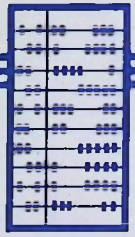
Generating A Standard Representation From Pascal Programs

by

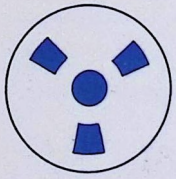
Kyung-Goo Doh James M. Bieman Albert L. Baker[1]

Technical Report #86-15

December 1986



DEPARTMENT OF COMPUTER SCIENCE
IOWA STATE UNIVERSITY/AMES, IOWA 50011



Generating A Standard Representation From Pascal Programs

by

Kyung-Goo Doh James M. Bieman Albert L. Baker[1]

Technical Report #86-15

December 1986

[1]Dr. Baker's current research has been supported in part by the Shell Companies Foundation, Inc.

Address Correspondence to Dr. James Bieman, Department of Computer Science, Iowa State University, Ames, IA. 50011, (515)294-4377.

Abstract

Software measures and software tools are often defined in terms of a particular, limited programming language. For example, a number of software measures are defined only for structured programs, and several approaches to program testing and debugging are defined using a specific simple language. As a result, implementing tools and measures so that they can be applied to "real" programs in "real" programming languages is difficult. Further, independent evaluation and comparison of measures and tools is difficult. In a supporting report, a standard representation of imperative language programs is formally described (Technical Report #86-17). The standard representation is independent of the syntax of any particular programming language, and supports the definition of a wide range of tools and measures. Additionally, the standard representation masks the actual program semantics. Thus the standard representation provides a vehicle by which large volumes of industrial software can be made available to researchers while protecting the proprietary nature of the programs.

This report describes the implementation of a mapping program which inputs 1985 ISO Standard Pascal programs and generates the corresponding standard representation. The implementation uses the YACC and LEX compiler generator tools and has been thoroughly tested. Design details, testing strategy, source code, test cases, and a user's manual are included in the report.

Contents

List of Figures

List of Tables

1 Introduction

The *StandardRep*[1] is a representation of imperative programs defined using an abstract data type approach. This representation provides a language independent basis for rigorous definitions of software tools and measures. While it hides the semantics of the actual programs, it preserves the control flow, data flow, and integration structures of a program unit. The protection of the semantics of programs allows us to use the *StandardRep* as a basis of a strategy for collecting large volumes of data on actual source programs. Such data can be used by software engineers interested in software tools and measures. In this project, we implement a *StandardRep* generator which takes the ISO Standard Pascal as an input and produces the corresponding *StandardRep* as an output. The generator translates a Pascal source program into a textual form of the *StandardRep*.

For this implementation, we use a software development tool, YACC[2] and LEX[3]. The C programming language[4] is used for supporting routines in order to be consistent with YACC and LEX. During a bottom-up parse, the translator constructs a data structure which represents the actual *StandardRep*. At the output generation phase, the data structure is searched through to produce the textual form of the *StandardRep*. Furthermore, we can easily derive an encoded form of this *StandardRep* from the data structure produced by the translator. The mapping strategy used throughout the implementation is based on [1]. The version of Pascal used in this implementation is the 1985 ISO Pascal Standard[5]. However, the implementation is designed so that it can be modified easily for non-standard features if needed.

In the next section, we present the *StandardRep* of Bieman, et. al. [1]. In section 3, we present the algorithms used in this implementation for generating the *StandardRep* from Pascal programs. The data structures themselves could be the input of the next stage of analysis tools for measures research. The algorithms are described abstractly

in a Pascal-like language form. In section 4, we discuss the testing strategy used to demonstrate the correctness of the generator. In the last section, we summarize the overall project and possible future work. Appendix A contains the actual code of the generator which consists of YACC and LEX specifications with their semantic routines, and the supporting C routines. In Appendix B, we present the test program segments we describe in Section 4 and the results. The purpose of inclusion of the test results in the Appendix is to help users understand how the *StandardRep* works and how Pascal programs are represented as the *StandardRep*. In Appendix C, we present the user's manual of the *StandardRep* generator.

2 A Standard Representation

In this section[1], we describe the definition of standard representation as it appears in [1]. The standard representation incorporates the concepts that are common to imperative language programs -control flow, data dependency, procedure interfaces, and the usage of operators and operands.

Program unit control flow is modeled by the familiar control flow graph in which nodes represent basic blocks. We provide a few definitions which are helpful in understanding the formal specification of the standard representation.

Definition 2.1 A *flowgraph* $G = (N, E, s, t)$ is a directed graph with a finite nonempty set of nodes N, a finite nonempty set of edges E, a start node $s \in N$, and a terminal node $t \in N$. The start node s is the unique node of N with indegree zero. The terminal node t is the unique node of N with outdegree zero. Each node $x \in N$ lies on some path in G from s to t.

[1] The material of this section is largely borrowed with permission from [1].

Definition 2.2 A *sequential block* of code in a source program P is a subsequence of tokens from P that is always executed in order starting with the first token and ending with the last token.

Definition 2.3 A *basic block* is a maximal length sequential block of code.

Program unit data flow is represented as the sequence of definitions and references in the nodes that represent basic blocks. The following definitions are derived from those by Hecht [6].

Definition 2.4 A variable *definition* is a sequence of tokens in a source program that, when executed, can (potentially) modify the value stored in a program variable.

Definition 2.5 A variable *reference* or variable *use* is a sequence of tokens in a source program that, when executed, references the value stored in a program variable.

Consider a program statement of the form $A := (X + Y) * Z$. The variable A is defined by the statement, and the variables X, Y, and Z are referenced.

We define the representation in terms of sets, sequences, tuples, reals, integers, booleans, and operations on these primitive types. The specification language has type declarations with syntax similar to Pascal types. For a detailed description of the specification language used, see [7]. To represent an entire program, we first break the program into its unit-level components, such as procedures or functions in a Pascal-like language. Each procedure or function has its own internal structure and a specific way in which it interfaces with the rest of the program. Thus we have:

Type Definition 1

$$StandardRep \quad = \quad set\ of\ UnitRepType$$

Type Definition 2

UnitRepType = *ordered pair(*
Interface: HeaderType,
UFS: UnitFlowStructure)

The *Interface* must contain the unique name of the program unit and the information necessary to determine the data interface with the rest of the program. Thus the *Interface* has three components – the procedure or function name, the list of formal parameters, and the set of global variables which are referenced or defined by the program unit.

Type Definition 3

HeaderType = *triple(*
UnitName: UnitID,
FormalParams: sequence of VarID,
Globals: set of VarID)

The *UnitFlowStructure* closely resembles a conventional control flowgraph with information concerning data dependency, unit interconnection, and software science measures [8] embedded within the nodes.

Type Definition 4

UnitFlowStructure = *4-tuple(*
Nodes: set of NodeType,
Edges: set of EdgeType,
Start: NodeID,
Terminal: NodeID)

Type Definition 5

EdgeType = *ordered pair(*
FromNode: NodeID,
ToNode: NodeID)

A node must contain information about the uses and definitions of variables within the corresponding basic block. A node must also contain information about the procedures and functions that are referenced in it. We retain the distinction between references that are used for definitions and references in predicates. We also include operator and operand counts from each basic block. These counts are used for software science measures[8] and are not accurately obtainable from the other node information. Thus our characterization of a node consists of four parts: a node identifier, a list of variable definitions, a list of predicate uses, and the software science information.

Type Definition 6

NodeType = *4-tuple(*
NID: NodeID,
LocalDefinitions: sequence of DefinitionType,
Predicate: ExpType,
Counts: HalsteadInfoType)

Type Definition 7

HalsteadInfoType = *ordered pair(*
Opcounts: set of OperatorCount,
Opandcounts: set of OperandCount)

Type Definition 8

OperatorCount = *ordered pair(*
OperatorName: OperatorID,
Occurrences: integer)

Type Definition 9

OperandCount = *ordered pair(*
OperandName: OperandID,
Occurrences: integer)

A definition of a variable occurs when either an assignment is made to that variable, or the variable is defined by a procedure call.

Type Definition 10

DefinitionType = *SimpleDefinition* | *ProceduralDefinition*

A *SimpleDefinition* has two components: the name of the variable being defined, and the list of items referenced in the definition. A procedure call is represented by the procedure's name and the sequence of actual parameters. The representation of the procedure call, combined with the control flow and data dependency information in the *UnitFlowStructure* of the called procedure, makes it possible to deduce potential data dependencies resulting from the call.

Type Definition 11

SimpleDefinition = *ordered pair(*
DefinedVariable: VarID,
Expr: ExpType)

Type Definition 12

ProceduralDefinition = *ordered pair(*
ProcName: UnitID,
ActualParams: sequence of ExpType)

For our purposes, an expression results in a particular sequence of references. The order is determined by the parsing.

Type Definition 13

ExpType = *sequence of ExprComponent*

Each item referenced in a *SimpleDefinition* may have any of three forms: the item may be a variable, a constant, or a function call.

Type Definition 14

ExprComponent = *VarID* | *ConstID* | *FunctionUse*

Type Definition 15

FunctionUse = *ordered pair(*
FunctionName: UnitID,
ActualParams: sequence of ExpType)

We now present a definition of the cyclomatic complexity measure[9] in terms of the *StandardRep* as an example. The ADT approach is also used to present the definition of cyclomatic complexity. We define the cyclomatic number of a *UFS* of type *UnitFlowStructure* as an abstract operation on the ADT *UnitFlowStructure* as follows:

Definition 2.6 *The cyclomatic number V of a program unit is:*

operation V(UFS: UnitFlowStructure): integer

post: V = | Edges(UFS) | - | Nodes(UFS) | + 2

For more details about the definitions of measures and tools in terms of the *StandardRep*, see [1].

3 Implementation

In this section, we describe the *StandardRep* generator which takes a syntactically correct Pascal program as an input and produces the corresponding *StandardRep* as output. The actual implementation is given in Appendix A. The *StandardRep* produced by the generator is in a textual form which uses the set, sequence, and tuple notation of Section 2. For this implementation, we use the software development tools, YACC[2] and LEX[3].

3.1 Using YACC and LEX

YACC is a parser generator, that is, a program for converting a grammatical specification of a language into a parser that will parse statements in the language. YACC provides

a way to associate meanings with the components of the grammar. Thus, as parsing takes place, the meaning associated with the grammar is evaluated. YACC accepts as input the grammar and the semantic actions, and produces a parsing function, named yyparse, and writes it out as a file of C code, named y.tab.c. The C code produced by YACC is an LALR(1) parser. The operation of this program is to call repeatedly upon the lexical analyzer for tokens, recognize the grammatical structure in the input, and perform the semantic actions as each grammatical rule is recognized. The lexical analyzer is created by LEX. The LEX source file is a specification of the lexical rules of the language to be parsed, using regular expressions and fragments of C to be executed when a matching string is found. LEX translates the specification into a routine, called yylex. The code produced by LEX recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. The stages in using YACC for development of this generator are as follows:

1. The appropriate grammar is selected, in this case the 1985 ISO Standard Pascal[5].

2. Each production rule is augmented with a semantic action – a statement of what to do when a particular grammatical form is found during parsing. This part is written in C code and defines how to construct data structures for the *StandardRep*.

3. A lexical analyzer is prepared by using LEX.

4. A routine for writing out the *StandardRep* is prepared. This routine searches through the data structure built by the parser and produces the textual form of *StandardRep*.

5. A controlling routine which calls the parser and the output generator is written. The parser, the lexical analyzer, output generating routine, and the control routine are compiled and linked together, and executed.

6. A preprocessor which converts all the upper-case letters into lower-case letters in the input Pascal program is written. The preprocessor also removes comments from the input program. The output of the preprocessor becomes the input of the lexical analyzer.

7. A shell program which runs the preprocessor and the *StandardRep* generator sequentially is prepared.

The remainder of Section 3 describes the mapping strategy and algorithms employed in this implementation.

3.2 Overall Program Structure

The *StandardRep* for a Pascal program is a set of procedure representations - with one procedure representation (of type *UnitRepType*) for each procedure or function, including the main procedure. For example, a Pascal program of the form:

```
program one;
    procedure a;
        procedure b;
            begin . . . end;{b}
        begin . . . end; {a}
    procedure c;
        begin . . . end;{c}
    begin . . . end {one}.
```

has a *StandardRep* of the form $\{ONE, A, B, C\}$ where ONE, A, B, and C are of type *UnitRepType* and represent the individual procedures. Each user-defined procedure or function in a Pascal program is treated as an individual program unit with its own *UnitRepType*[1].

Since procedures and functions in Pascal Program can be nested within each other, we need to use a stack in order to generate a set of *UnitRepType*'s. We use a global

stack named unitstack which has a *UnitRepType* as an element. The unitstack grows everytime the parser processes the following three tokens:

- *program*
- *procedure*
- *function*

Thus, the top of the unitstack always represents the procedure or function currently being processed. When the parser finishes processing each procedure or function unit, the top of the unitstack is popped off and added to the set of *UnitRepType*'s.

3.3 Interface Component of a UnitRepType

The *UnitRepType* for each Pascal program (main procedure), procedure, and function has an *Interface* component and a *UFS* component. The *Interface* component models the connections between an individual procedure and the rest of the program. Recall from Section 2 that the *Interface* consists of a *UnitID*, a *FormalParams* sequence, and a *Globals* set. The following examples contain segments of Pascal code and the *Interface* component of the representation[1].

1. Pascal Code: *program p (input,output)*

   Interface: $(p, \langle input, output \rangle, \{\})$

2. Pascal Code: *procedure q (a: integer; b,c: real; var d: char)*

   Interface: $(q, \langle a, b, c, d \rangle, \{v_1, v_2, \ldots, v_n\})$,

   where $v_1, v_2, ..., v_n$ are the *VarID*'s of all non-local variables that are referenced or set in the body of procedure q.

3. Pascal Code: *function r(function a: real; b,c: integer): real*

Interface: $(r, \langle a, b, c, r_{return} \rangle, \{\})$,

assuming r has no global definitions or references.

A *FormalParams* sequence is just a sequence of identifiers. However, since the sequence of definitions in the *Start* node in the *ProgramFlowStructure* represents the initialization of variables, i.e., the assignment of call-by-value parameters to local variables in Pascal, we need to distinguish value parameters from variable parameters. We use this information when we generate a simple definition in the *Start* node. Hence, we maintain another sequence of *VarID* which retains only value parameter identifiers. Then these value parameters can be referenced at the time of we generate the *Start* node.

In order to detect global definitions and references, we use a global stack named `memtab` which can also be used for finding the destination nodes of goto statements, distinguishing buffer variables from pointer variables, finding the object type of pointer variables, etc. Each item on the stack `memtab` is of type `MemoryTable`. The `Labels` component of the `MemoryTable` maintains information about *goto*'s. The `Const` component of the `MemoryTable` retains a set of constant identifiers defined. The `Type` component of the `MemoryTable` retains a set of type identifiers along with their types defined. The `Var` component of the `MemoryTable` retains a set of variable identifiers declared and their types. The ADT representation of the `MemoryTable` is as follows:

```
MemoryTable = 4-tuple(
              Labels : set of LabelTable,
              Const : set of ConstID,
              Type : set of TypeTable,
              Var : set of DecListType)
```

A new `MemoryTable` is pushed into `memtab` when entering each procedure unit and popped out when exiting each procedure unit. The top of the stack `memtab` represents

local declarations of the current unit, while the rest of the stack represents items that may be referenced now legally. Any identifiers which satisfy the following conditions are treated as global definitions or references. Such a global identifier is:

- recognized as an identifier by the lexical analyzer,
- not in the set of variable declarations of the top of the `memtab`,
- not in the set of constant definitions throughout the `memtab`, and
- in the set of variable declarations of at least one `MemoryTable` in the `memtab` stack except the top of the `memtab` stack.

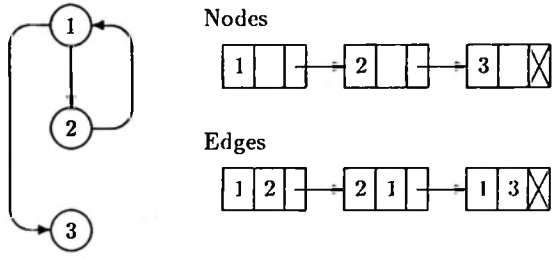
All global identifiers are added to the *Globals* in the the *Interface* component of *UnitRepType*.

3.4 UFS component

The *UFS* component of the representation models the intra-procedural structure of the program[1]. From Section 2, the *UFS* = (*Nodes*, *Edges*, *Start*, *Terminal*) has the basic structure of a flowgraph, where *Nodes* represent basic blocks and *Edges* represent possible control flow between *Nodes*.

In this implementation, we build the `UFS` as a `3-tuple(Nodes,Edges,ValueParameters)`. The `Nodes` are represented as a linked list of *NodeType*'s, and the `Edges` are represented as a linked list of *EdgeType*'s. `ValueParameters`, which is a list of `VarID`'s, stores the value parameter identifiers of a procedure unit. Later, the value parameters are used to build a *SimpleDefinition* in the *Start* node. Figure 1 shows how `Nodes` and `Edges` are represented in the actual data structure.

In the following discussion, we explain how basic blocks and control flow between basic blocks are constructed. First, we present the structure built by each statement.



Graphcal Form Actual Form

Figure 1: Example : Representation of Nodes and Edges

Then we discuss the algorithms to construct representations of basic blocks and control flow.

3.4.1 Structures built by Statements

Each grammar rule for statements returns a structure and passes it up to the ancestor in the parse tree. Pascal statements which return different structures are categorized as follows:

1. simple statement : assignment statement and procedural statement
2. *if* statement without *else* clause
3. *if* statement with *else* clause
4. *case* statement
5. *repeat* statement
6. *while* statement

7. *for* statement

8. *goto* statement

9. *label* statement

The structure each statement returns is described graphically in Figure 2. These statements are combined and linked by appropriate algorithms appearing in the semantic rules. Pascal constructs used to combine statements can be classified as either:

- a sequence of statements, or
- nested statements

In the next section, we describe the algorithms used to represent the basic blocks and the control flow between them.

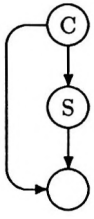
3.4.2 Building the Representation of Basic Blocks

In a bottom-up parse, we can view the partition of statements into basic blocks as a process of combining structures returned by descendents and passing the resulting structure up to the ancestor in the parse tree. Since we are using YACC, we explain the algorithms to build the representation of basic blocks by employing this concept. The returned structures of the right-hand side of the grammar rules for statements are combined or linked by appropriate algorithms in the semantic rules when reduced by the left-hand side of the rule.

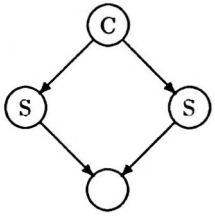
The generator must work for all possible combinations of sequences of statements. Algorithm `ConnectionSequence` takes two inputs, `n1` and `n2`, which are lists of `NodeType`'s, and returns a new list of `NodeType`'s. This algorithm is used for all combinations of sequencing. For most cases, we can build a new combined list just by merging the last node of the first list and the first node of the second list as shown in Figure 3(a). Exceptions occur in the following cases:

simple statement or a sequence of simple statement

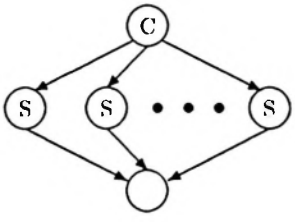
if statement (without *else*)



if statement (with *else*)



case statement



repeat statement

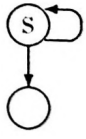
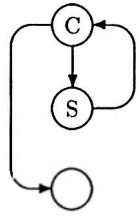
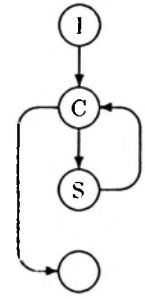


Figure 2: Structures built by Pascal Statements(cont.)

while statement



for statement



Symbol Notation:

C : predicate node

S : statement or statement sequence

◯ : empty node

I : initialization node for *for* statement

Figure 2: Structures built by Pascal Statements

1. a simple statement followed by *while* statement.

2. a simple statement followed by *repeat* statement.

3. any statement followed by a labelled statement or a sequence of statements where the first is a labelled statement.

4. any statement followed by *goto* statement.

5. a *goto* statement followed by any statement.

In the first three cases, we can build a new combined list by linking two lists as shown in Figure 3(b). The last two cases are rather complex and need additional explanations. We consider these cases separately in a later section. The algorithm ConnectionSequence is as follows:

```
function ConnectionSequence(n1,n2:list of NodeType's):list of NodeType's;
  begin
    if n2 is nil then  (* n2 is goto statement *)
      GotoFlag(Last(n1)) := ON;
    else if GotoFlag(Last(n1)) = ON then
      (* Last(n1) is goto statement *)
      begin
        GotoFlag(Last(n1)) := OFF;
        append n2 to n1;
      end
    else if n2 starts with repeat, while, or label statement
            and Last(n1) is empty then
      begin
        add an edge from Last(n1)
                    to First(n2);
        append n2 to n1;
      end
    else if n1 has only one NodeType then
      (* a simple statement or a sequence of simple statements *)
      begin
```

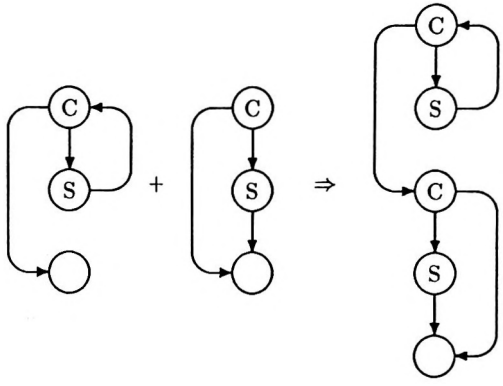
```
      move all contents of First(n2) to n1;
      remove First(n2) from n2;
      append n2 to n1;
      update edges connected with First(n2) to n1;
    end
  else
    begin
      move all contents of Last(n1) to First(n2);
      remove Last(n1) from n1;
      append n2 to n1;
      update edges connected with Last(n1) to First(n2);
    end;
  return n1;
end.
```

Now we consider nested Pascal statements. A simple statement, a sequence of simple statements, *if-then*, *if-then-else*, *case*, *repeat*, *while*, and *for* statements, and any sequence of the above statements(compound statement) may be nested in any of *if-then*, *if-then-else*, *case*, *repeat*, *while*, and *for* statements. What the structure each statement generates looks like is already seen. In order to describe algorithms for nesting, we denote each node in the structure as follows:

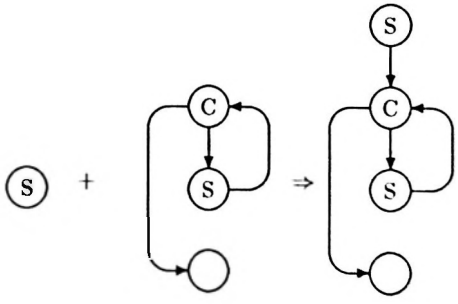
- **c** : node that contains conditional expressions
- **s** : a list of nodes that returned by nested statement (also s1,...,sn)
- **e** : last node which is always empty
- **i** : node that contains initialization part in *for* statement

Algorithms for combining the structures returned by each Pascal statement in case of nesting are as follows:

```
if-then statement :
  function ifthen(c,s,e:list of NodeType's):list of NodeType's;
```



(a). sequence of *while* statement and *if* statement



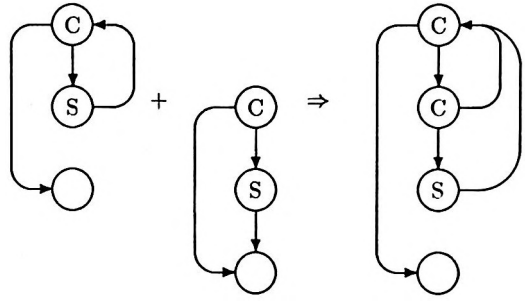
(b). sequence of simple statement and *while* statement

Figure 3: Examples: Sequencing

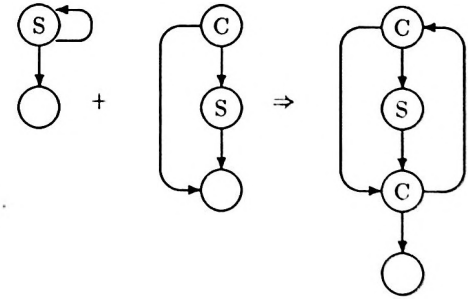
```
    begin
      if s is not empty then
        begin
          add an edge from c to First(s);
          if GotoFlag(Last(s)) = OFF then
            add an edge from Last(s) to e;
        end;
      add an edge from c to e;
      return Append(c,ConnectionI(s,e));
    end;
if-then-else statement :
  function ifthenelse(c,s1,s2,e:list of NodeType's):list of NodeType's;
    begin
      if s1 is not empty then
        begin
          add an edge from c to First(s1);
          if GotoFlag(Last(s1)) = OFF then
            add an edge from Last(s1) to e;
        end;
      if s2 is not empty then
        begin
          add an edge from c to First(s2);
          if GotoFlag(Last(s2)) = OFF then
            add an edge from Last(s2) to e;
        end;
      return Append(c,ConnectionI(s1,Connection(s2,e)));
    end;
case statement :
  function case(c,s1,...,sn,e:list of NodeType's):list of NodeType's;
    var new : list of NodeType's
    begin
      add edges from c to First(s1),....,First(sn)
        if si is not empty;    /* for all i = 1,...,n */
      add edges from Last(s1),....,Last(sn) to e
        if GotoFlag(Last(si)) = OFF;    /* for all i = 1,.,n */
      new := ConnectionII(...(ConnectionII(s1,s2),s3),....,sn);
      return Append(c,ConnectionI(new,e));
    end;
```

```
repeat statement :
  function repeat(s,e:list of NodeType's):list of NodeType's;
    begin
      add an edge from Last(s) to First(s);
      add an edge from Last(s) to e;
      return ConnectionII(s,e);
    end;
while statement :
  function while(c,s,e:list of NodeType's):list of NodeType's;
    begin
      add an edge from c to Start(s);
      add an edge from Last(s) to c;
      add an edge from c to e;
      return Append(c,ConnectionII(s,e));
    end;
for statement :
  function for(i,c,s,e:list of NodeType's):list of NodeType's;
    begin
      add an edge from i to c;
      add an edge from c to First(s);
      add an edge from Last(s) to c;
      add an edge from c to e;
      return Append(i,Append(c,ConnectionII(s,e)));
    end;

function Append(n1,n2:list of NodeType's):list of NodeType's;
  begin
    append n2 to n1;
    return n1;
  end;

function ConnectionI(n1,n2:list of NodeType's):list of NodeType's;
  begin
    if n1 is nil then  (* n1 is goto statement *)
      return n2;
    else if Last(n1) is not empty then
      begin
        append n2 to n1;
        return n1;
```

```
      end
    else
      begin
        move all contents of Last(n1) from n1;
        remove Last(n1) from n1;
        append n2 to n1;
        update edges connected with Last(n1);
        return n1;
      end
  end.

function ConnectionII(n1,n2:list of NodeType's):list of NodeType's;
  begin
    if n1 is nil then  (* n1 is goto statement *)
      return n2;
    else if n2 is nil then  (* n2 is goto statement *)
      return n1;
    else if n1 has only one NodeType then
      (* a simple statement or a sequence of simple statements *)
      begin
        append n2 to n1;
        return n1;
      end
    else if Last(n1) is not empty then
      begin
        append n2 to n1;
        return n1;
      end
    else
      begin
        lastnode := Last(n1);
        remove Last(n1) from n1;
        move all contents of lastnode to Last(n1);
        update edges connected with lastnode;
        append n2 to n1;
        return n1;
      end
  end.
```



(a). *if* statement nested in *while* statement



(b). *if* statement nested in *repeat* statement

Figure 4: Examples: Nesting

Figure 4 demonstrates these algorithms using simple examples. Details about dealing with *goto* statement are explained in the next subsection. For the details of the semantic actions and supporting routines, see the actual code in Appendix A.

3.4.3 Dealing with *goto* statements

The higher-level of the parse tree does not know whether the returning structure from the descendents is from a *goto* statement or not until it receives the structure from the descendent which may contain a *goto* statement. Since the parser needs to distinguish a *goto* statement from the other statements, the *goto* semantic routine returns a null pointer while every other statement always returns a non-null pointer to a list of *NodeType's*.

When a *goto* statement is parsed, an edge is added from the node having the *goto* statement to the destination node. However, since a *goto* statement does not generate a node, the edge should be connected from the most recently generated node. The most recently generated node can always be referenced by a global variable `ndptr`. At the time the *goto* statement is parsed, a `GotoFlag` component of `ndptr` is set to ON to denote that the node ends with a *goto* statement. By referencing this flag later in the upper-level of the parse tree, the parser knows if the node ends with a *goto* statement. Unfortunately, this rule does not always work. The exceptions will be discussed later in this section.

In Pascal, a *label* must be declared. During the parse of the *label* declaration section we construct a `LabelTable` for each declared *label*.

```
LabelTable = 4-tuple(
             Label : LabelType,
             To : NodeID,
             From : set of NodeID,
```

Dead : set of NodeID)

At this time, only the Label component is initialized and the other components are undefined or empty. This list of LabelTable is attached to the memtab. When the *label* statement itself is parsed, the generated *NodeID* is assigned to the To component of LabelTable. If a *goto* statement is parsed before the corresponding To component of LabelTable is set, i.e., in the case of a forward branch, the *NodeID* from which the *goto* branch starts is added to the component From. Later, when the *label* statement is parsed, edges from From to To which is just generated are added. If a *goto* statement is parsed, after the corresponding *label* statement, i.e., in case of backward branch, an edge from last generated *NodeID* to To.

The edges which are already in the list of *EdgeType*'s may be modified during the construction of the structures which represent basic blocks because the existing nodes can be merged by the operations of the basic block construction. Thus the node whose *NodeID* is already stored in the From component might be merged with another node before the *label* statement is parsed. Then the From component contains a *NodeID* of a non-existant node. Therefore, a "dead" *NodeID* is added to Dead component to indicate that the *NodeID* does not exist anymore.

This process works for most cases except when only a *goto* statement is in either an *else*-clause of an *if-then-else* statement or in a *case* statement. If an *else*-clause consists of solely a *goto* statement, then the From node of the branching edge should be the node containing conditional expressions. However, according to the above rule, the From node of the branching edge must be the last node of *then*-clause since ndptr points to it at the time of parsing *else*-clause. Thus, we have to treat a *goto* statement differently when it occurs in an *else*-clause or in a *case* statement. To solve this problem, we maintain a global stack named brrootstack which has BrRootStackType as an element. BrRootStackType is a 2-tuple(NodePointer, Flag). Everytime the parser processes an *if-then-else* or a *case* statement, a new BrRootStackType is pushed

onto brrootstack. Initially, a NodePointer points to the node having the conditional expressions and the Flag component is set to VALID. If any other statements except a *goto* statement occurs in the *else*-clause, ndptr is used for branching. On the other hand, if the Flag remains as VALID, i.e., only a *goto* statement is in an *else*-clause, the NodePointer of brrootstack is used. In the case of a *case* statement, we can apply the same method.

3.5 Variables in DefinitionType's and ExpType's

The value of an individual *DefinitionType* depends in part on the types of Pascal variables used in the statement represented. In this subsection, we describe the representation of Pascal simple statements that use simple data types and structured data types [1] as well as their implementation.

3.5.1 Simple Data Types

Simple Pascal variables include variables of type *real*, *integer*, *boolean*, and *char*. Sets, enumerated data types, and subrange types are also considered simple data types in defining the mapping of Pascal to the *StandardRep*. Variables of these simple types map directly to *VarID*'s. For example, consider a Pascal assignment of the form $y := x + y$, where y and x are simple variables. The assignment maps to a *SimpleDefinition* of the form $(y, \langle x, y \rangle)$. The Pascal procedure invocation *p(x, x + y)* maps to the *Procedural Definition* $(p, \langle \langle x \rangle, \langle x, y \rangle \rangle)$.

3.5.2 Structured Data Types

The representation of Pascal statements that contain variables of structured types is not so simple, because structured variables usually cannot map directly to *VarID*'s. We describe the mapping for each type of Pascal structured variable and its implemen-

tation.

The structured variables include array variables, record variables, pointer variables, and file variables. Since these variables are not directly mapped into *VarID*, we need a data structure to hold the structured variables in order to use them appropriately in the future. In this implementation, `ExtraIdType` is designed for this purpose. `ExtraIdType` is `2-tuple(ArrayIndex,Extras)` in which `ArrayIndex` is a list of ExpType's and `Extras` is a list of strings. Index variables of the array variable are stored in `ArrayIndex`. Thus as long as this list is not null, the variable represented by `ExtraIdType` is an array variable. `Extras` keeps field-identifiers of record variables and ↑ of pointer variable and buffer variable. Thus no ↑ at the head of the `Extras` means that the variable is not a pointer or buffer variable. The global stack `memtab` is used to distinguish pointer variables from buffer variables. Now we describe the mapping for each type of Pascal structured variables.

Array variables: In a static analysis, the actual cell that is defined by an assignment $A[i] := Z$ cannot be determined. In the *StandardRep*, an entire array is represented with one *VarID*. The index variable, i in the above example, is always a referenced variable. Since an indexed array assignment only modifies one element, the redefined array is dependent on its last state. Therefore the array itself is referenced. The above array assignment maps to the *SimpleDefinition* $(A, \langle A, i, Z \rangle)$. The implementation of the mapping of array variables to *StandardRep* is relatively simple. The array itself and all elements of `ArrayIndex` are inserted in front of the list of ExpType representing on the right-hand side of assignment statement.

Record variables: Following the strategy used for arrays, one *VarID* is used to represent an entire Pascal record. Thus, a Pascal variable reference $A.b$ is mapped to the same *VarID* as $A.c$. An assignment of the form $A.c := Y$ maps to the

SimpleDefinition $(A, \langle A, Y \rangle)$. We map records with fields that are arrays as follows: the assignment statement $A[i].b[j] := 7$ maps to the *SimpleDefinition* $(A, \langle A, i, j, 7 \rangle)$. The implementation of the mapping of record variables is similar to that of array variables. The record variable itself is inserted in front of the list of ExpType representing on the right-hand side of assignment statement.

Pointer based objects: In Pascal, pointers can only reference objects of a specified type that are dynamically allocated. Pointer values are either *nil* or are set via the *new* procedure which allocates the storage for the object referenced and sets the pointer value. We treat the collection of objects that a pointer may reference as one *VarID* in a manner similar to that used to represent array variables. During a static analysis we cannot determine which objects are referenced by a pointer or even how many such objects will exist at run time. However, in Pascal programs we can limit the range of a pointer reference to objects of a specified type that are allocated dynamically. Any variable reference made using a pointer refers to the collection of objects of the declared type that the pointer may possibly reference. Consider the following Pascal declarations:

```
type Cptr = ↑C;
     C = record
           v:    integer;
           next: Cptr
         end;
var x: Cptr;
```

Now we describe the representation of some Pascal statements using the above declarations:

- *new(x)*

  SimpleDefinition's are used to represent predefined procedures such as the *new(x)* command: $(x, \langle\rangle), (C, \langle x, C \rangle)$, where C represents the collection of

objects that x may reference. The actual implementation of this approach is discussed in Section 3.6.

- $x\uparrow.v := 7$

  This statement maps to a single *SimpleDefinition* $(C, \langle x, C, 7\rangle)$. We are again using the *VarID* C to represent the collection of objects that the pointer may be referencing.

- $x := x\uparrow.next$

  This is another single *SimpleDefinition* $(x, \langle x, C\rangle)$.

To find an object type, C, of a pointer variable, we also have to search through the global stack `memtab` from the top to the bottom. The algorithm for finding C is as follows:

```
function FindPointerObject(varid:VarID):VarID;
  var typeid : VarID;
  begin
    typeid := type of varid in Var component;
    if typeid is preceded by ^ then
      return removeuparrow(typeid)
    else
      begin
        while typeid is preceded by ^ do
          typeid := type of typeid in Type component;
        return removeuparrow(typeid);
      end;
  end;
```

The function `removeuparrow` takes a type identifier which includes an uparrow as an input parameter and returns the identifier without the uparrow.

File variables: Every file variable F has an associated, implicitly declared buffer variable $F\uparrow$. In the representation of file primitive procedures with a file variable argument, the implicit buffer variable is included explicitly in the representation.

Therefore, in our implementation, we represent the *write* statement, *write(F,a,b)* as the *SimpleDefinition*'s $(F, \langle F, a, b\rangle), (F\uparrow, \langle b\rangle)$. The other possibility of the representation of the *write* statement is discussed in [1]. The often implicit textfile program parameter *Output* and the buffer variable *Output*↑ will be explicitly included as a parameter in the representation (and analogously for *Input*). In assignment statements that reference or set the value of the buffer variable $F\uparrow$, the buffer variable itself is modified or set. Thus, the representation of "$X\uparrow$" depends upon whether X is a pointer or file variable. Buffer variables in assignment statements are treated the same way as simple variables.

3.5.3 Value Parameters

Any initialization of variables is represented by the sequence of definitions in the *Start* node in the *ProgramFlowStructure*. In Pascal, these definitions include the assignment of call-by-value parameters to local variables. Therefore, the *Start* node in the *UnitFlowStructure* of *procedure Q(a: integer, var b: integer)* has the *SimpleDefinition* $(a', \langle a\rangle)$. All references to a in the procedure are represented by $a'[1]$.

In our generator, value parameters of a procedure unit are detected when parsing the formal parameter list. However, at the time the value parameters are parsed, the corresponding unit block has not been defined. Hence, we need a data structure which saves the value of the parameters of the procedure unit. To do this, we have an additional list only for value parameters in *FormalParams* of *HeaderType*. This list is available via global stack `unitstack` during parsing the procedure block to recognize if an identifier is a local variable.

3.5.4 *With* Statement

The *with* statement in Pascal allows a convenient shorthand. After a record variable appears in a *with* statement's variable list, its field names denote fields for the remainder of the *with* statement's action.

Therefore, we need to distinguish field names from the ordinary variables in the body of the *with* statement. To do this, we maintain a global stack named `withvarstack`. The `withvarstack` has `WithVarType` as its element. The ADT representation of `WithVarType` is as follows:

```
WithVarType = ordered pair(
              ID : VarID,
              Extras : ExtraIdType)
```

When entering a *with* statement environment, every *with* variable name is pushed onto the `withvarstack`. Hence, when `withvarstack` is not empty, we examine each identifier parsed to determine whether it is an ordinary variable or a field of a record variable which is one of the elements of `withvarstack`. If the identifier examined is a field of record variable in `withvarstack`, then we have to use the record variable identifier instead of field identifier. When exiting *with* statement environment, the `withvarstack` is popped to restore the earlier environment.

