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** ................................................................. 2

**PART 1** ............................................................................ 3
1.1. Introduction ............................................................... 4
1.2. The Lambda - Calculus .................................................. 4
1.3. Systems of Lambda - Expression Definitions .................... 6
1.4. Primitives in the Model ............................................... 7
1.5. Functional Semantics of Variables and Statements .......... 9
1.6. Compound Statements and Blocks .................................. 9
1.7. Expressions and Assignments ....................................... 10
1.8. Files and Input-Output ............................................... 12
1.9. Example#1 ................................................................... 13
1.10. Conditional Statements ............................................. 15
1.11. Repetitive Statements ................................................ 15
1.12. Example#2 ................................................................... 16
1.13. Procedures ............................................................... 19
1.14. Example#3 ................................................................... 20
1.15. Remarks on Further Language Constructs .................... 21
1.15. Example#4 ................................................................... 22

**PART 2** ............................................................................ 27
2.1. Overall Features about the Compiler ............................ 28
2.2. The Lexical Scanner .................................................... 28
2.2.1. Identifiers ............................................................... 29
2.2.2. Numbers ............................................................... 29
2.3. An LL(1) Grammar for Standard PASCAL .................... 30
2.4. An Attributed Translation of Lists ............................... 39
2.5. The Main Data Structures of Translation ....................... 40
2.6. Strings of Fluid Length ............................................... 43
2.7. Summary of Attributes .............................................. 43
2.8. Error Diagnostics and Error Recovery ......................... 47
2.9. Example#5 ................................................................... 48
2.10. Implementation Notes ............................................... 50

**INDEX** ......................................................................... 52

**REFERENCES** ............................................................... 55

**APPENDICES** .............................................................. 56
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 Institute, and actually accomplishing more than in its original proposal, this compiler is part of a program verification project. Although 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.
| PPPPPPPPPPPP | AAAAAAAAAAAA | RRRRRRRRRRRR | TTTTTTTTTTTTT | 11 |
| PPPPPPPPPPPP | AAAAAAAAAAAA | RRRRRRRRRRRR | TTTTTTTTTTTTT | 111 |
| PP PP AA AA AA | AA  MM  MM | TT | 1111 |
| PP PP AA AA AA | AA  RR  RR | TT | 11 |
| PPPPPPPPPPPP | AAAAAAAAAAAA | RRRRRRRRRRRR | TT | 11 |
| PPPPPPPPPPPP | AAAAAAAAAAAA | RRRRRRRRRRRR | TT | 11 |
| PP AA AA AA | AA  RR  RR | TT | 11 |
| PP AA AA AA | AA  RR  RR | TT | 11 |
| PP AA AA AA | AA  RR  RR | TT | 11 |
| PP AA AA AA | AA  RR  RR | TT | 1111111111 |
| PP AA AA AA | AA  RR  RR | TT | 1111111111 |

---

The table shows a pattern with rows and columns of characters. The pattern repeats with a mix of capital and lowercase letters, along with some numbers. The sequence at the end indicates a count or identifier.
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) \[ \langle \text{indeterminate} \rangle ::= \langle \text{PASCAL-identifier including '$' in the letter set} \rangle \]

(2) \[ \langle \text{lambda-expression} \rangle ::= \langle \text{indeterminate} \rangle \]

(3) \[ \langle \text{lambda-expression} \rangle ::= \langle \text{application} \rangle \]

(4) \[ \langle \text{lambda-expression} \rangle ::= \langle \text{abstraction} \rangle \]

(5) \[ \langle \text{application} \rangle ::= (\langle \text{lambda-expression} \rangle \langle \text{lambda-expression} \rangle) \]

(6) \[ \langle \text{abstraction} \rangle ::= (\lambda \langle \text{binding indeterminates} \rangle : \langle \text{lambda-expression} \rangle) \]

(7) \[ \langle \text{binding indeterminates} \rangle ::= \langle \text{indeterminate} \rangle \]

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 occurrence is free in this lambda-expression. If \( e, f_1, f_2, \ldots, f_n \) are lambda-expressions and \( x_1, x_2, \ldots, x_n \) are pairwise distinct indeterminates\(^+\), then

\[
\text{sub}[f_1, x_1; f_2, x_2; \ldots; f_n, x_n; e]
\]

denotes the result of simultaneously substituting \( f_i \) for all

\(^+\) In the following, \( e, f, g, \ldots \) will denote lambda-expressions and \( x, y, \ldots \) indeterminates. Unless stated otherwise, they are always assumed universally quantified in definitions and theorems of the meta-language.
free occurrences of $x_i$ ($1 \leq i \leq n$) in $e$.

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

**Alpha-conversion (renaming bound variables):**

$$\lambda x : e \quad \overset{\text{(alpha)}}{\longrightarrow} \quad \lambda y : \text{sub[y,x; e]}$$ provided that $y$ has no free occurrence in $e$.

**Beta-contraction (substitution):**

$$\left( (\lambda x : e) f \right) \quad \overset{\text{(beta)}}{\longrightarrow} \quad \text{sub}[f,x; e]$$ if no free indeterminates in $f$ occur bound in $e$.

**Eta-contraction (extensionality):**

$$\lambda x : (e \ x) \quad \overset{\text{(eta)}}{\longrightarrow} \quad e$$ 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 "$\longrightarrow$"). Conversion ("$\longleftrightarrow$") 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, furthermore, $e \longrightarrow f$. 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 [7]: If $e \longleftrightarrow g$, then there is a conversion from $e$ into $g$ in which no expansion precedes any contraction.

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

**Lemma:**

$$\left( \ldots(\lambda x_1 : (\lambda x_2 : (\ldots(\lambda x_n : e)\ldots)f_1)\ldots f_n \right) \quad \longrightarrow \quad \text{sub}[f_1,x_1; f_2, x_2; \ldots; f_n, x_n; e]$$ provided no free indeterminate in any of the $f_i$'s is bound in $e$.

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

$$\langle \text{binding indeterminates} \rangle ::= \langle \text{binding indeterminates} \rangle, \langle \text{indeterminate} \rangle$$

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:

\((\lambda x_1, x_2, \ldots, x_n : e)\) means \((\lambda x_1 : (\lambda x_2 : (\ldots (\lambda x_n : e) \ldots)))\).

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

An "object"-program is then a \(<\text{list of definitions}>\). 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 preceding ''' character. This is the reason for including a '}' sign to the PASCAL letter set in the productions (1) and (13).

$ID = (\lambda X : X)$.

The identical ("do-nothing") function and empty list.

$CAT = (\lambda X, Y : (\lambda Z : ((X Z) Y)))$.

The concatenation of objects. If \((\lambda X : (((X y_1) y_2) \ldots) y_n)\) 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.
$\text{OMEGA} = (((\lambda X,Y: (X \ x))(\lambda X,Y: (X \ x))))$

The undefined value. It should be noted that ($\text{OMEGA} f$) $\rightarrow$ $\text{OMEGA}$ but $\text{OMEGA}$ does not possess a normal form.

$\text{Y} = (\lambda X:((\lambda Y: (X(Y \ Y)))(\lambda Y: (X(Y \ Y)))).$

The recursion operator. Since ($\text{Y} g$) $\rightarrow$ (g ($\text{Y} g$)), ($\text{Y} g$) is a solution to the recursive definition of $F = (g F)$, provided that g does not contain the name F.

$\text{Y}$ 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

$$(((g \langle \text{name}\rangle \langle \text{name}\rangle \ldots \langle \text{name}\rangle))$$

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

$$n_1 = (((g_1 \ n_1) \ n_2) \ldots) \ n_k),$$
$$n_2 = (((g_2 \ n_1) \ n_2) \ldots) \ n_k),$$
$$\quad \vdots$$
$$n_k = (((g_k \ n_1) \ n_2) \ldots) \ n_k),$$

is determined by

$$n_i = (((Y[i,k] g_1) \ g_2) \ldots) \ g_k), \text{ with}$$

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

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 $\text{Y}$ 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.

