPUT^12)INPUT^13)^14).
```

```
$STM10=(^2λ $PHI:(^1$STM11(^0$STM12 $PHI^0)^1)^2).
```

```
$STM13=(^20λ $PHI,P,D,B,V,S,C,I,OUTPUT,INPUT:(^19(^18(^17(^16(^15(^14(^1
^3(^12(^11$PHI (^10P(^9(^3(^2$REPLACE (^1(^0$MINUS C^0)192^1)^2)41^3)(^8(^4P(^3(
^2$RETRIEVE (^1(^0$MINUS C^0)192^1)^2)41^3)^4)(^7(^1(^0$REPLACE S^0)2^1)(^6(^5(
^4P(^3(^2$RETRIEVE (^1(^0$MINUS C^0)192^1)^2)41^3)^4)(^1(^0$RETRIEVE S^0)2^1)^5
)(^4(^3(^2$REPLACE (^1(^0$MINUS I^0)1^1)^2)6^3)(^3(^2$OR (^1(^0$LE I^0)2^1)^2)(^1
(^0$EQ C^0)(^0CHR 194^0)^1)^3)^4)^6)^7)^8)^9)^10)^11)D^12)B^13)V^14)S^15)C^16)I^17)
OUTPUT^18)INPUT^19)^20).
```

```
$STM14=(^20λ $PHI,P,D,B,V,S,C,I,OUTPUT,INPUT:(^19(^18(^17(^16(^15(^14(^1
^3(^1(^0$PHI P^0)D^1)(^12B(^11(^10(^9$REPLACE (^8(^7(^6(^5(^4P(^3(^2$RETRIEVE (^1
(^0$MINUS C^0)192^1)^2)41^3)^4)(^1(^0$RETRIEVE S^0)2^1)^5)(^3(^2$RETRIEVE (^1(
^0$MINUS I^0)1^1)^2)6^3)^6) 1^7)2^8)^9)2^10)1^11)^12)^13)V^14)S^15)C^16)I^17)OUTP
UT^18)INPUT^19)^20).
```

```
$STM1=(^12λ $PHI:((((((((((^11$STM2(^10$STM3(^9$STM4(^8$STM5(^7$STM6(^6
$STM7(^5$STM8(^4$STM9(^3$STM10(^2$STM13(^1$STM14(^0λ P,D,B,V,S,C,I,OUT
PUT,INPUT:$PHI^0)^1)^2)^3)^4)^5)^6)^7)^8)^9)^10)^11)(^3(^2(^1(^0$TUPINIT 3^0)41^1)
2^2)6^3))(^2(^1(^0$TUPINIT 2^0)4^1)6^2))(^1(^0$TUPINIT 1^0)2^1))(^1(^0$TUPINIT
 1^0)4^1))$OMEGA)$OMEGA)$OMEGA)$OMEGA)$OMEGA)^12).
```

```
$PROGRAM=(($STM1 $ID)$ID).
```

1.10. Conditional Statements

The reduction properties of $TRUE and $FALSE mentioned earlier, together with the definition $IF=$ID, imply a straightforward representation of if-statements:

```
{if <expression> then S1 else S2}/E =
   (λ $PHI, v1,..., vn: (...((((($IF {<expression>}/E) {S1}/E)
   {S2}/E) $PHI) v1)... vn).
```

```
{if <expression> then S1}/E =
   (λ $PHI, v1,..., vn: (...((((($IF {<expression>}/E) {S1}/E)
   $ID) $PHI) v1)... vn).
```

Case-statements could be represented as a sequence of if-statements.

1.11. Repetitive Statements

Any PASCAL loop can be transformed into a while loop. For instance repeat S until <expression> is equivalent to begin S; while <expression> do S end. While-statements themselves lead to recursive definitions. Let i be the statement number of the loop being represented:

$\{\underline{\text{while}}\ \langle\text{expression}\rangle\ \underline{\text{do}}\ S\}/E\ \underline{=}$
$\quad \$STMi=(\lambda\ \$PHI,\ v1,\ldots,\ vn:(\ldots(({}^{1}({}^{2}({}^{3}\$IF\ \ \{\langle\text{expression}\rangle\}/E^{3})$
$\quad ({}^{4}\{S\}/E\ ({}^{5}\$STMi\ \$PHI^{5})^{4})^{2})\ \$PHI^{1})\ v1)\ldots\ vn).\ .$

The small but essential difference between this if-construction and the one in the previous section 1.10 should be observed: The alternate clause has to be $PHI instead of $ID.

1.12. Example#2

The following program illustrates compilation of while loops and if statements:

```
Stmnr      Source code:
-----      ------------
           (*$U+,X- superscripts, no cross reference *)
           PROGRAM SORT(INPUT, OUTPUT);
           CONST LB=4; HB=9;
           VAR A: ARRAY[LB..HB] OF INTEGER;
               I, J, TEMP: INTEGER;
               NC: BOOLEAN;
           BEGIN
  2           I:=LB;
  3           WHILE (I<=HB) DO
  3              BEGIN
  5                 GET;
  6                 A[I]:=INPUT@
  6              END;
  7           J:=HB;
  8           NC:=FALSE;
  9           WHILE (J>LB) AND NOT NC DO
  9              BEGIN
 11                 I:=LB;
 12                 NC:=TRUE;
 13                 WHILE (I<J) DO
 14                    BEGIN
 15                       IF A[I]>A[I+1]
 15                          THEN BEGIN
 17                                  TEMP:=A[I];
 18                                  A[I]:=A[I+1];
 19                                  A[I+1]:=TEMP;
 20                                  NC:=FALSE
 20                               END;
 21                       I:=I+1
 21                    END;
 22                 J:=J-1
 22              END;
 23           I:=LB;
```

```
24           WHILE (I<=HB) DO
24              BEGIN
26                 OUTPUT@:=A[I];
26                 PUT
27              END
27        END.
```

```
* LAMBDA CODE FOR SORT

$STM2=(^7λ $PHI,NC,TEMP,J,I,A,OUTPUT,INPUT:(^6(^5(^4(^3(^2(^1(^0$PHI NC^0
)TEMP^1)J^2)4^3)A^4)OUTPUT^5)INPUT^6)^7).

$STM5=(^9λ $PHI,NC,TEMP,J,I,A,OUTPUT,INPUT:(^8λ $SPRINT,$SCARDS:(^7
(^6(^5(^4(^3(^2(^1(^0$PHI NC^0)TEMP^1)J^2)I^3)A^4)OUTPUT^5)$SCARDS^6)$SPRINT^7)
^8)^9).

$STM6=(^9λ $PHI,NC,TEMP,J,I,A,OUTPUT,INPUT:(^8(^7(^6(^3(^2(^1(^0$PHI NC^0
)TEMP^1)J^2)I^3)(^5A(^4(^3(^2$REPLACE (^1(^0$MINUS I^0)3^1)^2)6^3)INPUT^4)^5)^6)
OUTPUT^7)INPUT^8)^9).

$STM4=(^2λ $PHI:(^1$STM5(^0$STM6 $PHI^0)^1)^2).

$STM3=(^12λ $PHI,NC,TEMP,J,I,A,OUTPUT,INPUT:(^11(^10(^9(^8(^7(^6(^5(^4(^3(
^2$IF(^1(^0$LE I^0)9^1)^2)(^1$STM4(^0$STM3 $PHI^0)^1)^3)$PHI^4)NC^5)TEMP^6)J^7)
I^8)A^9)OUTPUT^10)INPUT^11)^12).

$STM7=(^7λ $PHI,NC,TEMP,J,I,A,OUTPUT,INPUT:(^6(^5(^4(^3(^2(^1(^0$PHI NC^0
)TEMP^1)9^2)I^3)A^4)OUTPUT^5)INPUT^6)^7).

$STM8=(^7λ $PHI,NC,TEMP,J,I,A,OUTPUT,INPUT:(^6(^5(^4(^3(^2(^1(^0$PHI $FA
LSE^0)TEMP^1)J^2)I^3)A^4)OUTPUT^5)INPUT^6)^7).

$STM11=(^7λ $PHI,NC,TEMP,J,I,A,OUTPUT,INPUT:(^6(^5(^4(^3(^2(^1(^0$PHI NC
^0)TEMP^1)J^2)4^3)A^4)OUTPUT^5)INPUT^6)^7).

$STM12=(^7λ $PHI,NC,TEMP,J,I,A,OUTPUT,INPUT:(^6(^5(^4(^3(^2(^1(^0$PHI $T
RUE^0)TEMP^1)J^2)I^3)A^4)OUTPUT^5)INPUT^6)^7).

$STM17=(^11λ $PHI,NC,TEMP,J,I,A,OUTPUT,INPUT:(^10(^9(^8(^7(^6(^5(^0$PHI
NC^0)(^4A(^3(^2$RETRIEVE (^1(^0$MINUS I^0)3^1)^2)6^3)^4)^5)J^6)I^7)A^8)OUTPUT^9)
INPUT^10)^11).

$STM18=(^12λ $PHI,NC,TEMP,J,I,A,OUTPUT,INPUT:(^11(^10(^9(^3(^2(^1(^0$PHI
 NC^0)TEMP^1)J^2)I^3)(^8A(^7(^3(^2$REPLACE (^1(^0$MINUS I^0)3^1)^2)6^3)(^6A(^5(^4
$RETRIEVE (^3(^2$MINUS (^1(^0$PLUS I^0)1^1)^2)3^3)^4)6^5)^6)^7)^8)^9)OUTPUT^10)
INPUT^11)^12).
```

```
$STM19=(^{11}λ $PHI,NC,TEMP,J,I,A,OUTPUT,INPUT:(^{10}(^{9}(^{8}(^{3}(^{2}(^{1}(^{0}$PHI
NC^{0})TEMP^{1})J^{2})I^{3})(^{7}A(^{6}(^{5}(^{4}$REPLACE (^{3}(^{2}$MINUS (^{1}(^{0}$PLUS I^{0})1^{1})^{2})3
^{3})^{4})6^{5})TEMP^{6})^{7})^{8})OUTPUT^{9})INPUT^{10})^{11}).

$STM20=(^{7}λ $PHI,NC,TEMP,J,I,A,OUTPUT,INPUT:(^{6}(^{5}(^{4}(^{3}(^{2}(^{1}(^{0}$PHI $F
ALSE^{0})TEMP^{1})J^{2})I^{3})A^{4})OUTPUT^{5})INPUT^{6})^{7}).

$STM16=(^{4}λ $PHI:(^{3}$STM17(^{2}$STM18(^{1}$STM19(^{0}$STM20 $PHI^{0})^{1})^{2})^{3})^{4}).

$STM15=(^{19}λ $PHI,NC,TEMP,J,I,A,OUTPUT,INPUT:(^{18}(^{17}(^{16}(^{15}(^{14}(^{13}(^{1
2}(^{11}(^{10}(^{9}(^{8}$IF(^{7}(^{5}$GT (^{4}A(^{3}(^{2}$RETRIEVE (^{1}(^{0}$MINUS I^{0})3^{1})^{2})6^{3})^{4})^{5}
)(^{6}A(^{5}(^{4}$RETRIEVE (^{3}(^{2}$MINUS (^{1}(^{0}$PLUS I^{0})1^{1})^{2})3^{3})^{4})6^{5})^{6})^{7})^{8})$ST
M16^{9})$ID^{10})$PHI^{11})NC^{12})TEMP^{13})J^{14})I^{15})A^{16})OUTPUT^{17})INPUT^{18})^{19}).

$STM21=(^{7}λ $PHI,NC,TEMP,J,I,A,OUTPUT,INPUT:(^{6}(^{5}(^{4}(^{3}(^{2}(^{1}(^{0}$PHI NC
^{0})TEMP^{1})J^{2})(^{1}(^{0}$PLUS I^{0})1^{1})^{3})A^{4})OUTPUT^{5})INPUT^{6})^{7}).

$STM14=(^{2}λ $PHI:(^{1}$STM15(^{0}$STM21 $PHI^{0})^{1})^{2}).