3.6 Predefined Procedures and Functions

In addition to those described previously, Pascal includes a number of primitive or predefined procedures and functions. The *StandardRep* could include a *UnitRepType* for each of the primitives used in the program. However, predefined procedures and functions represent language primitives. Predefined functions, e.g., *sin* and *abs*, are not conceptually different from operators. The program units that implement predefined

functions and procedures are part of the language and not part of a software implementation. We need not include the structure of these primitives in the *StandardRep*. Instead, we define one or more *SimpleDefinition*'s to represent each of these primitive program units.

Table 1 displays the complete mappings of predefined procedures. In Table 1, F represents a file variable and $F\uparrow$ represents an implicit file pointer variable. In the mapping of predefined procedures for pointers, T is an implicit variable representing the collection of objects that the pointer P may address. In addition to the Table 1, we can have variables of structured types as parameters of read, readln, write, and writeln. Furthermore, we can have pointer variables which are components of structured types; for example:

- A component points to the same object as p:

  new(p↑.next) $(T, \langle T \rangle)$

  where T is an object to which p points.

- A component points to a different object than p:

  new(p↑.ptr) $(T, \langle T \rangle), (C, \langle C, T \rangle)$

  where C is an object to which ptr points.

To treat the above examples properly, we employ the same method as we do for structured data types in section 3.5.

These predefined procedures are recognized when parsing a procedural statement. However, we must treat them differently than an ordinary procedure call. To do this, we use two supporting routines - one for procedures with parameters and the other for procedures without parameters - to recognize predefined procedure identifiers and generate *SimpleDefinition*'s as defined in Table 1.

Predefined Procedure	Sequence of *SimpleDefinitions*
rewrite(F)	$(F, \langle\rangle)$
put(F)	$(F, \langle F, F\uparrow\rangle), (F\uparrow, \langle\rangle)$
reset(F)	$(F\uparrow, \langle F\rangle)$
get(F)	$(F\uparrow, \langle F\rangle)$
read(F, V)	$(V, \langle F\uparrow\rangle), (F\uparrow, \langle F\rangle)$
read(F, $V_1, \ldots, V_n$**)**	equivalent to **read(F,** V_1**);** ...**; read(F,** V_n**)**
readln(F, V)	equivalent to **read(F, V)**
readln(F, $V_1, \ldots, V_n$**)**	equivalent to **read(F,** $V_1, \ldots, V_n$**)**
readln(F)	equivalent to **get(F)**
read(V)	$(V, \langle Input\uparrow\rangle), (Input\uparrow, \langle Input\rangle)$
read($V_1, \ldots, V_n$**)**	equivalent to **read(**V_1**);** ...**; read(**V_n**)**
readln(V)	equivalent to **read(V)**
readln($V_1, \ldots, V_n$**)**	equivalent to **read(**$V_1, \ldots, V_n$**)**
readln	equivalent to **get(Input)**
write(F, E)	$(F\uparrow, \langle E\rangle), (F, \langle F, F\uparrow\rangle)$
write(F, $E_1, \ldots, E_n$**)**	$(F, \langle F, E_1, \ldots, E_n\rangle), (F\uparrow, \langle E_n\rangle)$
writeln(F, E)	equivalent to **write(F, E); writeln(F)**
writeln(F, $E_1, \ldots, E_n$**)**	equivalent to **write(F,** $E_1, \ldots, E_n$**); writeln(F)**
writeln(F)	$(F, \langle F,$ *end-of-line*$\rangle)$
write(E)	$(\text{Output}\uparrow, \langle E\rangle), (\text{Output}, \langle Output, Output\uparrow\rangle)$
write($E_1, \ldots, E_n$**)**	$(\text{Output}, \langle\ \text{Output}, E_1, \ldots, E_n\rangle), (\text{Output}\uparrow, \langle E_n\rangle)$
writeln(E)	equivalent to **write(E); writeln**
writeln($E_1, \ldots, E_n$**)**	equivalent to **write(**$E_1, \ldots, E_n$**); writeln**
writeln	$(Output, \langle Output,$ *end-of-line*$\rangle)$
page(F)	$(F, \langle F,$ *end-of-page*$\rangle)$
page	$(Output, \langle Output,$ *end-of-page*$\rangle)$
pack(A, B, C)	$(C, \langle A, B\rangle)$
unpack(A, B, C)	$(B, \langle A, C\rangle)$
new(P)	$(P, \langle\rangle), (T, \langle P, T\rangle)$
new(P , $C_1, \ldots, C_n$**)**	$(P, \langle\rangle), (T, \langle P, T, C_1, \ldots, C_n\rangle)$
dispose(P)	$(T, \langle P, T\rangle), (P, \langle nil\rangle)$
dispose(P $C_1, \ldots, C_n$**)**	$(T, \langle P, T, C_1, \ldots, C_n\rangle), (P, \langle nil\rangle)$

Table 1: Mapping of Pascal Predefined Procedures

Pascal predefined functions each have one formal parameter and return a value without side effects. These functions include *abs(X)*, *sqr(X)*, *sqrt(X)*, *sin(X)*, *cos(X)*, *exp(X)*, *ln(X)*, *arctan(X)*, *odd(X)*, *eof(F)*, *eoln(F)*, *trun(X)*, *round(X)*, *ord(X)*, *chr(X)*, *succ(X)*, and *pred(X)*. We consider the use of a predefined function as a simple variable or constant reference. Let X and Y be variables of a simple type; the following are examples of the *SimpleDefinition*'s that represent statements that use the *abs* function:

1. Pascal Code: $Y :=$ *abs(X)*

   SimpleDef: $(Y, \langle X \rangle)$

2. Pascal Code: $Y :=$ *abs(X - Y)*

   SimpleDef: $(Y, \langle X, Y \rangle)$

In the case of *eof* and *eoln* without parameters, we use Input instead of file variable F as follows:

1. Pascal Code: *while not eoln do*

   Predicate: $\langle Input \rangle$

2. Pascal Code: *if not eof then*

   Predicate: $\langle Input \rangle$

When parsing expressions, predefined functions can be recognized in a similar manner as for predefined procedures.

3.7 Operator and Operand Counts

The *HalsteadInfoType* component of each node in a *UnitFlowStructure* is necessary to calculate the software science measures[8]. The determination of which tokens and groups of tokens constitute operators or operands is made according to the following

criteria of Bugh[10] with a little modification for this particular implementation. The counting strategy is originated by Salt[11]. The operator and operand counting strategy used in this implementation are as follows:

1. Only executable statements are considered. The program heading, declaration, and comments are ignored.

2. Variables, constants, filenames, labels, and the reserved word *nil* are counted as operands.

3. The following entities are counted as single operators:

*	/	*div*	*mod*	<	<=	=
<>	>=	>	:=	↑	,	..
not	*and*	*or*	*in*	*else*	*goto*	;

4. The following groups of entities are counted as single operators:

begin..end	*case..of..end*	*while..do*
with..do	*repeat..until*	*if..then*
for..to..do	*for..downto..do*	

5. The following entities or pairs of entities are counted as single operators:

   - unary and binary + are counted as the same operator except when unary + occurs as part of case label,
   - unary and binary - are counted as the same operator except when unary - occurs as part of case label,
   - . is counted as either a record component separator or a program terminator,

- () is counted as either an argument list operator or grouping operator,
- [] is counted as either a subscript operator or set operator.

6. Procedure and Function calls are counted as operators.

7. All *goto* statements are counted as operators and accompanying labels are counted as operands.

8. The colon when it occurs with a label is considered part of the label. It is counted as an operator only when it occurs as a field formatting symbol in a *write* or *writeln* statement. For example, in the code below,

```
            goto 100
            .
            .
100:        writeln(total:5:2,average:5:2);
```

   the label *100:* occurs twice and the operator *:* occurs four times.

9. Each *case* label, its accompanying colon, and the optional unary + and - are counted as a single operand. If a *case* label list is used to select a clause of the *case* statement, then each label is counted as a separate operand and the comma separating them are counted as occurrences of the same operator. For example,

```
case number of
   -777, 0 : stmt1;
   pos : stmt2;
   neg : stmt3;
end
```

-777:, *0:*, *pos:*, and *neg:* are operands and , and *case..of..end* are operators.

10. The := is not counted as a separate operator when it occurs within a *for* statement.

11. Literals are counted as operands. This includes the literal string and the enclosing quotes.

12. The executable portion of any procedure or function is counted with any local variable treated as a unique operand and any global variable treated as an occurrence of some previously defined operand.

In Bugh's strategy the unary + and - are counted as operators while in our implementation they can be counted as part of operands as we presented above. The main reason for this change is due to the difficulties in counting unary operands in a *case* label. At the time of parsing *case* label, the parser does not know which node the operator and operand belongs to. Hence, we need to return both the operand and operator as a result of the returned value of the grammar rule. This complicates the implementation since we have to maintain another global variable only for this. Furthermore, we think that this alternative way is also a reasonable counting strategy. Consequently, we decided to treat the each *case* label including unary operator and colon as one operand even though the original Bugh's strategy is possible to implement.

Counting operators and operands is done during the parsing of the program. As soon as an operator or operand is parsed, it is added to the list *Operators* or *Operands* of the global variable *ndptr*.

4 Testing Strategy

A perfect testing strategy is impossible. However, by selecting test data very carefully, we believe we can achieve reasonably reliable test results. To test our *StandardRep*

generator, we tried to find all possible combinations of program segments. We employ the following strategy for testing the representation of control flow:

1. Select the simplest program segments based on the basic structures discussed in Section 2 and test them. These program segments are as follows:
   - a simple statement,
   - an *if-then* statement with a simple statement in it,
   - an *if-then-else* statement with a simple statement in it,
   - a *case* statement with three cases, each of them has a simple statement,
   - a *repeat* statement with a simple statement in it,
   - a *while* statement with a simple statement in it,
   - a *for* statement with a simple statement in it.
2. Find all possible sequences of two statements selected above and test them.
3. Find all possible nesting cases of two statements selected above and test them. They are as follows: Each of the eight statements selected in strategy 1 nested in *if-then* statement, *if-then-else* statement, *case* statement, *repeat* statement, *while* statement, and *for* statement, respectively.

If we consider strategy 1 to be a basis and strategy 2 and 3 to be inductive clauses, we can show the correctness of the generator in representing control flow without testing any more combinations.

For testing the generator on programs with *goto* statements, we select the following program segments.

1. a *goto* statement in a statement sequence for both forward and backward branching.

2. a *goto* statement nested in an *if-then* statement for both forward and backward branching.

3. a *goto* statement nested in an *if-then-else* statement for both forward and backward branching. In this case, we consider the cases where the *goto* statement is in *then* clause, *else* clause, and both.

4. a *goto* statement that branches from the inside of the loop to the outside of the loop for both forward and backward branching.

For additional testing of the generator, we tested the following program segments.

1. expressions consisting of

    - real and integer numbers
    - *true* and *false*
    - literals
    - arithmetic, relational, boolean, and set operators
    - set notation
    - user-defined function call
    - predefined function call

2. empty statement

3. assignment statement consisting of

    - simple identifiers
    - identifiers with array designators
    - pointer variables

- buffer variables
- record variables

4. procedural statement
   - user-defined procedures with and without parameters
   - predefined procedures displayed in Table 1
5. *with* statement
   - nested *with* statement
6. sequencing and nesting of procedure and function declaration
7. detecting global variables
8. value parameters
9. comments

All the test program segments selected were tested completely. We found that all of them were working correctly. Hence, we have confidence that the *StandardRep* generator is correct with respect to all possible Pascal programs.

5 Conclusion

The mapping of ISO Standard Pascal programs to the Standard Representation is implemented and tested by selected Pascal program segments. The implementation was designed using the YACC and LEX compiler generator tools. An inductive approach was used to select test data, and Pascal test programs were tested successfully. This project demonstrates that a *StandardRep* of programs written in an actual programming

language can be constructed. The implementation works on full ISO Standard Pascal including pointers, structures, files, and goto statements.

As a basis for this implementation, we used the mapping strategy from Pascal programs to the *StandardRep* of Bieman, et. al.[1]. However, if one wants to define a different mapping strategy and use it in implementing a generator, one can still use similar algorithms to those developed here. Furthermore, the generator has been designed so that it can be easily modified for other local versions of Pascal. In order to deal with the non-standard features of Pascal, we can add and modify suitable semantic routines of the YACC specification of the generator. Hence, we believe that the actual code in the Appendix A and algorithms in Section 3 will help in developing different versions of the generator.

As a future project, we can develop generators for the other imperative languages including FORTRAN, COBOL, C, and Ada.

References

[1] J. M. Bieman, A. L. Baker, P. N. Clites, D. A. Gustafson, and A. C. Melton. A standard representation of imperative language programs. 1986.

[2] S. C. Johnson. *YACC - Yet Another Compiler Compiler.* Technical Report, AT & T Bell Laboratories, Murray Hill, New Jersey, 1975.

[3] M. E. Lesk. *Lex - a lexical analyzer generator.* Technical Report Computing Science Technical Report 39, AT & T Bell Laboratories, Murray Hill, New Jersey, 1975.

[4] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[5] K. Jensen and N. Wirth. *Pascal User Manual and Report.* Springer-Verlag, New York, 3rd edition, 1985.

[6] M. S. Hecht. *Flow Analysis of Computer Programs.* Elsevier, New York, 1977.

[7] A. L. Baker, J. M. Bieman, and P. N. Clites. *Implications for Formal Specifications - Results of Specifying a Software Engineering Tool.* Technical Report T.R. 86-9, Iowa State University, Dept. of Computer Science, Iowa State University, Ames, Iowa, 1986.

[8] M. H. Halstead. *Elements of Software Science.* Elsevier, New York, 1977.

[9] T. J. McCabe. A complexity measure. *IEEE Trans. on Software Engineering,* 2(4), 1976.

[10] R. A. Bugh. *An Empirical Investigation of Control Flow Complexity Measures.* Master's thesis, Iowa State University, 1984.

[11] N. F. Salt. Defining software science counting strategies. *ACM SIGPLAN Notices,* 17:58-67, March 1982.