$\texttt{MINUS}\texttt{UNARY}$. Integer negation.
$\texttt{PLUS}$. Integer addition.
$\texttt{MINUS}$. Integer subtraction.
$\texttt{MULT}$. Integer multiplication.
$\texttt{DIV}$. Integer division.
$\texttt{TRUE}$. Boolean value true. It is assumed that (((TRUE g) h) $\rightarrow$ g).
$\texttt{FALSE}$. Boolean value false. It is assumed that (((FALSE g) h) $\rightarrow$ h).
$\texttt{NOT}$. Boolean negation.
$\texttt{AND}$. Boolean conjunction.
$\texttt{OR}$. Boolean disjunction.
$\texttt{EQ}$. Integer comparison equal.
$\texttt{N}$. Integer comparison not equal.
$\texttt{GT}$. Integer comparison greater.
$\texttt{GE}$. Integer comparison greater than or equal to.
$\texttt{LT}$. Integer comparison less.
$\texttt{LE}$. 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 $\texttt{SCAT}$). It is not an easy task to obtain the lambda-expressions of these primitives [1].

$\texttt{STUPINIT}$. Initialization of an array. Its reduction property is $\ldots((((\texttt{STUPINIT} n) m1)\ldots mn) \rightarrow $"list of m1 lists of m2 lists of ... mn $\Omega$MEGA$\ldots$".

$\texttt{RETRIEVE}$. Indexing of vector elements. Its reduction property is $(f ((\texttt{RETRIEVE} i) k)) \rightarrow "$i-th element of f if f is a list of k items (lambda-expressions)"

$\texttt{REPLACE}$. Assigning a vector element. Its reduction property is $(f((\texttt{REPLACE} i) k) g) \rightarrow "$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

\[ S/(v_1, v_2, \ldots, v_n) = \text{abstraction}. \]

where \( S \) is a statement and \((v_1, v_2, \ldots, v_n)\) 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 "$STM\\( i \)$", 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:

\[
\{\text{begin; } S_1; S_2; \ldots; S_n \text{ end}\}/E = \\
(\lambda \$PHI:(\{S_1\}/E (\{S_2\}/E (\ldots (\{S_n\}/E \$PHI) \ldots))))).
\]

It should be noticed that the program remainder of each \( S_i \)
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 $F$ be the global and $F=\{u_1, \ldots, u_m, v_1, \ldots, v_n\}$ be the local environment (identifier conflicts already resolved):

$$\{\text{var } u_1: \text{type1}; \ldots ; u_m: \text{typem}; \begin{array}{ll} \text{begin} & S_1; \ldots ; S_k \text{ end}\end{array}\} / F =$$

$$(\lambda \text{ PHI:} \ldots \cdot ((\{S_1\} / F \cdot \{S_2\} / F \ldots \cdot \{S_k\} / F \cdot (\lambda u_1, \ldots , u_m : \text{PHI}))$$

$$\ldots )\{\text{init } u_1\} \ldots \{\text{init } u_2\} \ldots \{\text{init } u_m\} ) ;$$

where $\{\text{init } u\}$ is $\Omega$ if $u$ is a scalar variable,

is $((\text{TYPINIT } 1) \ p)$ if $u$ is a vector of $p$ items,

is $((\text{TYPINIT } 2) \ p_1) \ p_2)$ if $u$ is a $p_1 \times p_2$ 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:

\[
<\text{expr. 1}> <\text{binary operator}> <\text{expr. 2}> = \\
(' \text{"primitive of binary oper."} <\text{expr. 1}>) <\text{expr. 2}>.
\]

\[
<\text{unary operator}> <\text{expression}> = \\
(' \text{"primitive of unary oper."} <\text{expression}>).
\]

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 array \([\text{lb}..\text{hb}]\), \(b\) an array [\text{BOOLEAN}] and \(d\) be an array \([\text{lb1}..\text{hb1}, \text{lb2}..\text{hb2}]\):

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

\[
b \ [\text{expression} ] = \\
(' (1 ' \text{'$\text{RETRIEVE}^{-}$'} (1 ' \text{'$\text{MINUS}<$expression>}) \text{2} (2 ' 2)) \text{2})).
\]

\[
d \ [\text{expr. 1}, \ \text{expr. 2}] = \\
(' (1 ' \text{'$\text{RETRIEVE}^{-}$'} (1 ' \text{'$\text{MINUS}<$expr. 1>}) \text{lb-1} (2 ' 2)) \text{hb1-lb1+1} ' (1 ' \text{'$\text{RETRIEVE}^{-}$'} (1 ' \text{'$\text{MINUS}<$expr. 2>}) \text{lb2-1} (2 ' 2)) \text{hb2-lb2+1}).
\]

In the following, \text{CHR} and \text{ORD} are the standard \text{PASCAL} functions on scalars:

\[
<\text{character}> = \text{CHR} \text{"order of this character"}.
\]

\[
\text{CHR}(<\text{expression}>) = \text{CHR} (<\text{expression}>).
\]

\[
\text{ORD}(<\text{expression}>) = <\text{expression}>.
\]

\[
<\text{expression}> - <\text{expression}>.
\]

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:

\[
v_1 := <\text{expression}> / E = \\
(' 1 ' \text{'$\text{PHI}'} \text{, v1} \ldots \text{, vn: } (\ldots ([[\ldots ([[\ldots [[<\text{expression}> / E \text{)} \text{ vi+1} \ldots \text{ vn}}).\]

\[
v [ <\text{expr. 1}> := <\text{expr. 2}> ] / E = \\
(' 1 ' \text{'$\text{PHI}'} \text{, v1} \ldots \text{, vn: } (\ldots ([[\ldots [[<\text{expression}> / E \text{)} \text{ vj+1} \ldots \text{ vn}}).\]
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: ((...((Φ v1) v2)... vn) OUTPUT) $SCARDS) $SPRINT))

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

These representations only require $SPRINT to be arranged in list format (see definition of $SCAT) whereas the input elements merely have to follow this output list. The representation of a complete program which the compiler names $PROGRAM follows:
(program...; Si.)/(() = (({Si}/() $ID)$ID);
where Si is the outermost begin-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 i1)... ip) —> (1 X:(X o1).... 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:
----- -------------------------------
 1. ($U+$,X- superscripts, no cross reference *)
PROGRAM EXAMPLE1(INPUT, OUTPUT);
CONST
  LB=2; HB=5; (* bounds for V *)
  LR1=-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! *)
BEGIN
  (* The following statements make no sense *)
  (* but illustrate the compilation *)
  GET; I:=INPUT0;
  3
  V[I+1]:=-(I+1);
  4
  I:=I+V[ (LB+HB) DIV 2 ] * I;
  5
  OUTPUT@:=V[4]; PUT;
  6
  B[FALSE]:=THREE;
  7
  S:=B[ NOT(I<>0) AND (V[I]<I+1) ];
  8
  BEGIN
  9
  D[-2, I*2]:= 'Q';
  10
  C:=D[HR1-2][V[3]]
  11
  END;
  12
  (* some difficult assignments *)
  P[C,S,I]:=(I<=2) OR (C='B');
  13
  B[P[C,S,I]]:=TWO
  14
END.
The compiler generated the following code. Optionally, matching parentheses are identified by superscripts. An asterisk in column one signals a comment line and this line should be ignored by automatic evaluators.

* LAMBDA CODE FOR EXAMPLE 1

$\text{ST}M\,2=\lambda \,\text{PHI}, P, D, B, V, S, C, I, \text{OUTPUT}, \text{INPUT}: (10 \lambda \,\text{PRINT}, \text{SCARDS}: (8(7(6(5(4(3(2(1(\text{PHI}\,P)D)B)B)V)S)C)I)\text{OUTPUT}?)\text{SCARDS}8)\lambda \,\text{PRINT}9)10)11).

$\text{ST}M\,3=\lambda \,\text{PHI}, P, D, B, V, S, C, I, \text{OUTPUT}, \text{INPUT}: (8(7(6(5(4(3(2(1(\text{PHI}\,P)D)B)B)V)S)C)I)\text{OUTPUT}?)\text{INPUT}8)9).

$\text{ST}M\,4=\lambda \,\text{PHI}, P, D, B, V, S, C, I, \text{OUTPUT}, \text{INPUT}: (13(12(11(10(9(8(7(6(5\,\text{REPLACE}(4(3\,\text{MINUS}(2\,\text{PLUS}(1\,I)B)B)V)S)C)I)\text{OUTPUT}?)\text{INPUT}8)9).

$\text{ST}M\,5=\lambda \,\text{PHI}, P, D, B, V, S, C, I, \text{OUTPUT}, \text{INPUT}: (13(12(11(10(9(8(7(6(5\,\text{MINUS}(4(3\,\text{DIV}(2\,\text{PLUS}(5\,1)B)B)V)S)C)I)\text{OUTPUT}?)\text{INPUT}8)9).

