A Standard Representation of Imperative Language Programs

James M. Bieman
Albert L. Baker[1]
Paul N. Clites

Department of Computer Science
Iowa State University
Ames, Iowa 50011
(515) 294-4377

David A. Gustafson
Austin C. Melton

Computer Science Department
Kansas State University
Manhattan, Kansas 66506
(913) 532-6350

Keywords
software tools, software measures, program representation

Published in *Journal of Systems and Software*, vol. 8, no. 1, pp. 13-37, Jan. 1988.

Abstract

Numerous research results in the areas of software measures and software tools are predicated on a particular programming language, or on some characterizations of a programming language. For example, numerous software measures have been defined only for structured programming languages and several of the reported approaches to program testing define a specific language. However, this proliferation of languages upon which measures and tools are defined makes independent evaluation and comparison of measures and tools problematic.

We propose a standard representation of imperative language programs which is independent of the syntax of any particular programming language, yet the standard representation supports the definition of a wide range of tools and measures. Additionally, the standard representation will allow for the masking of actual program semantics. Thus the standard representation provides a vehicle by which large volumes of industrial software might be made available to researchers while protecting the proprietary nature of the programs.

[1]Dr. Baker is currently a Shell Faculty Fellow, supported in part by the Shell Companies Foundation, Inc.

1 Introduction

Programmers, managers and researchers need quantitative descriptions of software. Software continues to increase in complexity with a corresponding decrease in the ability of developers to understand major portions of projects. Developers need enhanced tools to help them understand modern software, and some of these enhanced tools must provide quantitative abstractions of software documents. The development of these tools is in a primitive state, and there is no general framework to guide researchers in this area [6].

Software tools and measures are often defined in terms of source code [7], underlying control flow graphs [11], system interfaces [10], textual position [16], and/or data dependencies [4]. For example, the cyclomatic number as defined by McCabe [13] is defined both in terms of the underlying flowgraph and the complexity of the expressions in the program predicates. This measure is also used as the basis for a testing tool [14]. Many researchers have defined simple programming languages for the purpose of defining tools and measures [15]. Because tools and measures are commonly defined in a non-standard, language dependent fashion, it is often difficult to implement and compare these software engineering tools.

Considering the non-standard definitions of tools and measures, it is not surprising that few convincing analytical or empirical validations of these facilities have been conducted. Another problem in conducting empirical validation studies is the lack of data. In order to validate the worth of software tools on large projects, they must be applied to large software systems. Yet the organizations that develop large systems are naturally reluctant to share proprietary software with outside researchers [8].

A standard, language independent representation of programs can provide a clear and precise basis for the definition of tools and measures. If the representation hides the semantics of the actual program, then proprietary interests are also protected.

We carefully define a representation of source programs that allows for the protection of the semantics of the programs and facilitates the definition of a wide range of tools and measures. All the information required to perform control flow analysis, data dependency analysis, and expression complexity analysis is preserved. Yet there is no way to infer, at any level, the purpose or functionality of the original source program. The semantics are protected because:

1. identifiers and operator names are encoded, and
2. the mapping is many to one. A program can be converted to its standard representation while the inverse translation is undefined, and cannot pragmatically be approximated.

The language independent representation is computable from source programs written in imperative programming languages in time similar to that required for traditional data flow analysis [9].

The representation should serve as the basis of a strategy for publishing large volumes of data on actual source programs. Such data can be used by a broad spectrum of software engineers interested in software tools and measures. If we define tools and measures in terms of the standard representation, we do not have to redefine them for each language. We only need to provide a mapping from programs in each language to the representation. Thus, a research environment for software tools or software measures might consist of:

1. the standard representation,

2. a set of tools or measures defined in terms of the representation, and

3. a set of translators that convert programs in a number of languages to the standard representation.

The standard representation is defined as an abstract data type. The domain of the abstract data type is a structure that incorporates the control flow, data flow, expression structures, and integration structures of a program unit (e.g. a Pascal procedure or function). Software measures are defined as operations on the abstract data type.