$STM13=(^{12}λ $PHI,NC,TEMP,J,I,A,OUTPUT,INPUT:(^{11}(^{10}(^{9}(^{8}(^{7}(^{6}(^{5}(^{4}(^{3}
(^{2}$IF(^{1}(^{0}$LT I^{0})J^{1})^{2})(^{1}$STM14(^{0}$STM13 $PHI^{0})^{1})^{3})$PHI^{4})NC^{5})TEMP^{6})
J^{7})I^{8})A^{9})OUTPUT^{10})INPUT^{11})^{12}).

$STM22=(^{7}λ $PHI,NC,TEMP,J,I,A,OUTPUT,INPUT:(^{6}(^{5}(^{4}(^{3}(^{2}(^{1}(^{0}$PHI NC
^{0})TEMP^{1})(^{1}(^{0}$MINUS J^{0})1^{1})^{2})I^{3})A^{4})OUTPUT^{5})INPUT^{6})^{7}).

$STM10=(^{4}λ $PHI:(^{3}$STM11(^{2}$STM12(^{1}$STM13(^{0}$STM22 $PHI^{0})^{1})^{2})^{3})^{4}).

$STM9=(^{14}λ $PHI,NC,TEMP,J,I,A,OUTPUT,INPUT:(^{13}(^{12}(^{11}(^{10}(^{9}(^{8}(^{7}(^{6}(
^{5}(^{4}$IF(^{3}(^{2}$AND (^{1}(^{0}$GT J^{0})4^{1})^{2})(^{0}$NOT NC^{0})^{3})^{4})(^{1}$STM10(^{0}$STM9 $P
HI^{0})^{1})^{5})$PHI^{6})NC^{7})TEMP^{8})J^{9})I^{10})A^{11})OUTPUT^{12})INPUT^{13})^{14}).

$STM23=(^{7}λ $PHI,NC,TEMP,J,I,A,OUTPUT,INPUT:(^{6}(^{5}(^{4}(^{3}(^{2}(^{1}(^{0}$PHI NC
^{0})TEMP^{1})J^{2})4^{3})A^{4})OUTPUT^{5})INPUT^{6})^{7}).

$STM26=(^{7}λ $PHI,NC,TEMP,J,I,A,OUTPUT,INPUT:(^{6}(^{5}(^{4}(^{3}(^{2}(^{1}(^{0}$PHI NC
^{0})TEMP^{1})J^{2})I^{3})A^{4})(^{4}A(^{3}(^{2}$RETRIEVE (^{1}(^{0}$MINUS I^{0})3^{1})^{2})6^{3})^{4})^{5})INPU
T^{6})^{7}).

$STM27=(^{9}λ $PHI,NC,TEMP,J,I,A,OUTPUT,INPUT:(^{8}λ $SPRINT:(^{7}(^{6}(^{5}(^{4}(
^{3}(^{2}(^{1}(^{0}$PHI NC^{0})TEMP^{1})J^{2})I^{3})A^{4})OUTPUT^{5})INPUT^{6})(($CAT $SPRINT)OUT
PUT)^{7})^{8})^{9}).

$STM25=(^{2}λ $PHI:(^{1}$STM26(^{0}$STM27 $PHI^{0})^{1})^{2}).

$STM24=(^{12}λ $PHI,NC,TEMP,J,I,A,OUTPUT,INPUT:(^{11}(^{10}(^{9}(^{8}(^{7}(^{6}(^{5}(^{4}(^{3}
(^{2}$IF(^{1}(^{0}$LE I^{0})9^{1})^{2})(^{1}$STM25(^{0}$STM24 $PHI^{0})^{1})^{3})$PHI^{4})NC^{5})TEMP^{6})
J^{7})I^{8})A^{9})OUTPUT^{10})INPUT^{11})^{12}).
```

```
$STM1=(^8λ $PHI:(((((((^7$STM2(^6$STM3(^5$STM7(^4$STM8(^3$STM9(^2$STM2
3(^1$STM24(^0λ NC,TEMP,J,I,A,OUTPUT,INPUT:$PHI^0)^1)^2)^3)^4)^5)^6)^7)$OME
GA)$OMEGA)$OMEGA)$OMEGA)(^1(^0$TUPINIT 1^0)6^1))$OMEGA)$OMEGA)^8).
```

```
$PROGRAM=(($STM1 $ID)$ID).
```

1.13. Procedures

The full modelling of procedures containing different kinds of parameter references (call by name, value, reference), global side effects and possible recursive invocations constitutes a major challenge to functional semantics [1]. Indeed, the actual implementations appear to be fairly involved. At this state of development only procedures with parameters passed by value are accepted by the code generation part of the compiler, but side-effects and recursive calls are permitted. This type of procedure will henceforth be referred to as a "V-procedure".

V-procedure definitions will be represented by names for blocks, and their parameters will be initialized dynamically with the argument values passed:

```
{V-procedure p(a1:<type1>;...; ak:<typek>); <block>}/E =
   p = (λ $VAL$a1,..., $VAL$ak: {<block>}/E). .
```

Inside the block representation (see section 1.6) a new initial value is chosen for all formal parameters ai:

{*init* ai} is VALai for all value parameters ai.

In the case that the environments of the calling statement and the V-procedure definition are the same, the representation of the call is simple:

```
{ p (<expr.1>,..., <expr.k>) }/E =
   (λ $PHI, v1,..., vn:(...(((...(p {<expr.1>}/E)...
   {<expr.k>}/E) $PHI) v1)... vn)).
```

Should the environments differ (e.g. if the V-procedure is called recursively), all additional variables of the calling environment have to be disposed of during the V-procedure execution and recovered upon return. This is done by including them into the continuation of the calling statement and re-establishing them when this continuation is accessed by the reduction process. Let G=(u1,..., um, v1,..., vn) be the calling and E=(v1,..., vn) the procedure environments.

```
{ p (<expr.1>,..., <expr.k>) }/G ≡
   (λ $PHI, u1,..., um, v1,..., vn̄:(...(((...(p {<expr.1>}/E)...
   {<expr.k>}/E)(...(($PHI u1) u2)... um)) v1)... vn)).
```

This representation solves the so-called environment conflict problem [1].

1.14. Example#3

The following example shows how procedures and their calls will be translated:

```
Stmnr       Source code:
-----       ------------
            (*$U+,X- superscripts, no cross reference *)
            PROGRAM EXAMPLE3(OUTPUT);
               VAR I, J: INTEGER;
               PROCEDURE P1(K, L: INTEGER);
                  VAR M: INTEGER;
                  BEGIN
  2                  IF 0<>L
  2                     THEN BEGIN
  6                             M:=K; K:=L; L:=M MOD L;
  7                             P1(K, L)
  7                          END
  8                     ELSE OUTPUT@:=K
  8               END; (* P1 *)
  8            PROCEDURE GCD(I, J: INTEGER);
 11               BEGIN P1(I, J); END; (* GCD *)
 11         BEGIN
 14            I:=28; J:=7;
 15            GCD(I, J+14);
 15            PUT
 16         END.
```

```
* LAMBDA CODE FOR EXAMPLE3

P1=(^0λ $VAL$L,$VAL$K:$STM1^0).

$STM4=(^6λ $PHI,M,K,L,J,I,OUTPUT:(^5(^4(^3(^2(^1(^0$PHI K^0)K^1)L^2)J^3)I^4)
OUTPUT^5)^6).

$STM5=(^6λ $PHI,M,K,L,J,I,OUTPUT:(^5(^4(^3(^2(^1(^0$PHI M^0)L^1)L^2)J^3)I^4)
OUTPUT^5)^6).

$STM6=(^6λ $PHI,M,K,L,J,I,OUTPUT:(^5(^4(^3(^2(^1(^0$PHI M^0)K^1)(^1(^0$MOD
M^0)L^1)^2)J^3)I^4)OUTPUT^5)^6).
```

```
$STM7=(^7λ $PHI,M,K,L,J,I,OUTPUT:(^6(^5(^4(^3(^1(^0P1 K^0)L^1)(^2(^1(^0$PHI
M^0)K^1)L^2)^3)J^4)I^5)OUTPUT^6)^7).

$STM3=(^4λ $PHI:(^3$STM4(^2$STM5(^1$STM6(^0$STM7 $PHI^0)^1)^2)^3)^4).

$STM8=(^6λ $PHI,M,K,L,J,I,OUTPUT:(^5(^4(^3(^2(^1(^0$PHI M^0)K^1)L^2)J^3)I^4)
K^5)^6).

$STM2=(^12λ $PHI,M,K,L,J,I,OUTPUT:(^11(^10(^9(^8(^7(^6(^5(^4(^3(^2$IF (^1(^0$
NE 0^0)L^1)^2)$STM3^3)$STM8^4)$PHI^5)M^6)K^7)L^8)J^9)I^10)OUTPUT^11)^12).

$STM1=(^2λ $PHI:((((^1$STM2(^0λ M,K,L:$PHI^0)^1)$OMEGA)$VAL$K)$VAL$L)
^2).

GCD=(^0λ $VAL$J$2,$VAL$I$2:$STM9^0).

$STM10=(^6λ $PHI,I$2,J$2,J$1,I$1,OUTPUT:(^5(^4(^3(^2(^1(^0P1 I$2^0)J$2^1)
(^1(^0$PHI I$2^0)J$2^1)^2)J$1^3)I$1^4)OUTPUT^5)^6).

$STM11=$ID.

$STM9=(^3λ $PHI:(((^2$STM10(^1$STM11(^0λ I$2,J$2:$PHI^0)^1)^2)$VAL$I$2)
$VAL$J$2)^3).

$STM13=(^3λ $PHI,J,I,OUTPUT:(^2(^1(^0$PHI J^0)28^1)OUTPUT^2)^3).

$STM14=(^3λ $PHI,J,I,OUTPUT:(^2(^1(^0$PHI 7^0)I^1)OUTPUT^2)^3).

$STM15=(^7λ $PHI,J,I,OUTPUT:(^6(^5(^4(^3(^2(^0GCD I^0)(^1(^0$PLUS J^0)14^1)^2
)$PHI ^3)J^4)I^5)OUTPUT^6)^7).

$STM16=(^5λ $PHI,J,I,OUTPUT:(^4λ $SPRINT:(^3(^2(^1(^0$PHI J^0)I^1)OUTPUT
^2)(($CAT $SPRINT)OUTPUT)^3)^4)^5).

$STM12=(^5λ $PHI:((((^4$STM13(^3$STM14(^2$STM15(^1$STM16(^0λ J,I,OUTPU
T:$PHI^0)^1)^2)^3)^4)$OMEGA)$OMEGA)$OMEGA)^5).

$PROGRAM=(($STM12 $ID)$ID).
```

1.15. Remarks on Further Language Constructs

Until now all representations described have been actually implemented in the compiler. Some remarks are in order regarding how some PASCAL features for which no lambda code is currently generated by the compiler could be translated.

Labels can be viewed as names for continuations. A goto statement then merely substitutes the representation of a the referenced label for the current program remainder. However, it seems a very tedious task to determine the continuation at a given point of a program at compilation time.

Function calls are similar to procedure calls. If no side effects occur their represention is actually very simple [1]. Otherwise many intermediate results have to be introduced because function calls can be made repeatedly within a single expression. Their representation is not theoretically difficult but it is rather hard to actually implement their translation.

The modelling of procedure and function parameters as well as pointer variables unfortunately lies beyond the reach of this compiler.

1.16. Example#4

Part 1 is concluded with a "real" PASCAL program example to multiply matrices:

```
Stmnr     Source code:
-----     ------------
          (*$U+,X- superscripts, no cross reference *)
          PROGRAM MATRIXMULT(INPUT, OUTPUT);
             CONST LB=5; HB=10;
             TYPE RANGE=LB..HB;
                  MATRIX=ARRAY[RANGE, RANGE] OF INTEGER;
             VAR A, B, C: MATRIX;
                 I, J, K: INTEGER;

             PROCEDURE READWRITE(SWITCH: BOOLEAN; C: MATRIX);
             (* Reads in A and B (global) or prints C *)
             (* according to the logical SWITCH       *)
                VAR I, J: INTEGER;
                BEGIN
   2               I:=LB;
   3               WHILE I<=HB DO
   3                  BEGIN
   5                     J:=LB;
   6                     WHILE J<=HB DO
   6                        BEGIN
   8                           IF SWITCH
   8                              THEN BEGIN
  11                                      GET; A[I, J]:=INPUT@;
  13                                      GET; B[I, J]:=INPUT@
  13                                   END
  13                              ELSE BEGIN
  15                                      OUTPUT@:=C[I][J]; PUT
  16                                   END;
  17                           J:=J+1
  17                        END;
  18                     I:=I+1
  18                  END
  18            END; (*READWRITE*)
  18
```

```
BEGIN (* of main program *)
   READWRITE(TRUE, C); (* C is just dummy *)
   I:=LB;
   WHILE (I<=HB) DO
      BEGIN
         J:=LB;
         WHILE (J<=HB) DO
            BEGIN
               C[I, J]:=0;
               K:=LB;
               WHILE (K<=HB) DO
                  BEGIN
                     C[I,J]:=C[I,J]+A[I,K]*B[K,J];
                     K:=K+1
                  END;
               J:=J+1
            END;
         I:=I+1
      END;
   READWRITE(FALSE, C)
END.
```

```
* LAMBDA CODE FOR MATRIXMULT

READWRITE=(^{0}λ $VAL$SWITCH,$VAL$C$2:$STM1^{0}).