A Source Program for StandardRep

```
/***** sr.h ***********************************************************/
#define      NULL             0
#define      DELETED         -1
#define      START           -2
#define      TERMINAL        -3
#define      CASEMID         -4
#define      IDTAG            0
#define      FUNCTAG          1
#define      PTRTAG           2
#define      SDEFTAG          0
#define      PDEFTAG          1
#define      BASICTYPE        1
#define      ENUMERATEDTYPE   2
#define      SUBRANGETYPE     3
#define      RECORDTYPE       4
#define      POINTERTYPE      5
#define      FILETYPE         6
#define      ON               1
#define      OFF              0
#define      VALID            1
#define      INVALID          0
#define      YES              1
#define      NO               0

typedef struct UnitRepType {
        struct HeaderType *interface;
        struct UnitFlowStruct *ufs;
        struct UnitRepType *next;
} UnitRepType;

typedef struct HeaderType {
        char *pname;
        struct FPListType *fparams;
        struct IdListType *globals;
} HeaderType;

typedef struct FPListType {
        struct IdListType *fp;
        struct IdListType *in;
} FPListType;

typedef struct IdListType {
        char *id;
        struct IdListType *next;
} IdListType;

typedef struct ExtraIdType {
        struct ExpType *u;
        struct IdListType *c;
} ExtraIdType;

typedef struct WithVarType {
        char *id;
        struct ExtraIdType *ext;
        short borderline;   /* ON or OFF */
        struct WithVarType *next;
} WithVarType;

typedef struct UnitFlowStruct {
```

```
        struct NodeType *nd;
        struct Edge *eg;
        struct IdListType *s;
} UnitFlowStruct;

typedef struct NodeType {
        int nid;
        struct LocalDef *ldef;
        struct ExpType *pexp;
        struct OpCount *operators;
        struct OpCount *operands;
        short gt;   /* ON or OFF */
        struct NodeType *next;
} NodeType;

typedef struct LocalDef {
        union {
            struct SimpleDef *sdef;
            struct ProcDef *pdef;
        } def;
        short tag;
        struct LocalDef *next;
} LocalDef;

typedef struct SimpleDef {
        char *varid;
        struct ExpType *sexp;
} SimpleDef;

typedef struct ProcDef {
        char *procid;
        struct ExpSequence *pexp;
} ProcDef;

typedef struct ExpType {
        union { char *id;
                struct FuncExp *fexp;
        } exp;
        short tag;
        struct ExpType *next;
} ExpType;

typedef struct FuncExp {
        char *funcid;
        struct ExpSequence *exps;
} FuncExp;

typedef struct ExpSequence {
        struct ExpType *exp;
        struct ExpSequence *next;
} ExpSequence;

typedef struct OpCount {
        char *name;
        int occur;
        struct OpCount *next;
} OpCount;

typedef struct Edge {
        int from;
```

```
        int to;
        struct Edge *next;
} Edge;

typedef struct MemoryTable {
        struct LabelTable *lab;
        struct IdListType *const;
        struct TypeTable *type;
        struct DecListType *var;
        struct MemoryTable *next;
} MemoryTable;

typedef struct LabelTable {
        int to;
        int nid;
        struct OriginList *from;
        struct OriginList *dead;
        struct LabelTable *next;
} LabelTable;

typedef struct TypeTable {
        char *id;
        struct TypeDenoterType *d;
        struct TypeTable *next;
} TypeTable;

typedef struct TypeDenoterType {
        short tag;  /* BASICTYPE, ENUMERATEDTYPE, SUBRANGETYPE, */
        union {     /* RECORDTYPE, POINTERTYPE, and FILETYPE    */
            char *id;
            struct IdListType *i;
            struct DecListType *f;
        } d;
        struct TypeDenoterType *next;
} TypeDenoterType;

typedef struct DecListType {
        struct IdListType *idl;
        struct TypeDenoterType *d;
        struct DecListType *next;
} DecListType;

typedef struct OriginList {
        int n;
        struct OriginList *next;
} OriginList;

typedef struct BrRootStack {
        struct NodeType *nd;
        short flag;
        struct BrRootStack *next;
} BrRootStack;


/***** sr.y ********************************************************/
%{
#include "sr.h"
%}
%union {
     char *str;
```

```
      IdListType *idlptr;
      FPListType *fptr;
      ExpType *expptr;
      ExtraIdType *extptr;
      WithVarType *wptr;
      ExpSequence *expseqptr;
      NodeType *nptr;
      UnitFlowStruct *ufs;
      UnitRepType *prep;
      TypeDenoterType *tdptr;
      LabelTable *labptr;
      TypeTable *typtr;
      DecListType *dptr;
}
%token <str> NUMBER ID LITERAL
%token <str> PROGRAM LABEL CONST TYPE PROCEDURE FUNCTION VAR
%token <str> BEGIN END ARRAY FILE RECORD SET PACKED
%token <str> CASE OF FOR TO DOWNTO DO IF THEN ELSE
%token <str> REPEAT UNTIL WHILE WITH GOTO ASGN NIL FORWARD
%token <str> LB RB PD CM CL SC AR LP RP SR SL
%type <str> semicolon todownto constant carrays arrinds caseconst
%type <idlptr> constdef consts idlist
%type <idlptr> caseconsts constlist
%type <extptr> extvars extvar
%type <wptr> withvars withvar
%type <fptr> progparams fparams fpmlist fparam dfparams
%type <expptr> aparam exprlist expr p
%type <expptr> setelems elemlist elem
%type <expseqptr> apmlist plist
%type <nptr> stmtseq stmts stmt caselist caseelem rtnnewnode
%type <prep> procfuncdec procfunc block
%type <tdptr> typedenoter typelist
%type <labptr> labeldec labels
%type <typtr> typedef types
%type <dptr> fieldlist fixedpart variantpart variants vardec vars
%left <str> IN EQ NE LT GT LE GE
%left <str> OR PLUS MINUS
%left <str> DIV MOD AND MULTIPLY DIVIDE
%left <str> UNARYMINUS UNARYPLUS NOT
%%

program  : PROGRAM
           ID
           progparams
           SC
           { pushmemtab(); }
           block
           PD
                                { addunitrep($2,$3);
                                  operatorcnt($6,"."); }
         ;

progparams: /* empty */         { $$ = NULL; }
         | LP
           idlist
           RP
                                { $$ = mkfplist($2); }
         ;
```

```
block    : { initunitrep(); }
           labeldec
           constdef
           typedef
           vardec
           procfuncdec
           BEGIN stmtseq END
                                { operatorcnt($8,"begin..end");
                                  addfirstedge($8->nid);
                                  addlastedge($8);
                                  rmlastnode($8);
                                  unitstack->ufs->nd = $8;
                                  popmemtab();
                                  $$ = $8; }
         ;

labeldec : /* empty */          { $$ = NULL; }
         | LABEL labels SC      { memtab->lab = $2;
                                  $$ = $2; }
         ;

labels   : NUMBER               { $$ = setlabtab($1); }
         | labels CM NUMBER     { $$ = addlabtab($1,setlabtab($3)); }
         ;

constdef : /* empty */          { $$ = NULL; }
         | CONST consts         { memtab->const = $2;
                                  $$ = $2; }
         ;

consts   : ID EQ constant SC    { $$ = mkidlist($1); }
         | consts ID EQ constant SC
                                { $$ = catidlist($1,mkidlist($2)); }
         ;

typedef  : /* empty */          { $$ = NULL; }
         | TYPE types           { memtab->type = $2;
                                  $$ = $2; }
         ;

types    : ID EQ typedenoter SC
                                { $$ = settype($1,$3); }
         | types ID EQ typedenoter SC
                                { $$ = addtype($1,settype($2,$4)); }
         ;

vardec   : /* empty */          { $$ = NULL; }
         | VAR vars             { addvars($2);
                                  $$ = memtab; }
         ;

vars     : idlist CL typedenoter SC
                                { $$ = mkdeclist($1,$3); }
         | vars idlist CL typedenoter SC
                                { $$ = catdeclist($1,mkdeclist($2,$4)); }
         ;

procfuncdec:    /* empty */     { $$ = NULL; }
         | procfuncdec procfunc SC
                                { $$ = $2; }
```

```
        ;
procfunc : PROCEDURE
           ID
           fparams
           SC
           block
                    { operatorcnt($5,";");
                      if ($3 != NULL)  unitstack->ufs->s = $3->in;
                      $$ = addunitrep($2,$3); }
         | PROCEDURE
           ID
           fparams
           SC
           FORWARD
                    { popmemtab(); $$ = NULL; }
         | FUNCTION
           ID
           SC
                    { pushmemtab(); }
           block
                    { operatorcnt($5,";");
                      $$ = addunitrep($2,mkfplist(mkidlist($2))); }
         | FUNCTION
           ID
           fparams
           CL
           ID
           SC
           block
            { operatorcnt($7,";");
              if ($3 != NULL)  unitstack->ufs->s = $3->in;
              $$ = addunitrep($2,catfplist($3,mkfplist(mkidlist($2))));}
         | FUNCTION
           ID
           fparams
           CL
           ID
           SC
           FORWARD
                    { popmemtab(); $$ = NULL; }
         ;
fparams  : /* empty */          { pushmemtab(); $$ = NULL; }
         | {pushmemtab();}
           LP fpmlist RP        { $$ = $3; }
         ;
fpmlist  : fparam
         | fpmlist SC fparam    { $$ = catfplist($1,$3); }
         ;
fparam   : VAR idlist CL ID
                    { addvars(mkdeclist($2,mkbasictype($4)));
                      $$ = mkfplist($2); }
         | idlist CL ID
                    { addvars(mkdeclist($1,mkbasictype($3)));
                      $$ = mkinfplist($1); }
         | VAR idlist CL carrays
                    { addvars(mkdeclist($2,mkbasictype($4)));
```

```
                    $$ = mkfplist($2); }
        | idlist CL carrays
                    { addvars(mkdeclist($1,mkbasictype($3)));
                      $$ = mkinfplist($1); }
        | PROCEDURE ID dfparams
                    { $$ = mkfplist(mkidlist($2)); }
        | FUNCTION ID dfparams CL ID
                    { $$ = mkfplist(mkidlist($2)); }
        ;

dfparams : /* empty */          { $$ = NULL; }
        | LP fpmlist RP         { $$ = NULL; }

carrays  : ARRAY LB arrinds RB OF ID
                                { $$ = $6;}
        | ARRAY LB arrinds RB OF carrays
                                { $$ = $6; }
        | PACKED ARRAY LB ID SR ID CL ID RB OF ID
                                { $$ = $11; }
        ;

arrinds  : ID SR ID CL ID
                                { $$ = NULL; }
        | arrinds SC ID SR ID CL ID
                                { $$ = NULL; }
        ;

constant : NUMBER               { $$ = NULL; }
        | ID                    { $$ = NULL; }
        | PLUS NUMBER           { $$ = NULL; }
        | MINUS NUMBER          { $$ = NULL; }
        | PLUS ID               { $$ = NULL; }
        | MINUS ID              { $$ = NULL; }
        | LITERAL               { $$ = NULL; }
        ;

typedenoter:   ID               { $$ = mkbasictype($1); }
        | LP idlist RP          { $$ = mkenumeratedtype($2); }
        | constant SR constant  { $$ = mksubrangetype(); }
        | ARRAY LB typelist RB OF typedenoter
                                { $$ = $6; }
        | PACKED ARRAY LB typelist RB OF typedenoter
                                { $$ = $7; }
        | RECORD fieldlist END { $$ = mkrecordtype($2); }
        | PACKED RECORD fieldlist END
                                { $$ = mkrecordtype($3); }
        | SET OF typedenoter
                                { $$ = $3; }
        | PACKED SET OF typedenoter
                                { $$ = $4; }
        | FILE OF typedenoter
                                { $$ = mkfiletype($3); }
        | PACKED FILE OF typedenoter
                                { $$ = mkfiletype($4); }
        | AR ID                 { $$ = mkpointertype($2); }
        ;

typelist : typedenoter          { $$ = NULL; }
```

```
        | typelist CM typedenoter
                                { $$ = NULL; }
        ;

fieldlist: /* empty */          { $$ = NULL; }
        | fixedpart
        | fixedpart SC          { $$ = $1; }
        | fixedpart SC variantpart
                                { $$ = catdeclist($1,$3); }
        | fixedpart SC variantpart SC
                                { $$ = catdeclist($1,$3); }
        | variantpart
        | variantpart SC        { $$ = $1; }
        ;

fixedpart: idlist CL typedenoter
                                { $$ = mkdeclist($1,$3);}
        | fixedpart SC idlist CL typedenoter
                { $$ = catdeclist($1,mkdeclist($3,$5));}
        ;

variantpart:  CASE ID OF variants
            { $$ = catdeclist(mkdeclist(mkidlist($2),NULL),$4); }
        | CASE ID CL ID OF variants
            { $$ = catdeclist(mkdeclist(mkidlist($2),$4),$6); }
        ;

variants : constlist CL LP fieldlist RP
                                { $$ = $4; }
        | variants SC constlist CL LP fieldlist RP
                                { $$ = catdeclist($1,$6); }
         ;

constlist: constant
                                { $$ = NULL; }
        | constlist CM constant
                                { $$ = NULL; }
        ;

stmtseq  : stmts
        | stmtseq
          SC
          stmts
                { operatorcnt(lastndptr($1),";");
                  $$ = linkflow1($1,$3); }
        ;

stmts    : /* empty */
          rtnnewnode
        | stmt  { rmfieldtype(); }
        | rtnnewnode
          NUMBER CL
                { operandcnt($1,mklabelopnd($2));
                  destlist($2,$1);
                  $$ = $1; }
        | rtnnewnode
          NUMBER CL stmt
                { operandcnt($1,mklabelopnd($2));
                  destlist($2,$1);
                  rmfieldtype();
```

```
                $$ = linklabel($1,$4); }
        ;

stmt    : rtnnewnode
          ID ASGN expr
                { operandcnt($1,$2);
                  operatorcnt($1,":=");
                  chkglb($2);
                  if (brroot != NULL) brroot->flag = INVALID;
                  if (withvarstack == NULL)
                      $$ = growdefs($1,mksimpledef($2,$4));
                  else $$ = withgrowdefs($1,$2,NULL,$4); }
        | rtnnewnode
          ID extvars ASGN expr
                { operandcnt($1,$2);
                  operatorcnt($1,":=");
                  chkglb($2);
                  if (brroot != NULL) brroot->flag = INVALID;
                  if (withvarstack == NULL)
                      $$ = growdefs($1,mksimpledef(determobj($2,$3->c),
                                   extraexplist1($2,$3,$5)));
                  else $$ = withgrowdefs($1,$2,$3,$5); }
        | rtnnewnode
          ID
                { operatorcnt($1,$2);
                  if (brroot != NULL) brroot->flag = INVALID;
                      $$ = growprocdefs1($1,$2); }
        | rtnnewnode
          ID
          LP apmlist RP
                { operatorcnt($1,$2);
                  operatorcnt($1,"()");
                  if (brroot != NULL) brroot->flag = INVALID;
                  $$ = growprocdefs2($1,$2,$4); }
        | GOTO NUMBER
                { if ((brroot != NULL) && (brroot->flag == VALID)) {
                      operatorcnt(brroot->nd,"goto");
                      operandcnt(brroot->nd,$2);
                      gotobranch($2,brroot->nd->nid);
                      brroot->flag = INVALID;
                  }
                  else {
                      operatorcnt(ndptr,"goto");
                      operandcnt(ndptr,$2);
                      gotobranch($2,ndptr->nid);
                  }
                  $$ = NULL; }
        | BEGIN
          stmtseq
          END
                { operatorcnt($2,"begin..end");
                  $$ = $2; }
        | rtnnewnode
          IF
          expr
          THEN
          stmts
```

```
  rtnnewnode
        { operatorcnt($1,"if..then");
          $1->pexp = $3;
          if ($5 != NULL) {
              addedge($1->nid,$5->nid);
              if (isgotoflagON($5) == NO)
                  addedge(last($5),$6->nid);
          }
          addedge($1->nid,$6->nid);
          $$ = append($1,linkflow2($5,$6)); }
| rtnnewnode
  IF
  expr
  THEN
  stmts
  { pushbrroot($1); }
  ELSE
  stmts
  rtnnewnode
        { popbrroot();
          operatorcnt($1,"if..then");
          operatorcnt($8,"else");
          $1->pexp = $3;
          if ($5 != NULL) {
              addedge($1->nid,$5->nid);
              if (isgotoflagON($5) == NO)
                  addedge(last($5),$9->nid);
          }
          if ($8!= NULL) {
              addedge($1->nid,$8->nid);
              if (isgotoflagON($8) == NO)
                  addedge(last($8),$9->nid);
          }
          $$ = linkifthenelse($1,$5,$8,$9); }
| rtnnewnode
  { pushbrroot($1); }
  CASE
  expr
  OF
  caselist
  semicolon
  rtnnewnode
  END
        { popbrroot();
          operatorcnt($1,"case..of..end");
          $1->pexp = $4;
          addcasedge($1->nid,$8->nid);
          if ($6 == NULL)  $$ = $1;
          else  $$ = append($1,linkflow2($6,$8)); }
| REPEAT
  stmtseq
  UNTIL
  expr
  rtnnewnode
        { operatorcnt($2,"repeat..until");
          setrepeatpexp($2,$4);
          addedge(last($2),$5->nid);
          addedge(last($2),$2->nid);
```

```
                $$ = linkflow3($2,$5); }
        | rtnnewnode
          WHILE
          expr
          DO
          stmts
          rtnnewnode
                { operatorcnt($1,"while..do");
                  $1->pexp = $3;
                  addedge($1->nid,$5->nid);
                  addedge(last($5),$1->nid);
                  addedge($1->nid,$6->nid);
                  $$ = append($1,linkflow3($5,$6)); }
        | rtnnewnode
          FOR
          ID
          ASGN
          expr
          rtnnewnode
          todownto
          expr
          DO
          stmts
          rtnnewnode
                { operandcnt($1,$3);
                  operatorcnt($1,$7);
                  $1->ldef = mksimpledef($3,$5);
                  $6->pexp = catexplist(mkexptype1($3),$8);
                  attachdef($10,mksimpledef($3,mkexptype1($3)));
                  addedge($1->nid,$6->nid);
                  addedge($6->nid,$10->nid);
                  addedge(last($10),$6->nid);
                  addedge($6->nid,$11->nid);
                  $$ = append($1,append($6,linkflow3($10,$11))); }
        | WITH withvars DO stmts
                { operatorcnt($4,"with..do");
                  popwithvar();
                  $$ = $4; }
        ;

todownto : TO           { $$ = "for..to..do"; }
         | DOWNTO       { $$ = "for..downto..do"; }
         ;

semicolon: /* empty */  { $$ = NULL; }
         | SC           { operatorcnt(ndptr,";"); }
         ;

apmlist  : aparam       { $$ = mkexpseq($1); }
         | apmlist CM aparam
                        { operatorcnt(ndptr,",");
                          $$ = catexpseqlist($1,mkexpseq($3)); }
         ;

aparam   : expr
         | expr CL expr
                        { operatorcnt(ndptr,":");
                          $$ = catexplist($1,$3); }
         | expr CL expr CL expr
```

```
                        { operatorcnt(ndptr,":");
                          operatorcnt(ndptr,":");
                          $$ = catexplist(catexplist($1,$3),$5); }
        | ID            { operandcnt(ndptr,$1);
                          chkglb($1);
                          if (withvarstack == NULL)
                               $$ = mkexptype1($1);
                          else $$ = mkwithexptype($1,NULL); }
        | ID extvars    { operandcnt(ndptr,$1);
                          chkglb($1);
                          mkfieldtype($1,$2);
                          if (withvarstack == NULL)
                               $$ = extraexplist2($1,$2);
                          else $$ = mkwithexptype($1,$2); }
        ;

caselist : caseelem
        | caselist SC caseelem
                        { operatorcnt(ndptr,";");
                          $$ = linkflow3($1,$3); }
        ;

caseelem : caseconsts
          CL
          { brroot->flag = VALID; }
          stmts
                        { caseopcnt($4,$1);
                          if ($4 != NULL) {
                              addedge(CASENID,$4->nid);
                              if (isgotoflagON($4) == NO)
                                  addedge(last($4),CASENID);
                          }
                          $$ = $4; }
        ;

caseconsts: caseconst   { $$ = mkidlist($1); }
        | caseconsts
          CM
          caseconst
                        { $$ = catidlist($1,mkidlist($3)); }
        ;

caseconst: NUMBER       { $$ = mkcaseconstopnd(NULL,$1); }
        | ID            { $$ = mkcaseconstopnd(NULL,$1); }
        | PLUS NUMBER   { $$ = mkcaseconstopnd("+",$2); }
        | MINUS NUMBER  { $$ = mkcaseconstopnd("-",$2); }
        | PLUS ID       { $$ = mkcaseconstopnd("+",$2); }
        | MINUS ID      { $$ = mkcaseconstopnd("-",$2); }
        | LITERAL       { $$ = mkcaseconstopnd(NULL,$1); }
        ;

extvars  : extvar
        | extvars extvar
                        { $$ = mkextralist($1,$2); }
        ;

extvar   : LB exprlist RB
                        { operatorcnt(ndptr,"[]");
                          $$ = mkextraid1($2); }
        | PD ID         { operatorcnt(ndptr,"\.");
```

```
                        operandcnt(ndptr,$2);
                        $$ = mkextraid2($2); }
        | AR            { operatorcnt(ndptr,"\^");
                        $$ = mkextraid2("\^"); }
        ;

exprlist : expr
        | exprlist CM expr
                        { operatorcnt(ndptr,",");
                        $$ = catexplist($1,$3); }
        ;

expr    : ID            { if ((strcmp($1,"eof") == 0) ||
                              (strcmp($1,"eoln") == 0)) {
                              operatorcnt(ndptr,$1);
                              $$ = mkexptype1("input"); }
                        else {
                              operandcnt(ndptr,$1);
                              chkglb($1);
                              if (withvarstack == NULL)
                                  $$ = mkexptype1($1);
                              else $$ = mkwithexptype($1,NULL); }}
        | ID extvars    { operandcnt(ndptr,$1);
                        chkglb($1);
                        if (withvarstack == NULL)
                            $$ = extraexplist2($1,$2);
                        else $$ = mkwithexptype($1,$2); }
        | NUMBER        { operandcnt(ndptr,$1);
                        $$ = mkexptype1($1); }
        | LITERAL       { operandcnt(ndptr,$1);
                        $$ = mkexptype1($1); }
        | NIL           { operandcnt(ndptr,"nil");
                        $$ = mkexptype1("nil"); }
        | ID LP plist RP
                        { operatorcnt(ndptr,$1);
                        operatorcnt(ndptr,"()");
                        $$ = mkfunctionexp($1,$3); }
        | LB setelems RB
                        { operatorcnt(ndptr,"[]");
                        $$ = $2; }
        | LP expr RP    { operatorcnt(ndptr,"()");
                        $$ = $2; }
        | expr PLUS expr
                        { operatorcnt(ndptr,"+");
                        $$ = catexplist($1,$3); }
        | expr MINUS expr
                        { operatorcnt(ndptr,"-");
                        $$ = catexplist($1,$3); }
        | expr MULTIPLY expr
                        { operatorcnt(ndptr,"*");
                        $$ = catexplist($1,$3); }
        | expr DIVIDE expr
                        { operatorcnt(ndptr,"/");
                        $$ = catexplist($1,$3); }
        | expr DIV expr
```

```
                        { operatorcnt(ndptr,"div");
                          $$ = catexplist($1,$3); }
        | expr MOD expr
                        { operatorcnt(ndptr,"mod");
                          $$ = catexplist($1,$3); }
        | expr EQ expr
                        { operatorcnt(ndptr,"=");
                          $$ = catexplist($1,$3); }
        | expr NE expr
                        { operatorcnt(ndptr,"<>");
                          $$ = catexplist($1,$3); }
        | expr LT expr
                        { operatorcnt(ndptr,"<");
                          $$ = catexplist($1,$3); }
        | expr GT expr
                        { operatorcnt(ndptr,">");
                          $$ = catexplist($1,$3); }
        | expr LE expr
                        { operatorcnt(ndptr,"<=");
                          $$ = catexplist($1,$3); }
        | expr GE expr
                        { operatorcnt(ndptr,">=");
                          $$ = catexplist($1,$3); }
        | expr AND expr
                        { operatorcnt(ndptr,"and");
                          $$ = catexplist($1,$3); }
        | expr OR expr
                        { operatorcnt(ndptr,"or");
                          $$ = catexplist($1,$3); }
        | expr IN expr
                        { operatorcnt(ndptr,"in");
                          $$ = catexplist($1,$3); }
        | PLUS expr     %prec UNARYPLUS
                        { operatorcnt(ndptr,"+");
                          $$ = $2; }
        | MINUS expr    %prec UNARYMINUS
                        { operatorcnt(ndptr,"-");
                          $$ = $2; }
        | NOT expr      { operatorcnt(ndptr,"not");
                          $$ = $2; }
        ;

plist   : p             { $$ = mkexpseq($1); }
        | plist CM p    { operatorcnt(ndptr,",");
                          $$ = catexpseqlist($1,mkexpseq($3)); }
        ;

p       : expr
        | ID extvars    { operandcnt(ndptr,$1);
                          chkglb($1);
                          if (withvarstack == NULL)
                               $$ = extraexplist2($1,$2);
                          else $$ = mkwithexptype($1,$2); }
        | ID            { operandcnt(ndptr,$1);
```

```
                                chkglb($1);
                                if (withvarstack == NULL)
                                $$ = mkexptype1($1);
                                else $$ = mkwithexptype($1,NULL); }
        ;

setelems : /* empty */  { $$ = NULL; }
        | elemlist
        ;

elemlist : elem
        | elemlist CM elem
                        { operatorcnt(ndptr,",");
                          $$ = catexplist($1,$3); }
        ;

elem    : expr
        | expr SR expr
                        { operatorcnt(ndptr,"..");
                          $$ = catexplist($1,$3); }
        ;

idlist  : ID            { operandcnt(ndptr,$1);
                          $$ = mkidlist($1); }
        | idlist CM ID
                        { operatorcnt(ndptr,",");
                          operandcnt(ndptr,$3);
                          $$ = catidlist($1,mkidlist($3)); }
        ;

withvars : withvar      { pushwithvar($1,ON);
                          $$ = NULL; }
        | withvars CM withvar
                        { operatorcnt(",");
                          pushwithvar($3,OFF);
                          $$ = NULL; }

withvar  : ID           { operandcnt(ndptr,$1);
                          chkglb($1);
                          $$ = mkwithvars($1,NULL); }
        | ID extvars
                        { operandcnt(ndptr,$1);
                          chkglb($1);
                          $$ = mkwithvars($1,$2); }
        ;

/* auxiliary grammar */

rtnnewnode:             { $$ = newnode(); }
        ;
%%

#include <stdio.h>
#include <ctype.h>
#include <strings.h>

UnitRepType *unitrep = NULL, *unitstack = NULL;
NodeType *ndptr = NULL;
MemoryTable *memtab = NULL;
LocalDef *newdef = NULL;
IdListType *glbls = NULL;
```

```
WithVarType *withvarstack = NULL;
BrRootStack *brroot = NULL;
char *fieldtype = NULL;
int n = 0;     /* node id counter */
main()
{
     yyparse();
     codegenerate();
}

char *emalloc(i)
     int i;
{
     char *m, *malloc();

     m = (char *) malloc(i+(4-i%4));
     return m;
}

yyerror(s)
     char *s;
{

}


/***** lex.l ***************************************************************/
%{
#include "sr.h"
#include "y.tab.h"
%}
delim                   [ \t\n]
ws                      {delim}+
letter                  [A-Za-z]
digit                   [0-9]
id                      {letter}({letter}|{digit})*
number                  {digit}+(\.{digit}+)?(e[+-]?{digit}+)?
moreliteral             '[^'\n]*'/'
literal                 '[^'\n]*'

%%
{ws}                    {;}
program                 {return PROGRAM;}
label                   {return LABEL;}
const                   {return CONST;}
type                    {return TYPE;}
procedure               {return PROCEDURE;}
function                {return FUNCTION;}
var                     {return VAR;}
begin                   {return BEGIN;}
end                     {return END;}
array                   {return ARRAY;}
file                    {return FILE;}
record                  {return RECORD;}
set                     {return SET;}
packed                  {return PACKED;}
case                    {return CASE;}
of                      {return OF;}
for                     {return FOR;}
to                      {return TO;}
```

```
downto                  {return DOWNTO;}
do                      {return DO;}
if                      {return IF;}
then                    {return THEN;}
else                    {return ELSE;}
repeat                  {return REPEAT;}
until                   {return UNTIL;}
while                   {return WHILE;}
with                    {return WITH;}
goto                    {return GOTO;}
forward                 {return FORWARD;}
nil                     {return NIL;}
"+"                     {return PLUS; }
"-"                     {return MINUS; }
"*"                     {return MULTIPLY; }
"/"                     {return DIVIDE; }
div                     {return DIV;}
mod                     {return MOD;}
"="                     {return EQ;}
"<"                     {return LT;}
">"                     {return GT;}
"<>"                    {return NE;}
"<="                    {return LE;}
">="                    {return GE;}
and                     {return AND;}
or                      {return OR;}
not                     {return NOT;}
in                      {return IN;}
"["                     {return LB;}
"]"                     {return RB;}
"."                     {return PD;}
","                     {return CM;}
":"                     {return CL;}
";"                     {return SC;}
"^"                     {return AR;}
"("                     {return LP;}
")"                     {return RP;}
":="                    {return ASGN;}
".."                    {return SR;}
"'"                     {return SL;}
{id}                    {char *emalloc();
                         yylval.str = (char *) emalloc(strlen(yytext)+1);
                         strcpy(yylval.str,yytext);
                         return ID;
                        }
{number}                {char *emalloc();
                         yylval.str = (char *) emalloc(strlen(yytext)+1);
                         strcpy(yylval.str,yytext);
                         return NUMBER;
                        }
{moreliteral}           {yymore();}
{literal}               {char *emalloc();
                         yylval.str = (char *) emalloc(strlen(yytext)+1);
                         strcpy(yylval.str,yytext);
                         return LITERAL;
                        }

/***** cnstrct.c ************************************************************/
```

```
#include "sr.h"
#include "y.tab.h"
#define         REWRITE         1
#define         RESET           2
#define         PUT             3
#define         GET             4
#define         READ            5
#define         READLN          6
#define         WRITE           7
#define         WRITELN         8
#define         PAGE            9
#define         NEW             10
#define         DISPOSE         11
#define         PACK            12
#define         UNPACK          13
#define         USERDEFINED     -1

extern UnitRepType *unitrep, *unitstack;
extern NodeType *ndptr;
extern IdListType *glbls;
extern WithVarType *withvarstack;
extern BrRootStack *brroot;
extern char *fieldtype;
extern int n;
extern short brrootflag, labelbefore;

initunitrep()
{
     UnitRepType *p;
     HeaderType *h;
     UnitFlowStruct *f;
     char *emalloc();

     p = (UnitRepType *) emalloc(sizeof(UnitRepType));
     h = (HeaderType *) emalloc(sizeof(HeaderType));
     h->pname = NULL;
     h->fparams = NULL;
     h->globals = NULL;
     p->interface = h;
     f = (UnitFlowStruct *) emalloc(sizeof(UnitFlowStruct));
     f->nd = NULL;
     f->eg = NULL;
     f->s = NULL;
     p->ufs = f;
     p->next = NULL;
     if (unitstack == NULL) unitstack = p;
     else {
          p->next = unitstack;
          unitstack = p;
     }
}

UnitRepType *addunitrep(id,p)
     char *id;
     FPListType *p;
{
     UnitRepType *t1, *t;
     IdListType *n, *mkidset();

     unitstack->interface->pname = (char *) emalloc(strlen(id)+1);
     strcpy(unitstack->interface->pname,id);
```

```
        unitstack->interface->fparams = p;
        unitstack->interface->globals = mkidset(glbls);
        t1 = unitstack;
        unitstack = unitstack->next;
        t1->next = NULL;
        if (unitrep == NULL)  unitrep = t1;
        else {
             for (t = unitrep; t->next != NULL; t = t->next);
             t->next = t1;
        }
        ndptr = NULL;
        glbls = NULL;
        return unitrep;
}

FPListType *mkfplist(p)
        IdListType *p;
{
        FPListType *f;
        char *emalloc();

        f = (FPListType *) emalloc(sizeof(FPListType));
        f->fp = p;
        f->in = NULL;
        return f;
}

FPListType *mkinfplist(p)
        IdListType *p;
{
        FPListType *f;
        IdListType *t, *l = NULL, *catidlist(), *mkidlist();
        char *emalloc();

        f = (FPListType *) emalloc(sizeof(FPListType));
        for (t = p; t != NULL; t = t->next)
             l = catidlist(l,mkidlist(t->id));
        f->fp = p;
        f->in = l;
        return f;
}

FPListType *catfplist(p1,p2)
        FPListType *p1, *p2;
{
        IdListType *t;

        if (p1 == NULL) return p1;
        else if (p2 == NULL) return p2;
        else if (p1->fp == NULL)  p1->fp = p2->fp;
        else {
             for (t = p1->fp; t->next != NULL; t = t->next);
             t->next = p2->fp;
        }
        if (p1->in == NULL)  p1->in = p2->in;
        else {
             for (t = p1->in; t->next != NULL; t = t->next);
             t->next = p2->in;
        }
```

```
    return p1;
}

isvalueparam(s,vp)
    char *s;
    IdListType *vp;
{
    IdListType *v;

    if (vp == NULL) return NO;
    for (v = vp; v != NULL; v = v->next)
        if (strcmp(v->id,s) == 0) return YES;
    return NO;
}

NodeType *newnode()
{
    NodeType *np;
    char *emalloc();

    np = (NodeType *) emalloc(sizeof(NodeType));
    np->nid = ++n;
    np->ldef = NULL;
    np->pexp = NULL;
    np->operators = NULL;
    np->operands = NULL;
    np->gt = OFF;
    np->next = NULL;
    ndptr = np;
    return np;
}

LocalDef *mksimpledef(id,u)
    char *id;
    ExpType *u;
{
    LocalDef *d;
    SimpleDef *s;
    char *emalloc();

    d = (LocalDef *) emalloc(sizeof(LocalDef));
    s = (SimpleDef *) emalloc(sizeof(SimpleDef));
    s->varid = (char *) emalloc(strlen(id)+1);
    strcpy(s->varid,id);
    s->sexp = u;
    d->def.sdef = s;
    d->tag = SDEFTAG;
    d->next = NULL;
    return d;
}

LocalDef *mkprocdef(id,u)
    char *id;
    ExpSequence *u;
{
    LocalDef *d;
    ProcDef *p;
    char *emalloc();

    d = (LocalDef *) emalloc(sizeof(LocalDef));
```

```
     p = (ProcDef *) emalloc(sizeof(ProcDef));
     p->procid = (char *) emalloc(strlen(id)+1);
     strcpy(p->procid,id);
     p->pexp = u;
     d->def.pdef = p;
     d->tag = PDEFTAG;
     d->next = 0;
     return d;
}

NodeType *growdefs(n,d)
     NodeType *n;
     LocalDef *d;
{
     LocalDef *t;

     if (n->ldef == NULL) n->ldef = d;
     else {
          for (t = n->ldef; t->next != NULL; t = t->next);
          t->next = d;
     }
     return n;
}

NodeType *growprocdefs1(n,s)
     NodeType *n;
     char *s;
{
     int p;
     ExpType *exp, *catexplist(), *mkexptype1();
     LocalDef *mksimpledef();

     if (strcmp(s,"writeln") == 0)  p = WRITELN;
     else if (strcmp(s,"readln") == 0)  p = READLN;
     else if (strcmp(s,"page") == 0)  p = PAGE;
     else p = USERDEFINED;
     switch (p) {
     case WRITELN:
          exp = catexplist(mkexptype1("output"),
                           mkexptype1("end-of-line"));
          return growdefs(n,mksimpledef("output",exp));
          break;
     case READLN:
          return growdefs(n,mksimpledef("input",
                                        mkexptype1("input")));
          break;
     case PAGE:
          exp = catexplist(mkexptype1("output"),
                           mkexptype1("end-of-page"));
          return growdefs(n,mksimpledef("output",exp));
          break;
     default:
          return growdefs(n,mkprocdef(s,NULL));
          break;
     }
}

NodeType *growprocdefs2(n,s,u)
     NodeType *n;
     char *s;
```

```
    ExpSequence *u;
{
    int p;
    char *id, *idw, *pid, *a, *b, *c, *emalloc();
    char *mkbuffervar(), *object();
    ExpType *ulist, *exp, *last, *mkexptype1(), *catexplist();
    ExpType *objexp, *fieldexp;
    ExpSequence *u1;
    LocalDef *mksimpledef(), *mkprocdef();
    NodeType *growdefs();

    if (strcmp(s,"rewrite") == 0)  p = REWRITE;
    else if (strcmp(s,"reset") == 0)  p = RESET;
    else if (strcmp(s,"put") == 0)  p = PUT;
    else if (strcmp(s,"get") == 0)  p = GET;
    else if (strcmp(s,"read") == 0)  p = READ;
    else if (strcmp(s,"readln") == 0)  p = READLN;
    else if (strcmp(s,"write") == 0)  p = WRITE;
    else if (strcmp(s,"writeln") == 0)  p = WRITELN;
    else if (strcmp(s,"page") == 0)  p = PAGE;
    else if (strcmp(s,"new") == 0)  p = NEW;
    else if (strcmp(s,"dispose") == 0)  p = DISPOSE;
    else if (strcmp(s,"pack") == 0)  p = PACK;
    else if (strcmp(s,"unpack") == 0)  p = UNPACK;
    else p = USERDEFINED;
    ulist = seqtoexp(u);
    id = ulist->exp.id;
    idw = mkbuffervar(id);
    switch (p) {
    case REWRITE:          /* rewrite(F) */
        return growdefs(n,mksimpledef(id,NULL));
        break;
    case PUT:              /* put(F) */
        exp = catexplist(mkexptype1(id),mkexptype1(idw));
        return growdefs(n,mksimpledef(id,exp));
        break;
    case RESET:
    case GET:              /* reset(F) or get(F) */
        return growdefs(n,mksimpledef(idw,u->exp));
        break;
    case READLN:
    case READ:
        if (isfilevar(id) == 0)  { /* standard input */
            id = "input";
            idw = "input^";
        }
        else  u = u->next;
        if (u == NULL)         /* readln(F) */
            return growdefs(n,mksimpledef(idw,mkexptype1(id)));
        /* read(F,V), readln(F,V), read(V), readln(V),
           read(F,V1,V2,...,Vn), readln(F,V1,V2,...,Vn),
           read(V1,V2,...,Vn), or readln(V1,V2,...,Vn) */
        for (u1 = u; u1 != NULL; u1 = u1->next) {
            if (withvarstack == NULL)
                n = growdefs(n,mksimpledef(object(u1->exp->exp.id),
                    catexplist(mkexptype1(idw),u1->exp->next)));
```

```
            else n = growdefs(n,mksimpledef(object(u1->exp->exp.id),
                     catexplist(mkexptype1(object(u1->exp->exp.id)),
                     catexplist(mkexptype1(idw),u1->exp->next))));
            n = growdefs(n,mksimpledef(idw,mkexptype1(id)));
        }
        return n;
        break;
case WRITELN:
        if (isfilevar(id) == 0)  { /* standard output */
            id = "output";
            idw = "output^";
        }
        else  u = u->next;
        if (u == NULL)      /* writeln(F) */
            return growdefs(n,mksimpledef(id,
                            catexplist(mkexptype1(id),
                            mkexptype1("end-of-line"))));
        if (u->next == NULL) {  /* writeln(E) or writeln(F,E) */
            n = growdefs(n,mksimpledef(idw,u->exp));
            n = growdefs(n,mksimpledef(id,
                         catexplist(mkexptype1(id),mkexptype1(idw))));
            return growdefs(n,mksimpledef(id,catexplist(mkexptype1(id),
                                          mkexptype1("end-of-line"))));
        }
        else { /* writeln(E1,E2,..,En) or writeln(F,E1,E2,..En) */
            exp = u->exp;
            for (u1 = u->next; u1->next != NULL; u1 = u1->next)
                exp = catexplist(exp,u1->exp);
            last = u1->exp;
            exp = catexplist(exp,last);
            n = growdefs(n,mksimpledef(id,catexplist(mkexptype1(id),exp)));
            n = growdefs(n,mksimpledef(idw,last));
            return growdefs(n,mksimpledef(id,catexplist(mkexptype1(id),
                                          mkexptype1("end-of-line"))));
        }
        break;
case WRITE:
        if (isfilevar(id) == 0)  {  /* standard output */
            id = "output";
            idw = "output^";
        }
        else  u = u->next;
        if (u->next == NULL) {   /* write(F,E) or write(E) */
            n = growdefs(n,mksimpledef(idw,u->exp));
            return growdefs(n,mksimpledef(id,catexplist(mkexptype1(id),
                                                mkexptype1(idw))));
        }
        else { /* write(F,E1,E2,..En) or write(E1,E2,..,En) */
            exp = u->exp;
            for (u1 = u->next; u1->next != NULL; u1 = u1->next)
                exp = catexplist(exp,u1->exp);
            last = u1->exp;
            exp = catexplist(exp,last);
            n = growdefs(n,mksimpledef(id,catexplist(mkexptype1(id),exp)));
            return growdefs(n,mksimpledef(idw,last));
        }
        break;
```

```
case PAGE:          /* page(F) */
     return growdefs(n,mksimpledef(id,catexplist(mkexptype1(id),
                                   mkexptype1("end-of-page"))));
     break;
case NEW:
     if (u->exp->next != NULL)  pid = object(id);
     else  pid = id;
     objexp = mkexptype1(object(id));
     fieldexp = mkexptype1(fieldtype);
     if (u->next == NULL) {     /* new(P) */
          if (fieldtype != NULL) {
              if (strcmp(object(id),fieldtype) == 0)
                   return growdefs(n,mksimpledef(object(id),
                                        mkexptype1(object(id))));
              else {
                   n = growdefs(n,mksimpledef(object(id),objexp));
                   return growdefs(n,mksimpledef(fieldtype,
                                   catexplist(fieldexp,objexp)));
              }
          }
          else {
               n = growdefs(n,mksimpledef(pid,NULL));
               return growdefs(n,mksimpledef(object(id),
                               catexplist(mkexptype1(pid),objexp)));
          }
     }
     else {  /* new(P,C1,C2,..,Cn) */
          exp = NULL;
          for (u1 = u->next; u1 != NULL; u1 = u1->next)
              exp = catexplist(exp,mkexptype1(u1->exp->exp.id));
          if (fieldtype != NULL) {
               if (strcmp(object(id),fieldtype) == 0)
                    return growdefs(n,mksimpledef(object(id),
                                    catexplist(objexp,exp)));
               else {
                    n = growdefs(n,mksimpledef(object(id),
                                        mkexptype1(object(id))));
                    return growdefs(n,mksimpledef(fieldtype,
                           catexplist(fieldexp,catexplist(objexp,exp))));
               }
          }
          else {
               exp = catexplist(catexplist(mkexptype1(pid),objexp),exp);
               n = growdefs(n,mksimpledef(pid,NULL));
               return growdefs(n,mksimpledef(object(id),exp));
          }
     }
     break;
case DISPOSE:
     if (u->exp->next != NULL)  pid = object(id);
     else  pid = id;
     objexp = mkexptype1(object(id));
     fieldexp = mkexptype1(fieldtype);
     if (u->next == NULL) {    /* dispose(P) */
          if (fieldtype != NULL) {
               if (strcmp(object(id),fieldtype) == 0)
```

```
                    return growdefs(n,mksimpledef(object(id),objexp));
               else {
                    n = growdefs(n,mksimpledef(fieldtype,
                              catexplist(fieldexp,objexp)));
                    return growdefs(n,mksimpledef(object(id),objexp));
               }
          }
          else {
               n = growdefs(n,mksimpledef(object(id),
                         catexplist(mkexptype1(pid),objexp)));
               return growdefs(n,mksimpledef(pid,mkexptype1("nil")));
          }
     }
     else {  /* dispose(P,C1,C2,..,Cn) */
          exp = NULL;
          for (u1 = u->next; u1 != NULL; u1 = u1->next)
               exp = catexplist(exp,mkexptype1(u1->exp->exp.id));
          if (fieldtype != NULL) {
               if (strcmp(object(id),fieldtype) == 0)
                    return growdefs(n,mksimpledef(object(id),
                                   catexplist(objexp,exp)));
               else {
                    n = growdefs(n,mksimpledef(fieldtype,
                        catexplist(fieldexp,catexplist(objexp,exp))));
                    return growdefs(n,mksimpledef(object(id),
                                   mkexptype1(object(id))));
               }
          }
          else {
               exp = catexplist(catexplist(mkexptype1(pid),objexp),exp);
               n = growdefs(n,mksimpledef(object(id),exp));
               return growdefs(n,mksimpledef(pid,mkexptype1("nil")));
          }
     }
     break;
case PACK:     /* pack(A,B,C) */
     a = (char *) emalloc(strlen(id)+1);
     strcpy(a,id);
     u = u->next;
     b = (char *) emalloc(strlen(u->exp->exp.id)+1);
     strcpy(b,u->exp->exp.id);
     u = u->next;
     c = (char *) emalloc(strlen(u->exp->exp.id)+1);
     strcpy(c,u->exp->exp.id);
     return growdefs(n,mksimpledef(c,catexplist(mkexptype1(a),
                                        mkexptype1(b))));
     break;
case UNPACK:    /* unpack(A,B,C) */
     a = (char *) emalloc(strlen(id)+1);
     strcpy(a,id);
     u = u->next;
     b = (char *) emalloc(strlen(u->exp->exp.id)+1);
     strcpy(b,u->exp->exp.id);
     u = u->next;
     c = (char *) emalloc(strlen(u->exp->exp.id)+1);
     strcpy(c,u->exp->exp.id);
     return growdefs(n,mksimpledef(b,catexplist(mkexptype1(a),
```

```
                                                    mkexptype1(c))));
            break;
        default:    /* expr defined procedure */
            return growdefs(n,mkprocdef(s,u));
            break;
        }
}

mkfieldtype(id,ex)
        char *id;
        ExtraIdType *ex;
{
        char *fobj;

        if (ex->c != NULL) {
            fobj = fieldobject(ex,id);
            fieldtype = (char *) emalloc(strlen(fobj)+1);
            strcpy(fieldtype,fobj);
        }
}

rmfieldtype()
{
        char *t;

        t = fieldtype;
        fieldtype = NULL;
        free(t);
}

attachdef(n,d)
        NodeType *n;
        LocalDef *d;
{
        NodeType *t;

        for (t = n; t->next != NULL; t = t->next);
        growdefs(t,d);
}

setrepeatpexp(n,u)
        NodeType *n;
        ExpType *u;
{
        NodeType *t;

        for (t = n; t->next != NULL; t = t->next);
        t->pexp = u;
}

WithVarType *mkwithvars(s,x)
        char *s;
        ExtraIdType *x;
{
        WithVarType *w;

        w = (WithVarType *) emalloc(sizeof(WithVarType));
        w->id = (char *) emalloc(strlen(s)+1);
        strcpy(w->id,s);
        w->ext = x;
        w->borderline = OFF;
```

```
    w->next = NULL;
    return w;
}

pushwithvar(w,b)
    WithVarType *w;
    short b;
{
    if (b == ON)  w->borderline = ON;
    if (withvarstack == NULL)  withvarstack = w;
    else {
        w->next = withvarstack;
        withvarstack = w;
    }
}

popwithvar()
{
    WithVarType *w;

    for (w = withvarstack; w->borderline != ON; w = w->next);
    withvarstack = w->next;
}

NodeType *withgrowdefs(n,s,x,u)
    NodeType *n;
    char *s;
    ExtraIdType *x;
    ExpType *u;
{
    WithVarType *ws;
    char *key, *keyobj;
    ExtraIdType *keyextra = NULL;
    short found = NO;

    for (ws = withvarstack;
        (ws != NULL) && (found == NO);
        ws = ws->next) {
        if ((ws->ext == NULL) && (isfamily(s,ws->id) == 1)) {
            found = YES;
            key = ws->id;
        }
        else if ((ws->ext != NULL) &&
                 (isfamily(s,determobj(ws->id,ws->ext->c)) == 1)) {
            found = YES;
            key = ws->id;
            keyobj = determobj(ws->id,ws->ext->c);
            keyextra = ws->ext;
        }
    }
    if (found == NO) {
        if (x == NULL)
            return growdefs(n,mksimpledef(s,u));
        else return growdefs(n,mksimpledef(determobj(s,x->c),
                             extraexplist1(s,x,u)));
    }
    else {    /* found == YES */
        if (x == NULL) {
            if (keyextra == NULL)
                return growdefs(n,mksimpledef(key,
                                catexplist(mkexptype1(key),u)));
```

```
                else if ((keyextra->u != NULL) && (keyextra->c == NULL))
                     return growdefs(n,mksimpledef(
                                     key,
                                     extraexplist1(key,keyextra,u)));
                else return growdefs(n,mksimpledef(
                                     keyobj,
                                     extraexplist1(key,keyextra,u)));
           }
           else {
                if (keyextra == NULL)
                     return growdefs(n,mksimpledef(key,
                                     catexplist(mkexptype1(key),
                                                catexplist(x->u,u))));
                else if ((keyextra->u != NULL) && (keyextra->c == NULL))
                     return growdefs(n,mksimpledef(
                                     key,
                                     extraexplist1(key,keyextra,
                                                   catexplist(x->u,u))));
                else return growdefs(n,mksimpledef(
                                     keyobj,
                                     extraexplist1(key,keyextra,
                                                   catexplist(x->u,u))));
           }
     }
}

ExpType *mkwithexptype(s,x)
     char *s;
     ExtraIdType *x;
{
     WithVarType *ws;
     char *key;
     ExtraIdType *keyextra;
     short found = NO;

     for (ws = withvarstack;
          (ws != NULL) && (found ==NO);
          ws = ws->next)
          if (isfamily(s,ws->id) == 1)  {
               found = YES;
               key = ws->id;
               keyextra = ws->ext;
          }
     if (found == NO) {
          if (x == NULL)
               return mkexptype1(s);
          else return extraexplist2(s,x);
     }
     else {     /* found == YES */
          if (x == NULL)
               return extraexplist2(key,keyextra);
          else
               return catexplist(extraexplist2(key,keyextra),
                                 x->u);
     }
}

NodeType *lastndptr(n)
```

```
     NodeType *n;
{
     NodeType *t;

     for (t = n; t->next != NULL; t = t->next);
     return t;
}

NodeType *append(n1,n2)
     NodeType *n1, *n2;
{
     NodeType *t;

     for (t = n1; t->next != NULL; t = t->next);
     t->next = n2;
     return n1;
}

NodeType *link1(n1,n2)
     NodeType *n1, *n2;
{
     NodeType *t, *t1;
     int dead;

     if (n1->next == NULL) {
          move1(n1,n2);
          if (n2->next == NULL) ndptr = n1;   /* for goto */
          n1->next = n2->next;
          updatedge1(n1->nid,n2->nid);
          free(n2);
          return n1;
     }
     else {
          for (t = n1; t->next != NULL; t = t->next);
          for (t1 = n1; t1->next->next != NULL; t1= t1->next);
          dead = t->nid;
          move2(n2,t);
          t1->next = n2;
          free(t);
          updatedge1(n2->nid,dead);
          return n1;
     }
}

NodeType *link2(n1,n2)
     NodeType *n1, *n2;
{
     NodeType *t, *t1;
     int dead;

     for (t = n1; t->next != NULL; t = t->next);
     for (t1 = n1; t1->next->next != NULL; t1 = t1->next);
     dead = t->nid;
     move2(n2,t);
     t1->next = n2;
     free(t);
     updatedge2(dead);
     return n1;
}

NodeType *linkflow1(n1,n2)
```

```
     NodeType *n1, *n2;
{
     NodeType *t;
     short found1 = 0, found2 = 0;
     OpCount *p;

     for (t = n1; t->next != NULL; t = t->next);
     if (n2 == NULL) {
          t->gt = ON;
          return n1;
     }
     else if (t->gt == ON) {
          t->gt = OFF;
          return append(n1,n2);
     }
     else {
          for (p = n2->operators;
               ((found1 == 0) && (p != NULL));
               p = p->next)
               if ((strcmp(p->name,"repeat..until") == 0) ||
                   (strcmp(p->name,"while..do") == 0))
                    found1 = 1;
          for (p = n2->operands;
               ((found2 == 0) && (p != NULL));
               p = p->next)
               if ((strlen(p->name) > 1) &&
                   (p->name[strlen(p->name)-1] == ':'))
                    found2 = 1;
          if (((t->ldef != NULL) && (found1 == 1)) ||
              ((t->ldef != NULL) && (found2 == 1))) {
               addedge(last(n1),n2->nid);
               return append(n1,n2);
          }
          else return link1(n1,n2);
     }
}

NodeType *linkflow2(n1,n2)
     NodeType *n1, *n2;
{
     NodeType *t;

     if (n1 == NULL) return n2;
     else {
          for (t = n1; t->next != NULL; t = t->next);
          if (t->ldef != NULL) return append(n1,n2);
          else  return link2(n1,n2);
     }
}

NodeType *linkflow3(n1,n2)
     NodeType *n1, *n2;
{
     NodeType *t, *t1;

     if (n1 == NULL) return n2;
     else if (n2 == NULL) return n1;
     else {
  if (n1->next == NULL)  return append(n1,n2);
```

```
          else {
               for (t = n1; t->next->next != NULL; t = t->next);
               for (t1 = n1; t1->next != NULL; t1 = t1->next);
               if ((t1->ldef != NULL) || (t1->pexp != NULL))
                    return append(n1,n2);
               else {
                    move1(t,t1);
                    updatedge2(t1->nid);
                    t->next = n2;
                    free(t1);
                    return n1;
               }
          }
     }
}

NodeType *linklabel(n1,n2)
     NodeType *n1, *n2;
{
     NodeType *lastndptr();

     if (n2 == NULL)     /* n2 is goto statement */
          return n1;
     else {
          move1(n1,n2);
          ndptr = lastndptr(n1);
          n1->next = n2->next;
          updatedge1(n1->nid,n2->nid);
          free(n2);
          return n1;
     }
}

NodeType *linkifthenelse(n1,n2,n3,n4)
     NodeType *n1, *n2, *n3, *n4;
{
     if ((n2 == NULL) && (n3 == NULL)) {
          free(n4);
          ndptr = n1;
          return n1;
     }
     else if ((n2 != NULL) && (n3 == NULL)) {
          if (isgotoflagON(n2) == YES) {
               free(n4);
               ndptr = n2;
               return append(n1,n2);
          }
          else return append(n1,linkflow2(n2,n4));
     }
     else if ((n2 == NULL) && (n3 != NULL)) {
          if (isgotoflagON(n3) == YES) {
               free(n4);
               ndptr = n3;
               return append(n1,n3);
          }
          else return append(n1,linkflow2(n3,n4));
     }
     else {
          if ((isgotoflagON(n2) == YES) &&
              (isgotoflagON(n3) == YES)) {
```

```
                free(n4);
                ndptr = n3;
                return append(n1,linkflow2(n2,n3));
           }
           else return append(n1,linkflow2(n2,linkflow2(n3,n4)));
      }
}

move1(n1,n2)
      NodeType *n1, *n2;
{
      NodeType *tndptr;
      LocalDef *t1;
      ExpType *t2;
      OpCount *t3, *t4;
      int c;

      if (n1->ldef == NULL)  n1->ldef = n2->ldef;
      else {
           for (t1 = n1->ldef; t1->next != NULL; t1 = t1->next);
           t1->next = n2->ldef;
      }
      if (n1->pexp == NULL)  n1->pexp = n2->pexp;
      else {
           for (t2 = n1->pexp; t2->next != NULL; t2 = t2->next);
           t2->next = n2->pexp;
      }
      if (n1->operators == NULL) n1->operators = n2->operators;
      else if (n2->operators != NULL) {
           for (t3 = n2->operators; t3 != NULL; t3 = t3->next)
                for (c = t3->occur; c > 0; c--)
                     operatorcnt(n1,t3->name);
      }
      if (n1->operands == NULL)  n1->operands = n2->operands;
      else if (n2->operands != NULL) {
           for (t4 = n2->operands; t4 != NULL; t4 = t4->next)
                for (c = t4->occur; c > 0; c--)
                     operandcnt(n1,t4->name);
      }
}

move2(n1,n2)
      NodeType *n1, *n2;
{
      NodeType *tndptr;
      LocalDef *t1;
      ExpType *t2;
      OpCount *t3, *t4;
      int c;

      if (n2 != NULL) {
           if (n1->ldef == NULL)  n1->ldef = n2->ldef;
           else if (n2->ldef != NULL) {
                for (t1 = n2->ldef; t1->next != NULL; t1 = t1->next);
                t1->next = n1->ldef;
                n1->ldef = n2->ldef;
           }
           if (n1->pexp == NULL) n1->pexp = n2->pexp;
           else if (n2->pexp != NULL) {
                for (t2 = n2->pexp; t2->next != NULL; t2 = t2 >next);
```

```
                t2->next = n1->pexp;
                n1->pexp = n2->pexp;
           }
           if (n1->operators = NULL) n1->operators = n2->operators;
           else if (n2->operators != NULL) {
                for (t3 = n2->operators; t3 != NULL; t3 = t3->next)
                     for (c = t3->occur; c > 0; c--)
                          operatorcnt(n1,t3->name);
           }
           if (n1->operands == NULL) n1->operands = n2->operands;
           else if (n2->operands != NULL) {
                for (t4 = n2->operands; t4 != NULL; t4 = t4->next)
                     for (c = t4->occur; c > 0; c--)
                         operandcnt(n1,t4->name);
           }
      }
}

last(n)
     NodeType *n;
{
     NodeType *t;

     for (t = n; t->next != NULL; t = t->next);
     return t->nid;
}

rmlastnode(n)     /* remove last empty node if it is empty */
     NodeType *n;
{
     NodeType *t, *t1;

     if (n->next != NULL) {
          for (t = n; t->next != NULL; t = t->next);
          for (t1 = n; t1->next->next != NULL; t1 = t1->next);
          if ((t->ldef == NULL) && (t->pexp == NULL)) {
               move1(t1,t);
               t1->next = NULL;
               free(t);
          }
     }
}

pushbrroot(n)
     NodeType *n;
{
     BrRootStack *t;
     char *emalloc();

     t = (BrRootStack *) emalloc(sizeof(BrRootStack));
     t->nd = n;
     t->flag = VALID;
     t->next = NULL;
     if (brroot == NULL) brroot = t;
     else { t->next = brroot;
            brroot = t;
     }
}

popbrroot()
```

```
{
     BrRootStack *t;

     t = brroot;
     brroot = brroot->next;
     free(t);
}


/***** memory.c ************************************************************/
#include "sr.h"
#include "y.tab.h"

extern MemoryTable *memtab;
extern IdListType *glbls;

pushmemtab()
{
     MemoryTable *mem;
     char *emalloc();

     mem = (MemoryTable *) emalloc(sizeof(MemoryTable));
     mem->lab = NULL;
     mem->const = NULL;
     mem->type = NULL;
     mem->var = NULL;
     mem->next = NULL;
     if (memtab == NULL)  memtab = mem;
     else {
          mem->next = memtab;
          memtab = mem;
     }
}

popmemtab()
{
     MemoryTable *tmp;

     tmp = memtab;
     memtab = memtab->next;
     free(tmp);
}

LabelTable *setlabtab(n)
     char *n;
{
     LabelTable *lab;
     char *emalloc();

     lab = (LabelTable *) emalloc(sizeof(LabelTable));
     lab->to = atoi(n);
     lab->nid = 0;
     lab->from = NULL;
     lab->dead = NULL;
     lab->next = NULL;
     return lab;
}

LabelTable *addlabtab(b1,b2)
     LabelTable *b1,*b2;
{
     LabelTable *t;
```

```
    for (t = b1; t->next != NULL; t = t->next);
    t->next = b2;
    return b1;
}

TypeTable *settype(str,td)
    char *str;
    TypeDenoterType *td;
{
    TypeTable *t;
    char *i, *emalloc();

    t = (TypeTable *) emalloc(sizeof(TypeTable));
    t->id = (char *) emalloc(strlen(str)+1);
    strcpy(t->id,str);
    free(str);
    t->d = td;
    t->next = NULL;
    return t;
}

TypeTable *addtype(t1,t2)
    TypeTable *t1, *t2;
{
    TypeTable *t;

    for (t = t1; t->next != NULL; t = t->next);
    t->next = t2;
    return t1;
}

TypeDenoterType *mkbasictype(i)
    char *i;
{
    TypeDenoterType *d;
    char *emalloc();

    d = (TypeDenoterType *) emalloc(sizeof(TypeDenoterType));
    d->tag =BASICTYPE;
    d->d.id = (char *) emalloc(strlen(i)+1);
    strcpy(d->d.id,i);
    free(i);
    d->next = NULL;
    return d;
}

TypeDenoterType *mkenumeratedtype(l)
    IdListType *l;
{
    TypeDenoterType *d;
    char *emalloc();

    d = (TypeDenoterType *) emalloc(sizeof(TypeDenoterType));
    d->tag = ENUMERATEDTYPE;
    d->d.i = l;
    d->next = NULL;
    return d;
}

TypeDenoterType *mksubrangetype()
{
```

```
     TypeDenoterType *d;
     char *emalloc();

     d = (TypeDenoterType *) emalloc(sizeof(TypeDenoterType));
     d->tag = SUBRANGETYPE;
     d->next = NULL;
     return d;
}

TypeDenoterType *mkfiletype(d)
     TypeDenoterType *d;
{
     TypeDenoterType *d1;
     char *emalloc();

     d1 = (TypeDenoterType *) emalloc(sizeof(TypeDenoterType));
     d1->tag = FILETYPE;
     d1->next = d;
     return d1;
}

TypeDenoterType *mkpointertype(i)
     char *i;
{
     TypeDenoterType *d, *d1;
     char *emalloc();

     d = (TypeDenoterType *) emalloc(sizeof(TypeDenoterType));
     d->tag = POINTERTYPE;
     d1 = (TypeDenoterType *) emalloc(sizeof(TypeDenoterType));
     d1->tag = BASICTYPE;
     d1->d.id = (char *) emalloc(strlen(i)+1);
     strcpy(d1->d.id,i);
     free(i);
     d1->next = NULL;
     d->next = d1;
     return d;
}

TypeDenoterType *mkrecordtype(f)
     DecListType *f;
{
     TypeDenoterType *d;
     char *emalloc();

     d = (TypeDenoterType *) emalloc(sizeof(TypeDenoterType));
     d->tag = RECORDTYPE;
     d->d.f = f;
     d->next = NULL;
     return d;
}

addvars(d)
     DecListType *d;
{
     DecListType *d1;

     if (memtab->var == NULL)  memtab->var =d;
     else {
          for (d1 = memtab->var; d1->next != NULL; d1 = d1->next);
          d1->next = d;
```

```
    }
}

DecListType *mkdeclist(i,d)
    IdListType *i;
    TypeDenoterType *d;
{
    DecListType *v;
    char *emalloc();

    v = (DecListType *) emalloc(sizeof(DecListType));
    v->idl = i;
    v->d = d;
    v->next = NULL;
    return v;
}

DecListType *catdeclist(v1,v2)
    DecListType *v1, *v2;
{
    DecListType *t;

    if (v1 == NULL)  return v2;
    else {
        for (t = v1; t->next != NULL; t = t->next);
        t->next = v2;
        return v1;
    }
}

chkglb(s)
    char *s;
{
    IdListType *c, *i, *mkidlist(), *catidlist();
    DecListType *v;
    MemoryTable *mem;

    if (memtab->next != NULL) {
        for (c = memtab->const; c != NULL; c = c->next)
            if (strcmp(c->id,s) == 0) return;
        for (v = memtab->var; v != NULL; v = v->next)
            for (i = v->idl; i != NULL; i = i->next)
                if (strcmp(i->id,s) == 0) return;
        for (mem = memtab->next; mem != NULL; mem = mem->next) {
            for (c = mem->const; c != NULL; c = c->next)
                if (strcmp(c->id,s) == 0)  return;
            for (v = mem->var; v != NULL; v = v->next)
                for (i = v->idl; i != NULL; i = i->next)
                    if (strcmp(i->id,s) == 0) {
                        if (glbls == NULL) glbls = mkidlist(s);
                        else glbls = catidlist(glbls,mkidlist(s));
                    }
        }
    }
}

TypeDenoterType *findrecordtype(d)
    TypeDenoterType *d;
{
    TypeDenoterType *d1;
```

```
    for (d1 = d; d1 != NULL; d1 = d1->next)
        if (d1->tag == RECORDTYPE)  return d1;
    return NULL;
}

char *findbasictype(d)
    TypeDenoterType *d;
{
    TypeDenoterType *d1;

    for (d1 = d; d1 != NULL; d1 = d1->next)
        if (d1->tag == BASICTYPE)  return d1->d.id;
}

isfilevar(v)
    char *v;
{
    MemoryTable *m;
    TypeTable *t;
    TypeDenoterType *s, *t1;
    DecListType *d;
    IdListType *i;
    char *stype = NULL, *findbasictype();

    for (m = memtab; m != NULL; m = m->next) {
        for (d = m->var; d != NULL; d = d->next)
            for (i = d->idl; i != NULL; i = i->next)
                if (strcmp(i->id,v) == 0) {
                    for (s = d->d; s != NULL; s = s->next) {
                        if ((s->tag == BASICTYPE) &&
                            strcmp(s->d.id,"text") == 0)
                            return 1;
                        if (s->tag == FILETYPE) return 1;
                    }
        stype = findbasictype(d->d);
        break;
    }
    if (stype != NULL)
        for (t = m->type; t != NULL; t = t->next)
            if (strcmp(stype,t->id) == 0) {
                for (t1 = t->d; t1 != NULL; t1 = t1->next)
                    if (t1->tag == FILETYPE) return 1;
            }
    return 0;
    }
}

isfamily(id,key)                /* if id is a field of recordtype variable */
    char *id, *key;             /* "key", then return 1.                   */
{                               /* return 0, otherwise.                    */
    TypeDenoterType *d1;
    TypeDenoterType *findrecordtype();
    MemoryTable *m;
    TypeTable *t;
    DecListType *d;
    IdListType *i;
    char *ktype = NULL, *btype, *findbasictype();

    for (m = memtab; m != NULL; m = m->next) {
```

```
          for (d = m->var; d != NULL; d = d->next)
               for (i = d->idl; i != NULL; i = i->next)
                    if (strcmp(i->id,key) == 0) {
                         if (d1 = findrecordtype(d->d) != NULL)
                              return inspectfields(d1,id);
                         else { ktype = findbasictype(d->d);
                                goto searchtype;
                         }
                    }
          searchtype:
          if (ktype != NULL) {
               for (t = m->type; t != NULL; t = t->next)
                    if (strcmp(t->id,ktype) == 0) {
                         if ((d1 = findrecordtype(t->d)) != NULL)
                              return inspectfields(d1,id);
                    }
          }
          else { ktype = key;
                 goto searchtype;
          }
     }
     return 0;
}

inspectfields(fd,id)
     TypeDenoterType *fd;
     char *id;
{
     TypeDenoterType *d1, *findrecordtype();
     DecListType *d;
     IdListType *i;
     char *findbasictype();

     if (fd != NULL)
          for (d = fd->d.f; d != NULL; d = d->next) {
               for (i = d->idl; i != NULL; i = i->next)
                    if (strcmp(i->id,id) == 0)  return 1;
               if ((d1 = findrecordtype(d->d)) != NULL)
                    return inspectfields(d1,id);
               else if (isfamily(id,findbasictype(d->d)) == 1)
               return 1;
          }
     return 0;
}

char *object(v)
     char *v;
{
     MemoryTable *m;
     TypeTable *p1;
     TypeDenoterType *t, *t1;
     DecListType *p;
     IdListType *i;
     char *vtype = NULL, *findbasictype();

     for (m = memtab; m != NULL; m = m->next) {
          for (p = m->var; p != NULL; p = p->next)
               for (i = p->idl; i != NULL; i = i->next)
```

```
                    if (strcmp(i->id,v) == 0) {
                         vtype = findbasictype(p->d);
                         for (t = p->d; t->next != NULL; t = t->next)
                              if (t->tag == POINTERTYPE) {
                                   t = t->next;
                                   return t->d.id;
                              }
                         break;
                    }
          for (p1 = m->type; p1 != NULL; p1 = p1->next)
               if (strcmp(p1->id,vtype) == 0)
                    for (t1 = p1->d; t1->next != NULL; t = t->next)
                         if (t1->tag == POINTERTYPE)
                              return findbasictype(p1->d);
     }
     return v;
}

char *ptrobject(td)
     TypeDenoterType *td;
{
     TypeDenoterType *d;
     MemoryTable *m;
     TypeTable *p1;
     char *ptype;

     for (d = td; d->next != NULL; d = d->next)
          if (d->tag == POINTERTYPE)
               return findbasictype(td);
     ptype = findbasictype(td);
     for (m = memtab; m != NULL; m = m->next)
          for (p1 = m->type; p1 != NULL; p1 = p1->next)
               if (strcmp(p1->id,ptype) == 0)
                    return ptrobject(p1->d);
}

char *inspectfieldobject(fd,id)
     TypeDenoterType *fd;
     char *id;
{
     TypeDenoterType *d1, *findrecordtype();
     DecListType *d;
     IdListType *i;
     char *findbasictype();

     if (fd != NULL)
          for (d = fd->d_f; d != NULL; d = d->next) {
               for (i = d->idl; i != NULL; i = i->next)
                    if (strcmp(i->id,id) == 0)
                         return ptrobject(d->d);
               if ((d1 = findrecordtype(d->d)) != NULL)
                    return inspectfieldobject(d1,id);
          }
     return NULL;
}

char *findobject(id,key)
     char *id, *key;
```

```
{
     TypeDenoterType *d1, *findrecordtype();
     MemoryTable *m;
     TypeTable *t;
     DecListType *d;
     IdListType *i;
     char *ktype = NULL, *btype, *findbasictype();

     for (m = memtab; m != NULL; m = m->next) {
          for (d = m->var; d != NULL; d = d->next)
               for (i = d->idl; i != NULL; i = i->next)
                    if (strcmp(i->id,key) == 0) {
                         if (d1 = findrecordtype(d->d) != NULL)
                              return inspectfieldobject(d1,id);
                         else { ktype = findbasictype(d->d);
                                goto searchtype;
                         }
                    }
          searchtype:
          if (ktype != NULL) {
               for (t = m->type; t != NULL; t = t->next)
                    if (strcmp(t->id,ktype) == 0) {
                         if ((d1 = findrecordtype(t->d)) != NULL)
                              return inspectfieldobject(d1,id);
                         ktype = findbasictype(t->d);
                         goto searchtype;
                    }
          }
     }
}

char *fieldobject(ex,key)
     ExtraIdType *ex;
     char *key;
{
     IdListType *x;

     for (x = ex->c; x->next != NULL; x = x->next);
     return findobject(x->id,key);
}


/***** exp.c ***************************************************************/
#include "sr.h"
#include "y.tab.h"
#define         BUILTIN         1
#define         USERDEFINED     2

IdListType *catidlist(i,j)
     IdListType *i, *j;
{
     IdListType *tmp;

     if (i == NULL)  i = j;
     else {
          for (tmp = i; tmp->next != NULL; tmp = tmp->next);
          tmp->next = j;
     }
     return i;
```

```
}
IdListType *mkidlist(s)
     char *s;
{
     IdListType *i;
     char *emalloc();

     i = (IdListType *) emalloc(sizeof(IdListType));
     i->id = (char *) emalloc(strlen(s)+1);
     strcpy(i->id,s);
     i->next = NULL;
     return i;
}

isinset(id,lst)
     char *id;
     IdListType *lst;
{
     IdListType *n;

     for (n = lst; n != NULL; n = n->next)
          if (strcmp(id,n->id) == 0) return 1;   /* yes */
     return 0;   /* no */
}

IdListType *mkidset(lst)
     IdListType *lst;
{
     IdListType *newlst = NULL, *n1, *n2, *mkidlist();

     for (n1 = lst; n1 != NULL; n1 = n1->next)
          if (newlst == NULL)
               newlst = mkidlist(n1->id);
          else if (isinset(n1->id,newlst) == 0) {  /* if not in the set */
               for (n2 = newlst; n2->next != NULL; n2 = n2->next);
               n2->next = mkidlist(n1->id);
          }
     return newlst;
}

ExpType *catexplist(u,v)
     ExpType *u, *v;
{
     ExpType *tmp;

     if (u == NULL)  u = v;
     else {
  for (tmp = u; tmp->next != NULL; tmp = tmp->next);
          tmp->next = v;
     }
     return u;
}

ExpType *mkexptype1(s)
     char *s;
{
     ExpType *u;
     char *emalloc();

     u = (ExpType *) emalloc(sizeof(ExpType));
```

```
    u->exp.id = (char *) emalloc(strlen(s)+1);
    strcpy(u->exp.id,s);
    u->tag = IDTAG;
    u->next = NULL;
    return u;
}

ExpType *mkexptype2(s,q)
    char *s;
    ExpSequence *q;
{
    ExpType *u;
    FuncExp *f;
    char *emalloc();

    u = (ExpType *) emalloc(sizeof(ExpType));
    f = (FuncExp *) emalloc(sizeof(FuncExp));
    f->funcid = (char *) emalloc(strlen(s)+1);
    strcpy(f->funcid,s);
    f->exps = q;
    u->exp.fexp = f;
    u->tag = FUNCTAG;
    u->next = NULL;
    return u;
}

ExpSequence *catexpseqlist(p,q)
    ExpSequence *p, *q;
{
    ExpSequence *tmp;

    if (p == NULL)  p = q;
    else {
  for (tmp = p; tmp->next != NULL; tmp = tmp->next);
        tmp->next = q;
    }
    return p;
}

ExpSequence *mkexpseq(u)
    ExpType *u;
{
    ExpSequence *q;

    q = (ExpSequence *) emalloc(sizeof(ExpSequence));
    q->exp = u;
    q->next = NULL;
    return q;
}

ExpType *seqtoexp(s)
    ExpSequence *s;
{
    ExpSequence *s1;
    ExpType *u = NULL, *n, *mkexptype1(), *catexplist();
    ExpType *seqtoexp();

    for (s1 = s; s1 != NULL; s1 = s1->next)
        for (n = s1->exp; n != NULL; n = n->next) {
            if (n->tag == IDTAG)
```

```
                u = catexplist(u,mkexptype1(n->exp.id));
            else if (n->tag == FUNCTAG)
                u = catexplist(u,seqtoexp(n->exp.fexp->exps));
        }
    return u;
}

ExpType *mkfunctionexp(s,q)
    char *s;
    ExpSequence *q;
{
    int p;

    if ((strcmp(s,"abs") == 0) ||
        (strcmp(s,"sqr") == 0) ||
        (strcmp(s,"sin") == 0) ||
        (strcmp(s,"cos") == 0) ||
        (strcmp(s,"exp") == 0) ||
        (strcmp(s,"ln") == 0) ||
        (strcmp(s,"sqrt") == 0) ||
        (strcmp(s,"arctan") == 0) ||
        (strcmp(s,"odd") == 0) ||
        (strcmp(s,"eof") == 0) ||
        (strcmp(s,"eoln") == 0) ||
        (strcmp(s,"trunc") == 0) ||
        (strcmp(s,"round") == 0) ||
        (strcmp(s,"ord")  == 0) ||
        (strcmp(s,"chr") == 0) ||
        (strcmp(s,"succ") == 0) ||
        (strcmp(s,"pred") == 0))
         p = BUILTIN;
    else    p = USERDEFINED;
    if (p == BUILTIN)
        return seqtoexp(q);
    if (p == USERDEFINED)
        return mkexptype2(s,q);
}

ExtraIdType *mkextraid1(u)
    ExpType *u;
{
    ExtraIdType *e;
    char *emalloc();

    e = (ExtraIdType *) emalloc(sizeof(ExtraIdType));
    e->u = u;
    e->c = NULL;
    return e;
}

ExtraIdType *mkextraid2(s)
    char *s;
{
    ExtraIdType *e;
    IdListType *mkidlist();
    char *emalloc();
```

```
    e = (ExtraIdType *) emalloc(sizeof(ExtraIdType));
    e->u = NULL;
    e->c = mkidlist(s);
    return e;
}

ExtraIdType *mkextralist(e1,e2)
    ExtraIdType *e1, *e2;
{
    IdListType *tc;

    e1->u = (IdListType *) catexplist(e1->u,e2->u);
    if (e1->c == NULL) e1->c = e2->c;
    else {
  for (tc = e1->c; tc->next != NULL; tc = tc->next);
        tc->next = e2->c;
    }
    return e1;
}

char *mkbuffervar(s1)
    char *s1;
{
    char *s;

    s = (char *) emalloc(strlen(s1)+1);
    strcpy(s,s1);
    strcat(s,"^");
    return s;
}

char *mklabelopnd(s1)
    char *s1;
{
    char *s;

    s = (char *) emalloc(strlen(s1)+1);
    strcpy(s,s1);
    strcat(s,":");
    return s;
}

char *mkcaseconstopnd(s1,s2)
    char *s1, *s2;
{
    char *s;

    if (s1 == NULL) {
        s = (char *) emalloc(strlen(s2)+1);
        strcpy(s,s2);
    }
    else if (strcmp(s1,"+") == 0) {
        s = (char *) emalloc(strlen(s1)+1);
        strcpy(s,s1);
        strcat(s,s2);
    }
    else if (strcmp(s1,"-") == 0) {
        s = (char *) emalloc(strlen(s1)+1);
        strcpy(s,s1);
        strcat(s,s2);
    }
```

```
     strcat(s,":");
     return s;
}

char *determobj(id,ex)
     char *id;
     IdListType *ex;
{
     if (ex == NULL) return id;              /* simple variable */
     else if ((strcmp(ex->id,"^") == 0) && (isfilevar(id) == 1))
          return mkbuffervar(id);            /* buffer variable */
     else if (strcmp(ex->id,"^") == 0)       /* pointer variable */
     return object(id);
     else return id;                         /* record variable */
}

ExpType *extraexplist1(s,x,u)
     char *s;
     ExtraIdType *x;
     ExpType *u;
{
     char *determobj();
     ExpType *mkexptype(), *catexplist();

     if ((x != NULL) && (x->c != NULL)) {
          if ((strcmp(x->c->id,"^") == 0) && (isfilevar(s) == 1)) /*buffer*/
               return catexplist(x->u,u);
          else if (strcmp(x->c->id,"^") == 0)   /* pointer */
               return catexplist(mkexptype1(s),
                                 catexplist(mkexptype1(determobj(s,x->c)),
                                            catexplist(x->u,u)));
          else /* record */
               return catexplist(mkexptype1(s),catexplist(x->u,u));
     }
     else if ((x != NULL) && (x->u != NULL))   /* array */
          return catexplist(mkexptype1(s),catexplist(x->u,u));
     else return u;
}

ExpType *extraexplist2(s,x)
     char *s;
     ExtraIdType *x;
{
     char *determobj();
     ExpType *mkexptype1(), *catexplist();

     if ((x != NULL) && (x->c != NULL)) {
          if ((strcmp(x->c->id,"^") == 0) && (isfilevar(s) == 1))
               return catexplist(mkexptype1(determobj(s,x->c)),x->u);
          if (strcmp(x->c->id,"^") == 0)
               return catexplist(mkexptype1(s),
                      catexplist(mkexptype1(determobj(s,x->c)),x->u));
     }
     if (x != NULL)
          return catexplist(mkexptype1(s),x->u);
     else return mkexptype1(s);
}
```

```
/**** branch.c ************************************************************/
#include "sr.h"
#include "y.tab.h"

extern NodeType *ndptr;
extern MemoryTable *memtab;

gotobranch(s,n)
     char *s;
     int n;
{
     LabelTable *tmp;
     OriginList *t, *tp;
     char *emalloc();

     for (tmp = memtab->lab; tmp->to != atoi(s); tmp = tmp->next);
     if (tmp->nid != 0)  addedge(n,tmp->nid);
     else if (tmp->from == NULL) {
          tmp->from = (OriginList *) emalloc(sizeof(OriginList));
          tmp->from->n = n;
          tmp->from->next = NULL;
     }
     else {
          t = (OriginList *) emalloc(sizeof(OriginList));
          tp = tmp->from;
          tmp->from = t;
          tmp->from->n = n;
          tmp->from->next = tp;
     }
}

destlist(s,n)
     char *s;
     NodeType *n;
{
     LabelTable *t;
     OriginList *tp, *td;

     for (t = memtab->lab; t->to != atoi(s); t = t->next);
     t->nid = n->nid;
     if (t->dead != NULL)
          for (td = t->dead; td != NULL; td = td->next)
               updatedge3(n->nid,td->n);
     for (tp = t->from; tp != NULL; tp = tp->next)
          addedge(tp->n,n->nid);
     t->from = NULL;
}

adddeads(dead)
     int dead;
{
     OriginList *d;
     char *emalloc();

     d = (OriginList *) emalloc(sizeof(OriginList));
     d->n = dead;
     d->next = NULL;
     if (memtab->lab == NULL)  memtab->dead = d;
     else {
          d->next = memtab->dead;
          memtab->dead = d;
```

```
    }
}

isgotoflagON(n)
    NodeType *n;
{
    NodeType *t;

    for (t = n; t->next != NULL; t = t->next);
    if (t->gt == ON)  return YES;
    else  return NO;
}


/***** edge.c ***************************************************************/
#include "sr.h"
#include "y.tab.h"

extern UnitRepType *unitstack;
extern MemoryTable *memtab;

addedge(x,y)
    int x, y;
{
    Edge *eg, *t;
    short found = 0;
    char *emalloc();

    eg = (Edge *) emalloc(sizeof(Edge));
    eg->from = x;
    eg->to = y;
    eg->next = NULL;
    if (unitstack->ufs->eg == NULL)  unitstack->ufs->eg = eg;
    else {
        for (t = unitstack->ufs->eg; t != NULL; t = t->next)
            if ((t->from == x) && (t->to == y))
                found = 1;
        if (found == 0) {
            for (t = unitstack->ufs->eg; t->next != NULL; t = t->next);
                t->next = eg;
        }
    }
}

updatedge1(alive,dead)
    int alive,dead;
{
    Edge *e;
    LabelTable *b;
    OriginList *f;

    for (e = unitstack->ufs->eg; e != NULL; e = e->next) {
        if (e->from == dead) {
            e->from = alive;
            if (e->from == e->to) {
                e->from = DELETED;
                e->to = DELETED;
            }
        }
    }
    for (e = unitstack->ufs->eg; e != NULL; e = e->next) {
```

```
            if (e->to == dead) {
                e->to = alive;
                if (e->from == e->to) {
                    e->from = DELETED;
                    e->to = DELETED;
                }
            }
        }
        for (b = memtab->lab; b != NULL; b = b->next) {
            if (b->nid == 0) {
                for (f = b->from; f != NULL; f = f->next)
                    if (f->n == dead)  f->n = alive;
            }
            else if (b->nid == dead)  b->nid = alive;
        }
}

updatedge2(dead)
        int dead;
{
        Edge *e;
        int backp;
        LabelTable *b;
        OriginList *f;

        for (e = unitstack->ufs->eg; e != NULL; e = e->next)
            if (e->from == dead) {
                backp = e->to;
                e->from = DELETED;
                e->to = DELETED;
            }
        for (e = unitstack->ufs->eg; e != NULL; e = e->next)
            if (e->to == dead) {  /* e->to = backup; */
                addedge(e->from,backp);
                e->from = DELETED;
                e->to = DELETED;
            }
        for (b = memtab->lab; b != NULL; b = b->next)
            if (b->nid == 0) {
                for (f = b->from; f != NULL; f = f->next)
                    if (f->n == dead)  adddeads(dead);
            }
}

updatedge3(alive,dead)
        int alive, dead;
{
        Edge *e;

        for (e = unitstack->ufs->eg; e != NULL; e = e->next)
            if (e->to == dead)  e->to = alive;
}

addcasedge(from,to)
        int from,to;
{
        Edge *e;

        for (e = unitstack->ufs->eg; e != NULL; e = e->next) {
            if (e->from == CASENID)  e->from = from;
            if (e->to == CASENID)  e->to = to;
        }
```

```
}

addfirstedge(n)
     int n;
{
     Edge *e;
     char *emalloc();

     e = (Edge *) emalloc(sizeof(Edge));
     e->from = START;
     e->to = n;
     e->next = NULL;
     if (unitstack->ufs->eg == NULL)  unitstack->ufs->eg = e;
     else {
          e->next = unitstack->ufs->eg;
          unitstack->ufs->eg = e;
     }
}

addlastedge(n)
     NodeType *n;
{
     NodeType *t;
     Edge *e;

     if (n != NULL) {
          for (t = n; t->next != NULL; t = t->next);
          if ((t->ldef == NULL) && (t->pexp == NULL)) {
          /* convert last node id into 't' if it is empty */
               for (e = unitstack->ufs->eg; e != NULL; e = e->next)
                    if (e->to == t->nid)  e->to = TERMINAL;
          }
          /* add edge (lastnode,'t') if it is not empty */
          else addedge(t->nid,TERMINAL);
     }
}


/***** opcnt.c ***************************************************************/
#include "sr.h"
#include "y.tab.h"

OpCount *oprtlookup(n,op)
     NodeType *n;
     char *op;
{
     OpCount *opp;

     for (opp = n->operators; opp != NULL; opp = opp->next)
          if (strcmp(opp->name,op) == 0)
               return opp;
     return NULL;
}

oprtinstall(n,op)
     NodeType *n;
     char *op;
{
     OpCount *opp, *t;
     char *emalloc();
```

```
    opp = (OpCount *) emalloc(sizeof(OpCount));
    opp->name = (char *) emalloc(strlen(op)+1);
    strcpy(opp->name,op);
    opp->occur = 1;
    opp->next = NULL;
    if (n->operators == NULL)  n->operators = opp;
    else {
        for (t = n->operators; t->next != NULL; t = t->next);
        t->next = opp;
    }
}

operatorcnt(n,op)
    NodeType *n;
    char *op;
{
    OpCount *opp, *oprtlookup();

    if (n != NULL) {
        if ((opp = oprtlookup(n,op)) == NULL)
            oprtinstall(n,op);
        else opp->occur += 1;
    }
}

OpCount *opndlookup(n,op)
    NodeType *n;
    char *op;
{
    OpCount *opp;

    for (opp = n->operands; opp != NULL; opp = opp->next)
        if (strcmp(opp->name,op) == 0)
            return opp;
    return NULL;
}

opndinstall(n,op)
    NodeType *n;
    char *op;
{
    OpCount *opp, *t;
    char *emalloc();

    opp = (OpCount *) emalloc(sizeof(OpCount));
    opp->name = (char *) emalloc(strlen(op)+1);
    strcpy(opp->name,op);
    opp->occur = 1;
    opp->next = NULL;
    if (n->operands == NULL)  n->operands = opp;
    else {
        for (t = n->operands; t->next != NULL; t = t->next);
        t->next = opp;
    }
}

operandcnt(n,op)
    NodeType *n;
```

```
     char *op;
{
     OpCount *opp, *opndlookup();

     if (n != NULL) {
          if ((opp = opndlookup(n,op)) == NULL)
               opndinstall(n,op);
          else opp->occur += 1;
     }
}

caseopcnt(n,d)
     NodeType *n;
     IdListType *d;
{
     char *s;
     IdListType *i;

     for (i = d; i != NULL; i = i->next) {
          operandcnt(n,i->id);
          if (i->next != NULL)  operatorcnt(n,",");
     }
}


/***** cdgen.c ***************************************************************/
#include "sr.h"
#include "y.tab.h"

extern UnitRepType *unitrep;

codegenerate()
{
     UnitRepType *tmp;

     printf("{");
     for (tmp = unitrep; tmp != NULL; tmp = tmp->next) {
          prunitrep(tmp);
          if (tmp->next != NULL)  printf(",\n ");
     }
     printf("}\n");
}

prunitrep(p)
     UnitRepType *p;
{
     printf("(");
     prheader(p->interface);
     printf(",\n");
     prufs(p->ufs);
     printf(")");
}

prheader(h)
     HeaderType *h;
{
     printf("(");
     printf("%s",h->pname);
     printf(",");
```

```
    prseqofid(h->fparams);
    printf(",");
    prsetofid(h->globals);
    printf(")");
}

prseqofid(i)
    FPListType *i;
{
    IdListType *t;

    printf("<");
    if (i != NULL)
        for (t = i->fp; t != NULL; t = t->next) {
            printf("%s",t->id);
            if (t->next != NULL) printf(",");
        }
    printf(">");
}

prsetofid(i)
    IdListType *i;
{
    IdListType *t;

    printf("{");
    for (t = i; t != NULL; t = t->next) {
        printf("%s",t->id);
        if (t->next != NULL)  printf(",");
    }
    printf("}");
}

prufs(p)
    UnitFlowStruct *p;
{
    printf("  (");
    prnodeset(p->nd,p->s);
    printf(",\n");
    predgeset(p->eg);
    printf(",\n");
    printf("   s,\n");
    printf("   t");
    printf(")");
}

prnodeset(n,vp)
    NodeType *n;
    IdListType *vp;
{
    NodeType *t;
    IdListType *vp1;

    printf("{");
    printf("(s,<");
    for (vp1 = vp; vp1 != NULL; vp1 = vp1->next) {
        printf("(");
```

```
            printf("%s'",vp1->id);
            printf(",<");
            printf("%s",vp1->id);
            printf(">)");
            if (vp1->next != NULL) printf(",");
        }
        printf(">,<>,0)\n");
        for (t = n; t != NULL; t = t->next) {
            prnode(t,vp);
            if (t->next != NULL)  printf(",\n");
        }
        printf("\n    (t,<>,<>,0)");
        printf("}");
}

prnode(n,vp)
        NodeType *n;
        IdListType *vp;
{
        printf("    (");
        printf("%d",n->nid);
        printf(",");
        prlocaldef(n->ldef,vp);
        printf(",\n       ");
        prexptype(n->pexp,vp);
        printf(",\n");
        printf("       (");
        prhalstinfo(n->operators);
        printf(",\n        ");
        prhalstinfo(n->operands);
        printf(")");
        printf(")");
}

prlocaldef(d,vp)
        LocalDef *d;
        IdListType *vp;
{
        LocalDef *t;
        int cnt = 0;

        printf("<");
        for (t = d; t != NULL; t = t->next) {
            if (t->tag == SDEFTAG)  prsimpledef(t->def.sdef,vp);
            else if (t->tag == PDEFTAG)  prprocdef(t->def.pdef,vp);
            if (t->next != NULL)  printf(",");
            cnt++;
            if ((cnt >= 3) && (t->next != NULL)) {
                printf("\n         ");
                cnt = 0;
            }
        }
        printf(">");
}

prsimpledef(d,vp)
        SimpleDef *d;
```

```
     IdListType *vp;
{
     printf("(");
     if (isvalueparam(d->varid,vp) == YES) printf("%s'",d->varid);
     else printf("%s",d->varid);
     printf(",");
     prexptype(d->sexp,vp);
     printf(")");
}

prprocdef(d,vp)
     ProcDef *d;
     IdListType *vp;
{
     printf("(");
     printf("%s",d->procid);
     printf(",");
     prexpseq(d->pexp,vp);
     printf(")");
}

prexptype(u,vp)
     ExpType *u;
     IdListType *vp;
{
     ExpType *t;

     printf("<");
     for (t = u; t != NULL; t = t->next) {
          if (t->tag == IDTAG) {
               if (isvalueparam(t->exp_id,vp) == YES)
                    printf("%s'",t->exp_id);
               else printf("%s",t->exp_id);
          }
          else if (t->tag == FUNCTAG)
               prfuncexp(t->exp_fexp,vp);
          if (t->next != NULL)  printf(",");
     }
     printf(">");
}

prfuncexp(u,vp)
     FuncExp *u;
     IdListType *vp;
{
     printf("(");
     printf("%s",u->funcid);
     printf(",");
     prexpseq(u->exps,vp);
     printf(")");
}

prexpseq(u,vp)
     ExpSequence *u;
     IdListType *vp;
{
     ExpSequence *t;
```

```
        printf("<");
        for (t = u; t != NULL; t = t->next) {
            prexptype(t->exp,vp);
            if (t->next != NULL)  printf(",");
        }
        printf(">");
}

prhalstinfo(h)
        OpCount *h;
{
        OpCount *t;
        int cnt = 0;

        printf("{");
        for (t = h; t != NULL; t = t->next) {
            printf("(");
            printf("%s",t->name);
            printf(",");
            printf("%d",t->occur);
            printf(")");
            cnt++;
            if (t->next != NULL)  printf(",");
            if ((cnt >= 8) && (t->next != NULL)) {
                printf("\n          ");
                cnt = 0;
            }
        }
        printf("}");
}

predgeset(e)
        Edge *e;
{
        Edge *t;
        int cnt = 0;

        printf("   {");
        for (t = e; t != NULL; t = t->next)
            if ((t->from != DELETED) || (t->to != DELETED)) {
                printf("(");
                prnodeid(t->from);
                printf(",");
                prnodeid(t->to);
                printf(")");
                cnt++;
                if (t->next != NULL) printf(",");
                if ((cnt >= 10) && (t->next != NULL)) {
                    printf("\n    ");
                    cnt = 0;
                }
            }
        printf("}");
}

prnodeid(n)
        int n;
{
```

```
     if (n == START) printf("s");
     else if (n == TERMINAL)  printf("t");
     else printf("%d",n);
}


/***** makefile ********************************************************/
YFLAGS = -d
CFLAGS = -w
OBJS = sr.o lex.o cnstrct.o memory.o branch.o exp.o edge.o opcnt.o cdgen.o

sr:     $(OBJS)
        cc $(CFLAGS) $(OBJS) -lm -ll -o sr

sr.o:   sr.h

lex.o cnstrct.o memory.o branch.o exp.o edge.o opcnt.o cdgen.o: sr.h y.tab.h


/***** pp.l ************************************************************/
%{
#include <ctype.h>
%}
%START SKIP
capital                 [A-Z]
moreliteral             '[^'\n]*'/'
literal                 '[^'\n]*'
begincomments           \(\*|\{
commentson              [^*}]*
endcomments             \*\)|\}
morecomments            \*/[^)]

%%
<SKIP>{commentson}          {;}
<SKIP>{endcomments}         {BEGIN 0;}
<SKIP>{morecomments}        {;}
{moreliteral}               {yymore();}
{literal}                   {printf("%s",yytext);}
{capital}                   {printf("%c",tolower(yytext[0]));}
{begincomments}             {BEGIN SKIP;}
```