We use Pascal as an example imperative language to demonstrate the translation to the representation. The mapping from Pascal programs to the representation is described in terms of the semantic routines that are inserted by a syntax directed compiler. Similar mappings can be defined for other imperative languages including C, FORTRAN, COBOL, Ada, assembler, etc.

The remainder of the paper is organized as follows. The representation is formally defined in Section 2. Section 3 describes the mapping from Pascal programs to the representation. Section 4 describes the encoding of program semantics. A set of measures and software tools are defined in terms of the representation in Section 5. Section 6 describes current problems and the direction of our research.

2 A Standard Representation

The standard representation incorporates the concepts of control flow, data dependency, integration structures, and expression complexity.

Program unit control flow is modeled by the familiar control flowgraph 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.

Definition 2.2 A *sequential block* of code in a source program P is a sequence of tokens in P that is always executed starting with the first token in the sequence, all the tokens in the sequence are always executed sequentially, and the sequence is always exited at the end.

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 [9].

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. For a detailed description of the specification language used, see [2]. To represent an entire program, we first break the program down 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

$$\textit{StandardRep} \quad = \quad \textit{set of UnitRepType}$$

Type Definition 2

$$\begin{aligned} \textit{UnitRepType} \quad = \quad & \textit{ordered pair(} \\ & \textit{Interface: HeaderType,} \\ & \textit{UFS: UnitFlowStructure)} \end{aligned}$$

The interface must contain a unique name by which the program unit is known[2], and the information necessary to determine the data interface with the rest of the program. Thus the interface should have 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

$$\begin{aligned} \textit{HeaderType} \quad = \quad & \textit{triple(} \\ & \textit{UnitName: UnitID,} \\ & \textit{FormalParams: sequence of VarID,} \\ & \textit{Globals: set of VarID)} \end{aligned}$$

In characterizing the control flow and data dependencies within a procedure, we follow closely the example of [15]. The *UnitFlowStructure* closely resembles a conventional control flowgraph with information concerning data dependency, integration, and software science measures [7] embedded within the nodes.

Type Definition 4

$$\begin{aligned} \textit{UnitFlowStructure} \quad = \quad & \textit{4-tuple(} \\ & \textit{Nodes: set of NodeType,} \\ & \textit{Edges: set of EdgeType,} \\ & \textit{Start: NodeID,} \\ & \textit{Terminal: NodeID)} \end{aligned}$$

Type Definition 5

[2]In the mapping from actual source programs to the standard representation, all identifiers, constants, and operators are to be encoded. We might map each distinct operator in a program to $O_1, O_2, \ldots$ and each operand to $V_1, V_2, \ldots$. We proceed with the specification of *StandardRep* simply assuming this encoding.

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

A node must contain information about the uses and definitions of variables within the corresponding basic block. We wish to retain the distinction between references that are used for definitions and references in predicates, since some published tools make this distinction [15]. We also include operator and operand counts from the corresponding basic block, since such counts are used in software science measures[7] 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: UseType,
Counts: HalsteadInfoType)

Type Definition 7

HalsteadInfoType = *ordered pair(*
opcounts: set of OperatorCount,
opandcounts: set of OperandCounts)

Type Definition 8

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

Type Definition 9

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

Effectively, 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

$$\mathit{DefinitionType} = \mathit{SimpleDefinition} \mid \mathit{ProceduralDefinition}$$

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

Type Definition 11

$$\mathit{ExprComponent} = \mathit{VarID} \mid \mathit{ConstID} \mid \mathit{FunctionUse}$$

The definition of a variable by assignment is represented by 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. The representation preserves the maximum amount of data dependency information possible with a static (compile time) analysis. We can apply the information-flow relations of Bergeretti and Carré[3] to the *StandardRep* of a program to determine the interprocedural dependencies.

Type Definition 12

$$\begin{aligned} \mathit{SimpleDefinition} = \ & \textit{ordered pair(} \\ & \textit{DefinedVariable: VarID,} \\ & \textit{Uses: UseType)} \end{aligned}$$

Type Definition 13

$$\begin{aligned} \mathit{ProceduralDefinition} = \ & \textit{ordered pair(} \\ & \textit{ProcName: UnitID,} \\ & \textit{ActualParams: sequence of UseType)} \end{aligned}$$

All that remains is to specify the types *UseType* and *FunctionUse*.