$STM2=(^{12}λ $PHI,J$2,I$2,C$2,SWITCH,K,J$1,I$1,C$1,B,A,OUTPUT,INPU
T:(^{11}(^{10}(^{9}(^{8}(^{7}(^{6}(^{5}(^{4}(^{3}(^{2}(^{1}(^{0}$PHI J$2^{0})5^{1})C$2^{2})SWITCH^{3})K^{4})J$1^{5})I$
1^{6})C$1^{7})B^{8})A^{9})OUTPUT^{10})INPUT^{11})^{12}).

$STM5=(^{12}λ $PHI,J$2,I$2,C$2,SWITCH,K,J$1,I$1,C$1,B,A,OUTPUT,INPU
T:(^{11}(^{10}(^{9}(^{8}(^{7}(^{6}(^{5}(^{4}(^{3}(^{2}(^{1}(^{0}$PHI 5^{0})I$2^{1})C$2^{2})SWITCH^{3})K^{4})J$1^{5})I$
1^{6})C$1^{7})B^{8})A^{9})OUTPUT^{10})INPUT^{11})^{12}).

$STM10=(^{14}λ $PHI,J$2,I$2,C$2,SWITCH,K,J$1,I$1,C$1,B,A,OUTPUT,INP
UT:(^{13}λ $SPRINT,$SCARDS:(^{12}(^{11}(^{10}(^{9}(^{8}(^{7}(^{6}(^{5}(^{4}(^{3}(^{2}(^{1}(^{0}$PHI J$2^{0})I
$2^{1})C$2^{2})SWITCH^{3})K^{4})J$1^{5})I$1^{6})C$1^{7})B^{8})A^{9})OUTPUT^{10})$SCARDS^{11})$SPR
INT^{12})^{13})^{14}).

$STM11=(^{12}λ $PHI,J$2,I$2,C$2,SWITCH,K,J$1,I$1,C$1,B,A,OUTPUT,INP
UT:(^{11}(^{10}(^{9}(^{8}(^{7}(^{6}(^{5}(^{4}(^{3}(^{2}(^{1}(^{0}$PHI J$2^{0})I$2^{1})C$2^{2})SWITCH^{3})K^{4})J$1^{5}
)I$1^{6})C$1^{7})B^{8})(^{7}A(^{6}(^{3}(^{2}$REPLACE (^{1}(^{0}$MINUS I$2^{0})4^{1})^{2})6^{3})(^{5}(^{4}A(^{3}(
^{2}$RETRIEVE (^{1}(^{0}$MINUS I$2^{0})4^{1})^{2})6^{3})^{4})(^{4}(^{3}(^{2}$REPLACE (^{1}(^{0}$MINUS J
$2^{0})4^{1})^{2})6^{3})INPUT^{4})^{5})^{6})^{7})^{9})OUTPUT^{10})INPUT^{11})^{12}).

$STM12=(^{14}λ $PHI,J$2,I$2,C$2,SWITCH,K,J$1,I$1,C$1,B,A,OUTPUT,INP
UT:(^{13}λ $SPRINT,$SCARDS:(^{12}(^{11}(^{10}(^{9}(^{8}(^{7}(^{6}(^{5}(^{4}(^{3}(^{2}(^{1}(^{0}$PHI J$2^{0})I
$2^{1})C$2^{2})SWITCH^{3})K^{4})J$1^{5})I$1^{6})C$1^{7})B^{8})A^{9})OUTPUT^{10})$SCARDS^{11})$SPR
INT^{12})^{13})^{14}).
```

```
$STM13=(^{12}λ $PHI,J$2,I$2,C$2,SWITCH,K,J$1,I$1,C$1,B,A,OUTPUT,INP
UT:(^{11}(^{10}(^{9}(^{8}(^{7}(^{6}(^{5}(^{4}(^{3}(^{2}(^{1}(^{0}$PHI J$2^{0})I$2^{1})C$2^{2})SWITCH^{3})K^{4})J$1^{5}
)I$1^{6})C$1^{7})(^{7}B(^{6}(^{3}(^{2}$REPLACE (^{1}(^{0}$MINUS I$2^{0})4^{1})^{2})6^{3})(^{5}(^{4}B(^{3}(^{2}$R
ETRIEVE (^{1}(^{0}$MINUS I$2^{0})4^{1})^{2})6^{3})^{4})(^{4}(^{3}(^{2}$REPLACE (^{1}(^{0}$MINUS J$2^{0}
)4^{1})^{2})6^{3})INPUT^{4})^{5})^{6})^{7})^{8})A^{9})OUTPUT^{10})INPUT^{11})^{12}).

$STM9=(^{4}λ $PHI:(^{3}$STM10(^{2}$STM11(^{1}$STM12(^{0}$STM13 $PHI^{0})^{1})^{2})^{3})^{4}).

$STM15=(^{12}λ $PHI,J$2,I$2,C$2,SWITCH,K,J$1,I$1,C$1,B,A,OUTPUT,INP
UT:(^{11}(^{10}(^{9}(^{8}(^{7}(^{6}(^{5}(^{4}(^{3}(^{2}(^{1}(^{0}$PHI J$2^{0})I$2^{1})C$2^{2})SWITCH^{3})K^{4})J$1^{5}
)I$1^{6})C$1^{7})B^{8})A^{9})(^{5}(^{4}C$2(^{3}(^{2}$RETRIEVE (^{1}(^{0}$MINUS I$2^{0})4^{1})^{2})6^{3})^{4})
(^{3}(^{2}$RETRIEVE (^{1}(^{0}$MINUS J$2^{0})4^{1})^{2})6^{3})^{5})^{10})INPUT^{11})^{12}).

$STM16=(^{14}λ $PHI,J$2,I$2,C$2,SWITCH,K,J$1,I$1,C$1,B,A,OUTPUT,INP
UT:(^{13}λ $SPRINT:(^{12}(^{11}(^{10}(^{9}(^{8}(^{7}(^{6}(^{5}(^{4}(^{3}(^{2}(^{1}(^{0}$PHI J$2^{0})I$2^{1})C$2^{2}
)SWITCH^{3})K^{4})J$1^{5})I$1^{6})C$1^{7})B^{8})A^{9})OUTPUT^{10})INPUT^{11})(($CAT $SPRINT
)OUTPUT)^{12})^{13})^{14}).

$STM14=(^{2}λ $PHI:(^{1}$STM15(^{0}$STM16 $PHI^{0})^{1})^{2}).

$STM8=(^{16}λ $PHI,J$2,I$2,C$2,SWITCH,K,J$1,I$1,C$1,B,A,OUTPUT,INPU
T:(^{15}(^{14}(^{13}(^{12}(^{11}(^{10}(^{9}(^{8}(^{7}(^{6}(^{5}(^{4}(^{3}(^{2}(^{1}(^{0}$IF SWITCH^{0})$STM9^{1})$STM1
4^{2})$PHI^{3})J$2^{4})I$2^{5})C$2^{6})SWITCH^{7})K^{8})J$1^{9})I$1^{10})C$1^{11})B^{12})A^{13})OUTP
UT^{14})INPUT^{15})^{16}).

$STM17=(^{14}λ $PHI,J$2,I$2,C$2,SWITCH,K,J$1,I$1,C$1,B,A,OUTPUT,INP
UT:(^{13}(^{12}(^{11}(^{10}(^{9}(^{8}(^{7}(^{6}(^{5}(^{4}(^{3}(^{2}$PHI (^{1}(^{0}$PLUS J$2^{0})1^{1})^{2})I$2^{3})C$2
^{4})SWITCH^{5})K^{6})J$1^{7})I$1^{8})C$1^{9})B^{10})A^{11})OUTPUT^{12})INPUT^{13})^{14}).

$STM7=(^{2}λ $PHI:(^{1}$STM8(^{0}$STM17 $PHI^{0})^{1})^{2}).

$STM6=(^{17}λ $PHI,J$2,I$2,C$2,SWITCH,K,J$1,I$1,C$1,B,A,OUTPUT,INPU
T:(^{16}(^{15}(^{14}(^{13}(^{12}(^{11}(^{10}(^{9}(^{8}(^{7}(^{6}(^{5}(^{4}(^{3}(^{2}$IF (^{1}(^{0}$LE J$2^{0})10^{1})^{2})(^{1}
$STM7(^{0}$STM6 $PHI^{0})^{1})^{3})$PHI^{4})J$2^{5})I$2^{6})C$2^{7})SWITCH^{8})K^{9})J$1^{10})I$1
^{11})C$1^{12})B^{13})A^{14})OUTPUT^{15})INPUT^{16})^{17}).

$STM18=(^{13}λ $PHI,J$2,I$2,C$2,SWITCH,K,J$1,I$1,C$1,B,A,OUTPUT,INP
UT:(^{12}(^{11}(^{10}(^{9}(^{8}(^{7}(^{6}(^{5}(^{4}(^{3}(^{2}(^{0}$PHI J$2^{0})(^{1}(^{0}$PLUS I$2^{0})1^{1})^{2})C$2^{3}
)SWITCH^{4})K^{5})J$1^{6})I$1^{7})C$1^{8})B^{9})A^{10})OUTPUT^{11})INPUT^{12})^{13}).

$STM4=(^{3}λ $PHI:(^{2}$STM5(^{1}$STM6(^{0}$STM18 $PHI^{0})^{1})^{2})^{3}).

$STM3=(^{17}λ $PHI,J$2,I$2,C$2,SWITCH,K,J$1,I$1,C$1,B,A,OUTPUT,INPU
T:(^{16}(^{15}(^{14}(^{13}(^{12}(^{11}(^{10}(^{9}(^{8}(^{7}(^{6}(^{5}(^{4}(^{3}(^{2}$IF (^{1}(^{0}$LE I$2^{0})10^{1})^{2})(^{1}
$STM4(^{0}$STM3 $PHI^{0})^{1})^{3})$PHI^{4})J$2^{5})I$2^{6})C$2^{7})SWITCH^{8})K^{9})J$1^{10})I$1
^{11})C$1^{12})B^{13})A^{14})OUTPUT^{15})INPUT^{16})^{17}).

$STM1=(^{3}λ $PHI:(((((^{2}$STM2(^{1}$STM3(^{0}λ J$2,I$2,C$2,SWITCH:$PHI^{0})^{1})
^{2})$OMEGA)$OMEGA)$VAL$C$2)$VAL$SWITCH)^{3}).
```

```
$STM20=( 11λ $PHI,K,J,I,C,B,A,OUTPUT,INPUT:( 10( 9( 8( 7( 6( 5( 4( 3( 2( 1(
0READWRITE $TRUE 0)C 1)$PHI  2)K 3)J 4)I 5)C 6)B 7)A 8)OUTPUT 9)INPUT 10) 11
).

$STM21=( 8λ $PHI,K,J,I,C,B,A,OUTPUT,INPUT:( 7( 6( 5( 4( 3( 2( 1( 0$PHI K0
)J 1)5 2)C 3)B 4)A 5)OUTPUT 6)INPUT 7) 8).