$\text{ST}M\,6=\lambda \,\text{PHI}, P, D, B, V, S, C, I, \text{OUTPUT}, \text{INPUT}: (8(7(6(5(4(3(2(1(\text{PHI}\,P)D)B)B)V)S)C)I)\text{OUTPUT}?)\text{INPUT}8)9).

$\text{ST}M\,7=\lambda \,\text{PHI}, P, D, B, V, S, C, I, \text{OUTPUT}, \text{INPUT}: (10 \lambda \,\text{PRINT}:(8(7(6(5(4(3(2(1(\text{PHI}\,P)D)B)B)V)S)C)I)\text{OUTPUT}?)\text{INPUT}8)9).

$\text{ST}M\,8=\lambda \,\text{PHI}, P, D, B, V, S, C, I, \text{OUTPUT}, \text{INPUT}: (12(11(10(9(8(7(6(5\,\text{REPLACE}(4(3\,\text{FALSE}(1\,I)B)B)V)S)C)I)\text{OUTPUT}?)\text{INPUT}8)9).

$\text{ST}M\,9=\lambda \,\text{PHI}, P, D, B, V, S, C, I, \text{OUTPUT}, \text{INPUT}: (17(16(15(14(13(12(11(10\,\text{REPLACE}(4(3\,\text{REPLACE}(2(1\,\text{PLUS}(4\,1)B)B)V)S)C)I)\text{OUTPUT}?)\text{INPUT}8)9).

$\text{ST}M\,10=\lambda \,\text{PHI}, P, D, B, V, S, C, I, \text{OUTPUT}, \text{INPUT}: (17(16(15(14(13(12(11\,\text{REPLACE}(4(3\,\text{REPLACE}(2(1\,\text{PLUS}(4\,1)B)B)V)S)C)I)\text{OUTPUT}?)\text{INPUT}8)9).
1.10. Conditional Statements

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

$$\text{if} <\text{expression}> \text{ then } S1 \text{ else } S2)/E = (\lambda \$\text{PHI}, v1, ..., vn: (...) (((\$\text{IF}-<\text{expression}>)/E) \{S1\}/E) \{S2\}/E) \$\text{PHI}) \text{ v1} \text{... vn}).$$

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

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 \textbf{repeat} \textit{S} \textbf{until} \textit{expression} is equivalent to begin \textit{S}; \textbf{while} \textit{expression} \textbf{do} \textit{S} \textbf{end}. While-statements themselves lead to
recursive definitions. Let $i$ be the statement number of the loop being represented:

$$\{\textbf{while } \langle \text{expression} \rangle \textbf{ do } S\} / E = \SSTM_i = (\lambda \$\Phi, v_1, \ldots, v_n:((\{^2(\lambda \$\Phi \langle \text{expression} \rangle / E \}) ((S)/E \ (\SSTM_i \ $\Phi \langle^3 \rangle)^2) \ $\Phi^1) \ v_1) \ldots \ v_n) \ldots$$

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 $\$T$.

1.12. Example #2

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

```plaintext
_PROGRAM SORT (INPUT, OUTPUT);
CONST LB = 1; HB = 9;
VAR A: ARRAY[L..HB] OF INTEGER;
    I, J, TEMP: INTEGER;
    NC: BOOLEAN;
BEGIN
    I := LB;
    WHILE (I <= HB) DO
        BEGIN
            GET;
            A[I] := INPUT;
        END;
    J := HB;
    NC := FALSE;
    WHILE (J > LB) AND NOT NC DO
        BEGIN
            I := LB;
            NC := TRUE;
            WHILE (I < J) DO
                BEGIN
                    THEN BEGIN
                        TEMP := A[I];
                        A[I] := A[I + 1];
                        A[I + 1] := TEMP;
                        NC := FALSE
                        END;
                    I := I + 1
                END;
            J := J - 1
        END;
    I := LB;
```
24       WHILE (I <= HR) DO
24       BEGIN
26       OUTPUT := A[I];
26       PUT
27       END

* LAMBDA CODE FOR SORT

$STM2=(\lambda PHI, NC, TEMP, J, I, A, OUTPUT, INPUT: (6(5(*3(2(1(0PHI NC 0
TEMP1)J 2)4 3)A *))OUTPUT 5)INPUT 6)7).

$STM5=(\lambda PHI, NC, TEMP, J, I, A, OUTPUT, INPUT: (8 \lambda SPRINT, $SCARDS: (7
(6(5(*3(2(1(0PHI NC 0)TEMP1)J 2)I 3)A *))OUTPUT 5)SWORDS 6)$SPRINT 7)8).

$STM6=(\lambda PHI, NC, TEMP, J, I, A, OUTPUT, INPUT: (6(5(*3(2(1(0PHI NC 0
TEMP1)J 2)I 3)A *))OUTPUT 5)INPUT 6)8).

$STM4=(\lambda PHI: (1STM5(0STM6 PHI 0)1)2).

$STM3=(\lambda PHI, NC, TEMP, J, I, A, OUTPUT, INPUT: (11(10(7(6(*3(2(1
$SPH I 0)NC 0)TEMP1)J 7)I 8)A 9))OUTPUT 10)INPUT 11)12).

$STM7=(\lambda PHI, NC, TEMP, J, I, A, OUTPUT, INPUT: (6(5(*3(2(1(0PHI NC 0
TEMP1)9 2)I 3)A *))OUTPUT 5)INPUT 6)7).

$STM8=(\lambda PHI, NC, TEMP, J, I, A, OUTPUT, INPUT: (6(5(*3(2(1(0PHI NC 0
TEMP1)J 2)I 3)A *))OUTPUT 5)INPUT 6)7).

$STM11=(\lambda PHI, NC, TEMP, J, I, A, OUTPUT, INPUT: (6(5(*3(2(1(0PHI NC 0
TEMP1)J 3)4)A *))OUTPUT 5)INPUT 6)7).

$STM12=(\lambda PHI, NC, TEMP, J, I, A, OUTPUT, INPUT: (6(5(*3(2(1(0PHI NC 0
TEMP1)J 2)I 3)A *))OUTPUT 5)INPUT 6)7).

$STM17=(\lambda PHI, NC, TEMP, J, I, A, OUTPUT, INPUT: (10(9(7(6(5(*3
($RETREIVE (1(0MINUS 1 0)3 2)6 3)*5)J 6)I 7)A 8))OUTPUT 9)INPUT 10)11).

$STM18=(\lambda PHI, NC, TEMP, J, I, A, OUTPUT, INPUT: (12(10(7(6(*3(2(1
($RETREIVE (3(2$MINUS (1(0PLUS 1 0)1 2)3 4)6 5)*7)8))OUTPUT 10)
INPUT 11)12).
$STM1 = \lambda \ PHI:\((((((\#STM2(\#STM3(\#STM7(\#STM8(\#STM9(\#STM2
3(\#NC, TEMP, J, I, A, OUTPUT, INPUT: \#PHI^0))^J)^3))^3))^3))^3))\#OMEGA)\#OMEGA)\#OMEGA)\#OMEGA)(\#TUPINIT 1^0)^J)\#OMEGA)\#OMEGA)^0).

$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:

\[\{V\text{-procedure} p(a1: \text{type1}; \ldots; ak: \text{typek}); <\text{block}>\}/E = \]
\[p = (\lambda \text{VAL}a1, \ldots, \text{VAL}ak: \{<\text{block}>\}/E)\] .

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

\[\text{init ai} \] is $\text{VAL}a1$ for all value parameters a1.

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:

\[\{ p (\text{expr.1}, \ldots, \text{expr.k}) \}/E = \]
\[\lambda \text{PHI}, v1, \ldots, vn: (((\ldots(\{p (\text{expr.1})/E\})\ldots \{\text{expr.k}]/E) \#PHI) v1)\ldots \text{vn})\] .

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 $G(s, u1, \ldots, un, v1, \ldots, vn)$ be the calling and $E(v1, \ldots, vn)$ the procedure environments.
\{ p (\langle expr \cdot 1 \rangle, \ldots, \langle expr \cdot k \rangle) \}/G =
(\lambda \$PHI, u_1, \ldots, u_m, v_1, \ldots, v_n:((\ldots((p \langle expr \cdot 1 \rangle)/E)\ldots
\langle expr \cdot k \rangle)/E)(\ldots((\$PHI u_1) u_2)\ldots u_m)) v_1)\ldots v_n)).