B Test Programs and Results

B.1 The Simplest Program Segments

```
program simplestmt;
begin
  stmt
end.

{((simplestmt,<>,{}),
  ({(s,<>,<>,0)
    (1,<(stmt,<>)>,
       <>,
       ({(stmt,1),(begin..end,1),(.,1)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}

program ifthenstmt;
begin
  if cond then stmt
end.

{((ifthenstmt,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <cond>,
       ({(if..then,1),(begin..end,1),(.,1)},
        {(cond,1)})),
    (2,<(stmt,<>)>,
       <>,
       ({(stmt,1)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(1,2),(2,t),(1,t)},
   s,
   t))}

program ifthenelsetest;
begin
  if cond then stmt1
  else stmt2;
end.

{((ifthenelsetest,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <cond>,
       ({(if..then,1),(begin..end,1),(.,1)},
        {(cond,1)})),
    (2,<(stmt1,<>)>,
       <>,
       ({(stmt1,1)},
        {})),
    (3,<(stmt2,<>)>,
       <>,
       ({(stmt2,1),(else,1),(;,1)},
        {}))
```

```
    (t,<>,<>,0)},
   {(s,1),(1,2),(2,t),(1,3),(3,t)},
   s,
   t))}

program casestmt;
begin
  case which of
    one : smtm1;
    two : stmt2;
    three : stmt3;
  end
end.

{((casestmt,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <which>,
       ({(case..of..end,1),(begin..end,1),(.,1)},
        {(which,1)})),
    (2,<(smtm1,<>)>,
       <>,
       ({(smtm1,1)},
        {(one:,1)})),
    (3,<(stmt2,<>)>,
       <>,
       ({(stmt2,1),(;,1)},
        {(two:,1)})),
    (4,<(stmt3,<>)>,
       <>,
       ({(stmt3,1),(;,2)},
        {(three:,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,2),(2,t),(1,3),(3,t),(1,4),(4,t)},
   s,
   t))}

program repeatstmt;
begin
  repeat
    stmt
  until cond
end.

{((repeatstmt,<>,{}),
  ({(s,<>,<>,0)
    (1,<(stmt,<>)>,
       <cond>,
       ({(stmt,1),(repeat..until,1),(begin..end,1),(.,1)},
        {(cond,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,t),(1,1)},
   s,
   t))}

program whilestmt;
begin
  while cond do
    stmt
end.

{((whilestmt,<>,{}),
```

```
  ({(s,<>,<>,0)
    (1,<>,
       <cond>,
       ({(while..do,1),(begin..end,1),(.,1)},
        {(cond,1)})),
    (2,<(stmt,<>)>,
       <>,
       ({(stmt,1)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(1,2),(2,1),(1,t)},
   s,
   t))}

program forstmt;
begin
  for c := 1 to 100 do
    dosomething
end.

{((forstmt,<>,{}),
  ({(s,<>,<>,0)
    (1,<(c,<i>)>,
       <>,
       ({(for..to..do,1),(begin..end,1),(.,1)},
        {(1,1),(c,1)})),
    (2,<>,
       <c,100>,
       ({},
        {(100,1)})),
    (3,<(dosomething,<>),(c,<c>)>,
       <>,
       ({(dosomething,1)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(1,2),(2,3),(3,2),(2,t)},
   s,
   t))}
```