$STM24=( 8λ $PHI,K,J,I,C,B,A,OUTPUT,INPUT:( 7( 6( 5( 4( 3( 2( 1( 0$PHI K0
)5 1)I 2)C 3)B 4)A 5)OUTPUT 6)INPUT 7) 8).

$STM27=( 13λ $PHI,K,J,I,C,B,A,OUTPUT,INPUT:( 12( 11( 10( 9( 8( 2( 1( 0$PH
I K0)J 1)I 2)( 7C( 6( 3( 2$REPLACE ( 1( 0$MINUS I 0)4 1) 2)6 3)( 5( 4C( 3( 2$RET
RIEVE ( 1( 0$MINUS I 0)4 1) 2)6 3) 4)( 4( 3( 2$REPLACE ( 1( 0$MINUS J 0)4 1) 2)
6 3)0 4) 5) 6) 7) 8)B 9)A 10)OUTPUT 11)INPUT 12) 13).

$STM28=( 8λ $PHI,K,J,I,C,B,A,OUTPUT,INPUT:( 7( 6( 5( 4( 3( 2( 1( 0$PHI 50
)J 1)I 2)C 3)B 4)A 5)OUTPUT 6)INPUT 7) 8).

$STM31=( 18λ $PHI,K,J,I,C,B,A,OUTPUT,INPUT:( 17( 16( 15( 14( 13( 2( 1( 0$
PHI K0)J 1)I 2)( 12C( 11( 3( 2$REPLACE ( 1( 0$MINUS I 0)4 1) 2)6 3)( 10( 4C( 3(
2$RETRIEVE ( 1( 0$MINUS I 0)4 1) 2)6 3) 4)( 9( 3( 2$REPLACE ( 1( 0$MINUS J 0)
4 1) 2)6 3)( 8( 6$PLUS ( 5( 4C( 3( 2$RETRIEVE ( 1( 0$MINUS I 0)4 1) 2)6 3) 4)( 3(
2$RETRIEVE ( 1( 0$MINUS J 0)4 1) 2)6 3) 5) 6)( 7( 6$MULT ( 5( 4A( 3( 2$RETRIEV
E ( 1( 0$MINUS I 0)4 1) 2)6 3) 4)( 3( 2$RETRIEVE ( 1( 0$MINUS K 0)4 1) 2)6 3) 5)
6)( 5( 4B( 3( 2$RETRIEVE ( 1( 0$MINUS K 0)4 1) 2)6 3) 4)( 3( 2$RETRIEVE ( 1( 0$
MINUS J 0)4 1) 2)6 3) 5) 7) 8) 9) 10) 11) 12) 13)B 14)A 15)OUTPUT 16)INPUT 17) 18
).

$STM32=( 10λ $PHI,K,J,I,C,B,A,OUTPUT,INPUT:( 9( 8( 7( 6( 5( 4( 3( 2$PHI (
1( 0$PLUS K 0)1 1) 2)J 3)I 4)C 5)B 6)A 7)OUTPUT 8)INPUT 9) 10).

$STM30=( 2λ $PHI:( 1$STM31( 0$STM32 $PHI 0) 1) 2).

$STM29=( 13λ $PHI,K,J,I,C,B,A,OUTPUT,INPUT:( 12( 11( 10( 9( 8( 7( 6( 5( 4(
3( 2$IF ( 1( 0$LE K 0)10 1) 2)( 1$STM30( 0$STM29 $PHI 0) 1) 3)$PHI 4)K 5)J 6)I
7)C 8)B 9)A 10)OUTPUT 11)INPUT 12) 13).

$STM33=( 9λ $PHI,K,J,I,C,B,A,OUTPUT,INPUT:( 8( 7( 6( 5( 4( 3( 2( 0$PHI K0
)( 1( 0$PLUS J 0)1 1) 2)I 3)C 4)B 5)A 6)OUTPUT 7)INPUT 8) 9).

$STM26=( 4λ $PHI:( 3$STM27( 2$STM28( 1$STM29( 0$STM33 $PHI 0) 1) 2) 3) 4).

$STM25=( 13λ $PHI,K,J,I,C,B,A,OUTPUT,INPUT:( 12( 11( 10( 9( 8( 7( 6( 5( 4(
3( 2$IF ( 1( 0$LE J 0)10 1) 2)( 1$STM26( 0$STM25 $PHI 0) 1) 3)$PHI 4)K 5)J 6)I
7)C 8)B 9)A 10)OUTPUT 11)INPUT 12) 13).

$STM34=( 8λ $PHI,K,J,I,C,B,A,OUTPUT,INPUT:( 7( 6( 5( 4( 3( 2( 1( 0$PHI K0
)J 1)( 1( 0$PLUS I 0)1 1) 2)C 3)B 4)A 5)OUTPUT 6)INPUT 7) 8).
```

```
$STM23=(^3λ $PHI:(^2$STM24(^1$STM25(^0$STM34 $PHI^0)^1)^2)^3).

$STM22=(^13λ $PHI,K,J,I,C,B,A,OUTPUT,INPUT:(^12(^11(^10(^9(^8(^7(^6(^5(^4(
^3(^2$IF (^1(^0$LE I^0)10^1)^2)(^1$STM23(^0$STM22 $PHI^0)^1)^3)$PHI^4)K^5)J^6)I
^7)C^8)B^9)A^10)OUTPUT^11)INPUT^12)^13).

$STM35=(^1(^0READWRITE $FALSE^0)C^1).

$STM19=(^5λ $PHI:(((((((((^4$STM20(^3$STM21(^2$STM22(^1$STM35(^0λ K,J,
I,C,B,A,OUTPUT,INPUT:$PHI^0)^1)^2)^3)^4)$OMEGA)$OMEGA)$OMEGA)(^2(^1(^0$T
UPINIT 2^0)6^1)6^2))(^2(^1(^0$TUPINIT 2^0)6^1)6^2))(^2(^1(^0$TUPINIT 2^0)6^1)6
^2))$OMEGA)$OMEGA)^5).

$PROGRAM=(($STM19 $ID)$ID).
```

```
PPPPPPPPPPP    AAAAAAAAAA   RRRRRRRRRRR   TTTTTTTTTTTT   2222222222
PPPPPPPPPPPP  AAAAAAAAAAAA  RRRRRRRRRRRR  TTTTTTTTTTTT  222222222222
PP        PP  AA        AA  RR        RR       TT       22        22
PP        PP  AA        AA  RR        RR       TT                 22
PP        PP  AA        AA  RR        RR       TT                 22
PPPPPPPPPPPP  AAAAAAAAAAAA  RRRRRRRRRRRR       TT                22
PPPPPPPPPPP   AAAAAAAAAAAA  RRRRRRRRRRR        TT              22
PP            AA        AA  RR    RR           TT            22
PP            AA        AA  RR     RR          TT          22
PP            AA        AA  RR      RR         TT        22
PP            AA        AA  RR       RR        TT       222222222222
PP            AA        AA  RR        RR       TT       222222222222
```

2.1. Overall Features about the Compiler

The compiler itself is written in standard PASCAL. It consists of 5400 source lines and its object module generated by the PASCAL 8000 compiler on the Michigan Terminal System at Rensselaer Polytechnic Institute occupies 160K bytes of storage (excluding a variable sized run-time stack). It is written in a structured style (102 procedures and functions) composing a one pass translation. Several layers of executional tasks are well distinguished: a finite state machine for lexical scanning, an attributed LL(1) parser for syntactic analysis and semantic activities including type checking procedures, and the generator of lambda-expressions employing a garbage collecting system for character strings of dynamic lengths. Due to the size and sparseness of the transition tables of both the finite state automaton and the pushdown machine correspondiong to the LL(1) grammar (73 possible stack symbols and 50 input tokens), implicit program code was used to realize them.

The compiler generates a source program listing including accumulated statement and and semicolon counts, as well as block levels and depths of nested loops, compound and case statements. The compiler can also produce a cross-reference of all identifiers with respect to the semicolon counts of their occurences and a specification of their explicit types. Context-sensitive error messages are recorded on a temporary file which is finally appended to the source listings. Currently, three compiler options are supported which may be specified in the usual way within comment braces [4]: X$\pm$, S$\pm$, and U$\pm$. X- will supress the printing of the cross-reference, S+ will extend the syntax of the language accepted (see formal parameters and function declarations), and U+ will cause the compiler to attach superscripts to paired parenthesis in the code generated. The default values of these options are "(*$X+,S+,U-*)".

It took the compiler 1.71 seconds of IBM 3033 CPU time to compile the program in section 1.16. The machine was running under the Michigan Terminal System, with 33 terminals active at the time of the experiment.

2.2. The Lexical Scanner

The lexical scanner advances through the input stream of characters until it recognizes a new token which it passes to the parser [6]. There are 50 different (parameterized) tokens which become the terminals of the later LL(1) grammar. Some have an associated parameter value. This value does not influence the parse but is used in later tasks. From a theoretical point of view, each token and its parameter value is obtained by a single finite automaton and the complete lexical scanner is then a parallel composition of these. The two most important automata are the scanners for identifiers and numbers:

2.2.1. Identifiers

This compiler distinguishes PASCAL identifiers by their first ten characters, which are entered into a hashing table and eventually padded by blanks. The hashing function is the sum of the numerical codes of the first, second, fourth and fifth character modulo a prime number which is close to half of the size of the whole table. Hashing collisions are resolved by a chaining algorithm using the second half of the hashing table as overflow area. The parameter value of the token IDENTIFIER is the hashed table index of each recognized identifier. If there is no danger of ambiguities, identifiers and their corresponding hashing table indices will not be further distinguished.

Keywords cannot be used as identifiers. A binary search is conducted through an alphabetically sorted table of the 35 standard PASCAL keywords and all but four (the operators AND, MOD, DIV, and IN) become tokens themselves.

2.2.2. Numbers

The compiler contains an explicit finite automaton to accept numbers [6]. In the following transition table each output symbol (denoted by a lower case letter) corresponds to a certain action specified below the table. The initial state is 1, and the final ("accepting") state 0:

STATE vs. INPUT CHARACTER

	'0'..'9'	'.'	'E'	'+', '-'	others
1	1 a	2 d	4 c	0 b	0 b
2	3 c	0 e	0 e	0 e	0 e
3	3 d	0 d	4 d	0 d	0 d
4	6 d	0 f	0 f	5 d	0 f
5	6 d	0 f	0 f	0 f	0 f
6	6 d	0 d	0 d	0 d	0 d

Actions:

a) Record a new digit in the integral part of number.
b) Unsigned integer terminated.
c) Unsigned real without fractional part encountered.
d) Process fractional and exponential part in unsigned real now.

e) If current character = ')' then unsigned integer terminated and current character := ']'.
   If current character = '.' then unsigned integer terminated and current character := double dot.
   Otherwise proceed like f).
f) Error in real constant: Digit expected but not found.

Two tokens, viz. UNSGINTEG and UNSGREAL, correspond to unsigned integers and real numbers, resp. . Their parameters contain their actual numerical value. Signs will be distinguished from "adding operators" on a later grammatical level.

There are six separate tokens for the various PASCAL operators. However, some may also serve another syntactic purpose. E.g. EQUALSYM in definitions of constants and types or PLUSMINUS in signed numbers.

Token:	Meaning of parameter values:
NOTSYM	None.
PLUSMINUS	1: '+', 2: '-'.
ORSYM	None.
MULTOPER	1: '*', 2: '/', 3: DIV, 4: MOD, 5: AND.
EQUALSYM	None.
RELOPER	2: '<>', 3: '<', 4: '>', 5: '<=', 6: '>=', 7: IN.

All literals are collected in a vector of characters, which is MAXSTRGL long. The token STRINGSYM associates the entry of a certain literal by its parameter field in the following fashion:

Parameter value = starting index * MAXSTRGL + length.

The remaining tokens correspond to special symbols without parameter values:

LPARASYM: '(', RPARASYM: ')', LBRACKSYM: '[', RBRACKSYM: ']',
SEMICSYM: ';', COMMASYM: ',', PERIODSYM: '.', DOUBLEDOT: '..',
COLONSYM: ':', BECOMES: ':=', POINTER: PASCAL pointer symbol.

Brackets may be also written as '(.' and '.)'. Comments are enclosed by braces or by '(*' and '*)'. The pointer symbol of this implemention is the ampersand.

2.3. An LL(1) Grammar for Standard PASCAL

Before proceeding with a compendious description of the attributed LL(1) translation, the underlying context-free grammar itself shall be scrutinized. It consists of 57 non-terminals, 50 terminals (namely all tokens described in section 2.2) and 135 productions. All but one non-terminnal yield disjoint selection sets [6] for different productions. The selection sets of the productions

(i) <else clause> ::= ELSESYM <statement>.
(ii) <else clause> ::= <empty>.

Are {ELSESYM} for (i) and {ENDSYM, SEMICSYM, UNTILSYM, ELSESYM} for (ii). This is a consequence of the well-known ambiguity

$$\underline{\text{if}}\ \text{e1}\ \underline{\text{then}}\ \underline{\text{if}}\ \text{e2}\ \underline{\text{then}}\ \text{S1}\ \underline{\text{else}}\ \text{S2}.$$

By definition [4], each else clause is paired with the last unmatched then clause. This is equivalent to removing the ELSESYM from the selection set of (ii). With respect to this modification the grammar becomes LL(1) [6].+

Now the complete grammar shall be given in BNF notation. In addition, the selection set of each production will be specified unless its right-hand side starts with a terminal. (In this case, the terminal is the only element of its selection set.) The starting symbol is <program>:

(1) <identifierlist> ::= COMMASYM IDENTIFIER <identifierlist>.

(2) <identifierlist> ::= <empty>.
Selset(2) = {RPARASYM, SEMICSYM, COLONSYM}.

(3) <labeldclremainder> ::= COMMASYM UNSGINTEG <labeldclremainder>.

(4) <labeldclremainder> ::= SEMICSYM.

(5) <labeldeclaration> ::= LABELSYM UNSGINTEG <labeldclremainder>.

(6) <labeldeclaration> ::= <empty>.
Selset(6) = {CONSTSYM, TYPESYM, VARSYM, PROCSYM, FUNCSYM, BEGINSYM}.

(7) <nonidentconstrem> ::= IDENTIFIER.

(8) <nonidentconstrem> ::= UNSGINTEG.

(9) <nonidentconstrem> ::= UNSGREAL.

(10) <nonidentconstant> ::= PLUSMINUS <nonidentconstrem>.

+ It is not known to the author whether there exists a "pure" LL(1) grammar for standard PASCAL. E.g. ALGOL 60 is known to be "inherently non-LL(1)" [5].