Type Definition 14

$$\mathit{UseType} = \textit{sequence of ExprComponent}$$

Type Definition 15

$$\begin{aligned} \mathit{FunctionUse} = \ & \textit{ordered pair(} \\ & \textit{FunctionName: UnitID,} \\ & \textit{ActualParams: sequence of UseType)} \end{aligned}$$

3 The Representation of Pascal Programs

As an example mapping from a program to its StandardRep, we describe the translation for Pascal programs. The version of Pascal that we use as the domain of the mapping is the 1985 ISO Pascal Standard described by Jensen and Wirth[12].

To ease the discussion and to keep examples clear we will not encode the program identifiers and operator names. Encoding issues will be described separately in Section 4. The StandardRep's in the examples are presented in textual form using the set, sequence, tuple notation of Section 2. Other concrete structures (possibly non-ASCII) may be used to store an StandardRep, but the textual form is adequate and reasonable for human consumption.

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 *UnitRepType*'s of the individual procedures. Each procedure or function in a Pascal program is treated as an individual program unit with its own $UnitRepType$.

Structure of Program Units

Interface Component

The *UnitRepType* for each Pascal program (main procedure), procedure, or function has an *Interface* component and a *UFS* component. The *Interface* component models the binding of an individual procedure with the rest of the program. The *Interface* is a triple which consists of

1. ProcName: the unique name of the program, procedure, or function,
2. FormalParams: a sequence of the unique names of formal parameters (we add a return value parameter for functions),
3. Globals: the set of global variables referenced or set in the body of the procedure.

The following examples contain segments of Pascal code and the *Interface* component of the representation.

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: $(q, \langle a, b, c, d\rangle, \{v_1, v_2, ...v_n\})$
   where $v_1, v_2, ..., v_n$ are the unique names of all non-local (to procedure q) 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 side effects.

UFS Component

The *UFS* component of the representation models the intra-procedural structure of the program. 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 $Nodes$. The construction of a *UFS* from a Pascal procedure or function is described in terms of a top-down syntax directed parser. Semantic routines are embedded in the grammar and executed when they are reached during a parse. These semantic routines build the *UFS* – nodes are added to the $Nodes$ set, edges to the $Edges$ set, definitions to the sequence of *DefinitionType*, etc. In this section we describe the *UFS* construction for Pascal **if, goto**, and labelled statements. A complete description (in pseudocode) is in Appendix A.

GLOBALS
ADDNODE

IF
GOTO
LABELS

Nodes in the UFS

Each basic block consists of a possible empty sequence of assignment statements or procedure calls followed by a possibly empty control flow predicate. The structure of a basic block is represented in the *StandardRep* as a sequence of definitions, a predicate, and a set of operator counts.

Variables and the Definition Sequence

In Pascal a variable is defined either via an assignment or a procedure invocation. The mapping from Pascal statements to *StandardRep* definitions is described for statements defining and referencing each type of Pascal variable.

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

Pascal variables of structured types usually cannot map directly to StandardRep VarId's because a VarId is essentially of a simple type. We describe the mapping of Pascal structured variables to Intrep VarId's.

Array variables: At compile time, the actual cell that is defined by an assignment *A[i] := Z* cannot be determined. In terms of the *StandardRep* an 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 very dependent on its last state. Therefore the array itself is referenced. The above array assignment maps to a *StandardRep SimpleDefinition* of the form $(A, \langle A, i, Z\rangle)$.

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 can map records with fields that are arrays. For example, *A[i].b[j] := 7* maps to the *SimpleDefinition* $(A, \langle A, i, j, 7 \rangle)$.