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 l, j: INTEGER;
  PROCEDURE P1(k, l: INTEGER);
  VAR m: INTEGER;
  BEGIN
    IF k < l
      THEN begin
        m := k; k := l; l := m MOD l;
        P1(k, l)
      END
    ELSE OUTPUT := k
    END; (* P1 *)
  PROCEDURE GCD(i, j: INTEGER);
  BEGIN P1(i, j); END; (* GCD *)
  BEGIN
    i := 28; j := 7;
    GCD(i, j+14);
    PUT
    END.

* LAMBDA CODE FOR EXAMPLE 3
P1=\(\theta \lambda \$VAL\$L, \$VAL\$K: \$ST M1\)\).

\$ST M4=\(\theta \lambda \$PHI, M, K, L, J, I, OUTPUT: (5*(5(5(5M0K4)L1L2J3I4))OUTPUT5)6)\).

\$ST M5=\(\theta \lambda \$PHI, M, K, L, J, I, OUTPUT: (5*(5(5(5M0PHI4)L1L2J3I4))OUTPUT5)6)\).

\$ST M6=\(\theta \lambda \$PHI, M, K, L, J, I, OUTPUT: (5*(5(5(5M0PHI4)L1L2J3I4))OUTPUT5)6)\).
$\text{STM7} = (\lambda \ $PHI, M, K, L, J, I, \text{OUTPUT}: (6(3((3((\text{P}1 \text{ K} 0)\text{L} 1)(2(1(0\text{PHI} M 0)K 1)L 2)3)J 0)I 0)\text{OUTPUT} 6))2).
$
$\text{STM3} = (\lambda \ $PHI: (3$\text{STM}$4(2$\text{STM}$5(1$\text{STM}$6(0$\text{STM}$7 $\text{PHI} 0)1)2)3)4).
$
$\text{STM8} = (\lambda \ $PHI, M, K, L, J, I, \text{OUTPUT}: (3((3((1(0\text{PHI} M 0)K 1)L 2)J 3)I 3)K 3)4).
$
$\text{STM2} = (\lambda \ $PHI, M, K, L, J, I, \text{OUTPUT}: (12(18(8(6(4(2$\text{IF}$ (1(0$\text{NE}$ 0 0)L 1)2)$\text{STM}$3)3)$\text{STM}$8)5)$\text{PHI}$ 5)6)K 7)L 8)I 9)I 10)\text{OUTPUT} 11)12).
$
$\text{STM1} = (\lambda \ $PHI: (3(3(1(0\text{PHI} M, K, L, $\text{PHI} 0)1)$\text{OMEGA}$)3$\text{VAL}$K)3$\text{VAL}$L)2).
$
$\text{GCD} = (\lambda \ $VAL$J $2, $VAL$I $2: $\text{STM}$9).
$
$\text{STM}10 = (\lambda \ $PHI, I$2, J$2, J$1, I$1, \text{OUTPUT}: (5(3(1(0\text{PHI} I 2 0)J 2 1)1(0\text{PHI} I 2 0)J 2 1)2)J 2 1)I 2 1)$\text{OUTPUT} 5).
$
$\text{STM11} = $\text{ID}$.
$
$\text{STM}9 = (3(3(2$\text{STM}$10(I$2, J$2: $\text{PHI} 0)1)2)$\text{VAL}$I $2)2)$.
$
$\text{STM13} = (\lambda \ $PHI, J, I, \text{OUTPUT}: (2((0\text{PHI} J 0)28 1)\text{OUTPUT} 2).
$
$\text{STM14} = (\lambda \ $PHI, J, I, \text{OUTPUT}: (2((0\text{PHI} J 0)28 1)\text{OUTPUT} 2).
$
$\text{STM15} = (\lambda \ $PHI, J, I, \text{OUTPUT}: (6(2(3(2(0$\text{GCD}$ I 0)(1(0$\text{PLUS}$ J 0)14 1)2)$\text{PHI}$ 3)I 4)I 4)\text{OUTPUT} 6).
$
$\text{STM15} = (5(3(4$\text{PRINT}$: (3(2(1(0\text{PHI} J 0)I 1)$\text{OUTPUT}$ 2)(($\text{CAT}$ $\text{PRINT}$)\text{OUTPUT}) 2)$)5).
$
$\text{STM12} = (5(3(4$\text{PRINT}$: (3$\text{STM}$13(2$\text{STM}$14(1$\text{STM}$16(0$\lambda$ J, I, \text{OUTPUT} $\text{PHI} 0)1)2)3)$\text{OMEGA}$)$\text{OMEGA}$)$\text{OMEGA}$)5).
$
$\text{PROGRAM} = (($\text{STM}$12 $\text{ID}$)$\text{ID}$).

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 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 representation 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:

<table>
<thead>
<tr>
<th>Stmnr</th>
<th>Source code:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(*$U+, X- superscripts, no cross reference *)</td>
</tr>
<tr>
<td></td>
<td>PROGRAM MATRIXMULT(INPUT, OUTPUT);</td>
</tr>
<tr>
<td></td>
<td>CONST LB=5; HB=10;</td>
</tr>
<tr>
<td></td>
<td>TYPE RANGE=LB..HB;</td>
</tr>
<tr>
<td></td>
<td>MATRIX=ARRAY[RANGE, RANGE] OF INTEGER;</td>
</tr>
<tr>
<td></td>
<td>VAR A, B, C: MATRIX;</td>
</tr>
<tr>
<td></td>
<td>I, J, K: INTEGER;</td>
</tr>
<tr>
<td></td>
<td>PROCEDURE READWRITE(SWITCH: BOOLEAN; C: MATRIX);</td>
</tr>
<tr>
<td></td>
<td>(* Reads in A and B (global) or prints C *)</td>
</tr>
<tr>
<td></td>
<td>(* according to the logical SWITCH *)</td>
</tr>
<tr>
<td></td>
<td>VAR I, J: INTEGER;</td>
</tr>
<tr>
<td>2</td>
<td>BEGIN</td>
</tr>
<tr>
<td>3</td>
<td>I:=LB;</td>
</tr>
<tr>
<td>3</td>
<td>WHILE I&lt;=HB DO</td>
</tr>
<tr>
<td>5</td>
<td>BEGIN</td>
</tr>
<tr>
<td>6</td>
<td>J:=LB;</td>
</tr>
<tr>
<td>6</td>
<td>WHILE J&lt;=HB DO</td>
</tr>
<tr>
<td>8</td>
<td>BEGIN</td>
</tr>
<tr>
<td>8</td>
<td>IF SWITCH</td>
</tr>
<tr>
<td>11</td>
<td>THEN BEGIN</td>
</tr>
<tr>
<td>13</td>
<td>GET; A[I, J]:=INPUT;</td>
</tr>
<tr>
<td>13</td>
<td>GET; B[I, J]:=INPUT</td>
</tr>
<tr>
<td>13</td>
<td>END</td>
</tr>
<tr>
<td>13</td>
<td>ELSE BEGIN</td>
</tr>
<tr>
<td>15</td>
<td>OUTPUT@:=C[I][J]; PUT</td>
</tr>
<tr>
<td>16</td>
<td>END;</td>
</tr>
<tr>
<td>17</td>
<td>J:=J+1</td>
</tr>
<tr>
<td>17</td>
<td>END;</td>
</tr>
<tr>
<td>18</td>
<td>I:=I+1</td>
</tr>
<tr>
<td>18</td>
<td>END</td>
</tr>
<tr>
<td>18</td>
<td>END; (<em>READWRITE</em>)</td>
</tr>
</tbody>
</table>
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
K := K + 1
END;
J := J + 1
END;
I := I + 1
END;
READWRITE(FALSE, C)
END.

* LAMBDA CODE FOR MATRIXMULT
READWRITE := (\$a \$val$switch, \$val$c $2: $stm10).

$stm2 := (12 \$phi, $j$2, $i$2, c$2, $switch, k, j$1, i$1, c$1, b, a, output, input:
(11(10(9(8(7(6(5(4(3(2(1($phi $j$2)5)c$2)swtich)k)j$1)i$1)1)c$1)7)b)8)9)output10)$cards11)$print12).