```
(11) <nonidentconstant> ::= UNSGINTEG.

(12) <nonidentconstant> ::= UNSGREAL.

(13) <nonidentconstant> ::= STRINGSYM.

(14) <constant> ::= IDENTIFIER.

(15) <constant> ::= <nonidentconstant>.
     Selset(15) = {UNSGINTEG, PLUSMINUS, UNSGREAL, STRINGSYM}.

(16) <constantlist> ::= COMMASYM <constant> <constantlist>.

(17) <constantlist> ::= <empty>.
     Selset(17) = {COLONSYM}.

(18) <constdefinpartrem> ::= IDENTIFIER EQUALSYM <constant>
                             SEMICSYM <constdefinpartrem>.

(19) <constdefinpartrem> ::= <empty>.
     Selset(19) = {TYPESYM, VARSYM, PROCSYM, FUNCSYM, BEGINSYM}.

(20) <constantdefinpart> ::= CONSTSYM IDENTIFIER EQUALSYM <cons-
                             tant> SEMICSYM <constdefinpartrem>.

(21) <constantdefinpart> ::= <empty>.
     Selset(21) = {TYPESYM, VARSYM, PROCSYM, FUNCSYM, BEGINSYM}.

(22) <simpletyperemaind> ::= DOUBLEDOT <constant>.

(23) <simpletyperemaind> ::= <empty>.
     Selset(23) = {RPARASYM, SEMICSYM, COMMASYM, RBRACKSYM,
                  ENDSYM}.

(24) <simpletype> ::= LPARASYM IDENTIFIER <identifierlist>
                      RPARASYM.

(25) <simpletype> ::= IDENTIFIER <simpletyperemaind>.

(26) <simpletype> ::= <nonidentconstant> DOUBLEDOT <constant>.
     Selset(26) = {UNSGINTEG, PLUSMINUS, UNSGREAL, STRINGSYM}.

(27) <simpletypelist> ::= COMMASYM <simpletype> <simpletype-
                          list>.

(28) <simpletypelist> ::= <empty>.
     Selset(28) = {RBRACKSYM}.
```

```
(29) <variant> ::= <constant> <constantlist> COLONSYM LPARASYM
                   <fieldlist> RPARASYM.
     Selset(29) = {IDENTIFIER, UNSGINTEG, PLUSMINUS, UNSGREAL,
                   STRINGSYM}.

(30) <variant> ::= <empty>.
     Selset(30) = {RPARASYM, SEMICSYM, ENDSYM}.

(31) <variantlist> ::= SEMICSYM <variant> <variantlist>.

(32) <variantlist> ::= <empty>.
     Selset(32) = {RPARASYM, ENDSYM}.

(33) <tagfieldremainder> ::= COLONSYM IDENTIFIER.

(34) <tagfieldremainder> ::= <empty>.
     Selset(34) = {OFSYM}.

(35) <fieldlistremaind> ::= SEMICSYM <fieldlist>.

(36) <fieldlistremaind> ::= <empty>.
     Selset(36) = {RPAASYM, ENDSYM}.

(37) <recordsection> ::= IDENTIFIER <identifierlist> COLONSYM
                         <type>.

(38) <recordsection> ::= <empty>.
     Selset(38) = {RPARASYM, SEMICSYM, ENDSYM}.

(39) <fieldlist> ::= <recordsection> <fieldlistremaind>.
     Selset(39) = {IDENTIFIER, RPARASYM, SEMICSYM, ENDSYM}.

(40) <fieldlist> ::= CASESYM IDENTIFIER <tagfieldremainder> OF-
                     SYM <variant> <variantlist>.

(41) <unpackstructtype> ::= ARRAYSYM LBRACKSYM <simpletype>
                            <simpletypelist> RBRACKSYM OFSYM
                            <type>.

(42) <unpackstructtype> ::= RECORDSYM <fieldlist> ENDSYM.

(43) <unpackstructtype> ::= FILESYM OFSYM <type>.

(44) <unpackstructtype> ::= SETSYM OFSYM <simpletype>.

(45) <type> ::= <simpletype>.
     Selset(45) = {IDENTIFIER, LPARASYM, UNSGINTEG, PLUSMINUS,
                   UNSGREAL, STRINGSYM}.

(46) <type> ::= PACKEDSYM <unpackstructtype>.
```

(47) <type> ::= <unpackstructtype>.
Selset(47) = {ARRAYSYM, RECORDSYM, FILESYM, SETSYM}.

(48) <type> ::= POINTER IDENTIFIER.

(49) <typedefinpartrem> ::= IDENTIFIER EQUALSYM <type> SEMICSYM <typedefinpartrem>.

(50) <typedefinpartrem> ::= <empty>.
Selset(50) = {VARSYM, PROCSYM, FUNCSYM, BEGINSYM}.

(51) <typedefinitionprt> ::= TYPESYM IDENTIFIER EQUALSYM <type> SEMICSYM <typedefinpartrem>.

(52) <typedefinitionprt> ::= <empty>.
Selset(52) = {VARSYM, PROCSYM, FUNCSYM, BEGINSYM}.

(53) <variabledclprtrem> ::= IDENTIFIER <identifierlist> COLONSYM <type> SEMICSYM <variabledclprtrem>.

(54) <variabledclprtrem> ::= <empty>.
Selset(54) = {PROCSYM, FUNCSYM, BEGINSYM}.

(55) <variabledeclarprt> ::= VARSYM IDENTIFIER <identifierlist> COLONSYM <type> SEMICSYM <variabledclprtrem>.

(56) <variabledeclarprt> ::= <empty>.
Selset(56) = {PROCSYM, FUNCSYM, BEGINSYM}.

(57) <formalparameter> ::= IDENTIFIER <identifierlist> COLONSYM IDENTIFIER.

(58) <formalparameter> ::= VARSYM IDENTIFIER <identifierlist> COLONSYM IDENTIFIER.

If the compiler option S+ is activated, an explicit <type> will be accepted in formal parameters (productions 57 and 58). Any implicitely defined scalar identifiers within this <type> are then global to the scope of the procedure or function body. No pointer references are forwarded.

(59) <formalparameter> ::= FUNCSYM IDENTIFIER <identifierlist> COLONSYM IDENTIFIER.

(60) <formalparameter> ::= PROCSYM IDENTIFIER <identifierlist>.

(61) <formparameterlist> ::= SEMICSYM <formalparameter> <formparameterlist>.

```
(62) <formparameterlist> ::= <empty>.
     Selset(62) = {RPARASYM}.

(63) <formparameterpart> ::= LPARASYM <formalparameter> <form-
                             parameterlist> RPARASYM.

(64) <formparameterpart> ::= <empty>.
     Selset(64) = {SEMICSYM, COLONSYM}.

(65) <procfuncdeclarat> ::= PROCSYM IDENTIFIER <formparameter-
                            part> SEMICSYM <block>.

(66) <procfuncdeclarat> ::= FUNCSYM IDENTIFIER <formparameter-
                            part> COLONSYM IDENTIFIER SEMICSYM
                            <block>.
```

If the compiler option S+ is activated, an explicit <type> will be accepted in the function return type. Any implicitely defined scalar identifiers within this <type> are then global to the scope of the function body. No pointer references are forwarded.

```
(67) <procfuncdclpart> ::= <procfuncdeclarat> SEMICSYM <proc-
                           funcdclpart>.
     Selset(67) = {PROCSYM, FUNCSYM}.

(68) <procfuncdclpart> ::= <empty>.
     Selset(68) = {BEGINSYM}.

(69) <expressionlist> ::= COMMASYM <expression>.

(70) <expressionlist> ::= <empty>.
     Selset(70) = {RPARASYM, RBRACKSYM}.

(71) <variableselector> ::= LBRACKSYM <expression> <expression-
                            list> RBRACKSYM <variableselector>.

(72) <variableselector> ::= PERIODSYM IDENTIFIER <variablese-
                            lector>.

(73) <variableselector> ::= POINTER <variableselector>.

(74) <variableselector> ::= <empty>.
     Selset(74) = {RPARASYM, SEMICSYM, COMMASYM, EQUALSYM,
                  RBRACKSYM, DOUBLEDOT, BECOMES, DOSYM, OFSYM,
                  ORSYM, TOSYM, ENDSYM, ELSESYM, THENSYM,
                  UNTILSYM, DOWNTOSYM, PLUSMINUS, RELOPER,
                  MULTOPER}.
```

```
(75) <actparameterpart> ::= LPARASYM <expression> <expression-
                             list> RPARASYM.

(76) <identifierremaind> ::= <variableselector>.
     Selset(76) ={PERIODSYM, RPARASYM, SEMICSYM, COMMASYM,
                  EQUALSYM, LBRACKSYM, RBRACKSYM, DOUBLEDOT, DO-
                  SYM, OFSYM, ORSYM, TOSYM, ENDSYM, ELSESYM,
                  THENSYM, UNTILSYM, DOWNTOSYM, POINTER, PLUS-
                  MINUS, RELOPER, MULTOPER}.

(77) <identifierremaind> ::= <actparameterpart>.
     Selset(77) = {LPARASYM}.

(78) <setelementremaind> ::= DOUBLEDOT <expression>.

(79) <setelementremaind> ::= <empty>.
     Selset(79) = {COMMASYM, RBRACKSYM}.

(80) <setelement> ::= <expression> <setelementremaind>.
     Selset(80) = {LPARASYM, NOTSYM, STRINGSYM, PLUSMINUS,
                   IDENTIFIER, UNSGINTEG}.