B.2 Sequencing

```
program simplesimple;
begin
  stmt1;
  stmt2
end.

{((simplesimple,<>,{}),
  ({(s,<>,<>,0)
    (1,<(stmt1,<>),(stmt2,<>)>,
       <>,
       ({(stmt1,1),(;,1),(stmt2,1),(begin..end,1),(.,1)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}

program simpleifthen;
```

```
begin
  stmt1;
  if cond then stmt2
end.

{((simpleifthen,<>,{}),
  ({(s,<>,<>,0)
    (1,<(stmt1,<>)>,
       <cond>,
       ({(stmt1,1),(;,1),(if..then,1),(begin..end,1),(.,1)},
        {(cond,1)})),
    (3,<(stmt2,<>)>,
       <>,
       ({(stmt2,1)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(1,3),(3,t),(1,t)},
   s,
   t))}

program simpleifthenelse;
begin
  stmt1;
  if cond then stmt2
  else stmt3
end.

{((simpleifthenelse,<>,{}),
  ({(s,<>,<>,0)
    (1,<(stmt1,<>)>,
       <cond>,
       ({(stmt1,1),(;,1),(if..then,1),(begin..end,1),(.,1)},
        {(cond,1)})),
    (3,<(stmt2,<>)>,
       <>,
       ({(stmt2,1)},
        {})),
    (4,<(stmt3,<>)>,
       <>,
       ({(stmt3,1),(else,1)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(1,3),(3,t),(1,4),(4,t)},
   s,
   t))}

program simplecase;
begin
  stmt1;
  case man of
    good : stmt1;
    bad : stmt2;
    normal : stmt3;
  end
end.

{((simplecase,<>,{}),
  ({(s,<>,<>,0)
    (1,<(stmt1,<>)>,
       <man>,
       ({(stmt1,1),(;,1),(case..of..end,1),(begin..end,1),(.,1)},
```

```
        {(man,1)})),
    (3,<(stmt1,<>)>,
       <>,
       ({(stmt1,1)},
        {(good:,1)})),
    (4,<(stmt2,<>)>,
       <>,
       ({(stmt2,1),(;,1)},
        {(bad:,1)})),
    (5,<(stmt3,<>)>,
       <>,
       ({(stmt3,1),(;,2)},
        {(normal:,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,3),(3,t),(1,4),(4,t),(1,5),(5,t)},
   s,
   t))}
```

```
program simplerepeat;
begin
  stmt1;
  repeat
    stmt2
  until cond
end.
```

```
{((simplerepeat,<>,{}),
  ({(s,<>,<>,0)
    (1,<(stmt1,<>)>,
       <>,
       ({(stmt1,1),(;,1),(begin..end,1),(.,1)},
        {})),
    (2,<(stmt2,<>)>,
       <cond>,
       ({(stmt2,1),(repeat..until,1)},
        {(cond,1)}))
    (t,<>,<>,0)},
   {(s,1),(2,t),(2,2),(1,2)},
   s,
   t))}
```

```
program simplewhile;
begin
  stmt1;
  while cond do stmt1
end.
```

```
{((simplewhile,<>,{}),
  ({(s,<>,<>,0)
    (1,<(stmt1,<>)>,
       <>,
       ({(stmt1,1),(;,1),(begin..end,1),(.,1)},
        {})),
    (2,<>,
       <cond>,
       ({(while..do,1)},
        {(cond,1)})),
    (3,<(stmt1,<>)>,
       <>,
       ({(stmt1,1)},
```

```
        {}))
    (t,<>,<>,0)},
   {(s,1),(2,3),(3,2),(2,t),(1,2)},
   s,
   t))}

program simplefor;
begin
  stmt1;
  for i := init to final do stmt2
end.

{((simplefor,<>,{}),
  ({(s,<>,<>,0)
    (1,<(stmt1,<>),(i,<init>)>,
       <>,
       ({(stmt1,1),(;,1),(for..to..do,1),(begin..end,1),(.,1)},
        {(init,1),(i,1)})),
    (3,<>,
       <i,final>,
       ({},
        {(final,1)})),
    (4,<(stmt2,<>),(i,<i>)>,
       <>,
       ({(stmt2,1)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(1,3),(3,4),(4,3),(3,t)},
   s,
   t))}

program ifthensimple;
begin
  if cond then stmt1;
  stmt2;
  stmt3;
end.

{((ifthensimple,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <cond>,
       ({(if..then,1),(begin..end,1),(.,1)},
        {(cond,1)})),
    (2,<(stmt1,<>)>,
       <>,
       ({(stmt1,1)},
        {})),
    (6,<(stmt2,<>),(stmt3,<>)>,
       <>,
       ({(;,3)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(1,2),(2,6),(1,6),(6,t)},
   s,
   t))}

program ifthenifthen;
begin
  if cond1 then stmt1;
  if cond2 then stmt2
```

```
end.

{((ifthenifthen,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <cond1>,
       ({(if..then,1),(begin..end,1),(.,1)},
        {(cond1,1)})),
    (2,<(stmt1,<>)>,
       <>,
       ({(stmt1,1)},
        {})),
    (4,<>,
       <cond2>,
       ({(;,1)},
        {(cond2,1)})),
    (5,<(stmt2,<>)>,
       <>,
       ({(stmt2,1)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(1,2),(2,4),(1,4),(4,5),(5,t),(4,t)},
   s,
   t))}

program ifthenifthenelse;
begin
  if cond1 then stmt1;
  if cond2 then stmt2
  else stmt3
end.

{((ifthenifthenelse,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <cond1>,
       ({(if..then,1),(begin..end,1),(.,1)},
        {(cond1,1)})),
    (2,<(stmt1,<>)>,
       <>,
       ({(stmt1,1)},
        {})),
    (4,<>,
       <cond2>,
       ({(;,1)},
        {(cond2,1)})),
    (5,<(stmt2,<>)>,
       <>,
       ({(stmt2,1)},
        {})),
    (6,<(stmt3,<>)>,
       <>,
       ({(stmt3,1),(else,1)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(1,2),(2,4),(1,4),(4,5),(5,t),(4,6),(6,t)},
   s,
   t))}

program ifthencase;
begin
```

```
  if cond then stmt1;
  case a of
    1: stmt2;
    2: stmt3;
    3: srmr4
  end
end.

{((ifthencase,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <cond>,
       ({(if..then,1),(begin..end,1),(.,1)},
        {(cond,1)})),
    (2,<(stmt1,<>)>,
       <>,
       ({(stmt1,1)},
        {})),
    (4,<>,
       <a>,
       ({(;,1)},
        {(a,1)})),
    (5,<(stmt2,<>)>,
       <>,
       ({(stmt2,1)},
        {(1:,1)})),
    (6,<(stmt3,<>)>,
       <>,
       ({(stmt3,1),(;,1)},
        {(2:,1)})),
    (7,<(srmr4,<>)>,
       <>,
       ({(srmr4,1),(;,1)},
        {(3:,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,2),(2,4),(1,4),(4,5),(5,t),(4,6),(6,t),(4,7),(7,t)},
   s,
   t))}

program ifrepeat;
begin
  if cond1 then stmt1;
  repeat
    stmt
  until cond2
end.

{((ifrepeat,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <cond1>,
       ({(if..then,1),(begin..end,1),(.,1)},
        {(cond1,1)})),
    (2,<(stmt1,<>)>,
       <>,
       ({(stmt1,1)},
        {})),
    (4,<(stmt,<>)>,
       <cond2>,
       ({(;,1)},
        {(cond2,1)}))
```

```
    (t,<>,<>,0)},
   {(s,1),(1,2),(2,4),(1,4),(4,t),(4,4)},
   s,
   t))}

program ifthenwhile;
begin
  if cond1 then stmt1;
  while cond2 do stmt2
end.

{((ifthenwhile,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <cond1>,
       ({(if..then,1),(begin..end,1),(.,1)},
        {(cond1,1)})),
    (2,<(stmt1,<>)>,
       <>,
       ({(stmt1,1)},
        {})),
    (4,<>,
       <cond2>,
       ({(;,1)},
        {(cond2,1)})),
    (5,<(stmt2,<>)>,
       <>,
       ({(stmt2,1)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(1,2),(2,4),(1,4),(4,5),(5,4),(4,t)},
   s,
   t))}

program iffor;
begin
  if cond then stmt1;
  for i := init downto final do stmt2
end.

{((iffor,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <cond>,
       ({(if..then,1),(begin..end,1),(.,1)},
        {(cond,1)})),
    (2,<(stmt1,<>)>,
       <>,
       ({(stmt1,1)},
        {})),
    (4,<(i,<init>)>,
       <>,
       ({(:=,1)},
        {(init,1),(i,1)})),
    (5,<>,
       <i,final>,
       ({},
        {(final,1)})),
    (6,<(stmt2,<>),(i,<i>)>,
       <>,
       ({(stmt2,1)},
```

```
        {}))
    (t,<>,<>,0)},
   {(s,1),(1,2),(2,4),(1,4),(4,5),(5,6),(6,5),(5,t)},
   s,
   t))}

program ifthenelsesimple;
begin
  if cond then stmt1 else stmt2;
  stmt3
end.

{((ifthenelsesimple,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <cond>,
       ({(if..then,1),(begin..end,1),(.,1)},
        {(cond,1)})),
    (2,<(stmt1,<>)>,
       <>,
       ({(stmt1,1)},
        {})),
    (3,<(stmt2,<>)>,
       <>,
       ({(stmt2,1),(else,1)},
        {})),
    (5,<(stmt3,<>)>,
       <>,
       ({(;,1)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(1,2),(2,5),(1,3),(3,5),(5,t)},
   s,
   t))}

program ifthenelseif;
begin
  if cond1 then stmt1
  else stmt2;
  if cond2 then stmt3
end.

{((ifthenelseif,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <cond1>,
       ({(if..then,1),(begin..end,1),(.,1)},
        {(cond1,1)})),
    (2,<(stmt1,<>)>,
       <>,
       ({(stmt1,1)},
        {})),
    (3,<(stmt2,<>)>,
       <>,
       ({(stmt2,1),(else,1)},
        {})),
    (5,<>,
       <cond2>,
       ({(;,1)},
        {(cond2,1)})),
    (6,<(stmt3,<>)>,
```

```
        <>,
        ({(stmt3,1)},
         {}))
     (t,<>,<>,0)},
    {(s,1),(1,2),(2,5),(1,3),(3,5),(5,6),(6,t),(5,t)},
    s,
    t))}

program ifthenelseifthenelse;
begin
  if cond1 then stmt1
  else stmt2;
  if cond2 then stmt3
  else stmt4
end.

{((ifthenelseifthenelse,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <cond1>,
       ({(if..then,1),(begin..end,1),(.,1)},
        {(cond1,1)})),
    (2,<(stmt1,<>)>,
       <>,
       ({(stmt1,1)},
        {})),
    (3,<(stmt2,<>)>,
       <>,
       ({(stmt2,1),(else,1)},
        {})),
    (5,<>,
       <cond2>,
       ({(;,1)},
        {(cond2,1)})),
    (6,<(stmt3,<>)>,
       <>,
       ({(stmt3,1)},
        {})),
    (7,<(stmt4,<>)>,
       <>,
       ({(stmt4,1),(else,1)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(1,2),(2,5),(1,3),(3,5),(5,6),(6,t),(5,7),(7,t)},
   s,
   t))}

program ifthenelsecase;
begin
  if cond then stmt1 else stmt2;
  case status of
    a : male;
    b : female;
    c : child
  end
end.

{((ifthenelsecase,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <cond>,
       ({(if..then,1),(begin..end,1),(.,1)},
```

```
        {(cond,1)})),
    (2,<(stmt1,<>)>,
       <>,
       ({(stmt1,1)},
        {})),
    (3,<(stmt2,<>)>,
       <>,
       ({(stmt2,1),(else,1)},
        {})),
    (5,<>,
       <status>,
       ({(;,1)},
        {(status,1)})),
    (6,<(male,<>)>,
       <>,
       ({(male,1)},
        {(a:,1)})),
    (7,<(female,<>)>,
       <>,
       ({(female,1),(;,1)},
        {(b:,1)})),
    (8,<(child,<>)>,
       <>,
       ({(child,1),(;,1)},
        {(c:,1)}))
    (t,<>,<>,0)},
  {(s,1),(1,2),(2,5),(1,3),(3,5),(5,6),(6,t),(5,7),(7,t),(5,8),
   (8,t)},
  s,
  t))}

program ifthenelserepeat;
begin
  if cond1 then stmt1 else stmt2;
  repeat
    stmt3
  until cond2
end.

{((ifthenelserepeat,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <cond1>,
       ({(if..then,1),(begin..end,1),(.,1)},
        {(cond1,1)})),
    (2,<(stmt1,<>)>,
       <>,
       ({(stmt1,1)},
        {})),
    (3,<(stmt2,<>)>,
       <>,
       ({(stmt2,1),(else,1)},
        {})),
    (5,<(stmt3,<>)>,
       <cond2>,
       ({(;,1)},
        {(cond2,1)}))
    (t,<>,<>,0)},
  {(s,1),(1,2),(2,5),(1,3),(3,5),(5,t),(5,5)},
  s,
  t))}
```

```
program ifthenelsewhile;
begin
  if cond1 then stmt1 else stmt2;
  while cond2 do stmt3
end.

{((ifthenelsewhile,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <cond1>,
       ({(if..then,1),(begin..end,1),(.,1)},
        {(cond1,1)})),
    (2,<(stmt1,<>)>,
       <>,
       ({(stmt1,1)},
        {})),
    (3,<(stmt2,<>)>,
       <>,
       ({(stmt2,1),(else,1)},
        {})),
    (5,<>,
       <cond2>,
       ({(;,1)},
        {(cond2,1)})),
    (6,<(stmt3,<>)>,
       <>,
       ({(stmt3,1)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(1,2),(2,5),(1,3),(3,5),(5,6),(6,5),(5,t)},
   s,
   t))}

program ifthenelsefor;
begin
  if cond then stmt1 else stmt2;
  for i := init to final do stmt3
end.

{((ifthenelsefor,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <cond>,
       ({(if..then,1),(begin..end,1),(.,1)},
        {(cond,1)})),
    (2,<(stmt1,<>)>,
       <>,
       ({(stmt1,1)},
        {})),
    (3,<(stmt2,<>)>,
       <>,
       ({(stmt2,1),(else,1)},
        {})),
    (5,<(i,<init>)>,
       <>,
       ({(;,1)},
        {(init,1),(i,1)})),
    (6,<>,
       <i,final>,
       ({},
```

```
        {(final,1)})),
    (7,<(stmt3,<>),(i,<i>)>,
       <>,
       ({(stmt3,1)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(1,2),(2,5),(1,3),(3,5),(5,6),(6,7),(7,6),(6,t)},
   s,
   t))}

program casesimple;
begin
  case what of
    1 : stmt1;
    2,4 : stmt2;
    3 : stmt3
  end;
  stmt4
end.

{((casesimple,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <what>,
       ({(case..of..end,1),(begin..end,1),(.,1)},
        {(what,1)})),
    (2,<(stmt1,<>)>,
       <>,
       ({(stmt1,1)},
        {(1:,1)})),
    (3,<(stmt2,<>)>,
       <>,
       ({(stmt2,1),(,,1),(;,1)},
        {(2:,1),(4:,1)})),
    (4,<(stmt3,<>)>,
       <>,
       ({(stmt3,1),(;,1)},
        {(3:,1)})),
    (6,<(stmt4,<>)>,
       <>,
       ({(;,1)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(1,2),(2,6),(1,3),(3,6),(1,4),(4,6),(6,t)},
   s,
   t))}

program caseifthen;
begin
  case what of
    1: stmt1;
    2: stmt2;
    3: stmt3
  end;
  if cond then stmt4
end.

{((caseifthen,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <what>,
       {{(case..of..end,1),(begin..end,1),(.,1)},
```

```
        {(what,1)})),
    (2,<(stmt1,<>)>,
       <>,
       ({(stmt1,1)},
        {(1:,1)})),
    (3,<(stmt2,<>)>,
       <>,
       ({(stmt2,1),(;,1)},
        {(2:,1)})),
    (4,<(stmt3,<>)>,
       <>,
       ({(stmt3,1),(;,1)},
        {(3:,1)})),
    (6,<>,
       <cond>,
       ({(;,1)},
        {(cond,1)})),
    (7,<(stmt4,<>)>,
       <>,
       ({(stmt4,1)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(1,2),(2,6),(1,3),(3,6),(1,4),(4,6),(6,7),(7,t),(6,t)},
   s,
   t))}

program caseifthenelse;
begin
  case what of
    1: stmt1;
    2: stmt2;
    3: stmt3;
  end;
  if cond then stmt4 else stmt5
end.

{((caseifthenelse,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <what>,
       ({(case..of..end,1),(begin..end,1),(.,1)},
        {(what,1)})),
    (2,<(stmt1,<>)>,
       <>,
       ({(stmt1,1)},
        {(1:,1)})),
    (3,<(stmt2,<>)>,
       <>,
       ({(stmt2,1),(;,1)},
        {(2:,1)})),
    (4,<(stmt3,<>)>,
       <>,
       ({(stmt3,1),(;,2)},
        {(3:,1)})),
    (6,<>,
       <cond>,
       ({(;,1)},
        {(cond,1)})),
    (7,<(stmt4,<>)>,
       <>,
       ({(stmt4,1)},
```

```
        {})),
    (8,<(stmt5,<>)>,
       <>,
       ({(stmt5,1),(else,1)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(1,2),(2,6),(1,3),(3,6),(1,4),(4,6),(6,7),(7,t),(6,8),
    (8,t)},
   s,
   t))}
```

```
program casecase;
begin
  case one of
    +1: stmt1;
    +2: stmt2;
    +3: stmt3
  end;
  case two of
    -1: stmt4;
    -2: stmt5;
    -3: stmt6
  end
end.
```

```
{((casecase,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <one>,
       ({(case..of..end,1),(begin..end,1),(.,1)},
        {(one,1)})),
    (2,<(stmt1,<>)>,
       <>,
       ({(stmt1,1)},
        {(+1:,1)})),
    (3,<(stmt2,<>)>,
       <>,
       ({(stmt2,1),(;,1)},
        {(+2:,1)})),
    (4,<(stmt3,<>)>,
       <>,
       ({(stmt3,1),(;,1)},
        {(+3:,1)})),
    (6,<>,
       <two>,
       ({(;,1)},
        {(two,1)})),
    (7,<(stmt4,<>)>,
       <>,
       ({(stmt4,1)},
        {(-1:,1)})),
    (8,<(stmt5,<>)>,
       <>,
       ({(stmt5,1),(;,1)},
        {(-2:,1)})),
    (9,<(stmt6,<>)>,
       <>,
       ({(stmt6,1),(;,1)},
        {(-3:,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,2),(2,6),(1,3),(3,6),(1,4),(4,6),(6,7),(7,t),(6,8),
```

```
    (8,t),(6,9),(9,t)},
   s,
   t))}

program caserepeat;
begin
  case rec of
    1: stmt1;
    2: stmt2;
    3: stmt3
  end;
  repeat stmt4 until cond
end.

{((caserepeat,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <rec>,
       ({(case..of..end,1),(begin..end,1),(.,1)},
        {(rec,1)})),
    (2,<(stmt1,<>)>,
       <>,
       ({(stmt1,1)},
        {(1:,1)})),
    (3,<(stmt2,<>)>,
       <>,
       ({(stmt2,1),(;,1)},
        {(2:,1)})),
    (4,<(stmt3,<>)>,
       <>,
       ({(stmt3,1),(;,1)},
        {(3:,1)})),
    (6,<(stmt4,<>)>,
       <cond>,
       ({(;,1)},
        {(cond,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,2),(2,6),(1,3),(3,6),(1,4),(4,6),(6,t),(6,6)},
   s,
   t))}

program casewhile;
begin
  case what of
    1: stmt1;
    2: stmt2;
    3: stmt3
  end;
  while cond do stmt4
end.

{((casewhile,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <what>,
       ({(case..of..end,1),(begin..end,1),(.,1)},
        {(what,1)})),
    (2,<(stmt1,<>)>,
       <>,
       ({(stmt1,1)},
        {(1:,1)})),
```

```
    (3,<(stmt2,<>)>,
       <>,
       ({(stmt2,1),(;,1)},
        {(2:,1)})),
    (4,<(stmt3,<>)>,
       <>,
       ({(stmt3,1),(;,1)},
        {(3:,1)})),
    (6,<>,
       <cond>,
       ({(;,1)},
        {(cond,1)})),
    (7,<(stmt4,<>)>,
       <>,
       ({(stmt4,1)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(1,2),(2,6),(1,3),(3,6),(1,4),(4,6),(6,7),(7,6),(6,t)},
   s,
   t))}
```

```
program casefor;
begin
  case how of
    1: stmt1;
    2: stmt2;
    3: stmt3
  end;
  for i := init to final do stmt4
end.
```

```
{((casefor,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <how>,
       ({(case..of..end,1),(begin..end,1),(.,1)},
        {(how,1)})),
    (2,<(stmt1,<>)>,
       <>,
       ({(stmt1,1)},
        {(1:,1)})),
    (3,<(stmt2,<>)>,
       <>,
       ({(stmt2,1),(;,1)},
        {(2:,1)})),
    (4,<(stmt3,<>)>,
       <>,
       ({(stmt3,1),(;,1)},
        {(3:,1)})),
    (6,<(i,<init>)>,
       <>,
       ({(;,1)},
        {(init,1),(i,1)})),
    (7,<>,
       <i,final>,
       ({},
        {(final,1)})),
    (8,<(stmt4,<>),(i,<i>)>,
       <>,
       ({(stmt4,1)},
        {}))
```

```
    (t,<>,<>,0)},
   {(s,1),(1,2),(2,6),(1,3),(3,6),(1,4),(4,6),(6,7),(7,8),(8,7),
    (7,t)},
   s,
   t))}

program repeatsimple;
begin
  repeat
    stmt1
  until cond;
  stmt2
end.

{((repeatsimple,<>,{}),
  ({(s,<>,<>,0)
    (1,<(stmt1,<>)>,
       <cond>,
       ({(stmt1,1),(repeat..until,1),(begin..end,1),(.,1)},
        {(cond,1)})),
    (3,<(stmt2,<>)>,
       <>,
       ({(;,1)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(1,3),(1,1),(3,t)},
   s,
   t))}

program repeatif;
begin
  repeat stmt1 until cond1;
  if cond2 then stmt2
end.

{((repeatif,<>,{}),
  ({(s,<>,<>,0)
    (1,<(stmt1,<>)>,
       <cond1>,
       ({(stmt1,1),(repeat..until,1),(begin..end,1),(.,1)},
        {(cond1,1)})),
    (3,<>,
       <cond2>,
       ({(;,1)},
        {(cond2,1)})),
    (4,<(stmt2,<>)>,
       <>,
       ({(stmt2,1)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(1,3),(1,1),(3,4),(4,t),(3,t)},
   s,
   t))}

program repeatifthenelse;
begin
  repeat stmt1 until cond1;
  if cond2 then stmt2 else stmt3
end.

{((repeatifthenelse,<>,{}),
```

```
  ({(s,<>,<>,0)
    (1,<(stmt1,<>)>,
       <cond1>,
       ({(stmt1,1),(repeat..until,1),(begin..end,1),(.,1)},
        {(cond1,1)})),
    (3,<>,
       <cond2>,
       ({(;,1)},
        {(cond2,1)})),
    (4,<(stmt2,<>)>,
       <>,
       ({(stmt2,1)},
        {})),
    (5,<(stmt3,<>)>,
       <>,
       ({(stmt3,1),(else,1)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(1,3),(1,1),(3,4),(4,t),(3,5),(5,t)},
   s,
   t))}

program repeatcase;
begin
  repeat
    stmt1
  until cond;
  case what of
    1: stmt2;
    2: stmt3;
    3: stmt4
  end
end.

{((repeatcase,<>,{}),
  ({(s,<>,<>,0)
    (1,<(stmt1,<>)>,
       <cond>,
       ({(stmt1,1),(repeat..until,1),(begin..end,1),(.,1)},
        {(cond,1)})),
    (3,<>,
       <what>,
       ({(;,1)},
        {(what,1)})),
    (4,<(stmt2,<>)>,
       <>,
       ({(stmt2,1)},
        {(1:,1)})),
    (5,<(stmt3,<>)>,
       <>,
       ({(stmt3,1),(;,1)},
        {(2:,1)})),
    (6,<(stmt4,<>)>,
       <>,
       ({(stmt4,1),(;,1)},
        {(3:,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,3),(1,1),(3,4),(4,t),(3,5),(5,t),(3,6),(6,t)},
   s,
   t))}
```

```
program repeatrepeat;
begin
  repeat stmt1 until cond1;
  repeat stmt1 until cond2
end.

{((repeatrepeat,<>,{}),
  ({(s,<>,<>,0)
    (1,<(stmt1,<>)>,
       <cond1>,
       ({(stmt1,1),(repeat..until,1),(begin..end,1),(.,1)},
        {(cond1,1)})),
    (3,<(stmt1,<>)>,
       <cond2>,
       ({(;,1)},
        {(cond2,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,3),(1,1),(3,t),(3,3)},
   s,
   t))}

program repeatwhile;
begin
  repeat stmt1 until cond1;
  while cond2 do stmt2
end.

{((repeatwhile,<>,{}),
  ({(s,<>,<>,0)
    (1,<(stmt1,<>)>,
       <cond1>,
       ({(stmt1,1),(repeat..until,1),(begin..end,1),(.,1)},
        {(cond1,1)})),
    (3,<>,
       <cond2>,
       ({(;,1)},
        {(cond2,1)})),
    (4,<(stmt2,<>)>,
       <>,
       ({(stmt2,1)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(1,3),(1,1),(3,4),(4,3),(3,t)},
   s,
   t))}

program repeatfor;
begin
  repeat stmt1 until cond1;
  for i:= init to final do stmt2
end.

{((repeatfor,<>,{}),
  ({(s,<>,<>,0)
    (1,<(stmt1,<>)>,
       <cond1>,
       ({(stmt1,1),(repeat..until,1),(begin..end,1),(.,1)},
        {(cond1,1)})),
    (3,<(i,<init>)>,
       <>,
```

```
      ({(;,1)},
       {(init,i),(i,1)})),
    (4,<>,
      <i,final>,
      ({},
       {(final,1)})),
    (5,<(stmt2,<>),(i,<i>)>,
      <>,
      ({(stmt2,1)},
       {}))
    (t,<>,<>,0)},
  {(s,1),(1,3),(1,1),(3,4),(4,5),(5,4),(4,t)},
  s,
  t))}

program whilesimple;
begin
  while cond do stmt1;
  stmt2;
end.

{((whilesimple,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
      <cond>,
      ({(while..do,1),(begin..end,1),(.,1)},
       {(cond,1)})),
    (2,<(stmt1,<>)>,
      <>,
      ({(stmt1,1)},
       {})),
    (5,<(stmt2,<>)>,
      <>,
      ({(;,2)},
       {}))
    (t,<>,<>,0)},
  {(s,1),(1,2),(2,1),(1,5),(5,t)},
  s,
  t))}

program whileif;
begin
  while cond1 do stmt1;
  if cond2 then stmt2
end.

{((whileif,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
      <cond1>,
      ({(while..do,1),(begin..end,1),(.,1)},
       {(cond1,1)})),
    (2,<(stmt1,<>)>,
      <>,
      ({(stmt1,1)},
       {})),
    (4,<>,
      <cond2>,
      ({(;,1)},
       {(cond2,1)})),
    (5,<(stmt2,<>)>,
```

```
      <>,
      ({(stmt2,1)},
       {}))
   (t,<>,<>,0)},
  {(s,1),(1,2),(2,1),(1,4),(4,5),(5,t),(4,t)},
  s,
  t))}
```

```
program whileifthenelse;
begin
  while cond1 do stmt1;
  if cond2 then stmt2 else stmt3
end.
```

```
{((whileifthenelse,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <cond1>,
       ({(while..do,1),(begin..end,1),(.,1)},
        {(cond1,1)})),
    (2,<(stmt1,<>)>,
       <>,
       ({(stmt1,1)},
        {})),
    (4,<>,
       <cond2>,
       ({(;,1)},
        {(cond2,1)})),
    (5,<(stmt2,<>)>,
       <>,
       ({(stmt2,1)},
        {})),
    (6,<(stmt3,<>)>,
       <>,
       ({(stmt3,1),(else,1)},
        {}))
   (t,<>,<>,0)},
  {(s,1),(1,2),(2,1),(1,4),(4,5),(5,t),(4,6),(6,t)},
  s,
  t))}
```

```
program whilecase;
begin
  while cond do something;
  case any of
    10: come;
    20: go;
    30: stop
  end
end.
```

```
{((whilecase,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <cond>,
       ({(while..do,1),(begin..end,1),(.,1)},
        {(cond,1)})),
    (2,<(something,<>)>,
       <>,
       ({(something,1)},
        {})),
```

```
    (4,<>,
       <any>,
       ({(;,1)},
        {(any,1)})),
    (5,<(come,<>)>,
       <>,
       ({(come,1)},
        {(10:,1)})),
    (6,<(go,<>)>,
       <>,
       ({(go,1),(;,1)},
        {(20:,1)})),
    (7,<(stop,<>)>,
       <>,
       ({(stop,1),(;,1)},
        {(30:,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,2),(2,1),(1,4),(4,5),(5,t),(4,6),(6,t),(4,7),(7,t)},
   s,
   t))}
```

```
program whilerepeat;
begin
  while cond1 do stmt1;
  repeat
    stmt2
  until cond2
end.
```

```
{((whilerepeat,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <cond1>,
       ({(while..do,1),(begin..end,1),(.,1)},
        {(cond1,1)})),
    (2,<(stmt1,<>)>,
       <>,
       ({(stmt1,1)},
        {})),
    (4,<(stmt2,<>)>,
       <cond2>,
       ({(;,1)},
        {(cond2,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,2),(2,1),(1,4),(4,t),(4,4)},
   s,
   t))}
```

```
program whilewhile;
begin
  while cond1 do stmt1;
  while cond2 do stmt2
end.
```

```
{((whilewhile,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <cond1>,
       ({(while..do,1),(begin..end,1),(.,1)},
        {(cond1,1)})),
```

```
    (2,<(stmt1,<>)>,
       <>,
       ({(stmt1,1)},
        {})),
    (4,<>,
       <cond2>,
       ({(;,1)},
        {(cond2,1)})),
    (5,<(stmt2,<>)>,
       <>,
       ({(stmt2,1)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(1,2),(2,1),(1,4),(4,5),(5,4),(4,t)},
   s,
   t))}

program whilefor;
begin
  while cond do stmt1;
  for i := init to final do stmt2
end.

{((whilefor,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <cond>,
       ({(while..do,1),(begin..end,1),(.,1)},
        {(cond,1)})),
    (2,<(stmt1,<>)>,
       <>,
       ({(stmt1,1)},
        {})),
    (4,<(i,<init>)>,
       <>,
       ({(;,1)},
        {(init,1),(i,1)})),
    (5,<>,
       <i,final>,
       ({},
        {(final,1)})),
    (6,<(stmt2,<>),(i,<i>)>,
       <>,
       ({(stmt2,1)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(1,2),(2,1),(1,4),(4,5),(5,6),(6,5),(5,t)},
   s,
   t))}

program forsimple;
begin
  for i := init to final do stmt1;
  stmt2
end.

{((forsimple,<>,{}),
  ({(s,<>,<>,0)
    (1,<(i,<init>)>,
       <>,
       ({(for..to..do,1),(begin..end,1),(.,1)},
```

```
        {(init,1),(i,1)})),
    (2,<>,
       <i,final>,
       ({},
        {(final,1)})),
    (3,<(stmt1,<>),(i,<i>)>,
       <>,
       ({(stmt1,1)},
        {})),
    (5,<(stmt2,<>)>,
       <>,
       ({(;,1)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(1,2),(2,3),(3,2),(2,5),(5,t)},
   s,
   t))}

program forifthen;
begin
  for i := init to final do stmt1;
  if cond then stmt2
end.

{((forifthen,<>,{}),
  ({(s,<>,<>,0)
    (1,<(i,<init>)>,
       <>,
       ({(for..to..do,1),(begin..end,1),(.,1)},
        {(init,1),(i,1)})),
    (2,<>,
       <i,final>,
       ({},
        {(final,1)})),
    (3,<(stmt1,<>),(i,<i>)>,
       <>,
       ({(stmt1,1)},
        {})),
    (5,<>,
       <cond>,
       ({(;,1)},
        {(cond,1)}))),
    (6,<(stmt2,<>)>,
       <>,
       ({(stmt2,1)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(1,2),(2,3),(3,2),(2,5),(5,6),(6,t),(5,t)},
   s,
   t))}

program forifthenelse;
begin
  for i := init to final do stmt1;
  if cond then stmt2 else stmt3
end.

{((forifthenelse,<>,{}),
  ({(s,<>,<>,0)
    (1,<(i,<init>)>,
       <>,
```

```
      ({(for..to..do,1),(begin..end,1),(.,1)},
       {(init,1),(i,1)})),
    (2,<>,
       <i,final>,
       ({},
        {(final,1)})),
    (3,<(stmt1,<>),(i,<i>)>,
       <>,
       ({(stmt1,1)},
        {})),
    (5,<>,
       <cond>,
       ({(;,1)},
        {(cond,1)})),
    (6,<(stmt2,<>)>,
       <>,
       ({(stmt2,1)},
        {})),
    (7,<(stmt3,<>)>,
       <>,
       ({(stmt3,1),(else,1)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(1,2),(2,3),(3,2),(2,5),(5,6),(6,t),(5,7),(7,t)},
   s,
   t))}
```

```
program forcase;
begin
  for i := 1 to 22 do stmt1;
  case x of
    const1 : stmt2;
    const2 : stmt3;
    const3 : stmt4
  end
end.
```

```
{((forcase,<>,{}),
  ({(s,<>,<>,0)
    (1,<(i,<i>)>,
       <>,
       ({(for..to..do,1),(begin..end,1),(.,1)},
        {(1,1),(i,1)})),
    (2,<>,
       <i,22>,
       ({},
        {(22,1)})),
    (3,<(stmt1,<>),(i,<i>)>,
       <>,
       ({(stmt1,1)},
        {})),
    (5,<>,
       <x>,
       ({(;,1)},
        {(x,1)})),
    (6,<(stmt2,<>)>,
       <>,
       ({(stmt2,1)},
        {(const1:,1)})),
    (7,<(stmt3,<>)>,
       <>,
```

```
        ({(stmt3,1),(;,1)},
         {(const2:,1)})),
     (8,<(stmt4,<>)>,
        <>,
        ({(stmt4,1),(;,1)},
         {(const3:,1)}))
     (t,<>,<>,0)},
    {(s,1),(1,2),(2,3),(3,2),(2,5),(5,6),(6,t),(5,7),(7,t),(5,8),
     (8,t)},
    s,
    t))}

program forrepeat;
begin
  for i := init to final do stmt1;
  repeat
    stmt2
  until cond
end.

{((forrepeat,<>,{}),
  ({(s,<>,<>,0)
    (1,<(i,<init>)>,
       <>,
       ({(for..to..do,1),(begin..end,1),(.,1)},
        {(init,1),(i,1)})),
    (2,<>,
       <i,final>,
       ({},
        {(final,1)})),
    (3,<(stmt1,<>),(i,<i>)>,
       <>,
       ({(stmt1,1)},
        {})),
    (5,<(stmt2,<>)>,
       <cond>,
       ({(;,1)},
        {(cond,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,2),(2,3),(3,2),(2,5),(5,t),(5,5)},
   s,
   t))}

program forwhile;
begin
  for i := init to final do stmt1;
  while cond do stmt2
end.

{((forwhile,<>,{}),
  ({(s,<>,<>,0)
    (1,<(i,<init>)>,
       <>,
       ({(for..to..do,1),(begin..end,1),(.,1)},
        {(init,1),(i,1)})),
    (2,<>,
       <i,final>,
       ({},
        {(final,1)})),
    (3,<(stmt1,<>),(i,<i>)>,
       <>,
```

```
      ({(stmt1,1)},
       {})),
    (5,<>,
      <cond>,
      ({(;,1)},
       {(cond,1)})),
    (6,<(stmt2,<>)>,
      <>,
      ({(stmt2,1)},
       {}))
    (t,<>,<>,0)},
   {(s,1),(1,2),(2,3),(3,2),(2,5),(5,6),(6,5),(5,t)},
   s,
   t))}
```

```
program forfor;
begin
  for i := 1 to n do
    stmt1;
  for j := 100 downto m do
    stmt2
end.
```

```
{((forfor,<>,{}),
  ({(s,<>,<>,0)
    (1,<(i,<i>)>,
      <>,
      ({(for..to..do,1),(begin..end,1),(.,1)},
       {(1,1),(i,1)})),
    (2,<>,
      <i,n>,
      ({},
       {(n,1)})),
    (3,<(stmt1,<>),(i,<i>)>,
      <>,
      ({(stmt1,1)},
       {})),
    (5,<(j,<100>)>,
      <>,
      ({(;,1)},
       {(100,1),(j,1)})),
    (6,<>,
      <j,m>,
      ({},
       {(m,1)})),
    (7,<(stmt2,<>),(j,<j>)>,
      <>,
      ({(stmt2,1)},
       {}))
    (t,<>,<>,0)},
   {(s,1),(1,2),(2,3),(3,2),(2,5),(5,6),(6,7),(7,6),(6,t)},
   s,
   t))}
```