$stm5 := (12 \$phi, $j$2, $i$2, c$2, $switch, k, j$1, i$1, c$1, b, a, output, input:
(11(10(9(8(7(6(5(4(3(2(1($phi $j$2)5)i$2)c$2)swtich)k)j$1)i$1)1)c$1)7)b)8)9)output10)$cards11)$print12).

$stm10 := (13 \$print, $cards:(12(11(10(9(8(7(6(5(4(3(2(1($phi $j$2)5)i$2)c$2)swtich)k)j$1)i$1)1)c$1)7)b)8)9)output10)$cards11)$print12).

$stm11 := (12 \$phi, $j$2, $i$2, c$2, $switch, k, j$1, i$1, c$1, b, a, output, input:
(11(10(9(8(7(6(5(4(3(2(1($phi $j$2)5)i$2)c$2)swtich)k)j$1)i$1)1)c$1)7)b)8)9)output10)$cards11)$print12).

$stm12 := (13 \$print, $cards:(12(11(10(9(8(7(6(5(4(3(2(1($phi $j$2)5)i$2)c$2)swtich)k)j$1)i$1)1)c$1)7)b)8)9)output10)$cards11)$print12).
$STM_{13} = {^{1 \lambda} \phi} J, I$, $I$, $C, I$, $A$, $B$, $A$, $C, I$, $A$, $I$, $O$, $T, U, I, N, P$

$STM_{0} = {^{1 \lambda} \phi} J, I$, $I$, $C, I$, $A$, $B$, $A$, $C, I$, $A$, $I$, $O$, $T, U, I, N, P$

$STM_{15} = {^{1 \lambda} \phi} J, I$, $I$, $C, I$, $A$, $B$, $A$, $C, I$, $A$, $I$, $O$, $T, U, I, N, P$

$STM_{14} = {^{2 \lambda} \phi} J, I$, $I$, $C, I$, $A$, $B$, $A$, $C, I$, $A$, $I$, $O$, $T, U, I, N, P$

$STM_{16} = {^{1 \lambda} \phi} J, I$, $I$, $C, I$, $A$, $B$, $A$, $C, I$, $A$, $I$, $O$, $T, U, I, N, P$

$STM_{17} = {^{1 \lambda} \phi} J, I$, $I$, $C, I$, $A$, $B$, $A$, $C, I$, $A$, $I$, $O$, $T, U, I, N, P$

$STM_{18} = {^{1 \lambda} \phi} J, I$, $I$, $C, I$, $A$, $B$, $A$, $C, I$, $A$, $I$, $O$, $T, U, I, N, P$

$STM_{19} = {^{1 \lambda} \phi} J, I$, $I$, $C, I$, $A$, $B$, $A$, $C, I$, $A$, $I$, $O$, $T, U, I, N, P$

$STM_{20} = {^{1 \lambda} \phi} J, I$, $I$, $C, I$, $A$, $B$, $A$, $C, I$, $A$, $I$, $O$, $T, U, I, N, P$

$STM_{21} = {^{1 \lambda} \phi} J, I$, $I$, $C, I$, $A$, $B$, $A$, $C, I$, $A$, $I$, $O$, $T, U, I, N, P$

$STM_{22} = {^{1 \lambda} \phi} J, I$, $I$, $C, I$, $A$, $B$, $A$, $C, I$, $A$, $I$, $O$, $T, U, I, N, P$

$STM_{23} = {^{1 \lambda} \phi} J, I$, $I$, $C, I$, $A$, $B$, $A$, $C, I$, $A$, $I$, $O$, $T, U, I, N, P$

$STM_{24} = {^{1 \lambda} \phi} J, I$, $I$, $C, I$, $A$, $B$, $A$, $C, I$, $A$, $I$, $O$, $T, U, I, N, P$

$STM_{25} = {^{1 \lambda} \phi} J, I$, $I$, $C, I$, $A$, $B$, $A$, $C, I$, $A$, $I$, $O$, $T, U, I, N, P$

$STM_{26} = {^{1 \lambda} \phi} J, I$, $I$, $C, I$, $A$, $B$, $A$, $C, I$, $A$, $I$, $O$, $T, U, I, N, P$

$STM_{27} = {^{1 \lambda} \phi} J, I$, $I$, $C, I$, $A$, $B$, $A$, $C, I$, $A$, $I$, $O$, $T, U, I, N, P$

$STM_{28} = {^{1 \lambda} \phi} J, I$, $I$, $C, I$, $A$, $B$, $A$, $C, I$, $A$, $I$, $O$, $T, U, I, N, P$

$STM_{29} = {^{1 \lambda} \phi} J, I$, $I$, $C, I$, $A$, $B$, $A$, $C, I$, $A$, $I$, $O$, $T, U, I, N, P$

$STM_{30} = {^{1 \lambda} \phi} J, I$, $I$, $C, I$, $A$, $B$, $A$, $C, I$, $A$, $I$, $O$, $T, U, I, N, P$

$STM_{31} = {^{1 \lambda} \phi} J, I$, $I$, $C, I$, $A$, $B$, $A$, $C, I$, $A$, $I$, $O$, $T, U, I, N, P$
$\text{STM23} = (\lambda \phi: (\text{STM24 (STM25 (STM34 \phi 0)^1)^1}^2)^2).

$\text{STM22} = (\lambda \phi, k, j, i, c, b, a, \text{output, input}: (\text{STM23 (STM22 \phi 0)^1)^1}^3 \phi k 5 j 6 i 7 c 8 e 9 a 10 \text{output} 11 \text{input} 12)^1^1).

$\text{STM35} = (\text{readwrite false}^1 c 1).

$\text{STM19} = (\lambda \phi: ((((((\text{STM20 (STM21 (STM22 (STM35 \lambda k, j, i, c, b, a, \text{output, input: \phi 0)^1}^2)^3 \omega \omega \omega \omega)^1)^1)^1)^1)^1)^1)^1^2).

$\text{program} = ((\text{STM19 id id})\text{id})$. 


| PPPPPPPPPPP | AAAA | RRRRRRRRRRR | TTTTTTTTTTT | 22222222222 |
| PPPPPPPPPPP | AAAA | RRRRRRRRRRR | TTTTTTTTTTT | 22222222222 |
| PP    PP | AA  AA | KK  KK | TT   | 22 |
| PP    PP | AA  AA | RR  RR | TT   | 22 |
| PP    PP | AA  AA | RR  RR | TT   | 22 |
| PPPPPPPPPPP | AAAA | RRRRRRRRRRR | TT   | 22 |
| PPPPPPPPPPP | AAAA | RRRRRRRRRRR | TT   | 22 |
| PP    PP | AA  AA | RR  RR | TT   | 22 |
| PP    PP | AA  AA | RR  RR | TT   | 22 |
| PP    PP | AA  AA | RR  RR | TT   | 22 |
| PP    PP | AA  AA | RR  RR | TT   | 22 |
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 Men's Mason 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 corresponding 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 occurrences 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]: $X$, $S$, and $U$. $X$ suppress 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 parentheses 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 2.16. 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:

<table>
<thead>
<tr>
<th>STATE</th>
<th>INPUT CHARACTER</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>'0'..'9'</td>
<td>1a</td>
<td>2d</td>
<td>4c</td>
<td>0b</td>
<td>0b</td>
<td>0b</td>
</tr>
<tr>
<td>2</td>
<td>'.'</td>
<td>3c</td>
<td>0e</td>
<td>0e</td>
<td>0e</td>
<td>0e</td>
<td>0e</td>
</tr>
<tr>
<td>3</td>
<td>'E'</td>
<td>3d</td>
<td>0d</td>
<td>4d</td>
<td>0d</td>
<td>0d</td>
<td>0d</td>
</tr>
<tr>
<td>4</td>
<td>'+' , '-'</td>
<td>6d</td>
<td>0f</td>
<td>0f</td>
<td>5d</td>
<td>0f</td>
<td>0f</td>
</tr>
<tr>
<td>5</td>
<td>others</td>
<td>6d</td>
<td>0d</td>
<td>0d</td>
<td>0d</td>
<td>0d</td>
<td>0d</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Token</th>
<th>Meaning of parameter values</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOTSYM</td>
<td>None.</td>
</tr>
<tr>
<td>PLUSMINUS</td>
<td>1: '+', 2: '-'</td>
</tr>
<tr>
<td>ORSYM</td>
<td>None.</td>
</tr>
<tr>
<td>MULTOOPER</td>
<td>1: '*', 2: '/', 3: DIV, 4: MOD, 5: AND.</td>
</tr>
<tr>
<td>EQUALSYM</td>
<td>None.</td>
</tr>
</tbody>
</table>

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

Parameter value = starting index \* MAXSTRCL + length.