(81) <setelementlist> ::= COMMASYM <setelement> <setelement-
                          list>.

(82) <setelementlist> ::= <empty>.
     Selset(82) = {RBRACKSYM}.

(83) <setrange> ::= <setelement> <setelementlist>.
     Selset(83) = {LPARASYM, NOTSYM, STRINGSYM, PLUSMINUS,
                   IDENTIFIER, UNSGINTEG}.

(84) <setrange> ::= <empty>.
     Selset(84) = {RBRACKSYM}.

(85) <factor> ::= NOTSYM <factor>.

(86) <factor> ::= IDENTIFIER <identifieremaind>.

(87) <factor> ::= STRINGSYM.

(88) <factor> ::= UNSGINTEG.

(89) <factor> ::= UNSGREAL.

(90) <factor> ::= NILSYM.

(91) <factor> ::= LBRACKSYM <setrange> RBRACKSYM.

(92) <factor> ::= LPARASYM <expression> RPARASYM.

(93) <factorlist> ::= MULTOPER <factor> <factorlist>.
```

```
(94) <factorlist> ::= <empty>.
     Selset(94) = {RPARASYM, SEMICSYM, COMMASYM, EQUALSYM,
                   RBRACKSYM, DOUBLEDOT, DOSYM, OFSYM, ORSYM,
                   TOSYM, ENDSYM, ELSESYM, THENSYM, UNTILSYM,
                   DOWNTOSYM, PLUSMINUS, RELOPER}.

(95) <term> ::= <factor> <factorlist>.
     Selset(95) = {LPARASYM, LBRACKSYM, NILSYM, NOTSYM, STRING-
                   SYM, IDENTIFIER, UNSGREAL, UNSGINTEG}.

(96) <termlist> ::= PLUSMINUS <term> <termlist>.

(97) <termlist> ::= ORSYM <term> <termlist>.

(98) <termlist> ::= <empty>.
     Selset(98) = {RPARASYM, SEMICSYM, COMMASYM, EQUALSYM,
                   RBRACKSYM, DOUBLEDOT, DOSYM, OFSYM, TOSYM,
                   ENDSYM, ELSESYM, THENSYM, UNTILSYM, DOWNTO-
                   SYM, RELOPER}.

(99) <simpleexpression> ::= PLUSMINUS <term>.

(100) <simpleexpression> ::= <term> <termlist>.
      Selset(100) = {LPARASYM, LBRACKSYM, NILSYM, NOTSYM,
                     STRINGSYM, IDENTIFIER, UNSGREAL, UNSG-
                     INTEG}.

(101) <simpleexpressrem> ::= EQUALSYM <simpleexpression>.

(102) <simpleexpressrem> ::= RELOPER <simpleexpression>.

(103) <simpleexpressrem> ::= <empty>.
      Selset(103) = {RPARASYM, SEMICSYM, COMMASYM, RBRACKSYM,
                     DOUBLEDOT, DOSYM, OFSYM, TOSYM, ENDSYM,
                     ELSESYM, THENSYM, UNTILSYM, DOWNTOSYM}.

(104) <expression> ::= <simpleexpression> <simpleexpressrem>.
      Selset(104) = {LPARASYM, LBRACKSYM, NILSYM, NOTSYM,
                     STRINGSYM, PLUSMINUS, IDENTIFIER, UNSGREAL,
                     UNSGINTEG}.

(105) <simplestatemntrem> ::= <variableselector> BECOMES <ex-
                              pression>.
      Selset(105) = {PERIODSYM, LBRACKSYM, POINTER, BECOMES}.

(106) <simplestatemntrem> ::= <actparameterpart>.
      Selset(106) = {LPARASYM}.

(107) <simplestatemntrem> ::= <empty>.
      Selset(107) = {SEMICSYM, ENDSYM, UNTILSYM, ELSESYM}.
```

```
(108) <compoundstmntrem> ::= SEMICSYM <statement> <compound-
                             stmntrem>.

(109) <compoundstmntrem> ::= ENDSYM.

(110) <elseclause> ::= ELSESYM <statement>.

(111) <elseclause> ::= <empty>.
      Selset(111) = {SEMICSYM, ENDSYM, UNTILSYM}.

(112) <caseelement> ::= <constant> <constantlist> COLONSYM
                        <statement>.
      Selset(112) = {IDENTIFIER, UNSGINTEG, PLUSMINUS, UNSGREAL,
                     STRINGSYM}.

(113) <caseelement> ::= <empty>.
      Selset(113) = {SEMICSYM, ENDSYM}.

(114) <caseelementlist> ::= SEMICSYM <caseelement> <caseelement-
                            list>.

(115) <caseelementlist> ::= ENDSYM.

(116) <repeatstatemntlst> ::= SEMICSYM <statement> <repeatstmnt-
                              list>.

(117) <repeatstatemntlst> ::= UNTIL.

(118) <forstatementrem> ::= TOSYM <expression> DOSYM <state-
                            ment>.

(119) <forstatementrem> ::= DOWNTOSYM <expression> DOSYM <state-
                            ment>.

(120) <withvariablelist> ::= COMMASYM IDENTIFIER <variablese-
                             lector> <withvariablelist>.

(121) <withvariablelist> ::= DOSYM.

(122) <unlabeledstatemnt> ::= IDENTIFIER <simplestatemntrem>.

(123) <unlabeledstatemnt> ::= BEGINSYM <statement> <compound-
                              stmntrem>.

(124) <unlabeledstatemnt> ::= IFSYM <expression> THENSYM <state-
                              ment> <elseclause>.

(125) <unlabeledstatemnt> ::= CASESYM <expression> OFSYM <case-
                              element> <caseelementlist>.
```

```
(126) <unlabeledstatemnt> ::= WHILESYM <expression> DOSYM
                              <statement>

(127) <unlabeledstatemnt> ::= REPEATSYM <statement> <repeat-
                              statemntlst> <expression>.

(128) <unlabeledstatemnt> ::= FORSYM IDENTIFIER BECOMES <expres-
                              sion> <forstatementrem>.

(129) <unlabeledstatemnt> ::= WITHSYM IDENTIFIER <variableselec-
                              tor> <withvariablelist> <state-
                              ment>.

(130) <unlabeledstatemnt> ::= GOTOSYM UNSGINTEG.

(131) <unlabeledstatemnt> ::= <empty>.
      Selset(131) = {SEMICSYM, ENDSYM, ELSESYM, UNTILSYM}.

(132) <statement> ::= UNSGINTEG COLONSYM <unlabeledstatemnt>.

(133) <statement> ::= <unlabeledstatemnt>.
      Selset(133) = {IDENTIFIER, SEMICSYM, ENDSYM, CASESYM, BE-
                    GINSYM, IFSYM, WHILESYM, REPEATSYM, FORSYM,
                    WITHSYM, GOTOSYM, ELSESYM, UNTILSYM}.

(134) <block> ::= <labeldeclaration> <constdefinpart> <type-
                  definitionprt> <variabledeclarprt> <procfunc-
                  declarat> BEGINSYM <statement> <compoundstmnt-
                  rem>.
      Selset(134) = {LABELSYM, CONSTSYM, TYPESYM, VARSYM, PROC-
                    SYM, FUNCSYM, BEGINSYM}.

(135) <program> ::= PROGRAMSYM IDENTIFIER LPARASYM IDENTIFIER
                    <identifierlist> RPARASYM SEMICSYM <block>
                    PERIODSYM.
```

Appendix B contains also the transition table of the corresponding one-state pushdown automaton. This table has been produced by a program which inspects given context-free grammars for being LL(1).

2.4. An Attributed Translation of Lists

The syntax of PASCAL is rich in lists of certain entities (e.g. identifier list, constant list, simple type list, expression list, simple type list, formal parameter list, variant list, etc.). One general scheme applies throughout the entire tranlation: Let's assume first that the non-terminals <entity>, <list> and <item> have the following synthesized and inherited attributes [6]:

DESCR....... a data structure containing the description of any

	item (synth.).
FIRST, SEC...	same as DESCR (inher.).
HEAD.........	The head pointer to a list of all item DESCRs (synth.).
CAR, CDR.....	pointers to lists of item DESCRs (synth.).

A suitable translation grammar [6] to build a list of item descriptors follows. Action symbols will be surrounded by dashes:

```
(i)   <entity>(HEAD) ::= <item>(FIRST) <list>(FIRST,HEAD).
(ii)  <list>(FIRST,CAR) ::= <separator> <item>(SEC)
                            <list>(SEC,CDR) -action-.
(iii) <list>(FIRST,CAR) ::= <terminator> -CDR:=nil- -action-.
```

Where -action- means:

-allocate CAR and put FIRST into it; catenate CDR to it-.

This particular grammar yields a simple method for recovering from a syntactic error: Suppose <list> is called but neither a <separator> nor a <terminator> appear as input tokens. Then CAR can be set to nil and the error handler may advance the input stream to a global synchronization symbol such as a SEMICSYM (see also section 2.8).

2.5. The Main Data Structures of Translation

Each block of the source program may introduce a new set of identifiers which must be disregarded when the parsing of this block is completed. A record describing each new identifier will contain the block level number of this identifier. It is pushed onto a stack of identifier descriptors as soon as the referred identifier is sufficiently defined. On leaving a block, all identifier descriptors of its level are popped off this stack. In order to find an identifier descriptor within this stack, its position is entered into the identifier's hashing table element. Should there already be an address of an identifier defined at a lower block level, a stacking mechanism is engaged. It should be noticed that this algorithm reduces the search time for an identifier to a look-up in the hashing table. Predefined identifiers are pushed onto this stack at the very beginning of the parse and possess the level number zero.

Six classes of identifiers are distinguished. Identifiers may denote scalars, types, variables, record fields, procedures, or functions. An identifier description record contains the following fields (some will be pointers to a record describing a certain type which will be discussed below):

IDNR......	the hashing index of the referenced identifier.
LEVNR.....	the block level of its definition.

IDCLASS... the class it belongs to.

Depending on the value of the field IDCLASS, the descriptor record contains the following additional fields:

For a scalar identifier:
   STYP.... a pointer to its (scalar) type descriptor.
   VALUE... its cardinality.

For a type identifier:
   TYP... a pointer to the type it denotes.

For a variable:
   TYP.... a pointer to its type descriptor.
   PARM... a flag signalling whether it is a formal parameter and if so what kind of parameter it is (variable or value).

For a record field:
   TYP... a pointer to its type record.

For a procedure or function:
   ARGTYPS.... a list of type records for its parameters.
   SWFORW..... a flag signalling whether forward declared.
   FORWARGS... a list of identifier descriptors for all formal parameters in case of forward declaration. These records will be pushed onto the stack as soon as the procedure or function body is specified.
   RETTYP..... the function's return type.
   PARM....... a flag similar to the PARM field in a variable.

There are five classes of types, namely scalar, array, record and pointer types, and the undefined type.+ Each type record contains a switch telling which class the type belongs to and the following corresponding fields:

For a scalar type:
   SCIDNR........... its type identifier's hashing index.
   SCLEVNR.......... the level of its definition.
   LOWBND, HIGBND... its range of cardinalities.
   No distinction is made between subrange types and their matching scalar types (encompassing the full range). The type INTEGER at level zero ranges over -MAXINT .. MAXINT. The type REAL at level zero has no associated bounds. If no explicit type identifier is given, a unique identifier "$IMPLTn" (where n is a certain number) will be used instead. The compiler internally translates a label N into a scalar identifier '$LABELN' and gives it the scalar type 'LABEL' of level

+ The compiler does not yet support set and file types. They are treated as undefined types.

zero ranging from 1 to 9999.

For an array type:
INDXTYP... the (scalar) type record of the array index.
COMPTYP... the type record of the array component.
SWPACK.... a flag signalling whether the array is packed.
Matrices and multidimensional arrays will always obtain an array type as their component types. E.g. array [it1, it2] of t will become array [it1] of array [it2] of t.

For a record type:
SECTNS... a list of field identifier descriptors.
SWPACK... a flag signalling whether the record is packed.

For a pointer type:
PTRIDNR.... referenced type identifier.
REFERTYP... referenced type record.
This compiler treats all pointer type references as forward defined. A list of all unresolved pointer type records is kept until all type definitions are parsed. Then the appropriate type record pointers corresponding to PTRIDNR are assigned to REFERTYP.+

For of the undefined type class:
UNDFIDNR... the undefined type identifier (if known).
From a theoretical point of view all compiler routines taking type descriptors as arguments (e.g. the type checking facilities) are partial functions in the sense that a single argument of the undefined type forces all results to be of this type also.

In order to reduce the number of attributes attached to symbols of the grammar involving the parsing of expressions, a global stack is constructed to serve the following purpose: Each stack element contains a flag which is set to false unless the expression currently derived is a single variable. Immediatly after the parse of an expression is completed the top element of this stack may be saved for later inspection. Thus non-variable arguments are discovered in place of variable parameters. The compiler actually uses a doubly linked list in order to re-use popped off stack elements.

+ The author believes that this is a more natural resolution of the following ambiguity in PASCAL: Given type R = record F:...; G: @R end. Then G should refer to R itself rather than a different type R defined on a lower block level.

2.6. Strings of Fluid Length

This compiler utilizes a system for character strings of fluid lengths. By this we mean that whenever during manipulation a string would become longer than the space reserved for it, it is re-allocated and its old position released for garbage collection.+

The compiler builds the object code by a recursive process very similar to the representation rules of part one. Thus the final length of a string of lambda-calculus code can by no means be estimated beforehand. The string managment subsystem is considered to be the essential tool for successful code generation. It resides completely independently of the compilation routines, and it can be used whereever it is required to work with strings whose lengths cannot be determined prior to their actual use.

The contents of all strings in use are put into a common area of core -- for instance a very long string itself. Then every string may be referred to by a record containing its starting address in the string workspace, its current length, the space currently reserved for it, and a marker signalling whether it is in use or free to be re-used. The system provides procedures for allocating new strings, assigning literals and other strings to strings, concatenating strings and releasing occupied string space. All string description records are linked together in a list for garbage collection purposes. As soon as a string is to be allocated but no more workspace is available, three passes of garbage collection and storage compaction are attempted to recover space for this request: first the list of string descriptors is searched for a string large enough to satisfy the request. If this does not succeed, then as many unused strings as necessary are removed from the list and the workspace is properly compacted. Finally a necessary amount of strings may have their allocated lengths reduced to their current lengths.

2.7. Summary of Attributes

Any symbol of a translation grammar may have one or several attributes associated with it [6]. The following briefly describes the meaning of these attributes with respect to the grammatical symbols, and whether they are inherited or synthesized:

<identifierlist>(FIRSTID, CARIDLST):

+ The term "fluid" was chosen by the author. The concept should not be confused with strings of varying lengths in programming languages such as PL/1.