B.3 Nesting

```
program ifnestedif;
begin
  if cond1 then
```

```
  if cond2 then stmt
end.

{((ifnestedif,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <cond1>,
       ({(if..then,1),(begin..end,1),(.,1)},
        {(cond1,1)})),
    (2,<>,
       <cond2>,
       ({(if..then,1)},
        {(cond2,1)})),
    (3,<(stmt,<>)>,
       <>,
       ({(stmt,1)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(2,3),(1,2),(1,t),(3,t),(2,t)},
   s,
   t))}

program ifthenelsenestedif;
begin
  if cond1 then
  if cond2 then stmt1
  else stmt2
end.

{((ifthenelsenestedif,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <cond1>,
       ({(if..then,1),(begin..end,1),(.,1)},
        {(cond1,1)})),
    (2,<>,
       <cond2>,
       ({(if..then,1)},
        {(cond2,1)})),
    (3,<(stmt1,<>)>,
       <>,
       ({(stmt1,1)},
        {})),
    (4,<(stmt2,<>)>,
       <>,
       ({(stmt2,1),(else,1)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(2,3),(2,4),(1,2),(1,t),(3,t),(4,t)},
   s,
   t))}

program casenestedif;
begin
  if cond then
    case what of
      one : stmt1;
      two : stmt2;
      three : stmt3;
    end
end.
```

```
{((casenestedif,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <cond>,
       ({(if..then,1),(begin..end,1),(.,1)},
        {(cond,1)})),
    (2,<>,
       <what>,
       ({(case..of..end,1)},
        {(what,1)})),
    (3,<(stmt1,<>)>,
       <>,
       ({(stmt1,1)},
        {(one:,1)})),
    (4,<(stmt2,<>)>,
       <>,
       ({(stmt2,1),(;,1)},
        {(two:,1)})),
    (5,<(stmt3,<>)>,
       <>,
       ({(stmt3,1),(;,2)},
        {(three:,1)}))
    (t,<>,<>,0)},
   {(s,1),(2,3),(2,4),(2,5),(1,2),(1,t),(3,t),(4,t),(5,t)},
   s,
   t))}

program repeatnestedif;
begin
  if cond1 then
    repeat
      stmt1;
      stmt2
    until cond2
end.

{((repeatnestedif,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <cond1>,
       ({(if..then,1),(begin..end,1),(.,1)},
        {(cond1,1)})),
    (2,<(stmt1,<>),(stmt2,<>)>,
       <cond2>,
       ({(stmt1,1),(;,1),(stmt2,1),(repeat..until,1)},
        {(cond2,1)}))
    (t,<>,<>,0)},
   {(s,1),(2,2),(1,2),(1,t),(2,t)},
   s,
   t))}

program whilenestedif;
begin
  if cond then
    while cond2 do
      stmt
end.

{((whilenestedif,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
```

```
      <cond>,
      ({(if..then,1),(begin..end,1),(.,1)},
       {(cond,1)})),
   (2,<>,
      <cond2>,
      ({(while..do,1)},
       {(cond2,1)})),
   (3,<(stmt,<>)>,
      <>,
      ({(stmt,1)},
       {}))
   (t,<>,<>,0)},
  {(s,1),(2,3),(3,2),(1,2),(1,t),(2,t)},
  s,
  t))}

program fornestedif;
begin
  if cond then
    for k := 1 to 10 do
      stmt
end.

{((fornestedif,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <cond>,
       ({(if..then,1),(begin..end,1),(.,1)},
        {(cond,1)})),
    (2,<(k,<1>)>,
       <>,
       ({(for..to..do,1)},
        {(1,1),(k,1)})),
    (3,<>,
       <k,10>,
       ({},
        {(10,1)})),
    (4,<(stmt,<>),(k,<k>)>,
       <>,
       ({(stmt,1)},
        {}))
    (t,<>,<>,0)},
  {(s,1),(2,3),(3,4),(4,3),(1,2),(1,t),(3,t)},
  s,
  t))}

program ifnestedifthenelse;
begin
  if cond1 then stmt1
  else if cond2 then stmt2
end.

{((ifnestedifthenelse,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <cond1>,
       ({(if..then,1),(begin..end,1),(.,1)},
        {(cond1,1)})),
    (2,<(stmt1,<>)>,
       <>,
       ({(stmt1,1)},
```

```
        {})),
    (3,<>,
       <cond2>,
       ({(if..then,1),(else,1)},
        {(cond2,1)})),
    (4,<(stmt2,<>)>,
       <>,
       ({(stmt2,1)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(3,4),(1,2),(2,t),(1,3),(4,t),(3,t)},
   s,
   t))}

program ifthenelsenestedifthenelse;
begin
  if cond1 then stmt1
  else if cond2 then stmt2
  else stmt3
end.

{((ifthenelsenestedifthenelse,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <cond1>,
       ({(if..then,1),(begin..end,1),(.,1)},
        {(cond1,1)})),
    (2,<(stmt1,<>)>,
       <>,
       ({(stmt1,1)},
        {})),
    (3,<>,
       <cond2>,
       ({(if..then,1),(else,1)},
        {(cond2,1)})),
    (4,<(stmt2,<>)>,
       <>,
       ({(stmt2,1)},
        {})),
    (5,<(stmt3,<>)>,
       <>,
       ({(stmt3,1),(else,1)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(3,4),(3,5),(1,2),(2,t),(1,3),(4,t),(5,t)},
   s,
   t))}

program casenestedifthenelse;
begin
  if cond then
    case boston of
      first : stmt1;
      second : stmt2;
      third : stmt3;
    end
  else
    case imagine of
      first : stmt4;
      second : stmt5;
      third : stmt6;
    end
```

```
end.

{((casenestedifthenelse,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <cond>,
       ({(if..then,1),(begin..end,1),(.,1)},
        {(cond,1)})),
    (2,<>,
       <boston>,
       ({(case..of..end,1)},
        {(boston,1)})),
    (3,<(stmt1,<>)>,
       <>,
       ({(stmt1,1)},
        {(first:,1)})),
    (4,<(stmt2,<>)>,
       <>,
       ({(stmt2,1),(;,1)},
        {(second:,1)})),
    (5,<(stmt3,<>)>,
       <>,
       ({(stmt3,1),(;,2)},
        {(third:,1)})),
    (7,<>,
       <imagine>,
       ({},
        {(imagine,1)})),
    (8,<(stmt4,<>)>,
       <>,
       ({(stmt4,1)},
        {(first:,1)})),
    (9,<(stmt5,<>)>,
       <>,
       ({(stmt5,1),(;,1)},
        {(second:,1)})),
    (10,<(stmt6,<>)>,
       <>,
       ({(stmt6,1),(;,2)},
        {(third:,1)}))
    (t,<>,<>,0)},
   {(s,1),(2,3),(2,4),(2,5),(7,8),(7,9),(7,10),(1,2),(1,7),(8,t),
    (9,t),(10,t),(3,t),(4,t),(5,t)},
   s,
   t)}

program repeatnestedifthenelse;
begin
  if lookingclean then fly
  else
    repeat wash until toldtodoso
end.

{((repeatnestedifthenelse,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <lookingclean>,
       ({(if..then,1),(begin..end,1),(.,1)},
        {(lookingclean,1)})),
    (2,<(fly,<>)>,
```

```
      <>,
      ({(fly,1)},
       {})),
    (3,<(wash,<>)>,
      <toldtodoso>,
      ({(wash,1),(repeat..until,1),(else,1)},
       {(toldtodoso,1)}))
    (t,<>,<>,0)},
   {(s,1),(3,3),(1,2),(2,t),(1,3),(3,t)},
   s,
   t))}
```

```
program whilenestedifthenelse;
begin
  if cond1 then
    while cond2 do
      stmt1
  else
    while cond3 do
      stmt2
end.
```

```
{((whilenestedifthenelse,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
      <cond1>,
      ({(if..then,1),(begin..end,1),(.,1)},
       {(cond1,1)})),
    (2,<>,
      <cond2>,
      ({(while..do,1)},
       {(cond2,1)})),
    (3,<(stmt1,<>)>,
      <>,
      ({(stmt1,1)},
       {})),
    (5,<>,
      <cond3>,
      ({},
       {(cond3,1)})),
    (6,<(stmt2,<>)>,
      <>,
      ({(stmt2,1)},
       {}))
    (t,<>,<>,0)},
   {(s,1),(2,3),(3,2),(5,6),(6,5),(1,2),(1,5),(5,t),(2,t)},
   s,
   t))}
```

```
program fornestedifthenelse;
begin
  if you then
    for count := 100 downto 1 do standup
  else sitdown
end.
```

```
{((fornestedifthenelse,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
      <you>,
      ({(if..then,1),(begin..end,1),(.,1)},
```

```
        {(you,1)})),
    (2,<(count,<100>)>,
       <>,
       ({(for..downto..do,1)},
        {(100,1),(count,1)})),
    (3,<>,
       <count,1>,
       ({},
        {(1,1)})),
    (4,<(standup,<>),(count,<count>)>,
       <>,
       ({(standup,1)},
        {})),
    (6,<(sitdown,<>)>,
       <>,
       ({},
        {})
    (t,<>,<>,0)},
   {(s,1),(2,3),(3,4),(4,3),(1,2),(1,6),(6,t),(3,t)},
   s,
   t))}
```

```
program ifnestedcase;
begin
  case special of
    1: if good then go;
    7: stmt := 3;
    2: if bad then stayhome;
  end
end.
```

```
{((ifnestedcase,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <special>,
       ({(case..of..end,1),(begin..end,1),(.,1)},
        {(special,1)})),
    (2,<>,
       <good>,
       ({(if..then,1)},
        {(good,1),(1:,1)})),
    (3,<(go,<>)>,
       <>,
       ({(go,1)},
        {})),
    (5,<(stmt,<3>)>,
       <>,
       ({(:=,1),(;,1)},
        {(3,1),(stmt,1),(7:,1)})),
    (6,<>,
       <bad>,
       ({(if..then,1)},
        {(bad,1),(2:,1)})),
    (7,<(stayhome,<>)>,
       <>,
       ({(stayhome,1),(;,2)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(2,3),(1,2),(1,5),(5,t),(3,t),(2,t),(6,7),(1,6),(7,t),
    (6,t)},
```

```
   s,
   t))}

program ifthenelsenestedcase;
begin
  case nnnn of
    3: if good then come else stop;
    5: anystatement;
    7: if good then stop else go;
  end
end.

{((ifthenelsenestedcase,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <nnnn>,
       ({(case..of..end,1),(begin..end,1),(.,1)},
        {(nnnn,1)})),
    (2,<>,
       <good>,
       ({(if..then,1)},
        {(good,1),(3:,1)})),
    (3,<(come,<>)>,
       <>,
       ({(come,1)},
        {})),
    (4,<(stop,<>)>,
       <>,
       ({(stop,1),(else,1)},
        {})),
    (6,<(anystatement,<>)>,
       <>,
       ({(anystatement,1),(;,1)},
        {(5:,1)})),
    (7,<>,
       <good>,
       ({(if..then,1)},
        {(good,1),(7:,1)})),
    (8,<(stop,<>)>,
       <>,
       ({(stop,1)},
        {})),
    (9,<(go,<>)>,
       <>,
       ({(go,1),(else,1),(;,2)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(2,3),(2,4),(1,2),(1,6),(6,t),(3,t),(4,t),(7,8),(7,9),
    (1,7),(8,t),(9,t)},
   s,
   t))}

program casenestedcase;
begin
  case outer of
    1: case inner1 of
 10: dead;
 okay: alive;
         noway: gotohell
       end;
```

```
    2: nothing;
    3: case inner2 of
 10: alive;
 20: dead;
        30: ill
      end;
  end
end.

{((casenestedcase,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <outer>,
       ({(case..of..end,1),(begin..end,1),(.,1)},
        {(outer,1)})),
    (2,<>,
       <inner1>,
       ({(case..of..end,1)},
        {(inner1,1),(1:,1)})),
    (3,<(dead,<>)>,
       <>,
       ({(dead,1)},
        {(10:,1)})),
    (4,<(alive,<>)>,
       <>,
       ({(alive,1),(;,1)},
        {(okay:,1)})),
    (5,<(gotohell,<>)>,
       <>,
       ({(gotohell,1),(;,1)},
        {(noway:,1)})),
    (7,<(nothing,<>)>,
       <>,
       ({(nothing,1),(;,1)},
        {(2:,1)})),
    (8,<>,
       <inner2>,
       ({(case..of..end,1)},
        {(inner2,1),(3:,1)})),
    (9,<(alive,<>)>,
       <>,
       ({(alive,1)},
        {(10:,1)})),
    (10,<(dead,<>)>,
       <>,
       ({(dead,1),(;,1)},
        {(20:,1)})),
    (11,<(ill,<>)>,
       <>,
       ({(ill,1),(;,3)},
        {(30:,1)}))
    (t,<>,<>,0)},
   {(s,1),(2,3),(2,4),(2,5),(8,2),(8,7),(8,9),(8,10),(8,11),(1,8),
    (7,t),(3,t),(4,t),(5,t),(9,t),(10,t),(11,t)},
   s,
   t))}

program casenestedrepeat;
begin
  case hey of
```

```
    1: repeat hello until satisfy;
    2: remaininlight;
    3: repeat bow until saystop;
  end
end.

{((casenestedrepeat,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <hey>,
       ({(case..of..end,1),(begin..end,1),(.,1)},
        {(hey,1)})),
    (2,<(hello,<>)>,
       <satisfy>,
       ({(hello,1),(repeat..until,1)},
        {(satisfy,1),(1:,1)})),
    (4,<(remaininlight,<>)>,
       <>,
       ({(remaininlight,1),(;,1)},
        {(2:,1)})),
    (5,<(bow,<>)>,
       <saystop>,
       ({(bow,1),(repeat..until,1),(;,2)},
        {(saystop,1),(3:,1)}))
    (t,<>,<>,0)},
   {(s,1),(2,2),(1,2),(1,4),(4,t),(2,t),(5,5),(1,5),(5,t)},
   s,
   t))}

program casenestedwhile;
begin
case what of
  0: while cond do stmt0;
  1: stmt1;
  2: while cond do stmt2;
end
end.

{((casenestedwhile,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <what>,
       ({(case..of..end,1),(begin..end,1),(.,1)},
        {(what,1)})),
    (2,<>,
       <cond>,
       ({(while..do,1)},
        {(cond,1),(0:,1)})),
    (3,<(stmt0,<>)>,
       <>,
       ({(stmt0,1)},
        {})),
    (5,<(stmt1,<>)>,
       <>,
       ({(stmt1,1),(;,1)},
        {(1:,1)})),
    (6,<>,
       <cond>,
       ({(while..do,1)},
        {(cond,1),(2:,1)})),
```

```
    (7,<(stmt2,<>)>,
       <>,
       ({(stmt2,1),(;,2)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(2,3),(3,2),(1,2),(1,5),(5,t),(2,t),(6,7),(7,6),(1,6),
    (6,t)},
   s,
   t))}

program casenestedfor;
begin
  case music of
    1: for j := bach to mahler do playingrecord;
    2: sleep;
    3: for i := beatles to direstraits do goconcert;
  end
end.

{((casenestedfor,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <music>,
       ({(case..of..end,1),(begin..end,1),(.,1)},
        {(music,1)})),
    (2,<(j,<bach>)>,
       <>,
       ({(for..to..do,1)},
        {(bach,1),(j,1),(1:,1)})),
    (3,<>,
       <j,mahler>,
       ({},
        {(mahler,1)})),
    (4,<(playingrecord,<>),(j,<j>)>,
       <>,
       ({(playingrecord,1)},
        {})),
    (6,<(sleep,<>)>,
       <>,
       ({(sleep,1),(;,1)},
        {(2:,1)})),
    (7,<(i,<beatles>)>,
       <>,
       ({(for..to..do,1)},
        {(beatles,1),(i,1),(3:,1)})),
    (8,<>,
       <i,direstraits>,
       ({},
        {(direstraits,1)})),
    (9,<(goconcert,<>),(i,<i>)>,
       <>,
       ({(goconcert,1),(;,2)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(2,3),(3,4),(4,3),(1,2),(1,6),(6,t),(3,t),(7,8),(8,9),
    (9,8),(1,7),(8,t)},
   s,
   t))}

program ifnestedrepeat;
```

```
begin
  repeat
    if vacation then wolf := 100
  until allyearlong
end.

{((ifnestedrepeat,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <vacation>,
       ({(if..then,1),(repeat..until,1),(begin..end,1),(.,1)},
        {(vacation,1)})),
    (2,<(wolf,<100>)>,
       <>,
       ({(:=,1)},
        {(100,1),(wolf,1)})),
    (3,<>,
       <allyearlong>,
       ({},
        {(allyearlong,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,2),(2,3),(1,3),(3,t),(3,1)},
   s,
   t))}

program ifthenelsenestedrepeat;
begin
  repeat
    if cond1 then stmt1
    else stmt2
  until cond2
end.

{((ifthenelsenestedrepeat,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <cond1>,
       ({(if..then,1),(repeat..until,1),(begin..end,1),(.,1)},
        {(cond1,1)})),
    (2,<(stmt1,<>)>,
       <>,
       ({(stmt1,1)},
        {})),
    (3,<(stmt2,<>)>,
       <>,
       ({(stmt2,1),(else,1)},
        {})),
    (4,<>,
       <cond2>,
       ({},
        {(cond2,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,2),(2,4),(1,3),(3,4),(4,t),(4,1)},
   s,
   t))}

program casenestedrepeat;
begin
  repeat
    case on of
      1: go;
```

```
      2: come;
      3: stay;
    end
  until stop
end.
```

```
{((casenestedrepeat,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <on>,
       ({(case..of..end,1),(repeat..until,1),(begin..end,1),(.,1)},
        {(on,1)})),
    (2,<(go,<>)>,
       <>,
       ({(go,1)},
        {(1:,1)})),
    (3,<(come,<>)>,
       <>,
       ({(come,1),(;,1)},
        {(2:,1)})),
    (4,<(stay,<>)>,
       <>,
       ({(stay,1),(;,2)},
        {(3:,1)})),
    (5,<>,
       <stop>,
       ({},
        {(stop,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,2),(2,5),(1,3),(3,5),(1,4),(4,5),(5,t),(5,1)},
   s,
   t))}
```

```
program repeatnestedrepeat;
begin
  repeat
    repeat
      stmt
    until cond1
  until cond2
end.
```

```
{((repeatnestedrepeat,<>,{}),
  ({(s,<>,<>,0)
    (1,<(stmt,<>)>,
       <cond1>,
       ({(stmt,1),(repeat..until,2),(begin..end,1),(.,1)},
        {(cond1,1)})),
    (2,<>,
       <cond2>,
       ({},
        {(cond2,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,2),(1,1),(2,t),(2,1)},
   s,
   t))}
```

```
program whilenestedrepeat;
begin
  repeat
    while cond1 do
```

```
      stmt1
  until cond2
end.

{((whilenestedrepeat,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <cond1>,
       ({(while..do,1),(repeat..until,1),(begin..end,1),(.,1)},
        {(cond1,1)})),
    (2,<(stmt1,<>)>,
       <>,
       ({(stmt1,1)},
        {})),
    (3,<>,
       <cond2>,
       ({},
        {(cond2,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,2),(2,1),(1,3),(3,t),(3,1)},
   s,
   t))}

program fornestedrepeat;
begin
  repeat
    for i := 1 to 3 do
      begin
        stand;
        sit
      end
  until stop
end.

{((fornestedrepeat,<>,{}),
  ({(s,<>,<>,0)
    (1,<(i,<i>)>,
       <>,
       ({(for..to..do,1),(repeat..until,1),(begin..end,1),(.,1)},
        {(1,1),(i,1)})),
    (2,<>,
       <i,3>,
       ({},
        {(3,1)})),
    (3,<(stand,<>),(sit,<>),(i,<i>)>,
       <>,
       ({(stand,1),(;,1),(sit,1),(begin..end,1)},
        {})),
    (5,<>,
       <stop>,
       ({},
        {(stop,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,2),(2,3),(3,2),(2,5),(5,t),(5,1)},
   s,
   t))}

program ifnestedwhile;
  begin
    while cond1 do
      if cond2 then stmt
```

```
  end.

{((ifnestedwhile,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <cond1>,
       ({(while..do,1),(begin..end,1),(.,1)},
        {(cond1,1)})),
    (2,<>,
       <cond2>,
       ({(if..then,1)},
        {(cond2,1)})),
    (3,<(stmt,<>)>,
       <>,
       ({(stmt,1)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(2,3),(1,2),(1,t),(3,1),(2,1)},
   s,
   t))}

program ifthenelsenestedwhile;
begin
  while cond do
    if cond2 then stmt1 else stmt2
end.

{((ifthenelsenestedwhile,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <cond>,
       ({(while..do,1),(begin..end,1),(.,1)},
        {(cond,1)})),
    (2,<>,
       <cond2>,
       ({(if..then,1)},
        {(cond2,1)})),
    (3,<(stmt1,<>)>,
       <>,
       ({(stmt1,1)},
        {})),
    (4,<(stmt2,<>)>,
       <>,
       ({(stmt2,1),(else,1)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(2,3),(2,4),(1,2),(1,t),(3,1),(4,1)},
   s,
   t))}

program casenestedwhile;
begin
  while cond do
    case cond of
      1 : stmt1;
      2 : stmt2;
      6 : stmt6
    end
end.

{((casenestedwhile,<>,{}),
  ({(s,<>,<>,0)
```

```
    (1,<>,
       <cond>,
       ({(while..do,1),(begin..end,1),(.,1)},
        {(cond,1)})),
    (2,<>,
       <cond>,
       ({(case..of..end,1)},
        {(cond,1)})),
    (3,<(stmt1,<>)>,
       <>,
       ({(stmt1,1)},
        {(1:,1)})),
    (4,<(stmt2,<>)>,
       <>,
       ({(stmt2,1),(;,1)},
        {(2:,1)})),
    (5,<(stmt6,<>)>,
       <>,
       ({(stmt6,1),(;,1)},
        {(6:,1)}))
    (t,<>,<>,0)},
   {(s,1),(2,3),(2,4),(2,5),(1,2),(1,t),(3,1),(4,1),(5,1)},
   s,
   t))}
```

```
program repeatnestedwhile;
begin
  while cond1 do
    repeat
      stmt;
    until cond2
end.
```

```
{((repeatnestedwhile,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <cond1>,
       ({(while..do,1),(begin..end,1),(.,1)},
        {(cond1,1)})),
    (2,<(stmt,<>)>,
       <cond2>,
       ({(stmt,1),(;,1),(repeat..until,1)},
        {(cond2,1)}))
    (t,<>,<>,0)},
   {(s,1),(2,2),(1,2),(1,t),(2,1)},
   s,
   t))}
```

```
program whilenestedwhile;
begin
  while cond1 do
    while cond2 do
      stmt
end.
```

```
{((whilenestedwhile,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <cond1>,
       ({(while..do,1),(begin..end,1),(.,1)},
        {(cond1,1)})),
```

```
    (2,<>,
       <cond2>,
       ({(while..do,1)},
        {(cond2,1)})),
    (3,<(stmt,<>)>,
       <>,
       ({(stmt,1)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(2,3),(3,2),(1,2),(1,t),(2,1)},
   s,
   t))}

program fornestedwhile;
begin
  while cond do
    for i := init downto final do
      stmt
end.

{((fornestedwhile,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <cond>,
       ({(while..do,1),(begin..end,1),(.,1)},
        {(cond,1)})),
    (2,<(i,<init>)>,
       <>,
       ({(for..downto..do,1)},
        {(init,1),(i,1)})),
    (3,<>,
       <i,final>,
       ({},
        {(final,1)})),
    (4,<(stmt,<>),(i,<i>)>,
       <>,
       ({(stmt,1)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(2,3),(3,4),(4,3),(1,2),(1,t),(3,1)},
   s,
   t))}

program ifnestedfor;
begin
  for i := 1 to 999 do
    if cond then stmt
end.

{((ifnestedfor,<>,{}),
  ({(s,<>,<>,0)
    (1,<(i,<1>)>,
       <>,
       ({(for..to..do,1),(begin..end,1),(.,1)},
        {(1,1),(i,1)})),
    (2,<>,
       <i,999>,
       ({},
        {(999,1)})),
    (3,<>,
       <cond>,
```

```
        ({(if..then,1)},
         {(cond,1)})),
     (4,<(stmt,<>)>,
        <>,
        ({(stmt,1)},
         {})),
     (5,<(i,<i>)>,
        <>,
        ({},
         {}))
     (t,<>,<>,0)},
   {(s,1),(3,4),(4,5),(3,5),(1,2),(2,3),(5,2),(2,t)},
   s,
   t))}

program ifthenelsenestedfor;
begin
  for i := 1 to 100 do
    if cond then stmt1 else stmt2
end.

{((ifthenelsenestedfor,<>,{}),
  {{(s,<>,<>,0)
    (1,<(i,<1>)>,
       <>,
       ({(for..to..do,1),(begin..end,1),(.,1)},
        {(1,1),(i,1)})),
    (2,<>,
       <i,100>,
       ({},
        {(100,1)})),
    (3,<>,
       <cond>,
       ({(if..then,1)},
        {(cond,1)})),
    (4,<(stmt1,<>)>,
       <>,
       ({(stmt1,1)},
        {})),
    (5,<(stmt2,<>)>,
       <>,
       ({(stmt2,1),(else,1)},
        {})),
    (6,<(i,<i>)>,
       <>,
       ({},
        {}))
    (t,<>,<>,0)},
  {(s,1),(3,4),(4,6),(3,5),(5,6),(1,2),(2,3),(6,2),(2,t)},
  s,
  t))}

program casenestedfor;
begin
  for j := i to k do
    case what of
      c1: stmt1;
      c2: stmt2;
      c3: stmt3
    end
end.
```

```
{((casenestedfor,<>,{}),
  ({(s,<>,<>,0)
    (1,<(j,<i>)>,
       <>,
       ({(for..to..do,1),(begin..end,1),(.,1)},
        {(i,1),(j,1)})),
    (2,<>,
       <j,k>,
       ({},
        {(k,1)})),
    (3,<>,
       <what>,
       ({(case..of..end,1)},
        {(what,1)})),
    (4,<(stmt1,<>)>,
       <>,
       ({(stmt1,1)},
        {(c1:,1)})),
    (5,<(stmt2,<>)>,
       <>,
       ({(stmt2,1),(;,1)},
        {(c2:,1)})),
    (6,<(stmt3,<>)>,
       <>,
       ({(stmt3,1),(;,1)},
        {(c3:,1)})),
    (7,<(j,<j>)>,
       <>,
       ({},
        {}))
    (t,<>,<>,0)},
   {(s,1),(3,4),(4,7),(3,5),(5,7),(3,6),(6,7),(1,2),(2,3),(7,2),
    (2,t)},
   s,
   t))}
```

```
program repeatnestedfor;
begin
  for i := 1 to 2 do
    repeat
      stmt;
    until cond
end.
```

```
{((repeatnestedfor,<>,{}),
  ({(s,<>,<>,0)
    (1,<(i,<1>)>,
       <>,
       ({(for..to..do,1),(begin..end,1),(.,1)},
        {(1,1),(i,1)})),
    (2,<>,
       <i,2>,
       ({},
        {(2,1)})),
    (3,<(stmt,<>)>,
       <cond>,
       ({(stmt,1),(;,1),(repeat..until,1)},
        {(cond,1)})),
    (5,<(i,<i>)>,
```

```
      <>,
      ({},
       {}))
   (t,<>,<>,0)},
  {(s,1),(3,5),(3,3),(1,2),(2,3),(5,2),(2,t)},
  s,
  t))}

program whileneatedfor;
begin
  for i := 1 to 100 do
    while cond do stmt
end.

{((whileneatedfor,<>,{}),
  ({(s,<>,<>,0)
    (1,<(i,<i>)>,
       <>,
       ({(for..to..do,1),(begin..end,1),(.,1)},
        {(1,1),(i,1)})),
    (2,<>,
       <i,100>,
       ({},
        {(100,1)})),
    (3,<>,
       <cond>,
       ({(while..do,1)},
        {(cond,1)})),
    (4,<(stmt,<>)>,
       <>,
       ({(stmt,1)},
        {})),
    (5,<(i,<i>)>,
       <>,
       ({},
        {}))
    (t,<>,<>,0)},
  {(s,1),(3,4),(4,3),(3,5),(1,2),(2,3),(5,2),(2,t)},
  s,
  t))}

program fornestedfor;
begin
  for i := init to final do
    for j := init to final do
      stmt := right[1,j]
end.

{((fornestedfor,<>,{}),
  ({(s,<>,<>,0)
    (1,<(i,<init>)>,
       <>,
       ({(for..to..do,1),(begin..end,1),(.,1)},
        {(init,1),(i,1)})),
    (2,<>,
       <i,final>,
       ({},
        {(final,1)})),
    (3,<(j,<init>)>,
       <>,
       ({(for..to..do,1)},
```

```
        {(init,1),(j,1)})),
    (4,<>,
       <j,final>,
       ({},
        {(final,1)})),
    (5,<(stmt,<right,1,j>),(j,<j>)>,
       <>,
       ({(,,1),([],1),(:=,1)},
        {(i,1),(j,1),(right,1),(stmt,1)})),
    (6,<(i,<i>)>,
       <>,
       ({},
        {}))
    (t,<>,<>,0)},
   {(s,1),(3,4),(4,5),(5,4),(4,6),(1,2),(2,3),(6,2),(2,t)},
   s,
   t))}
```

B.4 Goto Statement

```
program gotoifthenelse;
label 1,2,3;
begin
  if cond then goto 1 else goto 2;
  1: stmt1;
     goto 3;
  2: stmt2;
  3:
end.
```

```
{((gotoifthenelse,<>,{}),
  ({(s,<>,<>,0)
    (1,<(stmt1,<>)>,
       <cond>,
       ({(goto,3),(if..then,1),(;,3),(stmt1,1),(begin..end,1),(.,1)},
        {(cond,1),(1,1),(2,1),(1:,1),(3,1)})),
    (5,<(stmt2,<>)>,
       <>,
       ({(stmt2,1),(;,1)},
        {(2:,1),(3:,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,5),(1,t),(5,t)},
   s,
   t))}
```

```
program gotoifthen;
label 1;
begin
  if cond then goto 1;
  stmt1;
  1: stmt2;
end.
```

```
{((gotoifthen,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <cond>,
       ({(goto,1),(if..then,1),(begin..end,1),(.,1)},
        {(cond,1),(1,1)})),
```

```
    (3,<(stmt1,<>)>,
       <>,
       ({(;,2)},
        {})),
    (6,<(stmt2,<>)>,
       <>,
       ({(stmt2,1),(;,1)},
        {(1:,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,3),(1,6),(3,6),(6,t)},
   s,
   t))}
```

```
program gotofor;
label 1,2,3;
begin
  for i := 1 to 10 do
    if cond then goto 1;
  stmt1;
  1: stmt2;
end.
```

```
{((gotofor,<>,{}),
  ({(s,<>,<>,0)
    (1,<(i,<i>)>,
       <>,
       ({(for..to..do,1),(begin..end,1),(.,1)},
        {(1,1),(i,1)})),
    (2,<>,
       <i,10>,
       ({},
        {(i0,1)})),
    (3,<>,
       <cond>,
       ({(goto,1),(if..then,1)},
        {(cond,1),(1,1)})),
    (4,<(i,<i>)>,
       <>,
       ({},
        {})),
    (6,<(stmt1,<>)>,
       <>,
       ({(;,2)},
        {})),
    (9,<(stmt2,<>)>,
       <>,
       ({(stmt2,1),(;,1)},
        {(1:,1)}))
    (t,<>,<>,0)},
   {(s,1),(3,4),(1,2),(2,3),(4,2),(2,6),(3,9),(6,9),(9,t)},
   s,
   t))}
```

```
program gotofor2;
label 1,2,3;
begin
  for i := 1 to 10 do
    if cond then
      begin
stmt;
goto 1;
```

```
      end;
  stmt1;
  1: stmt2;
end.

{((gotofor2,<>,{}),
  ({(s,<>,<>,0)
    (1,<(i,<i>)>,
       <>,
       ({(for..to..do,1),(begin..end,1),(.,1)},
        {(1,1),(i,1)})),
    (2,<>,
       <i,10>,
       ({},
        {(10,1)})),
    (3,<>,
       <cond>,
       ({(if..then,1)},
        {(cond,1)})),
    (4,<(stmt,<>)>,
       <>,
       ({(stmt,1),(goto,1),(;,2),(begin..end,1)},
        {(1,1)})),
    (6,<(i,<i>)>,
       <>,
       ({},
        {})),
    (8,<(stmt1,<>)>,
       <>,
       ({(;,2)},
        {})),
    (11,<(stmt2,<>)>,
       <>,
       ({(stmt2,1),(;,1)},
        {(1:,1)}))
    (t,<>,<>,0)},
   {(s,1),(3,4),(3,6),(1,2),(2,3),(6,2),(2,8),(4,11),(8,11),(11,t)},
   s,
   t))}

program gotoforif;
label 1,2,3;
begin
  for i := 1 to 10 do
    if cond then
      begin
stmt;
goto 1;
      end;
  1: if cond2 then stmt2;
end.

{((gotoforif,<>,{}),
  ({(s,<>,<>,0)
    (1,<(i,<i>)>,
       <>,
       ({(for..to..do,1),(begin..end,1),(.,1)},
        {(1,1),(i,1)})),
    (2,<>,
       <i,10>,
       ({},
```

```
        {(10,1)})),
    (3,<>,
       <cond>,
       ({(if..then,1)},
        {(cond,1)})),
    (4,<(stmt,<>)>,
       <>,
       ({(stmt,1),(goto,1),(;,2),(begin..end,1)},
        {(1,1)})),
    (6,<(i,<i>)>,
       <>,
       ({},
        {})),
    (8,<>,
       <cond2>,
       ({(;,1)},
        {(1:,1),(cond2,1)})),
    (10,<(stmt2,<>)>,
       <>,
       ({(stmt2,1),(;,1)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(3,4),(3,6),(1,2),(2,3),(6,2),(2,8),(8,10),(10,t),(8,t),
    (4,8)},
   s,
   t))}

program goto1;
label 1;
begin
  for i := 1 to m do
    if x = list[i] then
      goto 1;
  list[i] := x;
  count[i] := 0;
  m := i;
  1:count[i] := count[i] + 1
end.

{((goto1,<>,{}),
  ({(s,<>,<>,0)
    (1,<(i,<i>)>,
       <>,
       ({(for..to..do,1),(begin..end,1),(:=,1)},
        {(1,1),(i,1)})),
    (2,<>,
       <i,m>,
       ({},
        {(m,1)})),
    (3,<>,
       <x,list,i>,
       ({([],1),(=,1),(goto,1),(if..then,1)},
        {(x,1),(i,1),(list,1),(1,1)})),
    (4,<(i,<i>)>,
       <>,
       ({},
        {})),
    (8,<(list,<list,i,x>),(count,<count,i,0>),(m,<i>)>,
       <>,
       ({(;,4)},
```

```
        {(i,3),(m,1),(0,1),(count,1),(x,1),(list,1)})),
    (9,<(count,<count,i,count,i,1>)>,
       <>,
       ({([],2),(+,1),(:=,1)},
        {(1:,1),(i,2),(count,2),(1,1)}))
    (t,<>,<>,0)},
   {(s,1),(3,4),(1,2),(2,3),(4,2),(2,8),(3,9),(8,9),(9,t)},
   s,
   t))}

program goto2;
label 30, 50, 60, 70;
begin
      if x < y then goto 30;
      if y < z then goto 50;
      small := z;
      goto 70;
  30: if x < z then goto 60;
      small := z;
      goto 70;
  50: small := y;
      goto 70;
  60: small := x;
  70:
end.

{((goto2,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <x,y>,
       ({(<,1),(goto,1),(if..then,1),(begin..end,1),(.,1)},
        {(x,1),(y,1),(30,1)})),
    (3,<>,
       <y,z>,
       ({(;,1)},
        {(y,1),(z,1),(50,1)})),
    (5,<(small,<z>)>,
       <>,
       ({(;,3),(goto,1)},
        {(z,1),(small,1),(70,1)})),
    (6,<>,
       <x,z>,
       ({(<,1),(goto,1),(if..then,1)},
        {(30:,1),(x,1),(z,1),(60,1)})),
    (9,<(small,<z>)>,
       <>,
       ({(;,3),(goto,1)},
        {(z,1),(small,1),(70,1)})),
    (10,<(small,<y>)>,
       <>,
       ({(:=,1),(goto,1),(;,2)},
        {(50:,1),(y,1),(small,1),(70,1)})),
    (12,<(small,<x>)>,
       <>,
       ({(:=,1),(;,1)},
        {(60:,1),(x,1),(small,1),(70:,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,3),(3,5),(6,9),(1,6),(3,10),(6,12),(10,t),(9,t),(5,t),
    (12,t)},
```

```
   s,
   t))}

program goto3;
label 19, 10;
begin
      for i := 1 to m do
        begin
          if (bp[i] + 1) > 0 then
            goto 10;
          if (bp[i] + 1) < 0 then
            goto 19;
          ibn1[i] := blnk;
          ibn2[i] := blnk;
          goto 10;
  19:     bp[i] := -1;
          ibn1[i] := blnk;
          ibn2[i] := blnk;
  10:   end
end.

{((goto3,<>,{}),
  ({(s,<>,<>,0)
    (1,<(i,<i>)>,
       <>,
       ({(for..to..do,1),(begin..end,1),(.,1)},
        {(1,1),(i,1)})),
    (2,<>,
       <i,m>,
       ({},
        {(m,1)})),
    (3,<>,
       <bp,i,1,0>,
       ({([],1),(+,1),((),1),(>,1),(goto,1),(if..then,1),(begin..end,1)},
        {(i,1),(bp,1),(1,1),(0,1),(10,1)})),
    (5,<>,
       <bp,i,1,0>,
       ({(;,1)},
        {(i,1),(bp,1),(1,1),(0,1),(19,1)})),
    (8,<(ibn1,<ibn1,i,blnk>),(ibn2,<ibn2,i,blnk>)>,
       <>,
       ({(;,4),(goto,1)},
        {(i,2),(blnk,2),(ibn2,1),(ibn1,1),(10,1)})),
    (12,<(bp,<bp,i,1>),(ibn1,<ibn1,i,blnk>),(ibn2,<ibn2,i,blnk>)>,
       <>,
       ({([],1),(-,1),(:=,1),(;,3)},
        {(i,3),(blnk,2),(ibn2,1),(ibn1,1),(19:,1),(1,1),(bp,1)})),
    (13,<(i,<i>)>,
       <>,
       ({},
        {(10:,1)}))
    (t,<>,<>,0)},
   {(s,1),(3,5),(5,8),(5,12),(8,13),(3,13),(12,13),(1,2),(2,3),(13,2),
    (2,t)},
   s,
   t))}

program goto4;
label 10;
```

```
var newin,large : integer;
begin
       large := 0;
  10:  readln(newin);
       if newin > large then
       large := newin;
       if newin >= 0 then
         goto 10;
       writeln(large);
end.
```

```
{((goto4,<>,{}),
  ({(s,<>,<>,0)
    (1,<(large,<0>)>,
       <>,
       ({(:=,1),(;,1),(begin..end,1),(.,1)},
        {(0,1),(large,1)})),
    (4,<(newin,<input^>),(input^,<input>)>,
       <newin,large>,
       ({(readln,1),((),1),(;,1)},
        {(newin,2),(large,1),(10:,1)})),
    (5,<(large,<newin>)>,
       <>,
       ({(:=,1)},
        {(newin,1),(large,1)})),
    (7,<>,
       <newin,0>,
       ({(;,1)},
        {(newin,1),(0,1),(10,1)})),
    (10,<(output^,<large>),(output,<output,output^>),(output,<output,end-of-line>)>,
       <>,
       ({(;,2)},
        {(large,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,4),(4,5),(5,7),(4,7),(7,4),(7,10),(10,t)},
   s,
   t))}
```

```
program goto5;
label 10;
begin
  found := false;
  for i := 1 to maxsize do
    if jobid = jobs[i] then
      begin
        found := true;
        goto 10
      end;
10: if found then
      errorlog[3] := ' '
    else errorlog[3] := '*';
end.
```

```
{((goto5,<>,{}),
  ({(s,<>,<>,0)
    (1,<(found,<false>),(i,<1>)>,
       <>,
       ({(:=,1),(;,1),(for..to..do,1),(begin..end,1),(.,1)},
        {(false,1),(found,1),(1,1),(i,1)})),
```

```
    (3,<>,
       <i,maxsize>,
       ({},
        {(maxsize,1)})),
    (4,<>,
       <jobid,jobs,i>,
       ({([],1),(=,1),(if..then,1)},
        {(jobid,1),(i,1),(jobs,1)})),
    (5,<(found,<true>)>,
       <>,
       ({(:=,1),(goto,1),(;,1),(begin..end,1)},
        {(true,1),(found,1),(10,1)})),
    (6,<(i,<i>)>,
       <>,
       ({},
        {})),
    (8,<>,
       <found>,
       ({(;,1)},
        {(10:,1),(found,1)})),
    (10,<(errorlog,<errorlog,3,' ')>,
       <>,
       ({([],1),(:=,1)},
        {(3,1),(' ',1),(errorlog,1)})),
    (11,<(errorlog,<errorlog,3,'*')>,
       <>,
       ({([],1),(:=,1),(else,1),(;,1)},
        {(3,1),('*',1),(errorlog,1)}))
    (t,<>,<>,0)},
   {(s,1),(4,5),(4,6),(1,3),(3,4),(6,3),(3,8),(8,10),(10,t),(8,11),
    (11,t),(5,8)},
   s,
   t))}
```

```
program goto6;
label 10;
begin
  for i := 1 to n do
    begin
10:   get(r);
      for j := 1 to i-1 do
        if a[j] = r then
          goto 10;
      a[i] := r
    end
end.
```

```
{((goto6,<>,{}),
  ({(s,<>,<>,0)
    (1,<(i,<1>)>,
       <>,
       ({(for..to..do,1),(begin..end,1),(.,1)},
        {(1,1),(i,1)})),
    (2,<>,
       <i,n>,
       ({},
        {(n,1)})),
    (3,<(r^,<r>),(j,<1>)>,
       <>,
       ({(get,1),((),1),(;,1),(for..to..do,1),(begin..end,1)},
```

```
        {(10:,1),(r,1),(1,1),(j,1)})),
    (6,<>,
       <j,i,1>,
       ({(-,1)},
        {(i,1),(1,1)})),
    (7,<>,
       <a,j,r>,
       ({([],1),(=,1),(goto,1),(if..then,1)},
        {(j,1),(a,1),(r,1),(10,1)})),
    (8,<(j,<j>)>,
       <>,
       ({},
        {})),
    (10,<(a,<a,i,r>),(i,<i>)>,
       <>,
       ({(;,1)},
        {(i,1),(r,1),(a,1)}))
    (t,<>,<>,0)},
   {(s,1),(7,3),(7,8),(3,6),(6,7),(8,6),(6,10),(1,2),(2,3),(10,2),
    (2,t)},
   s,
   t))}
```

```
program goto7;
label 10;
begin
  for i := 1 to m do
    if x = a[i] then
      goto 10;
  i := m + 1;
  m := i;
  a[i] := x;
  b[i] := 0;
10:  b[i] := b[i] + 1
end.
```

```
{((goto7,<>,{}),
  ({(s,<>,<>,0)
    (1,<(i,<i>)>,
       <>,
       ({(for..to..do,1),(begin..end,1),(.,1)},
        {(1,1),(i,1)})),
    (2,<>,
       <i,m>,
       ({},
        {(m,1)})),
    (3,<>,
       <x,a,i>,
       ({([],1),(=,1),(goto,1),(if..then,1)},
        {(x,1),(i,1),(a,1),(10,1)})),
    (4,<(i,<i>)>,
       <>,
       ({},
        {})),
    (9,<(i,<m,1>),(m,<i>),(a,<a,i,x>),
       (b,<b,i,0>)>,
       <>,
       ({(;,5)},
        {(i,4),(0,1),(b,1),(x,1),(a,1),(m,2),(1,1)})),
    (10,<(b,<b,i,b,i,1>)>,
```

```
      <>,
      ({([],2),(+,1),(:=,1)},
       {(10:,1),(i,2),(b,2),(1,1)}))
   (t,<>,<>,0)},
  {(s,1),(3,4),(1,2),(2,3),(4,2),(2,9),(3,10),(9,10),(10,t)},
  s,
  t))}
```

```
program goto8;
label 10;
begin
  i := h(x);
  while a[i] <> x do
    begin
      if a[i] = 0 then
        begin
          a[i] := x;
          b[i] := 0;
          goto 10
        end;
      i := i + 1;
      if i = 0 then i := m
    end;
10:b[i] := b[i] + 1
end.
```

```
{((goto8,<>,{}),
  ({(s,<>,<>,0)
    (1,<(i,<(h,<<x>>)>)>,
       <>,
       ({(h,1),((),1),(:=,1),(;,1),(begin..end,1),(.,1)},
        {(x,1),(i,1)})),
    (2,<>,
       <a,i,x>,
       ({([],1),(<>,1),(while..do,1)},
        {(i,1),(a,1),(x,1)})),
    (3,<>,
       <a,i,0>,
       ({([],1),(=,1),(if..then,1),(begin..end,1)},
        {(i,1),(a,1),(0,1)})),
    (4,<(a,<a,i,x>),(b,<b,i,0>)>,
       <>,
       ({([],2),(:=,2),(;,2),(goto,1),(begin..end,1)},
        {(i,2),(x,1),(a,1),(0,1),(b,1),(10,1)})),
    (8,<(i,<i,1>)>,
       <i,0>,
       ({(;,2)},
        {(i,3),(0,1),(1,1)})),
    (9,<(i,<m>)>,
       <>,
       ({(:=,1)},
        {(m,1),(i,1)})),
    (12,<(b,<b,i,b,i,1>)>,
       <>,
       ({(;,1)},
        {(10:,1),(i,2),(b,2),(1,1)}))
    (t,<>,<>,0)},
  {(s,1),(3,4),(3,8),(8,9),(2,3),(2,12),(9,2),(8,2),(1,2),(4,12),
   (12,t)},
  s,
```

```
   t))}

program goto9;
label 10;
begin
  read(x);
  if x = slash then
    begin
      read(x);
      if x = slash then
        begin
          writeln;
          goto 10
        end
      else
        tabulate
    end;
  write(x);
  if x = period then
    write(blank);
10:
end.

{((goto9,<>,{}),
  ({(s,<>,<>,0)
    (1,<(x,<input^>),(input^,<input>)>,
       <x,slash>,
       ({(read,1),((),1),(;,1),(=,1),(if..then,1),(begin..end,1),(.,1)}
        {(x,2),(slash,1)})),
    (3,<(x,<input^>),(input^,<input>)>,
       <x,slash>,
       ({(read,1),((),1),(;,1),(=,1),(if..then,1),(begin..end,1)},
        {(x,2),(slash,1)})),
    (5,<(output,<output,end-of-line>)>,
       <>,
       ({(writeln,1),(goto,1),(;,1),(begin..end,1)},
        {(10,1)})),
    (6,<(tabulate,<>)>,
       <>,
       ({(tabulate,1),(else,1)},
        {})),
    (10,<(output^,<x>),(output,<output,output^>)>,
       <x,period>,
       ({(;,2)},
        {(x,2),(period,1)})),
    (11,<(output^,<blank>),(output,<output,output^>)>,
       <>,
       ({(write,1),((),1),(;,1)},
        {(blank,1),(10:,1)}))
    (t,<>,<>,0)},
   {(s,1),(3,5),(3,6),(1,3),(1,10),(6,10),(10,11),(11,t),(10,t),(5,t)},
   s,
   t))}

program goto10;
label 20;
begin
  read(igirl);
  for i := 1 to 8 do
    read(fem[i]);
```

```
20:read(iboy);
  for i := 1 to 8 do
    read(male[i]);
  for i := 1 to 8 do
    if (fem[i] and not male[i]) or
       (male[i] and not fem[i]) then
         goto 20;
  writeln(iboy:5);
  goto 20;
end.
```

```
{((goto10,<>,{}),
  ({(s,<>,<>,0)
    (1,<(igirl,<input^>),(input^,<input>),(i,<1>)>,
       <>,
       ({(read,1),(().1),(;,1),(for..to..do,1),(begin..end,1),(.,1)},
        {(igirl,1),(1,1),(i,1)})),
    (3,<>,
       <i,8>,
       ({},
        {(8,1)})),
    (4,<(fem,<input^,i>),(input^,<input>),(i,<i>)>,
       <>,
       ({([],1),(read,1),((),1)},
        {(i,1),(fem,1)})),
    (8,<(iboy,<input^>),(input^,<input>),(i,<1>)>,
       <>,
       ({(;,2)},
        {(1,1),(i,1),(20:,1),(iboy,1)})),
    (9,<>,
       <i,8>,
       ({},
        {(8,1)})),
    (10,<(male,<input^,i>),(input^,<input>),(i,<i>)>,
       <>,
       ({([],1),(read,1),((),1)},
        {(i,1),(male,1)})),
    (12,<(i,<1>)>,
       <>,
       ({(;,1)},
        {(1,1),(i,1)})),
    (13,<>,
       <i,8>,
       ({},
        {(8,1)})),
    (14,<>,
       <fem,i,male,i,male,i,fem,i>,
       ({([],4),(not,2),(and,2),((),2),(or,1),(goto,1),(if..then,1)},
        {(i,4),(fem,2),(male,2),(20,1)})),
    (15,<(i,<i>)>,
       <>,
       ({},
        {})),
    (17,<(output^,<iboy,5>),(output,<output,output^>),(output,<output,end-of-line>)>,
       <>,
       ({(;,3),(goto,1)},
        {(iboy,1),(5,1),(20,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,3),(3,4),(4,3),(3,8),(8,9),(9,10),(10,9),(9,12),(14,8),
    (14,15),(12,13),(13,14),(15,13),(13,17),(17,8)},
```

```
   s,
   t))}

program goto11;
label 10;
begin
  sparse := 0;
  for k := 1 to n do
    if (nrow[k] = i) and
       (ncol[k] = j) then
       begin
         sparse := value[k];
         goto 10
       end;
10:
end.

{((goto11,<>,{}),
  ({(s,<>,<>,0)
    (1,<(sparse,<0>),(k,<1>)>,
       <>,
       ({(:=,1),(;,1),(for..to..do,1),(begin..end,1),(.,1)},
        {(0,1),(sparse,1),(1,1),(k,1)})),
    (3,<>,
       <k,n>,
       ({},
        {(n,1)})),
    (4,<>,
       <nrow,k,i,ncol,k,j>,
       ({([],2),(=,2),((),2),(and,1),(if..then,1)},
        {(k,2),(nrow,1),(i,1),(ncol,1),(j,1)})),
    (5,<(sparse,<value,k>)>,
       <>,
       ({([],1),(:=,1),(goto,1),(;,1),(begin..end,1)},
        {(k,1),(value,1),(sparse,1),(10,1)})),
    (6,<(k,<k>)>,
       <>,
       ({(;,1)},
        {(10:,1)}))
    (t,<>,<>,0)},
   {(s,1),(4,5),(4,6),(1,3),(3,4),(6,3),(3,t),(5,t)},
   s,
   t))}

program goto12;
label 1;
begin
1:if data[index] < newdata then
    if left[index] = 0 then
      left[index] := newindex
    else
      begin
        index := left[index];
        goto 1
      end
  else
    if right[index] = 0 then
      right[index] := newindex
    else
      begin
```

```
        index := right[index];
        goto 1
      end;
  data[index] := newdata
end.
```

```
{((goto12,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <data,index,newdata>,
       ({([],1),(<,1),(if..then,1),(begin..end,1),(.,1)},
        {(1:,1),(index,1),(data,1),(newdata,1)})),
    (3,<>,
       <left,index,0>,
       ({([],1),(=,1),(if..then,1)},
        {(index,1),(left,1),(0,1)})),
    (4,<(left,<left,index,newindex>)>,
       <>,
       ({([],1),(:=,1)},
        {(index,1),(newindex,1),(left,1)})),
    (5,<(index,<left,index>)>,
       <>,
       ({([],1),(:=,1),(goto,1),(;,1),(begin..end,1),(else,1)},
        {(index,2),(left,1),(1,1)})),
    (7,<>,
       <right,index,0>,
       ({},
        {(index,1),(right,1),(0,1)})),
    (8,<(right,<right,index,newindex>)>,
       <>,
       ({([],1),(:=,1)},
        {(index,1),(newindex,1),(right,1)})),
    (9,<(index,<right,index>)>,
       <>,
       ({([],1),(:=,1),(goto,1),(;,1),(begin..end,1),(else,1)},
        {(index,2),(right,1),(1,1)})),
    (12,<(data,<data,index,newdata>)>,
       <>,
       ({(;,1)},
        {(index,1),(newdata,1),(data,1)}))
    (t,<>,<>,0)},
   {(s,1),(3,4),(3,5),(7,8),(7,9),(1,3),(1,7),(8,12),(4,12),(9,1),
    (5,1),(12,t)},
   s,
   t))}
```