The remaining tokens correspond to special symbols without
parameter values:

<table>
<thead>
<tr>
<th>Token</th>
<th>Meaning of parameter values</th>
</tr>
</thead>
<tbody>
<tr>
<td>LPARASYM</td>
<td>'(', RPARASYM: ')', LBRACKSYM: '['., RBRACKSYM: ']'</td>
</tr>
<tr>
<td>SEMICSYM</td>
<td>';', COMMASYM: ',', PERIODSYM: '.', DOUBLEDOT: '..',</td>
</tr>
<tr>
<td>COLONSYM</td>
<td>'!:', BECOMES: ':=', POINTER: PASCAL pointer symbol</td>
</tr>
</tbody>
</table>

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

2.3. An LL(1) Grammar for Standard PASCAL

Before proceeding with a compendious description of the at-
tributed 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-termininnal 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

\[ \text{if } e_1 \text{ then if } e_2 \text{ then } S_1 \text{ else } S_2. \]

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) [5].

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>.

\[ \text{+ It is not known to the author whether there exists a "pure" LL(1) grammar for standard PASCAL. E.g. ALGOL 60 is known to be \textquotedblleft\text{inherently non-LL(1)}\textquotedblright.} [5]. \]
(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) = {OLONSYM}.
(18) <constdefinpartrem> ::= IDENTIFIER EQUALSYM <constant>
   SEMICSYM <constdefinpartrem>.
(19) <constdefinpartrem> ::= <empty>.
   Selset(19) = {TYPESYM, VARSYM, PROCSYM, FUNCSYM, BEGINSYM}.
(20) <constdefinpart> ::= CONSTSYM IDENTIFIER EQUALSYM <constant>
   SEMICSYM <constdefinpartrem>.
(21) <constdefinpart> ::= <empty>.
   Selset(21) = {TYPESYM, VARSYM, PROCSYM, FUNCSYM, BEGINSYM}.
(22) <simpletyperemaind> ::= DOUBLEDOT <constant>.
(23) <simpletyperemaind> ::= <empty>.
   Selset(23) = {RPARASYM, SEMICSYM, COMMASYM, RBRACKSYM, ENDSYM}.
(24) <simpletypte> ::= LPARASYM IDENTIFIER <identifierlist>
   RPARASYM.
(25) <simpletype> ::= IDENTIFIER <simpletyperemaind>.
(26) <simpletype> ::= <nonidentconstant> DOUBLEDOT <constant>.
   Selset(26) = {UNSGINTEG, PLUSMINUS, UNSGREAL, STRINGSYM}.
(27) <simpletypleist> ::= COMMASYM <simpletype> <simpletype-
   list>.
(28) <simpletypleist> ::= <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) <fieldlistremainder> ::= SEMICSYM <fieldlist>.

(36) <fieldlistremainder> ::= <empty>.
Selset(36) = {RPARASYM, ENDSYM}.

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

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

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

(40) <fieldlist> ::= CASESYM IDENTIFIER <tagfieldremainder> OFSYM
            <variant> <variantlist>.

(41) <unpackstructtype> ::= ARARRAYSYM 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) \texttt{<type>} ::= \texttt{<unpackstructtype>}.  
\hspace*{1em}Selset(47) = \{\texttt{ARRAYSYM, RECORDSYM, FILESYM, SETSYM}\}.

(48) \texttt{<type>} ::= \texttt{POINTER IDENTIFIER}.

(49) \texttt{<typedefinpartrem>} ::= \texttt{IDENTIFIER EQUALSYM <type> SEMICSYM <typedefinpartrem>}.

(50) \texttt{<typedefinpartrem>} ::= \texttt{<empty>}.  
\hspace*{1em}Selset(50) = \{\texttt{VARSYM, PROCSYM, FUNCSYM, BEGINSYM}\}.

(51) \texttt{<typedefinitionpart> ::= TYPESYM IDENTIFIER EQUALSYM <type> SEMICSYM <typedefinpartrem>}.

(52) \texttt{<typedefinitionpart> ::= <empty>}.  
\hspace*{1em}Selset(52) = \{\texttt{VARSYM, PROCSYM, FUNCSYM, BEGINSYM}\}.

(53) \texttt{<variabledeclprtem>} ::= \texttt{IDENTIFIER <identifierlist} COLONSYM <type> SEMICSYM <variabledeclprtem>.

(54) \texttt{<variabledeclprtem>} ::= \texttt{<empty>}.  
\hspace*{1em}Selset(54) = \{\texttt{PROCSYM, FUNCSYM, BEGINSYM}\}.

(55) \texttt{<variabledeclprtem>} ::= \texttt{VARSYM IDENTIFIER <identifierlist} COLONSYM <type> SEMICSYM <variabledeclprtem>.

(56) \texttt{<variabledeclprtem>} ::= \texttt{<empty>}.  
\hspace*{1em}Selset(56) = \{\texttt{PROCSYM, FUNCSYM, BEGINSYM}\}.

(57) \texttt{<formalparameter>} ::= \texttt{IDENTIFIER <identifierlist} COLONSYM IDENTIFIER.

(58) \texttt{<formalparameter>} ::= \texttt{VARSYM IDENTIFIER <identifierlist} COLONSYM IDENTIFIER.

\hspace*{1em}If the compiler option S+ is activated, an explicit \texttt{<type>}  
\hspace*{1em}will be accepted in formal parameters (productions 57 and 58).  
\hspace*{1em}Any implicitly defined scalar identifiers within this \texttt{<type>}  
\hspace*{1em}are then global to the scope of the procedure or function body.  
\hspace*{1em}No pointer references are forwarded.

(59) \texttt{<formalparameter>} ::= \texttt{FUNCSYM IDENTIFIER <identifierlist} COLONSYM IDENTIFIER.

(60) \texttt{<formalparameter>} ::= \texttt{PROCSYM IDENTIFIER <identifierlist>}.

(61) \texttt{<formparameterlist>} ::= SEMICSYM \texttt{<formalparameter> <formparameterlist>}.  

(62) \texttt{<formparameterlist> ::= <empty>.
Selset(62) = [RPARASYM].}

(63) \texttt{<formparameterpart> ::= LPARASYM <formalparameter> <formparameterlist> RPARASYM.}

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

(65) \texttt{<procfuncdeclarat> ::= PROCSYM IDENTIFIER <formparameterpart> SEMICSYM <block>.}

(66) \texttt{<procfuncdeclarat> ::= FUNCSYM IDENTIFIER <formparameterpart> COLONSYM IDENTIFIER SEMICSYM <block>.}

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

(67) \texttt{<procfuncdeclpart> ::= <procfuncdeclarat> SEMICSYM <procfuncdeclpart>.
Selset(67) = \{PROCSYM, FUNCSYM\}.}

(68) \texttt{<procfuncdeclpart> ::= <empty>.
Selset(68) = \{BEGINSYM\}.}

(69) \texttt{<expressionlist> ::= COMMASYM <expression>.
}

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

(71) \texttt{<variablesselector> ::= LBRACKSYM <expression> <expressionlist> RBRACKSYM <variablesselector>.
}

(72) \texttt{<variablesselector> ::= PERIODSYM IDENTIFIER <variablesselector>.
}

(73) \texttt{<variablesselector> ::= POINTER <variablesselector>.
}

(74) \texttt{<variablesselector> ::= <empty>.
Selset(74) = \{RPARASYM, SEMICSYM, COMMASYM, EQUALSYM, RBRACKSYM, DOUBLEDOT, BECOMES, DOSYM, OFSYM, ORSYM, TOSYM, ENDSYM, ELSESYM, THENSYM, INTTILSYM, DOWNTOSYM, PLUSMINUS, RELOPER, MULTOPER\}.}
(75) <actparameterpart> ::= LPARASYM <expression> <expression-list> RPARASYM.