Pointer based objects: In Pascal, pointers can only reference objects of a specified type that are allocated at run time. Pointer values are either *null* 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 were 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:

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

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

- *new(x)*
  In a strict sense, the *new* statement is represented as a *ProceduralDefinition* of the form $(new, \langle x \rangle)$. However, instead of including a *UFS* of the *new* procedure and other "primitive" procedures and functions in a standard library, we provide *SimpleDefinition*'s to represent these primitives. The *SimpleDefinition* sequence representing the *new(x)* command is $(x, \langle\rangle), (C, \langle C, x \rangle)$, where C represents the collection of objects that x may reference.
- $x\uparrow.v := 7$
  This statement maps to a *SimpleDefinition* $(C, \langle C, x, 7 \rangle)$. We are using the type C to represent the collection of objects that the pointer may be referencing.

- *x := x↑.next*
  This is another SimpleDefinition $(x, \langle C, x\rangle)$.

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, *writeln(F, a, b)* is represented as the *ProceduralDefinition* $(write, \langle F, F\uparrow, a, b\rangle)$ or the sequence of *SimpleDefinition*'s $(F, \langle F, a, b\rangle), (F\uparrow, \langle b\rangle)$ The often implicit textfile program parameter *output* and the buffer variable *output* ↑ will be explicitly included as a parameter in the representation. 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.

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. Therefor, the *Start* node in the *UFS* of *procedure Q(a: integer, var b: integer)* has a *SimpleDefinition* $(a', \langle \angle a\rangle)$. All references to a in the program are represented by a'.

By adding appropriate definitions to the *Start* and *Terminal* nodes we can represent additional parameter passing mechanisms including call-by-value/return.

Operator counts

The *HalsteadInfoType* component of each node in a *UFS* is necessary to calculate the software science measures [7]. The determination of which tokens and groups of tokens constitute operators or operands is made according to the criteria of [5]. Briefly, control constructs such as *while... do...* or *case... of... end* are treated as single operators, and their counts are associated with the block where the first token appears.

MORE ON OPERANDS

Primitive 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. Instead we define a set of *SimpleDefinition* sequences that represent most of these primitive program units.

Predefined procedures.

rewrite(F) : $(F, \langle\rangle)$

put(F) : $(F, \langle F, F\uparrow\rangle)$

reset(F) : $(F\uparrow, \langle F\rangle)$

get(F) : $(F\uparrow, \langle F\rangle)$

read(F, $V_1, V_2, \ldots, V_n$) : equivalent to *read(F, V_1); read(F, V_2); ...; read(F, V_n)*

read(F, V) : $(V, \langle F\uparrow\rangle), (F\uparrow, \langle F\rangle)$

write(F, $E_1, E_2, \ldots, E_n$) : $(F, \langle F, E_1, E_2, \ldots, E_n\rangle), (F\uparrow, \langle E_n\rangle)$

write(F, E) : $(F\uparrow, \langle E\rangle), (F, \langle F, F\uparrow\rangle)$

pack(A, B, C) : $(C, \langle A, B\rangle)$

unpack(A, B, C) : $(B, \langle A, C\rangle)$

For the following pointer primitives, T is an implicit variable representing the collection of objects (the type) that the pointer P may address:

new(P) : $(P, \langle\rangle), (T, \langle T, P\rangle)$

new(P , $C_1, C_2, \ldots, C_n$) : $(P, \langle\rangle), (T, \langle T, P, C_1, C_2, \ldots, C_n\rangle)$

dispose(P) : $(T, \langle T, P\rangle), (P, \langle nil\rangle)$

dispose(P $C_1, C_2, \ldots, C_n$) : $(T, \langle T, P, C_1, C_2, \ldots, C_n\rangle), (P, \langle nil\rangle)$

Predefined functions. The functions that are predefined have one formal parameter and return a value without side effects. These functions include *abs(X), sqr(X), sin(X), cos(X), exp(X), ln(X), sqrt(X), arctan(X), odd(X), eof(X), eoln(X), trun(X), round(X), ord(X), chr(X), succ(X),* and *pred(X)*. Instead of using a *FunctionUse* to represent the invocation of these functions and a complete *UnitRepType* to to represent the function, we can consider the use of a predifined 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:

Y := abs(X) : $(Y, \langle X\rangle)$

Y := abs(X - Y) : $(Y, \langle X, Y\rangle)$

The other predefined Pascal functions are represented in a similar manner.

```
procedure Shellsort (var A: array[1..n] of integer );
var
    i, j, incr: integer;
begin
    incr := n div 2;
    while incr > 0 do begin
        for i := incr + 1 to n do begin
            j := i − incr;
            while j > 0 do
                if A[j] > A[j + incr] then begin
                    swap(A[j], A[j + incr]);
                    j := j − incr
                end
                else
                    j := 0 (* break *)
        end;
        incr := incr div 2
    end
end; (* Shellsort *)
```

Figure 1: Pascal Code for *Shellsort*

Example IntRep

We now provide a concrete example of the *UnitRepType* for a particular procedure, *Shellsort*[1]. Again, for reasons of clarity, we have not encoded identifiers, constants, or operators. Figure 1 contains source code for *Shellsort,* and the breakdown of the procedure into basic blocks is shown in Figure 2. The control flowgraph and a textual representation of the corresponding *UnitRepType* are given in Figure 3. Note that the *UnitRepType* is intended as a basis for automated tools and measures, and is not intended for human consumption.

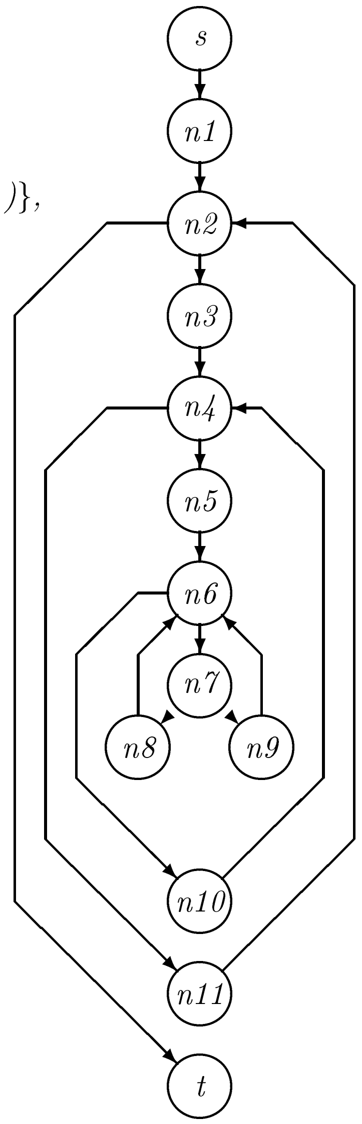
Basic block n1:	$incr := n$ **div** 2;
Basic block n2:	**while** $incr > 0$ **do**
Basic block n3:	**for** $i := incr + 1$
Basic block n4:	**to** n **do**
Basic block n5:	$j := i - incr$;
Basic block n6:	**while** $j > 0$ **do**
Basic block n7:	**if** $A[j] > A[j + incr]$ **then**
Basic block n8:	$swap(A[j], A[j + incr])$; $j := j - incr$
Basic block n9:	**else** $j := 0$
Basic block n10:	; (* representing the increment in the **for** loop *)
Basic block n11:	$incr := incr$ **div** 2

Figure 2: Basic Blocks for *Shellsort*

Implementation of A Pascal to Representation Translator

In this paper, we describe the translation process from the perspective of a top-down parser. A top-down description of the translation process is comparatively easy to understand. However, automated compiler generator tools such as YACC and LEX employ a bottom-up translation technique. Such a translator for Pascal has been implemented using YACC and LEX[**?**]. This translator will produce a Standard Representation from arbitrary Pascal programs.