B.5 Expressions

```
program empty;
begin
end.
```

```
{((empty,<>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <>,
       ({(begin..end,1),(.,1)},
        {}))
```

```
    (t,<>,<>,0)},
   {(s,t)},
   s,
   t))}

program idwitharraydesignator;
begin
   a[i] := b[j] + 1;
end.

{((idwitharraydesignator,<>,{}),
  ({(s,<>,<>,0)
    (1,<(a,<a,i,b,j,1>)>,
       <>,
       ({([],2),(+,1),(:=,1),(;,1),(begin..end,1),(.,1)},
        {(i,1),(j,1),(b,1),(1,1),(a,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}

program expression;
begin
  a := (42 - 13) * 2 / (23 +3);
  b(this,32 mod 2);
  real := (integer or real) and not boolean
end.

{((expression,<>,{}),
  ({(s,<>,<>,0)
    (1,<(a,<42,13,2,23,3>),(b,<<this>,<32,2>>),(real,<integer,real,boolean>)>,
       <>,
       ({(-,1),((),4),(*,1),(+,1),(/,1),(:=,2),(;,2),(mod,1),
         (,,1),(b,1),(or,1),(not,1),(and,1),(begin..end,1),(.,1)},
        {(42,1),(13,1),(2,2),(23,1),(3,1),(a,1),(this,1),(32,1),
         (integer,1),(real,2),(boolean,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}

program expression2;
begin
  a := f(a,wt(c*b))
end.

{((expression2,<>,{}),
  ({(s,<>,<>,0)
    (1,<(a,<(f,<<a>,<(wt,<<c,b>>)>>)>)>,
       <>,
       ({(*,1),(wt,1),((),2),(,,1),(f,1),(:=,1),(begin..end,1),(.,1)},
        {(a,2),(c,1),(b,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}

program literals;
begin
  a := 'I am an angels'' boy.'
```

```
end.

{((literals,<>,{}),
  ({(s,<>,<>,0)
    (1,<(a,<'I am an angels'' boy.'>)>,
       <>,
       ({(:=,1),(begin..end,1),(.,1)},
        {('I am an angels'' boy.',1),(a,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}
```

B.6 Predefined Procedures and Functions

```
program rewritetest;
begin
  rewrite(f);
end.

{((rewritetest,<>,{}),
  ({(s,<>,<>,0)
    (1,<(f,<>)>,
       <>,
       ({(rewrite,1),((),1),(;,1),(begin..end,1),(.,1)},
        {(f,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}

program  resettest;
begin
  reset(f)
end.

{((resettest,<>,{}),
  ({(s,<>,<>,0)
    (1,<(f^,<f>)>,
       <>,
       ({(reset,1),((),1),(begin..end,1),(.,1)},
        {(f,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}

program puttest;
begin
  put(grocerystore)
end.

{((puttest,<>,{}),
  ({(s,<>,<>,0)
    (1,<(grocerystore,<grocerystore,grocerystore^>)>,
       <>,
       ({(put,1),((),1),(begin..end,1),(.,1)},
        {(grocerystore,1)}))
    (t,<>,<>,0)},
```

```
   {(s,1),(1,t)},
   s,
   t))}

program gettest;
begin
  get(f)
end.

{((gettest,<>,{}),
  ({(s,<>,<>,0)
    (1,<(f^,<f>)>,
       <>,
       ({(get,1),((),1),(begin..end,1),(.,1)},
        {(f,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}

program readlnwritelntest(input,output);
var x, y : integer;
begin
  readln(x);
  writeln(y);
end.

{((readlnwritelntest,<input,output>,{}),
  ({(s,<>,<>,0)
    (1,<(x,<input^>),(input^,<input>),(output^,<y>),
        (output,<output,output^>),(output,<output,end-of-line>)>,
       <>,
       ({(readln,1),((),2),(;,2),(writeln,1),(begin..end,1),(.,1)},
        {(x,1),(y,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}

program readwritetest(input,output);
var x, y : ^integer;
begin
  read(x[i,j]^);
  write(y[i,j]^);
end.

{((readwritetest,<input,output>,{}),
  ({(s,<>,<>,0)
    (1,<(integer,<input^,integer,i,j>),(input^,<input>),(output^,<y,integer,i,j>),
       (output,<output,output^>)>,
       <>,
       ({(.,2),([],2),(^,2),(read,1),((),2),(;,2),(write,1),(begin..end,1),
         (.,1)},
        {(i,2),(j,2),(x,1),(y,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}
```

```
program readlnwritelntest2(input,output);
var a, b, c, d, e, f: real;
begin
  readln(a,b,c);
  writeln(d,e,f)
end.

{((readlnwritelntest2,<input,output>,{}),
  ({(s,<>,<>,0)
    (1,<(a,<input^>),(input^,<input>),(b,<input^>),
        (input^,<input>),(c,<input^>),(input^,<input>),
        (output,<output,d,e,f>),(output^,<f>),(output,<output,end-of-line>)>,
       <>,
       ({(,,4),(readln,1),((),2),(;,1),(writeln,1),(begin..end,1),(.,1)},
        {(a,1),(b,1),(c,1),(d,1),(e,1),(f,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}

program readwritetest2(input,output);
var a, b, c, d, e, f: real;
begin
  read(a,b,c);
  write(d,e,f)
end.

{((readwritetest2,<input,output>,{}),
  ({(s,<>,<>,0)
    (1,<(a,<input^>),(input^,<input>),(b,<input^>),
        (input^,<input>),(c,<input^>),(input^,<input>),
        (output,<output,d,e,f>),(output^,<f>)>,
       <>,
       ({(,,4),(read,1),((),2),(;,1),(write,1),(begin..end,1),(.,1)},
        {(a,1),(b,1),(c,1),(d,1),(e,1),(f,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}

program readlnwritelntest3(bang);
var bang : text;
begin
  readln(bang,x);
  writeln(bang,x)
end.

{((readlnwritelntest3,<bang>,{}),
  ({(s,<>,<>,0)
    (1,<(x,<bang^>),(bang^,<bang>),(bang^,<x>),
        (bang,<bang,bang^>),(bang,<bang,end-of-line>)>,
       <>,
       ({(,,2),(readln,1),((),2),(;,1),(writeln,1),(begin..end,1),(.,1)},
        {(bang,2),(x,2)}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}
```

```
program readlnwritelntest4(boom);
type boomtype = file of char;
var boom : boomtype;
begin
  readln(boom,s);
  writeln(boom,s)
end.

{((readlnwritelntest4,<boom>,{}),
  ({(s,<>,<>,0)
    (1,<(s,<boom^>),(boom^,<boom>),(boom^,<s>),
        (boom,<boom,boom^>),(boom,<boom,end-of-line>)>,
       <>,
       ({(,,2),(readln,1),((),2),(;,1),(writeln,1),(begin..end,1),(.,1)},
        {(boom,2),(s,2)}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}

program readlntest(music);
type musictype = array[1..100] of file of char;
var music : musictype;
begin
  readln(music,chopin,patmetheny,beatles)
end.

{((readlntest,<music>,{}),
  ({(s,<>,<>,0)
    (1,<(chopin,<music^>),(music^,<music>),(patmetheny,<music^>),
        (music^,<music>),(beatles,<music^>),(music^,<music>)>,
       <>,
       ({(,,3),(readln,1),((),1),(begin..end,1),(.,1)},
        {(music,1),(chopin,1),(patmetheny,1),(beatles,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}

program writelntest(music);
type musictype = array[1..100] of file of char;
var music : musictype;
begin
  writeln(music,chopin,patmetheny,beatles)
end.

{((writelntest,<music>,{}),
  ({(s,<>,<>,0)
    (1,<(music,<music,chopin,patmetheny,beatles>),(music^,<beatles>),(music,<music,end-
of-line>)>,
       <>,
       ({(,,3),(writeln,1),((),1),(begin..end,1),(.,1)},
        {(music,1),(chopin,1),(patmetheny,1),(beatles,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}

program readlntest2(music);
```

```
type musictype = file of char;
var music : musictype;
begin
  readln(music);
end.
```

```
{((readlntest2,<music>,{}),
  ({(s,<>,<>,0)
    (1,<(music^,<music>)>,
       <>,
       ({(readln,1),((),1),(;,1),(begin..end,1),(.,1)},
        {(music,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}
```

```
program readlntest(music);
begin
  readln
end.
```

```
{((readlntest,<music>,{}),
  ({(s,<>,<>,0)
    (1,<(input^,<input>)>,
       <>,
       ({(readln,1),(begin..end,1),(.,1)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}
```

```
program writelntest(music);
type musictype = array[1..100] of file of char;
var music : musictype;
begin
  writeln(music)
end.
```

```
{((writelntest,<music>,{}),
  ({(s,<>,<>,0)
    (1,<(music,<music,end-of-line>)>,
       <>,
       ({(writeln,1),((),1),(begin..end,1),(.,1)},
        {(music,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}
```

```
program writelntest(music);
begin
  writeln
end.
```

```
{((writelntest,<music>,{}),
  ({(s,<>,<>,0)
    (1,<(output,<output,end-of-line>)>,
       <>,
       ({(writeln,1),(begin..end,1),(.,1)},
```

```
        {}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}

program pagetest(music);
type musictype = array[1..100] of file of char;
var music : musictype;
begin
  page(music)
end.

{((pagetest,<music>,{}),
  ({(s,<>,<>,0)
    (1,<(music,<music,end-of-page>)>,
       <>,
       (({(page,1),((),1),(begin..end,1),(.,1)},
        {(music,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}

program pagetest(music);
begin
  page
end.

{((pagetest,<music>,{}),
  ({(s,<>,<>,0)
    (1,<(output,<output,end-of-page>)>,
       <>,
       (({(page,1),(begin..end,1),(.,1)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}

program newtest;
type ptrtype = ^guitar;
var  p : ptrtype;
begin
  new(p)
end.

{((newtest,<>,{}),
  ({(s,<>,<>,0)
    (1,<(p,<>),(guitar,<p,guitar>)>,
       <>,
       (({(new,1),((),1),(begin..end,1),(.,1)},
        {(p,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}

program newtest;
var p : ^integer;
```

```
begin
  new(p)
end.

{((newtest,<>,{}),
  ({(s,<>,<>,0)
    (1,<(p,<>),(integer,<p,integer>)>,
       <>,
       ({(new,1),((),1),(begin..end,1),(.,1)},
        {(p,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}

program disposetest;
var p : ^char;
begin
  dispose(p)
end.

{((disposetest,<>,{}),
  ({(s,<>,<>,0)
    (1,<(char,<p,char>),(p,<nil>)>,
       <>,
       ({(dispose,1),((),1),(begin..end,1),(.,1)},
        {(p,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}

program newtest;
type ptrtype = ^guitar;
var  p : ptrtype;
begin
  new(p,a,b,c)
end.

{((newtest,<>,{}),
  ({(s,<>,<>,0)
    (1,<(p,<>),(guitar,<p,guitar,a,b,c>)>,
       <>,
       ({(,,3),(new,1),((),1),(begin..end,1),(.,1)},
        {(p,1),(a,1),(b,1),(c,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}

program disposetest;
type ptrtype = ^guitar;
var  p : ptrtype;
begin
  dispose(p,a,b,c)
end.

{((disposetest,<>,{}),
  ({(s,<>,<>,0)
```

```
    (1,<(guitar,<p,guitar,a,b,c>),(p,<nil>)>,
       <>,
       ({(,,3),(dispose,1),((),1),(begin..end,1),(.,1)},
        {(p,1),(a,1),(b,1),(c,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}
```

```
program packtest;
begin
  pack(a,b,c);
  unpack(a,b,c)
end.
```

```
{((packtest,<>,{}),
  ({(s,<>,<>,0)
    (1,<(c,<a,b>),(b,<a,c>)>,
       <>,
       ({(,,4),(pack,1),((),2),(;,1),(unpack,1),(begin..end,1),(.,1)},
        {(a,2),(b,2),(c,2)}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}
```

```
program newtest;
type ptrtype = ^guitar;
     guitar = record
                electric : char;
                acoustic : char;
                next : ptrtype;
              end;
var  p : ptrtype;
begin
  new(p^.next)
end.
```

```
{((newtest,<>,{}),
  ({(s,<>,<>,0)
    (1,<(guitar,<guitar>)>,
       <>,
       ({(^,1),(.,2),(new,1),((),1),(begin..end,1)},
        {(next,1),(p,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}
```

```
program newtest;
type ptrtype = ^guitar;
     guitar = record
                electric : ^integer;
                acoustic : ^integer;
                next : ptrtype;
              end;
var  p : ptrtype;
begin
  new(p^.electric)
```

```
end.

{((newtest,<>,{}),
  ({(s,<>,<>,0)
    (1,<(guitar,<guitar>),(integer,<integer,guitar>)>,
       <>,
       ({(^,1),(.,2),(new,1),((),1),(begin..end,1)},
        {(electric,1),(p,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}

program newtest;
type ptrtype = ^guitar;
     guitar = record
                electric : char;
                acoustic : char;
                next : ptrtype;
              end;
var  p : ptrtype;
begin
  new(p^.next,a,b,c)
end.

{((newtest,<>,{}),
  ({(s,<>,<>,0)
    (1,<(guitar,<guitar,a,b,c>)>,
       <>,
       ({(^,1),(.,2),(,,3),(new,1),((),1),(begin..end,1)},
        {(next,1),(p,1),(a,1),(b,1),(c,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}

program newtest;
type ptrtype = ^guitar;
     guitar = record
                electric : ^integer;
                acoustic : ^integer;
                next : ptrtype;
              end;
var  p : ptrtype;
begin
  new(p^.electric,a,b,c)
end.

{((newtest,<>,{}),
  ({(s,<>,<>,0)
    (1,<(guitar,<guitar>),(integer,<integer,guitar,a,b,c>)>,
       <>,
       ({(^,1),(.,2),(,,3),(new,1),((),1),(begin..end,1)},
        {(electric,1),(p,1),(a,1),(b,1),(c,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}

program disposetest;
```

```
type ptrtype = ^guitar;
     guitar = record
                electric : char;
                acoustic : char;
                next : ptrtype;
              end;
var  p : ptrtype;
begin
  dispose(p^.next)
end.
```

```
{((disposetest,<>,{}),
  ({(s,<>,<>,0)
    (1,<(guitar,<guitar>)>,
       <>,
       ({(^,1),(.,2),(dispose,1),((),1),(begin..end,1)},
        {(next,1),(p,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}
```

```
program disposetest;
type ptrtype = ^guitar;
     guitar = record
                electric : ^integer;
                acoustic : ^integer;
                next : ptrtype;
              end;
var  p : ptrtype;
begin
  dispose(p^.electric)
end.
```

```
{((disposetest,<>,{}),
  ({(s,<>,<>,0)
    (1,<(integer,<integer,guitar>),(guitar,<guitar>)>,
       <>,
       ({(^,1),(.,2),(dispose,1),((),1),(begin..end,1)},
        {(electric,1),(p,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}
```

```
program disposetest;
type ptrtype = ^guitar;
     guitar = record
                electric : char;
                acoustic : char;
                next : ptrtype;
              end;
var  p : ptrtype;
begin
  dispose(p^.next,a,b,c)
end.
```

```
{((disposetest,<>,{}),
  ({(s,<>,<>,0)
```

```
    (1,<(guitar,<guitar,a,b,c>)>,
       <>,
       ({(^,1),(.,2),(,,3),(dispose,1),((),1),(begin..end,1)},
        {(next,1),(p,1),(a,1),(b,1),(c,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}

program disposetest;
type ptrtype = ^guitar;
     guitar = record
                electric : ^integer;
                acoustic : ^integer;
                next : ptrtype;
              end;
var  p : ptrtype;
begin
  dispose(p^.electric,a,b,c)
end.

{((disposetest,<>,{}),
  ({(s,<>,<>,0)
    (1,<(integer,<integer,guitar,a,b,c>),(guitar,<guitar>)>,
       <>,
       ({(^,1),(.,2),(,,3),(dispose,1),((),1),(begin..end,1)},
        {(electric,1),(p,1),(a,1),(b,1),(c,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}

program predfunctest1(input,output);
begin
  while eoln do stmt;
end.

{((predfunctest1,<input,output>,{}),
  ({(s,<>,<>,0)
    (1,<>,
       <input>,
       ({(eoln,1),(while..do,1),(begin..end,1),(.,1)},
        {})),
    (2,<(stmt,<>)>,
       <>,
       ({(stmt,1),(;,1)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(1,2),(2,1),(1,t)},
   s,
   t))}

program predfunctest2;
begin
  a := abs(x);
  b := sqr(u-x);
  c := odd(d)
end.

{((predfunctest2,<>,{}),
```

```
  ({(s,<>,<>,0)
    (1,<(a,<x>),(b,<u,x>),(c,<d>)>,
       <>,
       ({(abs,1),((),3),(:=,3),(;,2),(-,1),(sqr,1),(odd,1),(begin..end,1),
         (.,1)},
        {(x,2),(a,1),(u,1),(b,1),(d,1),(c,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}

program predfunctest3;
begin
  a := trunc(round(s mod r))
end.

{((predfunctest3,<>,{}),
  ({(s,<>,<>,0)
    (1,<(a,<s,r>)>,
       <>,
       ({(mod,1),(round,1),((),2),(trunc,1),(:=,1),(begin..end,1),(.,1)},
        {(s,1),(r,1),(a,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}

program predfunctest4;
begin
  y := abs(round(x)+trun(k))+sqr(s*w);
  y := abs(x-y) * c mod inte(w+e/r);
  k := sqrt(abs(x))*a;
  z := inter(a+b)
end.

{((predfunctest4,<>,{}),
  ({(s,<>,<>,0)
    (1,<(y,<x,k,s,w>),(y,<x,y,c,(inte,<<w,e,r>>)>),(k,<x,a>),
       (z,<(inter,<<a,b>>)>)>,
       <>,
       ({(round,1),((),9),(trun,1),(+,4),(abs,3),(*,3),(sqr,1),(:=,4),
         (;,3),(-,1),(/,1),(inte,1),(mod,1),(sqrt,1),(inter,1),(begin..end,1),
         (.,1)},
        {(x,3),(k,2),(s,1),(w,2),(y,3),(c,1),(e,1),(r,1),
         (a,2),(b,1),(z,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}
```

B.7 Structured Data Types

```
program structuredtype;
type cptr = ^c;
     c = record
           v : integer;
           next : cptr
         end;
```

```
var x : ^c;
begin
  new(x);
  x^.v := 7;
  x := x^.next
end.
```

```
{((structuredtype,<>,{}),
  ({(s,<>,<>,0)
    (1,<(x,<>),(c,<x,c>),(c,<x,c,7>),
       (x,<x,c>)>,
      <>,
      ({(new,1),((),1),(;,2),(^,2),(.,3),(:=,2),(begin..end,1)},
       {(x,4),(v,1),(7,1),(next,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}
```

```
program structuredtype;
type apartptr = ^apartment;
     apartment = record
                   floor : integer;
                   letter : lettertype;
                   wing : (north,south,east,west)
                 end;
     lettertype = record
                    paper : char;
                    envelope : integer
                  end;

var tolet, forlease : ^apartment;
begin
  tolet^.floor := 2;
  tolet^.wing := east;
  tolet^.letter := a
end.
```

```
{((structuredtype,<>,{}),
  ({(s,<>,<>,0)
    (1,<(apartment,<tolet,apartment,2>),(apartment,<tolet,apartment,east>),
       (apartment,<tolet,apartment,a>)>,
      <>,
      ({(^,3),(.,4),(:=,3),(;,2),(begin..end,1)},
       {(floor,1),(2,1),(tolet,3),(wing,1),(east,1),(letter,1),(a,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}
```

```
program structuredtype;
type
  apartment = record
                floor : integer;
                letter : char;
                wing : (north,south,east,west)
              end;
var tolet, forlease : apartment;
begin
  tolet.floor := 2;
```

```
  tolet.wing := east;
  forlease.wing := tolet.wing;
  writeln(tolet.wing);
end.

program withtest;
type apartment = record
                   floor : integer;
                   letter : char;
                   wing : (north,south,east,west)
                 end;
var tolet, forlease : apartment;
begin
  with tolet do
    begin
      read(floor);
      arrange(wing);
      forlease := tolet;
      floor := forlease.floor;
      forlease.wing := tolet.wing;
      letter := 'c';
      wing := east
    end
end.

{((withtest,<>,{}),
  ({(s,<>,<>,0)
    (1,<(tolet,<tolet,input^>),(input^,<input>),(arrange,<<tolet>>),
        (forlease,<tolet>),(tolet,<tolet,forlease>),(forlease,<forlease,tolet>),
        (tolet,<tolet,'c'>),(tolet,<tolet,east>)>,
       <>,
       ({(read,1),((),2),(;,6),(arrange,1),(:=,5),(.,4),(begin..end,2),(with..do,1)},
        {(floor,3),(wing,4),(tolet,2),(forlease,3),('c',1),(letter,1),(east,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}

program structuredtype;
type elementptr = ^element;
     element = record
                 data : integer;
                 left,right : elementptr
               end;
var current, saved : elementptr;
begin
  new(saved);
  saved^.right := nil;
  current := saved^.left;
end.

{((structuredtype,<>,{}),
  ({(s,<>,<>,0)
    (1,<(saved,<>),(element,<saved,element>),(element,<saved,element,nil>),
        (current,<saved,element>)>,
       <>,
       ({(new,1),((),1),(;,3),(^,2),(.,3),(:=,2),(begin..end,1)},
        {(saved,3),(right,1),(nil,1),(left,1),(current,1)}))
    (t,<>,<>,0)},
```

```
   {(s,1),(1,t)},
   s,
   t))}

program structuredtype;
type elementptr = ^element;
     element = record
                 data : integer;
                 left,right : elementptr
               end;
var current, saved: elementptr;
begin
  new(current^.left);
  read(current^.data);
end.

{((structuredtype,<>,{}),
  ({(s,<>,<>,0)
    (1,<(element,<element>),(element,<input^,element>),(input^,<input>)>,
       <>,
       ({(^,2),(.,3),(new,1),((),2),(;,2),(read,1),(begin..end,1)},
        {(left,1),(current,2),(data,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}

program structuredtype;
type elementptr = ^element;
     element = record
                 data : integer;
                 tag : pointer;
                 left,right : elementptr
               end;
     pointer = ^integer;
var current, saved: elementptr;
begin
  new(current^.tag);
  current^.right^.left := current;
  current := current^.right;
end.

{((structuredtype,<>,{}),
  ({(s,<>,<>,0)
    (1,<(element,<element>),(integer,<integer,element>),(element,<current,element,current>),
       (current,<current,element>)>,
       <>,
       ({(^,4),(.,5),(new,1),((),1),(;,3),(:=,2),(begin..end,1)},
        {(tag,1),(current,5),(right,2),(left,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}
program structuredtype;
type elementptr = ^element;
     element = record
                 data : integer;
                 left,right : elementptr
               end;
var current, saved: elementptr;
```

```
begin
  current^.next^.data := saved^.data;
  current := current^.right;
  current^.right := current;
end.

{((structuredtype,<>,{}),
  ({(s,<>,<>,0)
    (1,<(element,<current,element,saved,element>),(current,<current,element>),
        (element,<current,element,current>)>,
       <>
       ({(^,5),(.,6),(:=,3),(;,3),(begin..end,1)},
        {(next,1),(data,2),(saved,1),(current,5),(right,2)}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}

program structuredtype;
type ptr = ^elem;
     elem = integer;
     goody = file of char;
var  good : goody;
     p : ptr;
begin
  writeln(good^);
  good^ := 'z';
  p^ := 2;
end.

{((structuredtype,<>,{}),
  ({(s,<>,<>,0)
    (1,<(output^,<good^>),(output,<output,output^>),(output,<output,end-of-line>),
        (good^,<'z'>),(elem,<p,elem,2>)>,
       <>
       ({(^,3),(writeln,1),((),1),(;,3),(:=,2),(begin..end,1),(.,1)},
        {(good,2),('z',1),(2,1),(p,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t))}

program withtest;
type elementptr = ^element;
     element = record
                 data : integer;
                 left,right : elementptr
               end;
var current, saved: elementptr;
begin
  with current^ do
    begin
      left^.data := saved^.data;
      current := right;
      right := current;
    end
end.

{((withtest,<>,{}),
```

```
 ({(s,<>,<>,0)
   (1,<(element,<current,element,saved,element>),(current,<right>),
       (element,<current,element,current>)>,
      <>,
      ({(^,2),(.,3),(:=,3),(;,3),(begin..end,2),(with..do,1)},
       {(data,2),(saved,1),(left,1),(right,2),(current,2)}))
   (t,<>,<>,0)},
  {(s,1),(1,t)},
  s,
  t))}
```

B.8 Seuencing and Nesting of Procedure and Function Declaration

```
program proceduretest;
  procedure a;
    begin stmt1 end;
  procedure b;
    procedure c;
      begin stmt2 end;
    procedure d;
      function e:integer;
        begin e := something end;
      begin stmt3 end;
    begin stmt4 end;
  begin
    stmt5
  end.
```

```
{((a,<>,{}),
  ({(s,<>,<>,0)
    (1,<(stmt1,<>)>,
       <>,
       ({(stmt1,1),(begin..end,1),(;,1)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t)),
 ((c,<>,{}),
  ({(s,<>,<>,0)
    (2,<(stmt2,<>)>,
       <>,
       ({(stmt2,1),(begin..end,1),(;,1)},
        {}))
    (t,<>,<>,0)},
   {(s,2),(2,t)},
   s,
   t)),
 ((e,<>,{}),
  ({(s,<>,<>,0)
    (3,<(e,<something>)>,
       <>,
       ({(:=,1),(begin..end,1),(;,1)},
        {(something,1),(e,1)}))
    (t,<>,<>,0)},
   {(s,3),(3,t)},
```

```
    s,
    t)),
  ((d,<>,{}),
   ({(s,<>,<>,0)
     (4,<(stmt3,<>)>,
        <>,
        ({(stmt3,1),(begin..end,1),(;,1)},
         {}))
     (t,<>,<>,0)},
    {(s,4),(4,t)},
    s,
    t)),
  ((b,<>,{}),
   ({(s,<>,<>,0)
     (5,<(stmt4,<>)>,
        <>,
        ({(stmt4,1),(begin..end,1),(;,1)},
         {}))
     (t,<>,<>,0)},
    {(s,5),(5,t)},
    s,
    t)),
  ((proceduretest,<>,{}),
   ({(s,<>,<>,0)
     (6,<(stmt5,<>)>,
        <>,
        ({(stmt5,1),(begin..end,1),(.,1)},
         {}))
     (t,<>,<>,0)},
    {(s,6),(6,t)},
    s,
    t))}
```

```
program proceduretest;
  procedure a;
    begin stmt1 end;
  procedure b;
    begin stmt2 end;
  procedure c;
    begin stmt3 end;
  begin
    stmt4
  end.
```

```
{((a,<>,{}),
  ({(s,<>,<>,0)
    (1,<(stmt1,<>)>,
       <>,
       ({(stmt1,1),(begin..end,1),(;,1)},
        {}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t)),
  ((b,<>,{}),
   ({(s,<>,<>,0)
     (2,<(stmt2,<>)>,
        <>,
        ({(stmt2,1),(begin..end,1),(;,1)},
         {}))
```

```
    (t,<>,<>,0)},
   {(s,2),(2,t)},
   s,
   t)),
 ((c,<>,{}),
  ({(s,<>,<>,0)
    (3,<(stmt3,<>)>,
       <>,
       ({(stmt3,1),(begin..end,1),(;,1)},
        {}))
    (t,<>,<>,0)},
   {(s,3),(3,t)},
   s,
   t)),
 ((proceduretest,<>,{}),
  ({(s,<>,<>,0)
    (4,<(stmt4,<>)>,
       <>,
       ({(stmt4,1),(begin..end,1),(.,1)},
        {}))
    (t,<>,<>,0)},
   {(s,4),(4,t)},
   s,
   t))}
```

B.9 Detecting Glabal Variable

```
program globaltest;
const max = 10;
var x, y, z : integer;
procedure first(a:integer; var b:integer);
begin
  b := a + x - y;
end;
function second:integer;
var z : integer;
begin
  z := 3;
  x := y + z;
  second := x;
end;
begin
  x := 1; y := 2; z := 3;
  first(x,z);
  y := second;
end.
```

```
{((first,<a,b>,{x,y}),
  ({(s,<(a',<a>)>,<>,0)
    (1,<(b,<a',x,y>)>,
       <>,
       ({(+,1),(-,1),(:=,1),(;,2),(begin..end,1)},
        {(a,1),(x,1),(y,1),(b,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,t)},
   s,
   t)),
 ((second,<>,{y,x}),
```

```
  ({(s,<>,<>,0)
    (3,<(z,<3>),(x,<y,z>),(second,<x>)>,
       <>,
       ({(:=,3),(;,4),(+,1),(begin..end,1)},
        {(3,1),(z,2),(y,1),(x,2),(second,1)}))
    (t,<>,<>,0)},
   {(s,3),(3,t)},
   s,
   t)),
 ((globaltest,<>,{}),
  ({(s,<>,<>,0)
    (7,<(x,<1>),(y,<2>),(z,<3>),
       (first,<<x>,<z>>),(y,<second>)>,
       <>,
       ({(:=,4),(;,5),(,,1),(first,1),((),1),(begin..end,1),(.,1)},
        {(1,1),(x,2),(2,1),(y,2),(3,1),(z,2),(second,1)}))
    (t,<>,<>,0)},
   {(s,7),(7,t)},
   s,
   t))}
```