(76) <identifierremainder> ::= <variableselector>.
Selset(76) = {PERIODSYM, RPARASYM, SEMICSYM, COMMASYM, EQUALSYM, LBRACKSYM, RBRACKSYM, DOUBLEDOT, DOSYM, OFSYM, ORSYM, TOSYM, ENDSYM, ELSESYM, THENSYM, UNTILSYM, DOWNTOSYM, POINTER, PLUSMINUS, RELOPER, MULTOPER}.

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

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

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

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

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

(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 <identifierremainder>.

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

(95) \(\langle\text{term}\rangle::=\langle\text{factor}\rangle\langle\text{factorlist}\rangle.\)
    Selset(95) = \{LPARASYM, LBRACKSYM, NILSYM, NOTSYM, STRING-
    SYM, IDENTIFIER, UNSGREAL, UNSGINTEG\}.

(96) \(\langle\text{termlist}\rangle::=\text{PLUSMINUS}\langle\text{term}\rangle\langle\text{termlist}\rangle.\)

(97) \(\langle\text{termlist}\rangle::=\text{ORSYM}\langle\text{term}\rangle\langle\text{termlist}\rangle.\)

(98) \(\langle\text{termlist}\rangle::=\langle\text{empty}\rangle.\)
    Selset(98) = \{RPARASYM, SEMICSYM, COMMASYM, EQUALSYM,
    RBRACKSYM, DOUBLEDOT, DOSYM, OFSYM, TOSYM,
    ENDSYM, ELSESYM, THENSYM, UNTILSYM, DOWNTOSYM, RELOPER\}.

(99) \(\langle\text{simpleexpression}\rangle::=\text{PLUSMINUS}\langle\text{term}\rangle.\)

(100) \(\langle\text{simpleexpression}\rangle::=\langle\text{term}\rangle\langle\text{termlist}\rangle.\)
    Selset(100) = \{LPARASYM, LBRACKSYM, NILSYM, NOTSYM,
    STRINGSYM, IDENTIFIER, UNSGREAL, UNSGINTEG\}.

(101) \(\langle\text{simpleexpressionrem}\rangle::=\text{EQUALSYM}\langle\text{simpleexpression}\rangle.\)

(102) \(\langle\text{simpleexpressionrem}\rangle::=\text{RELOPER}\langle\text{simpleexpression}\rangle.\)

(103) \(\langle\text{simpleexpressionrem}\rangle::=\langle\text{empty}\rangle.\)
    Selset(103) = \{RPARASYM, SEMICSYM, COMMASYM, RBRACKSYM,
    DOUBLEDOT, DOSYM, OFSYM, TOSYM, ENDSYM,
    ELSESYM, THENSYM, UNTILSYM, DOWNTOSYM\}.

(104) \(\langle\text{expression}\rangle::=\langle\text{simpleexpression}\rangle\langle\text{simpleexpressionrem}\rangle.\)
    Selset(104) = \{LPARASYM, LBRACKSYM, NILSYM, NOTSYM,
    STRINGSYM, PLUSMINUS, IDENTIFIER, UNSGREAL,
    UNSGINTEG\}.

(105) \(\langle\text{simplestatemntrem}\rangle::=\langle\text{variableselector}\rangle\text{BECOMES}\langle\text{expression}\rangle.\)
    Selset(105) = \{PERIODSYM, LBRACKSYM, POINTER, BECOMES\}.

(106) \(\langle\text{simplestatemntrem}\rangle::=\langle\text{actparameterpart}\rangle.\)
    Selset(106) = \{LPARASYM\}.

(107) \(\langle\text{simplestatemntrem}\rangle::=\langle\text{empty}\rangle.\)
    Selset(107) = \{SEMICSYM, ENDSYM, UNTILSYM, ELSESYM\}.
(108) \(<\text{compoundstmntrem}> \::= \text{SEMICSYM} \ <\text{statement}> \ <\text{compoundstmntrem}>\).

(109) \(<\text{compoundstmntrem}> \::= \text{ENDSYM}\).

(110) \(<\text{elseclause}> \::= \text{ELSESYM} \ <\text{statement}>\).

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

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

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

(114) \(<\text{caseelementlist}> \::= \text{SEMICSYM} \ <\text{caseelement}> \ <\text{caseelementlist}>\).

(115) \(<\text{caseelementlist}> \::= \text{ENDSYM}\).

(116) \(<\text{repeatstmtntlist}> \::= \text{SEMICSYM} \ <\text{statement}> \ <\text{repeatstmtntlist}>\).

(117) \(<\text{repeatstmtntlist}> \::= \text{UNTIL}\).

(118) \(<\text{forstatementrem}> \::= \text{TOSYM} \ <\text{expression}> \text{DOSYM} \ <\text{statement}>\).

(119) \(<\text{forstatementrem}> \::= \text{DOWNTOSYM} \ <\text{expression}> \text{DOSYM} \ <\text{statement}>\).

(120) \(<\text{withvariablelist}> \::= \text{COMMASYM IDENTIFIER} \ <\text{variablesselector}> \ <\text{withvariablelist}>\).

(121) \(<\text{withvariablelist}> \::= \text{DOSYM}\).

(122) \(<\text{unlabeledstmtnt}> \::= \text{IDENTIFIER} \ <\text{simplestmtntrem}>\).

(123) \(<\text{unlabeledstmtnt}> \::= \text{BEGINSYM} \ <\text{statement}> \ <\text{compoundstmtntrem}>\).

(124) \(<\text{unlabeledstmtnt}> \::= \text{IFSYM} \ <\text{expression}> \text{THENSEM} \ <\text{statement}> \ <\text{elseclause}>\).

(125) \(<\text{unlabeledstmtnt}> \::= \text{CASESYM} \ <\text{expression}> \text{OFSYM} \ <\text{caseelement}> \ <\text{caseelementlist}>\).
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 translation: 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; concatenate 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:
ARCTYPS... a list of type records for its parameters.
SWFORD... 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.
SCELEVNR....... 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:

UNDFNTDNR... 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. Immediately 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 management 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 wherever 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, CARCONSLLT, MATCHTYP):
  FIRSTTYP.... pointer to type record (inher.).
  FIRSTVAL.... explicit value of constant (inher.).
  CARCONSLLT.. 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):
  FIRSTTID..... 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, CARSMLPLST, SWPACK)
  FIRSTTYP.... pointer to type record (inher.).
  CARSMLPLST.. 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 preceding 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 preceding tag field (inher.).

<tagfieldremainder>(FIRSTID, REC, MATCHTYP):
  FIRSTID..... identifier which begins tag field (inher.).
  REC.......... tag field identifier descriptor (if any otherwise 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):  
   FFLDLST...... 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 <unpackstructtype>.

<formalparameter>(RETARG, RETACT):  
   RETARG...... type record pointer list of parameters as re-  
   quired in the ARCTYPES 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>(ARGLIST, ACTLIST):  
   ARGLIST..... list of all parameter types (if any otherwise  
   nil) (synth.).  
   ACTLIST..... 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 variable (inher.).  
   CAREXPLST..... list of expression constituents which consist of  
   their type records, string pointers to their  
   code and "variable flags" like FIRSTSWVAR above  
   (synth.).

<variablesselector>(SWASSIGN, REPLLST, CODIN, TYPIN,  
   CODOUT, TYPOUT):  
   SWASSIGN..... flag signalling if called at left hand side of  
   assignment (inher.).  
   REPLLST..... list of subcripts 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>.

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

<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>.

<simpleexpressionrem>(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.).

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

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

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

<caseelementlist>(FIRSTCONS LST, CARCONS LST, MATCH TYP):  
FIRSTCONS LST.. list of expl. values of case labels (inher.).  
CARCONS LST.. list of expl. values of all case labels(synth.).  
MATCH TYP..... same as in <caseelement>.