)},



```
((Shellsort,⟨A⟩,{n}),
 ({(s,⟨⟩,⟨⟩,∅),
   (n1,⟨(incr,⟨n,2⟩)⟩,⟨⟩,
       ({(:=,1),(div,1),(;,1),(begin...end,1)},
        {(n,1),(incr,1),(1,1) }) ),
   (n2,⟨⟩,⟨incr,0⟩,
       ({(while...do,1),(>,1)},{(incr,1),(1,1)} )),
   (n3,⟨(i,⟨incr,1⟩)⟩,⟨⟩,
       ({(+,1),(begin...end,1),
           (for...to...do,1)},

           {(incr,1),(i,1),(1,1)}) )
   (n4,⟨⟩,⟨i,n⟩, (∅,{(n,1)}) ),
   (n5,⟨(j,⟨i,incr⟩)⟩,⟨⟩,
       ({(:=,1),(-,1),(;,1),(begin...end,1)},
        {(i,1),(incr,1),(j,1)}) ),
   (n6,⟨⟩,⟨j,0⟩, ({(while...do,1),(>,1)},{(j,1)}) ),
   (n7,⟨⟩,⟨A,j,A,j,incr⟩,
       ({(if...then,1),([],2),(+,1),(>,1)},
        {(j,2),(a,2),(incr,1)}) ),
   (n8,⟨(swap,⟨⟨A,j⟩, ⟨A,j,incr⟩⟩),(j,⟨j,incr⟩)⟩, ⟨⟩,
       ({(swap,1),((),1),([],2),(,,1),(+,1),
        (;,1),(:=,1),(-,1),(begin...end,1)}{(j,2),(a,2),(incr,1)}) ),
   (n9,⟨(j,⟨0⟩)⟩,⟨⟩,({(else,1), (:=,1)},{(j,1),(0,1)}) ),
   (n10,⟨(i,⟨i⟩)⟩,⟨⟩,(∅,∅) ),
   (n11,⟨(incr,⟨incr,2⟩)⟩,⟨⟩,
       ({(:=,1),(div,1),(;,1)},
        {(incr,2),(2,1)}) ),
   (t,⟨⟩,⟨⟩,∅)},
  {(s,n1),(n1,n2),(n2,n3),(n2,t),(n3,n4),
       (n4,n5),(n4,n11),(n5,n6),(n6,n7),
       (n6,n10),(n7,n8),(n7,n9),(n8,n6),
       (n9,n6),(n10,n4),(n11,n2)},
  s,
  t))
```

(a) Corresponding control flowgraph *(b) Textual representation of UnitRepType*

Figure 3: *UnitRepType* for *Shellsort*

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, Reading, MA, 1983.

[2] 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, Dept. of Computer Science, Iowa State University, Ames, Iowa, 1986.

[3] J. Bergeretti and B. A. Carre. Information-flow and data-flow analysis of while-programs. *ACM Transactions on Programming Languages and Systems*, 7(1):37–61, January 1985.

[4] J. M. Bieman and W. R. Edwards. Experimental evaluation of the data dependency graph for use in measuring software clarity. *Proc. 18th Hawaii International Conference on Systems Science*, 18:271–276, 1985.

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

[6] B. Curtis. Conceptual issues in software metrics. In *Proceedings of the Nineteenth Hawaii International Conference on Systems Sciences*, January 1986.

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

[8] W. A. Harrison and C. Cook. A method of sharing industrial software complexity data. *SIGPLAN Notices*, 20(2):42–51, February 1985.

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

[10] S. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE Trans. Software Engineering*, SE-7(5):510–518, 1981.

[11] J. W. Howatt and A. L. Baker. Definition and design of a tool for program control structure measures. *Proc. COMPSAC85*, 214–220, 1985.

[12] H. A. Jensen and K. Vairavan. An experimental study fo software metrics for real-time software. *IEEE Trans. Software Engineering*, SE-11(2):231–234, 1985.

[13] T. J. McCabe. A complexity measure. *IEEE Trans. Software Engineering*, SE-2(4):308–320, 1976.

[14] T. J. McCabe. *A Testing Methodology Using the McCabe Complexity Metric*. IEEE Computer Society Press, Silver Spring, MD, 1982.

[15] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. Software Engineering*, SE-11(4):367–375, 1985.

[16] M. R. Woodward, M. A. Hennell, and D. Hedley. A measure of control flow complexity in program text. *IEEE Trans. Software Engineering*, SE-5(1):45–50, 1979.