B.10 Value Parameters

```
program divide(input,output);
var a,b,x,y,q,r : integer;
function max(a,b:integer):integer;
begin
  max := a;
  if a < b then
    max := b;
end;
function min(a,b: integer):integer;
begin
  min := b;
  if a < b then
    min := a
end;
procedure quot(x,y:integer;var q,r:integer);
begin
  q := 0;
  if y > x then
    begin
      q := 0;
      r := x;
    end
    else
      if y = x then
        begin
          q := 1;
          r := 0
        end
      else
        begin
          quot(x-y,y,q,r);
          q := q + 1
        end
end;
begin
```

```
  readln(x,y);
  if (x>0) and (y>0)  then begin
    a := max(x,y);
    b := min(x,y);
    quot(a,b,q,r);
    writeln('the quotient is ',q,'the remainder is ',r)
  end
  else writeln('error in input')
end.
```

```
{((max,<a,b,max>,{}),
  ({(s,<(a',<a>),(b',<b>)>,<>,0)
    (1,<(max,<a'>)>,
       <a',b'>,
       ({(:=,1),(;,2),(<,1),(if..then,1),(begin..end,1)},
        {(a,2),(max,1),(b,1)})),
    (3,<(max,<b'>)>,
       <>,
       ({(:=,1),(;,1)},
        {(b,1),(max,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,3),(3,t),(1,t)},
   s,
   t)),
 ((min,<a,b,min>,{}),
  ({(s,<(a',<a>),(b',<b>)>,<>,0)
    (6,<(min,<b'>)>,
       <a',b'>,
       ({(:=,1),(;,2),(<,1),(if..then,1),(begin..end,1)},
        {(b,2),(min,1),(a,1)})),
    (8,<(min,<a'>)>,
       <>,
       ({(:=,1)},
        {(a,1),(min,1)}))
    (t,<>,<>,0)},
   {(s,6),(6,8),(8,t),(6,t)},
   s,
   t)),
 ((quot,<x,y,q,r>,{}),
  ({(s,<(x',<x>),(y',<y>)>,<>,0)
    (10,<(q,<0>)>,
       <y',x'>,
       ({(:=,1),(;,2),(>,1),(if..then,1),(begin..end,1)},
        {(0,1),(q,1),(y,1),(x,1)})),
    (12,<(q,<0>),(r,<x'>)>,
       <>,
       ({(:=,2),(;,2),(begin..end,1)},
        {(0,1),(q,1),(x,1),(r,1)})),
    (15,<>,
       <y',x'>,
       ({(=,1),(if..then,1),(else,1)},
        {(y,1),(x,1)})),
    (16,<(q,<1>),(r,<0>)>,
       <>,
       ({(:=,2),(;,1),(begin..end,1)},
        {(1,1),(q,1),(0,1),(r,1)})),
    (18,<(quot,<<x',y'>,<y'>,<q>,<r>>),(q,<q,1>)>,
       <>,
```

```
      ({(-,1),(,,3),(quot,1),((),1),(;,1),(+,1),(:=,1),(begin..end,1),
        (else,1)},
       {(x,1),(y,2),(q,3),(r,1),(1,1)}))
    (t,<>,<>,0)},
  {(s,10),(15,16),(15,18),(10,12),(12,t),(10,15),(16,t),(18,t)},
  s,
  t)),
 ((divide,<input,output>,{}),
  ({(s,<>,<>,0)
    (22,<(x,<input^>),(input^,<input>),(y,<input^>),
        (input^,<input>)>,
       <x,0,y,0>,
       ({(,,1),(readln,1),((),3),(;,1),(>,2),(and,1),(if..then,1),(begin..end,1),
         (.,1)},
        {(x,2),(y,2),(0,2)})),
    (24,<(a,<(max,<<x>,<y>>)>),(b,<(min,<<x>,<y>>)>),(quot,<<a>,<b>,<q>,<r>>),
        (output,<output,'the quotient is ',q,'the remainder is ',r>),
        (output^,<r>),(output,<output,end-of-line>)>,
       <>,
       ({(,,8),(max,1),((),4),(:=,2),(;,3),(min,1),(quot,1),(writeln,1),
         (begin..end,1)},
        {(x,2),(y,2),(a,2),(b,2),(q,2),(r,2),('the quotient is ',1),
         ('the remainder is ',1)})),
    (28,<(output^,<'error in input'>),(output,<output,output^>),
        (output,<output,end-of-line>)>,
       <>,
       ({(writeln,1),((),1),(else,1)},
        {('error in input',1)}))
    (t,<>,<>,0)},
  {(s,22),(22,24),(24,t),(22,28),(28,t)},
  s,
  t))}
```

B.11 Selected Pascal programs

```
program reverse(input,output);
const n = 100;
var a : array [1..n] of real;
    temp : real;
    i, j, k : integer;
    n : integer;
begin
  readln(n);
  if n>100 then n := 100;
  for i := 1 to n do
    read(a[i]);
  readln;
  for i := 1 to n-1 do
    begin
      k := i;
      for j := i + 1 to n do
        if a[k] < a[j] then k := j;
      temp := a[i];
      a[i] := a[k];
      a[k] := temp
    end;
```

```
  for i := n downto 1 do
    writeln(a[i]);
  writeln
end.

{((reverse,<input,output>,{}),
  ({(s,<>,<>,0)
    (1,<(n,<input^>),(input^,<input>)>,
       <n,100>,
       ({(readln,1),((),1),(;,1),(>,1),(if..then,1),(begin..end,1),(.,1)},
        {(n,2),(100,1)})),
    (3,<(n,<100>)>,
       <>,
       ({(:=,1)},
        {(100,1),(n,1)})),
    (5,<(i,<1>)>,
       <>,
       ({(;,1)},
        {(i,1),(i,1)})),
    (6,<>,
       <i,n>,
       ({},
        {(n,1)})),
    (7,<(a,<input^,i>),(input^,<input>),(i,<i>)>,
       <>,
       ({([],1),(read,1),((),1)},
        {(i,1),(a,1)})),
    (10,<(input^,<input>),(i,<1>)>,
       <>,
       ({(;,2)},
        {(i,1),(i,1)})),
    (11,<>,
       <i,n,1>,
       ({(-,1)},
        {(n,1),(1,1)})),
    (12,<(k,<i>),(j,<i,1>)>,
       <>,
       ({(:=,1),(;,1),(+,1),(for..to..do,1),(begin..end,1)},
        {(i,2),(k,1),(1,1),(j,1)})),
    (14,<>,
       <j,n>,
       ({},
        {(n,1)})),
    (15,<>,
       <a,k,a,j>,
       ({([],2),(<,1),(if..then,1)},
        {(k,1),(a,2),(j,1)})),
    (16,<(k,<j>)>,
       <>,
       ({(:=,1)},
        {(j,1),(k,1)})),
    (17,<(j,<j>)>,
       <>,
       ({},
        {})),
    (21,<(temp,<a,i>),(a,<a,i,a,k>),(a,<a,k,temp>),
        (i,<i>)>,
       <>,
       ({(;,3)},
```

```
        {(k,2),(temp,2),(a,4),(i,2)})),
    (23,<(i,<n>)>,
       <>,
       ({(;,1)},
        {(n,1),(i,1)})),
    (24,<>,
       <i,i>,
       ({},
        {(i,1)})),
    (25,<(output^,<a,i>),(output,<output,output^>),(output,<output,end-of-line>),
       (i,<i>)>,
       <>,
       ({([],1),(writeln,1),((),1)},
        {(i,1),(a,1)})),
    (27,<(output,<output,end-of-line>)>,
       <>,
       ({(;,1)},
        {})),
    (t,<>,<>,0)},
   {(s,1),(1,3),(3,5),(1,5),(5,6),(6,7),(7,6),(6,10),(15,16),(16,17),
    (15,17),(12,14),(14,15),(17,14),(14,21),(10,11),(11,12),(21,11),(11,23),(23,24),
    (24,25),(25,24),(24,27),(27,t)},
   s,
   t))}
```

```
program search(input,output);
var n : integer;
    x : array [1..10] of integer;
    i,j,h,l : integer;
    item : integer;
procedure binary(h,l : integer; var j : integer);
var loc : integer;
begin
  j := 0;
  loc := round((h+l)/2);
  if item = x[loc] then j := loc;
  if item < x[loc] then
    begin
      l := loc + 1;
      if h >= l then binary(h,l,j)
    end;
  if item > x[loc] then
    begin
      h := loc-1;
      if h >= l then binary(h,l,j)
    end
end;
begin
  read(n);
  for i := 1 to n do
    read(x[i]);
  readln;
  readln(item);
  while item <> 0 do
    begin
      h := n;
      l := 1;
      binary(h,l,j);
      if j = 0 then writeln('not found')
```

```
      else writeln(item,'found at ',j);
      readln(item)
    end
end.

{((binary,<h,1,j>,{item,x}),
  ({(s,<(h',<h>),(1',<1>)>,<>,0)
    (1,<(j,<0>),(loc,<h',1',2>)>,
       <item,x,loc>,
       ({(:=,2),(;,3),(+,1),((),2),(/,1),(round,1),([],1),(=,1),
         (if..then,1),(begin..end,1)},
        {(0,1),(j,1),(h,1),(1,1),(2,1),(loc,2),(item,1),(x,1)})),
    (4,<(j,<loc>)>,
       <>,
       ({(:=,1)},
        {(loc,1),(j,1)})),
    (6,<>,
       <item,x,loc>,
       ({(;,1)},
        {(item,1),(loc,1),(x,1)})),
    (7,<(1',<loc,1>)>,
       <h',1'>,
       ({(+,1),(:=,1),(;,1),(>=,1),(if..then,1),(begin..end,1)},
        {(loc,1),(1,1),(1,2),(h,1)})),
    (9,<(binary,<<h'>,<1'>,<j>>)>,
       <>,
       ({(,,2),(binary,1),((),1)},
        {(h,1),(1,1),(j,1)})),
    (12,<>,
       <item,x,loc>,
       ({(;,1)},
        {(item,1),(loc,1),(x,1)})),
    (13,<(h',<loc,1>)>,
       <h',1'>,
       ({(-,1),(:=,1),(;,1),(>=,1),(if..then,1),(begin..end,1)},
        {(loc,1),(1,1),(h,2),(1,1)})),
    (15,<(binary,<<h'>,<1'>,<j>>)>,
       <>,
       ({(,,2),(binary,1),((),1)},
        {(h,1),(1,1),(j,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,4),(4,6),(1,6),(7,9),(6,7),(6,12),(9,12),(7,12),(13,15),
    (12,13),(12,t),(15,t),(13,t)},
   s,
   t)),
 ((search,<input,output>,{}),
  ({(s,<>,<>,0)
    (18,<(n,<input^>),(input^,<input>),(i,<1>)>,
       <>,
       ({(read,1),((),1),(;,1),(for..to..do,1),(begin..end,1),(.,1)},
        {(n,1),(1,1),(i,1)})),
    (20,<>,
       <i,n>,
       ({},
        {(n,1)})),
    (21,<(x,<input^,i>),(input^,<input>),(i,<i>)>,
       <>,
       ({([],1),(read,1),((),1)},
        {(i,1),(x,1)})),
```

```
    (24,<(input^,<input>),(item,<input^>),(input^,<input>)>,
       <>,
       ({(;,3)},
        {(item,1)})),
    (25,<>,
       <item,0>,
       ({(<>,1),(while..do,1)},
        {(item,1),(0,1)})),
    (26,<(h,<n>),(l,<l>),(binary,<<h>,<l>,<j>>)>,
       <j,0>,
       ({(:=,2),(;,3),(,,2),(binary,1),(().1),(=,1),(if..then,1),(begin..end,1)},
        {(n,1),(h,2),(1,1),(1,2),(j,2),(0,1)})),
    (30,<(output^,<'not found'>),(output,<output,output^>),(output,<output,end-of-line>)>,
       <>,
       ({(writeln,1),(().1)},
        {('not found',1)})),
    (31,<(output,<output,item,'found at ',j>),(output^,<j>),(output,<output,end-of-line>)>,
       <>,
       ({(,,2),(writeln,1),(().1),(else,1)},
        {(item,1),('found at ',1),(j,1)})),
    (33,<(item,<input^>),(input^,<input>)>,
       <>,
       ({(;,1)},
        {(item,1)}))
    (t,<>,<>,0)},
   {(s,18),(18,20),(20,21),(21,20),(20,24),(26,30),(30,33),(26,31),(31,33),(25,26),
    (33,25),(25,t),(24,25)},
   s,
   t))}
```

```
program code(input,output);
var codemat : packed array['a'..'0',1..40] of boolean;
    i : integer;
    ch :char;
    error : boolean;
begin
  error := false;
  for i := 1 to 40 do
    codemat['0',i] := false;
  for i := 1 to 40 do
    for ch := 'a' to 'z' do
      codemat[ch,i] := false;
  i := 1;
  while not eoln do
    begin
      read(ch);
      if (ch < 'a') or (ch > 'z') then
        if ch = ' ' then ch := '0'
        else
          begin
            writeln('error in input');
            error := true
          end;
      if not error then
        codemat[ch,i] := true;
      i := i + 1
    end;
end.
```

```
{((code,<input,output>,{}),
  ({(s,<>,<>,0)
```

```
    (1,<(error,<false>),(i,<1>)>,
       <>,
       ({:=,1),(;,1),(for..to..do,1),(begin..end,1),(.,1)},
        {(false,1),(error,1),(1,1),(i,1)})),
    (3,<>,
       <i,40>,
       ({},
        {(40,1)})),
    (4,<(codemat,<codemat,'0',i,false>),(i,<i>)>,
       <>,
       ({(,,1),([],1),(:=,1)},
        {('0',i),(i,1),(false,1),(codemat,1)})),
    (6,<(i,<1>)>,
       <>,
       ({(;,1)},
        {(1,1),(i,1)})),
    (7,<>,
       <i,40>,
       ({},
        {(40,1)})),
    (8,<(ch,<'a'>)>,
       <>,
       ({(for..to..do,1)},
        {('a',1),(ch,1)})),
    (9,<>,
       <ch,'z'>,
       ({},
        {('z',1)})),
    (10,<(codemat,<codemat,ch,i,false>),(ch,<ch>)>,
       <>,
       ({(,,1),([],1),(:=,1)},
        {(ch,1),(i,1),(false,1),(codemat,1)})),
    (11,<(i,<i>)>,
       <>,
       ({},
        {})),
    (13,<(i,<1>)>,
       <>,
       ({(;,2)},
        {(i,1),(i,1)})),
    (14,<>,
       <input>,
       ({(eoln,1),(not,1),(while..do,1)},
        {})),
    (15,<(ch,<input^>),(input^,<input>)>,
       <ch,'a',ch,'z'>,
       ({(read,1),((),3),(;,1),(<,1),(>,1),(or,1),(if..then,1),(begin..end,1)},
        {(ch,3),('a',1),('z',1)})),
    (17,<>,
       <ch,' '>,
       ({(=,1),(if..then,1)},
        {(ch,1),(' ',1)})),
    (18,<(ch,<'0'>)>,
       <>,
       ({(:=,1)},
        {('0',1),(ch,1)})),
    (19,<(output^,<'error in input'>),(output,<output,output^>),(output,<output,end-of-
line>),
        (error,<true>)>,
       <>,
```

```
      ({(writeln,1),((),1),(;,1),(:=,1),(begin..end,1),(else,1)},
       {('error in input',1),(true,1),(error,1)})),
    (23,<>,
       <error>,
       ({(;,1)},
        {(error,1)})),
    (24,<(codemat,<codemat,ch,i,true>)>,
       <>,
       ({(,,1),([],1),(:=,1)},
        {(ch,1),(i,1),(true,1),(codemat,1)})),
    (26,<(i,<i,1>)>,
       <>,
       ({(;,2)},
        {(i,2),(1,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,3),(3,4),(4,3),(3,6),(8,9),(9,10),(10,9),(9,11),(6,7),
    (7,8),(11,7),(7,13),(17,18),(17,19),(15,17),(15,23),(18,23),(19,23),(23,24),
    (24,26),(23,26),(14,15),(26,14),(14,t),(13,14)},
   s,
   t))}
```

```
program positiveonly(file1,file2);
type ftype = array[1..10] of integer;
var file1, file2 : file of ftype;
    a : ftype;
begin
  reset(file1);
  rewrite(file2);
  while not eof(file1) do
    begin
      a := file1^;
      if a[1] > 0 then
        begin
          file2^ := file1^;
          put(file2)
        end;
      get(file1);
    end
end.
```

```
{((positiveonly,<file1,file2>,{}),
  ({(s,<>,<>,0)
    (1,<(file1^,<file1>),(file2,<>)>,
       <>,
       ({(reset,1),((),2),(;,2),(rewrite,1),(begin..end,1),(.,1)},
        {(file1,1),(file2,1)})),
    (3,<>,
       <file1>,
       ({(eof,1),((),1),(not,1),(while..do,1)},
        {(file1,1)})),
    (4,<(a,<file1^>)>,
       <a,i,0>,
       ({(^,1),(:=,1),(;,1),([],1),(>,1),(if..then,1),(begin..end,1)},
        {(file1,1),(a,2),(1,1),(0,1)})),
    (6,<(file2^,<file1^>),(file2,<file2,file2^>)>,
       <>,
       ({(^,2),(:=,1),(;,1),(put,1),((),1),(begin..end,1)},
        {(file1,1),(file2,2)})),
    (10,<(file1^,<file1>)>,
```

```
        <>,
        ({(;,2)},
         {(file1,1)}))
     (t,<>,<>,0)},
   {(s,1),(4,6),(6,10),(4,10),(3,4),(10,3),(3,t),(1,3)},
   s,
   t))}
```

```
program positiveonly2(file1,file2);
type ftype = array[1..10] of integer;
var file1, file2 : file of ftype;
    a : ftype;
begin
  reset(file1);
  rewrite(file2);
  while  not eof(file1) do
    begin
      read(file1,a);
      if a[1] > 0 then
        write(file2,a)
    end
end.
```

```
{((positiveonly2,<file1,file2>,{}),
  ({(s,<>,<>,0)
    (1,<(file1^,<file1>),(file2,<>)>,
       <>,
       ({(reset,1),((),2),(;,2),(rewrite,1),(begin..end,1),(.,1)},
        {(file1,1),(file2,1)})),
    (3,<>,
       <file1>,
       ({(eof,1),((),1),(not,1),(while..do,1)},
        {(file1,1)})),
    (4,<(a,<file1^>),(file1^,<file1>)>,
       <a,1,0>,
       ({(,,1),(read,1),((),1),(;,1),([],1),(>,1),(if..then,1),(begin..end,1)},
        {(file1,1),(a,2),(1,1),(0,1)})),
    (6,<(file2^,<a>),(file2,<file2,file2^>)>,
       <>,
       ({(,,1),(write,1),((),1)},
        {(file2,1),(a,1)}))
    (t,<>,<>,0)},
  {(s,1),(4,6),(3,4),(3,t),(6,3),(4,3),(1,3)},
   s,
   t))}
```

```
program exp(input,f);
var a : packed array[1..10] of char;
    f : text;
    i : integer;
begin
  rewrite(f);
  for i := 1 to 10 do
    read(a[i]);
  write(f,a)
end.
```

```
{((exp,<input,f>,{}),
  ({(s,<>,<>,0)
    (1,<(f,<>),(i,<1>)>,
```

```
        <>,
        ({(rewrite,1),((),1),(;,1),(for..to..do,1),(begin..end,1),(.,1)},
         {(f,1),(1,1),(i,1)})),
     (3,<>,
        <i,10>,
        ({},
         {(10,1)})),
     (4,<(a,<input^,i>),(input^,<input>),(i,<i>)>,
        <>,
        ({([],1),(read,1),((),1)},
         {(i,1),(a,1)})),
     (6,<(f^,<a>),(f,<f,f^>)>,
        <>,
        ({(;,1)},
         {(f,1),(a,1)}))
     (t,<>,<>,0)},
    {(s,1),(1,3),(3,4),(4,3),(3,6),(6,t)},
    s,
    t))}
```

```
program dumpdata(data,output);
var ch : char;
    data : text;
begin
  reset(data);
  while not eof(data) do
    begin
      while not eoln(data) do
        begin
          read(data,ch);
          write(ch)
        end;
      readln(data);
      writeln;
    end
end.
```

```
{((dumpdata,<data,output>,{}),
  ({(s,<>,<>,0)
     (1,<(data^,<data>)>,
        <>,
        ({(reset,1),((),1),(;,1),(begin..end,1),(.,1)},
         {(data,1)})),
     (2,<>,
        <data>,
        ({(eof,1),((),1),(not,1),(while..do,1)},
         {(data,1)})),
     (3,<>,
        <data>,
        ({(eoln,1),((),1),(not,1),(while..do,1),(begin..end,1)},
         {(data,1)})),
     (4,<(ch,<data^>),(data^,<data>),(output^,<ch>),
        (output,<output,output^>)>,
        <>,
        ({(,1),(read,1),((),2),(;,1),(write,1),(begin..end,1)},
         {(data,1),(ch,2)})),
     (9,<(data^,<data>),(output,<output,end-of-line>)>,
        <>,
        ({(;,3)},
         {(data,1)}))
```

```
    (t,<>,<>,0)},
   {(s,1),(3,4),(4,3),(3,9),(2,3),(9,2),(2,t),(1,2)},
   s,
   t))}

program pointers(input,output,dfile);
type line = packed array [1..80] of char;
     reclink = ^rectype;
     rectype = record
                 data : line;
                 link : reclink
               end;
var list : rectype;
    dfile : text;
    head, p, next : reclink;
    i : integer;
begin
  reset(dfile);
  new(p);
  head := p;
  while (not eof(dfile)) do
    begin
      i := 1;
      while (i <= 80) and (not eoln(dfile)) do
        begin
          read(dfile,p^.data[i]);
          i := i + 1
        end;
      readln(dfile);
      next := p;
      new(p);
      next^.link := p;
    end;
  next^.link := nil
end.

{((pointers,<input,output,dfile>,{}),
  ({(s,<>,<>,0)
    (1,<(dfile^,<dfile>),(p,<>),(rectype,<p,rectype>),
       (head,<p>)>,
      <>,
      ({(reset,1),((),2),(;,3),(new,1),(:=,1),(begin..end,1),(.,1)},
       {(dfile,1),(p,2),(head,1)})),
    (4,<>,
       <dfile>,
       ({(eof,1),((),2),(not,1),(while..do,1)},
        {(dfile,1)})),
    (5,<(i,<1>)>,
       <>,
       ({(:=,1),(;,1),(begin..end,1)},
        {(1,1),(i,1)})),
    (6,<>,
       <i,80,dfile>,
       ({(<=,1),((),3),(eoln,1),(not,1),(and,1),(while..do,1)},
        {(i,1),(80,1),(dfile,1)})),
    (7,<(rectype,<dfile^,rectype,i>),(dfile^,<dfile>),(i,<i,1>)>,
       <>,
       ({(^,1),(.,1),([],1),(,,1),(read,1),((),1),(;,1),(+,1),
         (:=,1),(begin..end,1)},
```

```
        {(dfile,1),(data,1),(i,3),(p,1),(1,1)})),
    (14,<(dfile^,<dfile>),(next,<p>),(p,<>),
        (rectype,<p,rectype>),(rectype,<next,rectype,p>)>,
        <>,
        ({(;,5)},
         {(link,1),(p,3),(next,2),(dfile,1)})),
    (16,<(rectype,<next,rectype,nil>)>,
        <>,
        ({(;,1)},
         {(link,1),(nil,1),(next,1)}))
    (t,<>,<>,0)},
  {(s,1),(6,7),(7,6),(6,14),(5,6),(4,5),(14,4),(4,16),(1,4),(16,t)},
  s,
  t))}
```

```
program primes(input,output);
const n = 50;
var sett, prime : set of 2..n;
    next, j : integer;
begin
  sett := [2..50];
  prime := [];
  next := 2;
  repeat
    while not (next in sett) do
      next := succ(next);
    prime := prime + [next];
    j := next;
    while j <= n do
      begin
        sett := sett - [j];
        j := j + next
      end
  until sett = [];
  for j := 1 to n do
    if j in prime then
      writeln(j)
end.
```

```
{((primes,<input,output>,{}),
  ({(s,<>,<>,0)
    (1,<(sett,<2,50>),(prime,<>),(next,<2>)>,
       <>,
       ({(..,1),([],2),(:=,3),(;,3),(begin..end,1),(.,1)},
        {(2,2),(50,1),(sett,1),(prime,1),(next,1)})),
    (4,<>,
       <next,sett>,
       ({(in,1),((),1),(not,1),(while..do,1),(repeat..until,1)},
        {(next,1),(sett,1)})),
    (5,<(next,<next>)>,
       <>,
       ({(succ,1),((),1),(:=,1)},
        {(next,2)})),
    (8,<(prime,<prime,next>),(j,<next>)>,
       <>,
       ({(;,3)},
        {(next,2),(j,1),(prime,2)})),
    (9,<>,
       <j,n>,
```

```
        ({(<=,1),(while..do,1)},
         {(j,1),(n,1)})),
     (10,<(sett,<sett,j>),(j,<j,next>)>,
        <>,
        ({([],1),(-,1),(:=,2),(;,1),(+,1),(begin..end,1)},
         {(sett,2),(j,3),(next,1)})),
     (12,<>,
        <sett>,
        ({([],1),(=,1)},
         {(sett,1)})),
     (14,<(j,<1>)>,
        <>,
        ({;,1)},
         {(1,1),(j,1)})),
     (15,<>,
        <j,n>,
        ({},
         {(n,1)})),
     (16,<>,
        <j,prime>,
        ({(in,1),(if..then,1)},
         {(j,1),(prime,1)})),
     (17,<(output^,<j>),(output,<output,output^>),(output,<output,end-of-line>)>,
        <>,
        ({(writeln,1),((),1)},
         {(j,1)})),
     (18,<(j,<j>)>,
        <>,
        ({},
         {}))
     (t,<>,<>,0)},
    {(s,1),(4,5),(5,4),(4,8),(9,10),(10,9),(9,12),(8,9),(12,14),(12,4),
     (1,4),(16,17),(17,18),(16,18),(14,15),(15,16),(18,15),(15,t)},
    s,
    t))}
```

```
program record1(input,output);
type name = record
              last : packed array[1..12] of char;
              first : packed array[1..10] of char;
              init : char
            end;
     address = record
                 street : packed array[1..20] of char;
                 cities : packed array[1..20] of char
               end;
     person = record
                pname : name;
                paddress : address;
                ssn : integer;
                depend : integer;
                mar : boolean
              end;
var student : person;
    i : integer;
begin
  with student do
    begin
      for i := 1 to 12 do
```

```
        read(pname.last[i]);
      for i := 1 to 10 do
        read(pname.first[i]);
      read(pname.init);
      for i := 1 to 20 do
        read(paddress.street[i]);
      for i := 1 to 20 do
        read(paddress.cities[i]);
      read(ssn);
      read(depend);
      read(mar)
    end
end.

{((record1,<input,output>,{}),
  ({(s,<>,<>,0)
    (1,<(i,<1>)>,
       <>,
       ({(for..to..do,1),(begin..end,2),(with..do,1),(.,1)},
        {(1,1),(i,1)})),
    (2,<>,
       <i,12>,
       ({},
        {(12,1)})),
    (3,<(student,<student,input^,i>),(input^,<input>),(i,<i>)>,
       <>,
       ({(.,1),([],1),(read,1),((),1)},
        {(last,1),(i,1),(pname,1)})),
    (5,<(i,<1>)>,
       <>,
       ({(;,1)},
        {(1,1),(i,1)})),
    (6,<>,
       <i,10>,
       ({},
        {(10,1)})),
    (7,<(student,<student,input^,i>),(input^,<input>),(i,<i>)>,
       <>,
       ({(.,1),([],1),(read,1),((),1)},
        {(first,1),(i,1),(pname,1)})),
    (10,<(student,<student,input^>),(input^,<input>),(i,<1>)>,
       <>,
       ({(;,2)},
        {(1,1),(i,1),(init,1),(pname,1)})),
    (11,<>,
       <i,20>,
       ({},
        {(20,1)})),
    (12,<(student,<student,input^,i>),(input^,<input>),(i,<i>)>,
       <>,
       ({(.,1),([],1),(read,1),((),1)},
        {(street,1),(i,1),(paddress,1)})),
    (14,<(i,<1>)>,
       <>,
       ({(;,1)},
        {(1,1),(i,1)})),
    (15,<>,
       <i,20>,
       ({},
        {(20,1)})),
```

```
    (16,<(student,<student,input^,i>),(input^,<input>),(i,<i>)>,
        <>,
        ({(.,1),([],1),(read,1),((),1)},
         {(cities,1),(i,1),(paddress,1)})),
    (20,<(student,<student,input^>),(input^,<input>),(student,<student,input^>),
        (input^,<input>),(student,<student,input^>),(input^,<input>)>,
        <>,
        ({(;,3)},
         {(mar,1),(depend,1),(ssn,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,2),(2,3),(3,2),(2,5),(5,6),(6,7),(7,6),(6,10),(10,11),
    (11,12),(12,11),(11,14),(14,15),(15,16),(16,15),(15,20),(20,t)},
   s,
   t))}
```

```
program record2(input,output);
type name = record
              last : packed array[1..12] of char;
              first : packed array[1..10] of char;
              init : char
            end;
     address = record
                 street : packed array[1..20] of char;
                 cities : packed array[1..20] of char
               end;
     person = record
                pname : name;
                paddress : address;
                ssn : integer;
                depend : integer;
                mar : boolean
              end;
var student : person;
    i : integer;
begin
  with student do
    begin
      with pname do
        begin
          for i := 1 to 12 do
            read(last[i]);
          for i := 1 to 10 do
            read(first[i]);
          read(init);
        end;
      with paddress do
        begin
          for i := 1 to 20 do
            read(street[i]);
          for i := 1 to 20 do
            read(cities[i]);
        end;
      read(ssn);
      read(depend);
      read(mar)
    end
end.
```

```
{((record2,<input,output>,{}),
  ({(s,<>,<>,0)
```

```
      (1,<(i,<1>)>,
         <>,
         ({(for..to..do,1),(begin..end,3),(with..do,2),(.,1)},
          {(1,1),(i,1)})),
      (2,<>,
         <i,12>,
         ({},
          {(i2,1)})),
      (3,<(student,<student,input^,i>),(input^,<input>),(i,<i>)>,
         <>,
         ({([],1),(read,1),((),1)},
          {(i,1),(last,1)})),
      (5,<(i,<1>)>,
         <>,
         ({;,1)},
          {(i,1),(i,1)})),
      (6,<>,
         <i,10>,
         ({},
          {(10,1)})),
      (7,<(student,<student,input^,i>),(input^,<input>),(i,<i>)>,
         <>,
         ({([],1),(read,1),((),1)},
          {(i,1),(first,1)})),
      (11,<(student,<student,input^>),(input^,<input>),(i,<1>)>,
         <>,
         ({;,3)},
          {(i,1),(i,1),(init,1),(paddress,1)})),
      (12,<>,
         <i,20>,
         ({},
          {(20,1)})),
      (13,<(student,<student,input^,i>),(input^,<input>),(i,<i>)>,
         <>,
         ({([],1),(read,1),((),1)},
          {(i,1),(street,1)})),
      (15,<(i,<1>)>,
         <>,
         ({;,1)},
          {(i,1),(i,1)})),
      (16,<>,
         <i,20>,
         ({},
          {(20,1)})),
      (17,<(student,<student,input^,i>),(input^,<input>),(i,<i>)>,
         <>,
         ({([],1),(read,1),((),1)},
          {(i,1),(cities,1)})),
      (22,<(student,<student,input^>),(input^,<input>),(student,<student,input^>),
          (input^,<input>),(student,<student,input^>),(input^,<input>)>,
         <>,
         ({;,4)},
          {(mar,1),(depend,1),(ssn,1)}))
     (t,<>,<>,0)},
    {(s,1),(1,2),(2,3),(3,2),(2,5),(5,6),(6,7),(7,6),(6,11),(11,12),
     (12,13),(13,12),(12,15),(15,16),(16,17),(17,16),(16,22),(22,t)},
    s,
    t))}

program record3(input,output);
```

```
type nametype = packed array [1..20] of char;
     addresstype = packed array [1..30] of char;
     status = (m,s,d);
     person = record
                name : nametype;
                address : addresstype;
                case tag : status of
                  m : (spousename : nametype;
                       children : integer);
                  s : (sex : char);
                  d : (ddate : packed array[1..8] of char;
                       children : integer)
              end;
var tagg : status;
    employee : person;
    students : array[1..100] of person;
begin
  with students do
    begin
      for i := 1 to 20 do
        read(name[i]);
      for i := 1 to 30 do
        read(address[i]);
      if tag = m then
        writeln(spousename,children)
      else if tag = s then
        writeln(sex)
      else if tag = d then
        writeln(ddate,children);
    end
end.
```

```
{((record3,<input,output>,{}),
  ({(s,<>,<>,0)
    (1,<(i,<1>)>,
       <>,
       ({(for..to..do,1),(begin..end,2),(with..do,1),(:=,1)},
        {(1,1),(i,1)})),
    (2,<>,
       <i,20>,
       ({},
        {(20,1)})),
    (3,<(students,<students,input^,i>),(input^,<input>),(i,<i>)>,
       <>,
       ({([],1),(read,1),((),1)},
        {(i,1),(name,1)})),
    (5,<(i,<1>)>,
       <>,
       ({(:=,1)},
        {(1,1),(i,1)})),
    (6,<>,
       <i,30>,
       ({},
        {(30,1)})),
    (7,<(students,<students,input^,i>),(input^,<input>),(i,<i>)>,
       <>,
       ({([],1),(read,1),((),1)},
        {(i,1),(address,1)})),
    (9,<>,
```

```
       <students,m>,
       ({(;,1)},
        {(tag,1),(m,1)})),
    (10,<(output,<output,students,students>),(output^,<students>),(output,<output,end-of-
line>)>,
       <>,
       ({(,,1),(writeln,1),((),1)},
        {(spousename,1),(children,1)})),
    (11,<>,
       <students,s>,
       ({(=,1),(if..then,1),(else,1)},
        {(tag,1),(s,1)})),
    (12,<(output^,<students>),(output,<output,output^>),(output,<output,end-of-line>)>,
       <>,
       ({(writeln,1),((),1)},
        {(sex,1)})),
    (13,<>,
       <students,d>,
       ({(=,1),(if..then,1),(else,1)},
        {(tag,1),(d,1)})),
    (14,<(output,<output,students,students>),(output^,<students>),(output,<output,end-of-
line>)>,
       <>,
       ({(,,1),(writeln,1),((),1),(;,1)},
        {(ddate,1),(children,1)}))
   (t,<>,<>,0)},
  {(s,1),(1,2),(2,3),(3,2),(2,5),(5,6),(6,7),(7,6),(6,9),(13,14),
   (11,12),(11,13),(9,10),(10,t),(9,11),(12,t),(14,t),(13,t)},
  s,
  t))}

program record4(input,output);
type nametype = packed array [1..20] of char;
     addresstype = packed array [1..30] of char;
     status = (m,s,d);
     person = record
                name : nametype;
                address : addresstype;
                case tag : status of
                  m : (spousename : nametype;
                       children : integer);
                  s : (sex : char);
                  d : (ddate : packed array[1..8] of char;
                       children : integer)
              end;
var tagg : status;
    employee : person;
    students : array[1..100] of person;
begin
  with students do
    begin
      for i := 1 to 20 do
        read(name[i]);
      for i := 1 to 30 do
        read(address[i]);
      case tag of
        m : writeln(spousename,children);
        s : writeln(sex);
```

```
        d : writeln(ddate,children);
      end
    end
end.

{((record4,<input,output>,{}),
  ({(s,<>,<>,0)
    (1,<(i,<i>)>,
       <>,
       ({(for..to..do,1),(begin..end,2),(with..do,1),(.,1)},
        {(1,1),(i,1)})),
    (2,<>,
       <i,20>,
       ({},
        {(20,1)})),
    (3,<(students,<students,input^,i>),(input^,<input>),(i,<i>)>,
       <>,
       ({([],1),(read,1),((),1)},
        {(i,1),(name,1)})),
    (5,<(i,<i>)>,
       <>,
       ({(;,1)},
        {(i,1),(i,1)})),
    (6,<>,
       <i,30>,
       ({},
        {(30,1)})),
    (7,<(students,<students,input^,i>),(input^,<input>),(i,<i>)>,
       <>,
       ({([],1),(read,1),((),1)},
        {(i,1),(address,1)})),
    (9,<>,
       <students>,
       ({(;,1)},
        {(tag,1)})),
    (10,<(output,<output,students,students>),(output^,<students>),(output,<output,end-of-
line>)>,
       <>,
       ({(,,1),(writeln,1),((),1)},
        {(spousename,1),(children,1),(m:,1)})),
    (11,<(output^,<students>),(output,<output,output^>),(output,<output,end-of-line>)>,
       <>,
       ({(writeln,1),((),1),(;,1)},
        {(sex,1),(s:,1)})),
    (12,<(output,<output,students,students>),(output^,<students>),(output,<output,end-of-
line>)>,
       <>,
       ({(,,1),(writeln,1),((),1),(;,2)},
        {(ddate,1),(children,1),(d:,1)}))
    (t,<>,<>,0)},
   {(s,1),(1,2),(2,3),(3,2),(2,5),(5,6),(6,7),(7,6),(6,9),(9,10),
    (10,t),(9,11),(11,t),(9,12),(12,t)},
   s,
   t))}
```

C User's Manual

NAME

srp – generator of *StandardRep* for Pascal programs

SYNOPSIS

srp pascalfilename

DESCRIPTION

srp generates a textual form of the *StandardRep* using the set, sequence, and tuple notation of [1] from an ISO Standard Pascal program. The identifiers and operators in the source program are mapped directly to the *StandardRep* without any changes. The Pascal input file must contain a syntactically correct Pascal program and must have only standard Pascal features. To deal with the non-standard features of Pascal programs, we have to modify the source program of the generator. Since the generator does not detect syntactical errors, if there is a syntax error in input programs, the generator may produce a set of *UnitRepType*'s only for procedure units which do not have errors and are processed before the error occurs. Hence, we have to make sure that we have an input program which is error-free and solely based on the Standard Pascal. Otherwise, we have to revise the generator source program for the non-standard features or for the syntactical error-detection. Thus we suggest that the user must check the input Pascal program by a pascal compiler before running the **srp**. The **srp** produces the output on the corresponding file suffixed **.sr** which contains the *StandardRep* for the input Pascal program.