<forstatementrem>(CONTR TYP):  
CONTR TYP..... 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

"VARIABLE SELECTOR 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:

```
PROGRAM ERRONEOUS(INPUT);
(* This program tests error diagnostics and error recovery *)
LABEL 1, 2, 1;
CONST C1='A'; C2='YZ';
TYPE SR1=C1..'Z'; SR2=-5..0;
PTR=RREC2;
REC1=RECORD RF1, RF2: CHAR;
CASE PTR OF
'B': (RF3: @REC1);
'Z': (RF1: INTEGER)
END;
VAR V1: BOOLEAN; V2: @REC1;
V3: (TRUE, FALSE);
V4: PACKED ARRAY (.SR1, (ONE, TWO)). OF
RECORD VF1: ONE..TWO;
VF2: RECORD VF3: @REC1; END;
C2: INTEGER;
PROCEDURE P(A: SR2); FORWARD;
(*$S+ allow extented syntax *)
FUNCTION F(VAR P1, P2: BOOLEAN): 1..100;
(*$S- inhibit extensions *)
PROCEDURE P(A: SR2);
BEGIN NEW(V2, 'B');
3: IF TRUE
THEN V4(.C2, ONE..VF2.VF5:=V2;
5:2:1:26
P( F(NOT V1, V1) );
6:2:1:27
END;
BEGIN PUT;
FOR V4:=3 DOWNTO -5 DO
CASE V2@.RF1 OF
'A', 'B': PP(-23);
'C': GOTO 4;
'D', 'A':;
'E': WITH V4('.C', THREE..), VF2 DO
VF3@.RF3@.RF1 := SR2;
15:3:2:33
END;
IF F(V1)<5 AND C1=''' THEN I:=I+1;
X:=5.E-7;
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 ONLY)), OF CHAR 0 .. 255 (ORDERS ONLY) 18, 24,
20 F ENTRY( VAR, BOOLEAN 0 .. 1 (ORDERS ONLY); VAR, BOOLEAN 0 .. 1 (ORDERS ONLY); ); INTEGER 1 .. 10 0 (ORDERS ONLY) *** UNRESOLVED FORWARD REFERENCE
35 X VARIABLE, *** UNDEFINED ***
12 FALSE SCALAR, $IMPLT1 0 .. 1 (ORDERS ONLY)
0 INPUT VARIABLE, @INTEGER
9 INTEGER TYPE, INTEGER -2147483647 .. 2147483647 (ORDERS ONLY) 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)
3 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 (ORDERS ONLY) ), OF RECORD VF1 :, VF2 :, 24, 28, 32,
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 MULTIDINED 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 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 from 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

Abstraction, 4
\textit{actparameterpart}, 36, 46
ALGOL, 4
Alpha-conversion, 5
\$\textit{AND}, 8
Application, 4
Array, 8, 42
Assignment, 10, 11
Attribute, 43, 51
Attributed Translation, 39

BECOMES, 30
Beta-contraction, 5
Binding, 4, 5
Block, 9, 19, 39, 40
BNF-definition, 4
BOOLEAN, 50
Bound, 4

\textit{caseelement}, 38, 47
\textit{caseelementlist}, 38, 47
Case-statement, 15
\$\textit{CAT}, 6, 12
CHAR, 50
Character String, 43
CHR, 9, 11, 50
Church-Rosser Theorem, 5
Code Generation, 43
COLONSYM, 30
COMMASYM, 30
Compound Statement, 9
\textit{compoundstmtrem}, 38, 46
Conditional Statement, 15
\textit{constant}, 32, 44
\textit{constantdefinpart}, 32
\textit{constantlist}, 32, 44
\textit{constdefinpartrem}, 32
Continuation, 9, 10, 19, 21
Conversion, 5

Definition, 6
\$\textit{DIV}, 8
DOUBLEDOT, 30

\textit{elseclause}, 38, 47
Entire Variable, 10
\textit{entity}, 39
Environment, 9
\$\textit{EQ}, 8

EQUALSYM, 30
Error Diagnostic, 47
Error Recovery, 2, 47
Eta-contraction, 5
Expansion, 5
Expression, 10, 37, 46
\textit{expressionlist}, 35, 45

\textit{factor}, 36, 46
\textit{factorlist}, 36, 37, 46
\$\textit{FALSE}, 8, 15, 50
Field Identifier, 42
\textit{fieldlist}, 33, 45
\textit{fieldlistremaind}, 33, 45
File, 12
Fluid Length, 43
\textit{formalparameter}, 34, 45
\textit{formparameterlist}, 34, 35, 45
\textit{formparameterpart}, 35, 45
\textit{forstatementrem}, 38, 47
Free, 4
Function, 22, 41

Garbage Collection, 43
\$\textit{GE}, 8
GET, 12, 50
\$\textit{GT}, 8

Hashing Collision, 29
Hashing Function, 29
Hashing Table, 29

\$\textit{ID}, 6, 15
IDCLASS, 40
Identifier, 29, 39, 40, 41, 42, 43, 44, 45, 50
\textit{identifierlist}, 31, 43
\textit{identifierremaind}, 36, 46
IDNR, 40
\$\textit{IF}, 15
If-statement, 15
\$\textit{IMPLTn}, 41
Indeterminate, 4, 5, 6, 9, 10, 11, 12
Inherited, 39, 51
INPUT, 12, 50
Input-Output, 12
INTEGER, 41, 50
Irreducible, 5
<item>, 39

Keyword, 29

Label, 21, 41
<labeldeclar>, 31
<labeldeclaration>, 31
Lambda-calculus, 2, 4, 5, 7, 8, 9, 10, 12, 43, 50
Lambda-expression, 2, 4, 5, 6, 7, 8, 12, 28
LBRACKSYM, 30
$LE, 8
LEVRN, 40
Lexical Scanner, 28
List, 6, 39
List of Definitions, 6
Literal, 30
LL(1), 2, 23, 30, 51
LPARASYM, 30
$LT, 8

MAXSTRGL, 30
Meta-variable, 6
$MINUS, 8
$MINUSUNARY, 8, 11
$MULT, 8
MULTOPER, 30

Name, 6
$NAME, 8
<nonidentconstant>, 31, 32, 44
<nonidentconstrem>, 31, 44
Non-terminal, 30, 51
Normal Form, 5
$NOT, 8
NOTSYM, 30
Number, 29

$OMEGA, 7, 10
One Pass, 28
Operator, 30
$OR, 8
ORD, 11, 50
ORSYM, 30
OUTPUT, 12, 50

Packed, 42
PASCAL, 2, 4, 6, 8, 9, 10, 11, 12, 15, 21, 22, 28, 29, 30, 31, 39, 42, 47, 50,

51
PERIODSYM, 30
$PHI, 9
$PLUS, 8
PLUSMINUS, 30
Pointer, 22, 30, 42
PRED, 50
Primitive, 7
Procedure, 19, 41
<procfuncdelpart>, 35
<procfuncdeclar>, 35
<program>, 31, 39
Program Remainder, 9
Pushdown Automaton, 39
PUT, 12, 50

RBRACKSYM, 30
REAL, 41, 50
Record, 42
Record Field, 41
<recordsection>, 33, 45
Recursion, 7
Reduction, 5
RELOPER, 30
<repeatstatementlist>, 38
repetitive Statement, 15
$REPLACE, 8, 11
Representation Rule, 9
$RETRIEVE, 8, 11
RPARASYM, 30

Scalar Identifier, 41
Scalar Type, 41
$SCARDS, 12
Selection Set, 30
SEMSYSYM, 30
<setelement>, 36
<setelementlist>, 36
<setelementremainder>, 36
<setrange>, 35
Side-effects, 19
<simpleexpression>, 37, 46
<simpleexpressionrem>, 37, 46
<simplestatementrem>, 37, 46
<simpletype>, 32, 44
<simpletypelist>, 32, 44
<simpletypeoflist>, 32, 44
$SPRINT, 12
SPUNCH, 50
Statement, 9, 39
$STMi, 9
Storage Compaction, 43
String Pointer, 45
STRINGSYM, 30
Subrange Type, 41
SUCC, 50
Synchronizing Symbol, 47
Synthesized, 39, 51
S+ , 28
<tagfieldremainder>, 33, 44
<term>, 37, 46
Terminal, 30, 47
<termlist>, 37, 46
Token, 28
$TRUE, 8, 15, 50
$TUPINIT, 8
<type>, 33, 34, 45
<typedefinitionp t>, 34
<typedefinpartrem>, 34
Type Identifier, 41
Undefined Type, 42
<unlabeledstate mnt>, 38, 39
<unpackstructtype>, 33, 45
UNSGINTEG, 30
UNSGREAL, 30
UL, 28
Variable, 41
<variabledelp t rem>, 34
<variabledeclarp t>, 34
Variable Parameter, 42
<variables elector>, 35, 45
<variant>, 33, 44
<variantlist>, 33, 44
V-procedure, 19
While-statement, 15
<withvariablelist>, 38
X+, 28

$Y, 7
REFERENCES


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.