```
   FIRSTID..... identifier index (inher.).
   CARIDLST.... list of identifiers (synth.).

<nonidentconstrem>(SIGN, CONSTYP, CONSVAL):
   SIGN........ sign of non identifier constant (inher.).
   CONSTYP..... pointer to its type record (synth.).
   CONSVAL..... its explicit value (synth.).

<nonidentconstant>(CONSTYP, CONSVAL):
   Same as for <nonidentconstrem> (both synth.).

<constant>(CONSTYP, CONSVAL):
   Same as for <nonidentconstrem> (both synth.).

<constantlist>(FIRSTTYP, FIRSTVAL, CARCONSLST, MATCHTYP):
   FIRSTTYP.... pointer to type record (inher.).
   FIRSTVAL.... explicit value of constant (inher.).
   CARCONSLST.. list of explicit values of constants (synth.).
   MATCHTYP.... pointer to a (scalar) type record which must
                match to all constant types incl. FIRSTTYP
                (inher.).

<simpletyperemaind>(FIRSTID, RETTYP, REFID):
   FIRSTID..... identifier which begins <type> (inher.).
   RETTYP...... pointer to completed type record (synth.).
   REFID....... index of type identifier which is to be defined
                (if explicit type definition, otherwise zero);
                necessary for scalar types. (inher.).

<simpletype>(RETTYP, REFID):
   Same as for <simpletyperemaind>.

<simpletypelist>(FIRSTTYP, CARSMPLST, SWPACK)
   FIRSTTYP.... pointer to type record (inher.).
   CARSMPLST... list of (scalar) type records (synth.).
   SWPACK...... flag whether corresponding array is packed or
                not (inher.).

<variant>(RECLST, MATCHTYP):
   RECLST...... list of field identifier descriptors (synth.).
   MATCHTYP.... type record of preceeding tag field (inher.).

<variantlist>(FIRSTVAR, CARRECLST, MATCHTYP):
   FIRSTVAR.... list of field identifier descriptors (inher.).
   CARRECLST... list of field identifier descriptors (synth.).
   MATCHTYP.... type record of preceeding tag field (inher.).

<tagfieldremainder>(FIRSTID, REC, MATCHTYP):
   FIRSTID..... identifier which begins tag field (inher.).
   REC......... tag field identifier descriptor (if any other-
                wise nil) (synth.).
   MATCHTYP.... pointer to type record of tag field (synth.).
```

```
<fieldlistremaind>(RECLST):
   RECLST...... list of field  identifier  descriptors  (if  any
                otherwise nil) (synth.).

<recordsection>(RECLST):
   Same as for <fieldlistremaind>.

<fieldlist>(FLDLST):
   FLDLST...... list of field identifier descriptors (synth.).

<unpackstructtype>(RETTYP, SWPACK, REFID):
   RETTYP...... pointer to complete type record (synth.).
   SWPACK...... flag whether type is packed or not (inher.).
   REFID....... same as REFID in <simpletypremaind> (inher.).

<type>(RETTYP, REFID):
   Same as in <unpackstructype>.

<formalparameter>(RETARG, RETACT):
   RETARG...... type  record  pointer  list of parameters as re-
                quired in the ARGTYPS field of  a  procedure  or
                function identifier description. (synth.).
   RETACT...... list of parameter desriptions (synth.).

<formparameterlist>(FIRSTARG, FIRSTACT, CARARGLST, CARACTLST):
   FIRSTARG.... list of type record pointers (inher.).
   FIRSTACT.... list of identifier record pointers (inher.).
   CARARGLST... list of parameter types (synth.).
   CARACTLST... list of parameter descriptions (synth.).

<formparameterpart>(ARGLST, ACTLST):
   ARGLST...... list  of  all  parameter types (if any otherwise
                nil) (synth.).
   ACTLST...... list  of  all  parameter  descriptions  (if  any
                otherwise nil) (synth.).

<expressionlist>(FIRSTCOD, FIRSTTYP, FIRSTSWVAR, CAREXPLST):
   FIRSTCOD.... string pointer (see section 2.6) (inher.).
   FIRSTTYP.... type record pointer (inher.).
   FIRSTSWVAR.. flag  whether  expression  is a variable or not;
                necessary to determine if an argument is a vari-
                able (inher.).
   CAREXPLST... list of expression constituents which consist of
                their type records,  string  pointers  to  their
                code  and "variable flags" like FIRSTSWVAR above
                (synth.).

<variableselector>(SWASSIGN, REPLLST, CODIN, TYPIN,
                   CODOUT, TYPOUT):
   SWASSIGN.... flag signalling if called at left hand  side  of
                assignment (inher.).
   REPLLST..... list  of  subscripts  and  array  bounds in case
```

```
                SWASSIGN is true; needed to compile according to
                the last representation rule of section 1.7
                (inher. and synth.).
   CODIN....... string pointer to current code (inher.).
   TYPIN....... type record pointer of current type (inher.).
   CODOUT...... string pointer to new code (synth.).
   TYPOUT...... type record pointer of new type (synth.).

<actparameterpart>(ACTEXPLST):
   ACTEXPLST... same as CAREXPLST in <expressionlist>.

<identifierremaind>(ID, CODOUT, TYPOUT):
   ID.......... identifier in front of it (inher.).
   CODOUT, TYPOUT.. same as in <variableselectors>.

<factor>(FACCOD, FACTYP):
   FACCOD...... string pointer to code of factor (synth.).
   FACTYP...... type record pointer of its type (synth.).

<factorlist>(PRIORCOD, PRIORTYP, RETCOD, RETTYP):
   PRIORCOD.... string pointer (inher.).
   PRIORTYP.... type record pointer (inher.).
   RETCOD...... string pointer to complete code of factors
                (synth.).
   RETTYP...... type record pointer of their resulting type
                (synth.).

<term>(TERMCOD, TERMTYP):
   Similar to attributes of <factor>.

<termlist>(PRIORCOD, PRIORTYP, RETCOD, RETTYP):
   Similar to attributes of <factorlist>.

<simpleexpression>(SEXPCOD, SEXPTYP):
   Similar to attributes of <factor>.

<simpleexpressrem>(PRIORCOD, PRIORTYP, RETCOD, RETTYP):
   Similar to attributes of <factorlist>.

<expression>(EXPCOD, EXPTYP):
   EXPCOD...... string pointer to expression code (synth.).
   EXPTYP...... pointer to its type record (synth.).

<simplestatemntrem>(ID):
   ID.......... identifier in front of it (inher.).

<compoundstmntrem>(FIRSTSTMNR, COMPSTMCOD):
   FIRSTSTMNR.. statement number (inher.).
   COMPSTMCOD.. string pointer to code of compound statement
                (see also section 1.6) (inher. and synth.).
```

```
<elseclause>(ELSESTMNR):
   ELSESTMNR... statement number (inher.).

<caseelement>(CONSLST, MATCHTYP):
   CONSLST..... list of explicit values of case labels (synth.).
   MATCHTYP.... pointer to a (scalar) type record which must
                match all case label types (inher.).

<caseelementlist>(FIRSTCONSLST, CARCONSLST, MATCHTYP):
   FIRSTCONSLST.. list of expl. values of case labels (inher.).
   CARCONSLST.. list of expl. values of all case labels(synth.).
   MATCHTYP.... same as in <caseelement>.

<forstatementrem>(CONTRTYP):
   CONTRTYP.... type rec. pointer of control variable (inher.).
```

Each terminal (viz. each token) has only one (synthesized) attribute: its parameter value, if any was assigned.

2.8. Error Diagnostics and Error Recovery

The error diagnostic routines take advantage of the LL(1) property of the underlying grammar. Illegal tokens are discovered as soon as they are obtained [6] namely if they are not contained in the selection set of a non-terminal which is to be derived. A typical error message is then

"VARIABLESELECTOR STARTS WITH IDENTIFIER"

or if terminals do not match

"; EXPECTED,BUT : FOUND"

All other error messages are adjusted to standard PASCAL [4] though their text usually includes some specific information such as an incorrectly used identifier. The error messages are enumerated according to [4].

The error recovery is probably the most complicated process of this compiler. Some general guidelines shall be explained now. For more details the reader is referred to the compiler source (Appendix A). The lexical scanner treats illegal characters like blank spaces. On encountering a bad token the parser proceeds until it finds a synchronizing symbol (SEMICSYM, ENDSYM, ELSESYM, UNTILSYM). Then it ignores all symbols of the current derivation ("pops the stack") until continuation by the synchronizing token is possible. Some tokens should not be passed during the synchronization (PROCSYM, FUNCSYM, RECORDSYM, BEGINSYM, CASESYM, REPEATSYM), because the essential program structure would be lost. In such cases the compilation is halted. Semantic errors are repaired quite thoroughly. A separate undefined type was introduced for this purpose (see also UNDFIDNR in section 2.5). However, in some cases the type might be constructed from the context.

2.9. Example#5

The following sample program contains many errors which the compiler detected and reported:

```
 STMNR LEV NST SEMIC   SOURCE CODE:

    0    0   0    1    PROGRAM ERRONEOUS(INPUT);
    0    1   0    1    (* This program tests error diagnostics
    0    1   0    1       and error recovery *)
    0    1   0    2     LABEL 1, 2, 1;
    0    1   0    4     CONST C1='A'; C2='YZ';
    0    1   0    6     TYPE SR1=C1..'Z'; SR2=-5..0;
    0    1   0    7      PTR=@REC2;
    0    1   1    8      REC1=RECORD RF1, RF2: CHAR;
    0    1   1    8            CASE PTR OF
    0    1   1    9             'B': (RF3: @REC1);
    0    1   1    9             'Z': (RF1: INTEGER)
    0    1   1   10           END;
    0    1   0   12     VAR V1: BOOLEAN; V2: @REC1;
    0    1   0   13      V3: (TRUE, FALSE);
    0    1   0   13      V4: PACKED ARRAY (.SR1, (ONE, TWO).) OF
    0    1   1   14          RECORD VF1: ONE..TWO;
    0    1   2   16           VF2: RECORD VF3: @REC1; END;
    0    1   1   17          END;
    0    1   0   18      C2: INTEGER;
    0    1   0   20     PROCEDURE P(A: SR2); FORWARD;
    0    1   0   20     (*$S+ allow extented syntax *)
    0    2   0   21     FUNCTION F(VAR P1, P2: BOOLEAN): 1..100;
    0    1   0   22      FORWARD; (*$S- inhibit extensions *)
    0    2   0   23     PROCEDURE P(A: SR2);
    2    2   1   24      BEGIN NEW(V2, 'B');
    3    2   1   24       3: IF TRUE
    4    2   1   25           THEN V4(.C2, ONE.).VF2.VF5:=V2;
    5    2   1   26       P( F(NOT V1, V1) );
    6    2   1   27      END;
    8    1   1   28     BEGIN PUT;
    9    1   1   28      FOR V4:=3 DOWNTO -5 DO
   10    1   2   28       CASE V2@.RF1 OF
   11    1   2   29        'A', 'B': PP(-23);
   12    1   2   30        'C': GOTO 4;
   13    1   2   32        'D', 'A':;;
   14    2   2   32        'E': WITH V4(.'C', THREE.), VF2 DO
   15    3   2   33              VF3@.RF3@.RF1 := SR2;
   15    1   2   34       END;
   16    1   1   35    IF F(V1)<>5 AND C1='''' THEN I:=I+1;
   17    1   1   36    X:=5.E-7;
   18    1   1   36   END.
```

```
REF IDENTIFIER CLASS, TYPE, REFERENCES:    ***ERRONEOUS***

  1  $LABEL1    SCALAR, LABEL 0 .. 9999 (ORDERS ONLY) 2,
  1  $LABEL2    SCALAR, LABEL 0 .. 9999 (ORDERS ONLY)
 24  $LABEL3    SCALAR, *** UNDEFINED ***
 29  $LABEL4    SCALAR, *** UNDEFINED ***
 18  A          VARIABLE, INTEGER -5 .. 0 (ORDERS ONLY)
 10  BOOLEAN    TYPE, BOOLEAN 0 .. 1 (ORDERS ONLY) 20,
  7  CHAR       TYPE, CHAR 0 .. 255 (ORDERS ONLY)
  2  C1         SCALAR, CHAR 0 .. 255 (ORDERS ONLY) 4, 34,
  3  C2         SCALAR, PACKED ARRAY (. INTEGER 1 .. 2 (ORDERS ON
                LY) .) OF CHAR 0 .. 255 (ORDERS ONLY) 18, 24,
 20  F          ENTRY( VAR, BOOLEAN 0 .. 1 (ORDERS ONLY) ; VAR, B
                OOLEAN 0 .. 1 (ORDERS ONLY) ; ) : INTEGER 1 .. 10
                0 (ORDERS ONLY) *** UNRESOLVED FORWARD REFERENCE
                *** 25, 34,
 12  FALSE      SCALAR, $IMPLT1 0 .. 1 (ORDERS ONLY)
  0  INPUT      VARIABLE, @INTEGER
  9  INTEGER    TYPE, INTEGER -2147483647 .. 2147483647 (ORDERS O
                NLY) 17,
 13  ONE        SCALAR, $IMPLT2 0 .. 1 (ORDERS ONLY) 13, 24,
 27  OUTPUT     VARIABLE, *** UNDEFINED ***
 18  P          ENTRY( INTEGER -5 .. 0 (ORDERS ONLY) ; ) 23, 25,
 28  PP         ENTRY( INTEGER -2147483647 .. 2147483647 (ORDERS
                ONLY) ; )
  6  PTR        TYPE, @REC2 8,
  7  REC1       TYPE, RECORD RF2 :, RF1 :, RF3 :, RF1 :, 8, 11, 1
                4,
  6  REC2       TYPE, REC2 , *** UNDEFINED ***
  8  RF1        RECORD FIELD, CHAR 0 .. 255 (ORDERS ONLY) 28, 32,
  9  RF1        RECORD FIELD, INTEGER -2147483647 .. 2147483647 (
                ORDERS ONLY)
  8  RF2        RECORD FIELD, CHAR 0 .. 255 (ORDERS ONLY)
  8  RF3        RECORD FIELD, @REC1 32,
  4  SR1        TYPE, CHAR 193 .. 233 (ORDERS ONLY) 13,
  5  SR2        TYPE, INTEGER -5 .. 0 (ORDERS ONLY) 18, 22, 32,
 32  THREE      VARIABLE, *** UNDEFINED ***
 12  TRUE       SCALAR, $IMPLT1 0 .. 1 (ORDERS ONLY) 24,
 13  TWO        SCALAR, $IMPLT2 0 .. 1 (ORDERS ONLY) 13,
 32  VF1        RECORD FIELD, $IMPLT2 0 .. 1 (ORDERS ONLY)
 14  VF1        RECORD FIELD, $IMPLT2 0 .. 1 (ORDERS ONLY)
 32  VF2        RECORD FIELD, RECORD VF3 :, 32,
 16  VF2        RECORD FIELD, RECORD VF3 :, 24,
 32  VF3        RECORD FIELD, @REC1
 15  VF3        RECORD FIELD, @REC1
 10  V1         VARIABLE, BOOLEAN 0 .. 1 (ORDERS ONLY) 25, 25, 34
 11  V2         VARIABLE, @REC1 23, 24, 28,
 12  V3         VARIABLE, $IMPLT1 0 .. 1 (ORDERS ONLY)
 13  V4         VARIABLE, PACKED ARRAY (. CHAR 193 .. 233 (ORDERS
                 ONLY) .) OF PACKED ARRAY (. $IMPLT2 0 .. 1 (ORDE
                RS ONLY) .) OF RECORD VF1 :, VF2 :, 24, 28, 32,
 35  X          VARIABLE, *** UNDEFINED ***
```

```
ERRNR SEMIC COL ERROR MESSAGE LISTING:      ***ERRONEOUS***

 101      2  44  IDENTIFIER '$LABEL1 ' DECLARED TWICE
 398      8  13  VARIANTS WILL BE TREATED AS RECORDS
 110      8  44  TAGFIELD TYPE MUST BE SCALAR OR SUBRANGE
 101      9  11  TWO RECORDFIELDS 'RF1 '
 104     10   5  IDENTIFIER 'REC2 ' UNDECLARED
 101     18  44  IDENTIFIER 'C2 ' DECLARED TWICE
 119     23   8  FORW. DCL.:MUST NOT REPEAT ARGUMENT LIST
 398     24  44  TAGFIELDVALUES IN PROC. 'NEW' IGNORED
 104     24   5  IDENTIFIER '$LABEL3 ' UNDECLARED
 135     24  12  TYPE OF OPERAND MUST BE BOOLEAN
 134     24  26  TYPE CONFLICT:'SCALAR ' VERSUS 'ARRAY '
 152     24  36  RECORD FIELD 'VF5 ' NOT FOUND
 154     25  22  ACTUAL PARAMETER MUST BE A VARIABLE
 134     26  23  TYPE CONFLICT:'SCALAR ' VERSUS 'SCALAR '
 104     28  44  IDENTIFIER 'OUTPUT ' UNDECLARED
 143     28  11  ILLEGAL TYPE OF LOOP CONTROL VARIABLE
 104     28  18  IDENTIFIER 'PP ' UNDECLARED
 104     30  44  IDENTIFIER '$LABEL4 ' UNDECLARED
 104     32  31  IDENTIFIER 'THREE ' UNDECLARED
 103     33  44  IDENTIFIER 'SR2 ' OF WRONG CLASS
 156     34  44  MULTIDEFINED CASE LABEL
 126     34  12  ACTUAL NUMBER OF ARGUMENTS UNEQUALS DCL.
 134     34  21  TYPE OF OPERAND(S) MUST BE BOOLEAN
  25     34  21  THEN EXPECTED,BUT = FOUND
 104     35   5  IDENTIFIER 'X ' UNDECLARED
 201     35   7  ERROR IN REAL CONSTANT: DIGIT EXPECTED
 398     35   7  REAL NUMBERS ARE NOT IMPLEMENTED
  26     35   8  FACTORLIST STARTS WITH IDENTIFIER
 167     36  44  UNSPECIFIED LABEL '4 '
 167     36  44  UNSPECIFIED LABEL '2 '
 167     36  44  UNSPECIFIED LABEL '1 '
 117     36  44  UNSATISF. FORWARD REFERENCE 'F '
```

2.10. Implementation Notes

The compiler was written by the author alone in approximately one year starting in September 1978. It is a single PASCAL program using three external files: INPUT for the program to be compiled, OUTPUT for compiler listings and messages, and SPUNCH for the lambda-calculus code produced. Only the first 100 characters of an input line are analyzed, and the maximum number of characters on code lines is currently set to 72. But these numbers can be changed easily. The routines generating a cross-reference were added for debugging purposes only and are coded rather inefficiently.

The following identifiers are pre-defined as in standard PASCAL: BOOLEAN, CHAR, CHR, FALSE, GET, INTEGER, ORD, PRED, PUT, REAL, SUCC, TRUE. INPUT or OUTPUT are files of INTEGER when specified as program parameters.

During the development of the compiler, the parser was actually generated automatically using a program supplied by the the author himself. It employs recursive PASCAL procedures, one for each non-terminal, in place of a pushdown stack [6]. Synthesized and inherited attributes become variable and value parameters respectively. A global switch is set if the parsing process attempts to recover from an erroneous input token. In this case, the body of a procedure is skipped if its corresponding symbol is supposed to be popped of the stack. It should be noted that this is just a certain way of coding an LL(1) parser and must not be confused with recursive descent methods [8].

The compiler itself can be successfully compiled by the parser, including type checking procedures.

INDEX

REFERENCES

[1] Abdali, S.K. A Combinatory Logic Model of Programming Languages. Univ. Of Wisconsin: Ph.D. Dissertation, 1974.

[2] Abdali, S.K. "CLONE" - a Combinatory Logic Normal Form Evaluator. Rensselaer Polytechnic Institute: User Manual, 1978.

[3] Church, A. The Calculi of Lambda-Conversion. Princeton: Princeton Univ. Press, 1941.

[4] Jensen, K., and Wirth, N. PASCAL User Manual and Report. Lecture Notes in Comp. Sci. 18. Berlin-Heidelberg-New York: Springer Verlag, 1974.

[5] Lewis, P.M., and Rosenkrantz, D.J. "An ALGOL Compiler Using Automata Theory." General Electric: Repport #71-C-176, 1971.

[6] Lewis, P.M., Rosenkrantz, D.J., and Stearns, R.E. Compiler Design Theory. Reading, MA: Addison-Wesley Publishing Company, 1976.

[7] Morris, J.H. Lambda-Calculus Models of Programming Languages. Massachussetts Institute of Technology: Ph.D. Dissertation, 1968.

[8] Nori, K.V., Amman, U., Jensen, K., and Naegli, H.H. The PASCAL P Compiler: Implementation Notes. ETH Zuerich: Technical Report #10, 1974.

[9] Petznik, G.W. Introduction to Combibatory Logic. In: Brainerd, W.S., and Landweber, L.H. Theory of Computation. New York: John Wiley & Sons Inc., 1974.

[10] Wirth, N. "The Design of a PASCAL Compiler." Software Practice and Experience 1, 4, pp. 309-333, 1971.

APPENDICES

The compiler can be obtained on tape from the author. Following listings may be helpful to understand the working of the compiler:

A) Compiler source listing (file 3 on distribution tape), 103 print pages.

B) Grammar and PDA (file 4 on distribution tape), 33 print pages.