Analysis of TEMPO
Using the Denotational Semantics Approach

by

Wan-Hong Cheng

B. S., National Chiao-Tung University, Taiwan, R. O. C., 1980

A MASTER'S REPORT

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1982

Approved by:

Major Professor
David A. Schmidt

ACKNOWLEDGEMENTS

I wish to express my sincere appreciation to my major professor, Dr. David A. Schmidt, for his time, guidance and direction in completion of this report. Also thanks are due to my other committee members, Dr. Rodney M. Bates and Dr. Paul S. Fisher.

CONTENTS

1. Introduction	1
1.1 Main Purposes Of Formal Semantics	1
1.2 Different Kinds Of Formal Semantics	2
1.3 Denotational Semantics	3
2. TEMPO-- A Language With Late Binding Times	9
2.1 General Description	9
2.2 Important Features Of TEMPO	10
3. Formal Syntactic Definition Of TEMPO	20
3.1 Concrete Syntax	20
3.2 Abstract Syntax	22
4. Formal Semantic Definition Of TEMPO	25
4.1 Semantic Domains	25
4.2 Semantic Functions	27
5. Analysis Of Implementation Of TEMPO	40
5.1 Original Implementation	40
5.2 Realization Through Semantic Functions	43
5.3 Improvements In Efficiency	53
6. Conclusions	56
Bibliography	58

1. INTRODUCTION

1.1 Main Purposes Of Formal Semantics

On many occasions, we tend to explain and describe programming languages intuitively or based on some practical computer system, which is used to give a detailed, low-level description of the languages' operations. Sometimes the effort is satisfactorily rewarded. But as the complexity of a language increases, we need more abstract, complete, comprehensible and machine-independent specification methods with rigorous theory for specifying programming language semantics.

With a formal semantic definition, we can not only describe the semantics of a programming language but also verify various properties of the language. In the past, the compiler has been regarded as the final definition of a programming language. But now, the formal semantics can serve as a precise description for implementers of the language to construct a "correct" compiler.

For the programmers, the formal semantics can give a sufficiently rigorous statement for determining the behaviour of the programs they write. Language designers can also use simple formal descriptions to design better languages. Analysis and modification of a language is more easily done by studying and

modifying a formal definition than by studying and modifying a physical implementation.

1.2 Different Kinds Of Formal Semantics

There are several different kinds of formal semantics, each with different specific goals. The following are the three most popular kinds of formal semantics:

AXIOMATIC SEMANTICS [Hoare]

The language is defined using a formal system, as in mathematical logic. Rules are given for constructing well formed programs and claims about the programs' logical properties. Axioms and rules of inference are supplied for constructing theorems of true logical properties of a program. These logical properties typically describe value ranges of program variables. The meaning of the program can be viewed as a transformer which transforms the initial logical formula describing the relationship between the initial values of the variables of the program state to a final logical formula.

DENOTATIONAL SEMANTICS [Stoy]

This method is model-theoretic in nature. Given appropriate notions of semantic domain and continuous function, the program's meaning is a continuous function mapping initial arguments from a

program state domain to final program states. How the function is achieved (through what computational process) is not considered in this approach (beyond the fact that the function is actually computable).

OPERATIONAL SEMANTICS [Wegner]

We can determine the meaning of a program using an abstract machine which "interprets" the program text accompanied by the initial program state to generate a state-transition sequence leading to the final program state (termination). The program's meaning is the sequence of program states describing the transition from the initial relationship between the variables and their corresponding values to the final relationship. The state transition sequence is computed by the language's interpreter, an abstracted computer. Note that the meaning of a program is only determined in tandem with some initial state.

1.3 Denotational Semantics

In this report, we will use the Scott-Strachey approach as in [Stoy]. The reason why we choose denotational semantics as the approach is that we want to give an abstract description about the language so that we can understand the features of the language and the difficulties we are going to have in implementation without getting into very low-level simulation of the language (as

in operational semantics). The principle of denotational semantics is to provide simple semantic objects as the meaning of simple syntactic forms directly and then express the meaning of compound syntactic objects through the composition of the meanings of their immediate constituents.

The denotational semantic description of a language will contain the syntax and semantic specifications. The syntax specification is the abstracted form of the language's grammar (we are considering only context free grammars). The semantic specification will contain the semantic domains and continuous functions which are used to map the syntactic objects to their abstract meanings.

SYNTAX SPECIFICATION

Since there are infinitely many grammars which can recognize/generate a specific language, we use abstract syntax [McCarthy2] as the definition method for specifying the syntax of a language. Abstract syntax is a BNF-like notation with the syntactic constructs defined as hierarchical objects, so we can identify the syntactic objects hierarchically by using the definition.

The idea of abstract syntax is to allow the designer to concentrate on analyzing the characteristics of the language without worrying about the actual recognition problems of the language. Ordinarily, the parser will do the recognition process

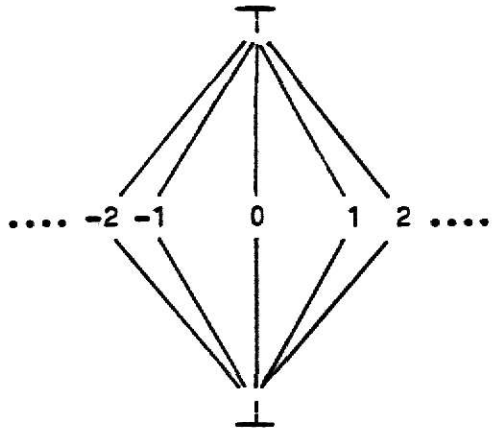
to determine the hierarchical structure of the program, i.e., to identify the abstract syntax of the program. Meanings are assigned to the subparts of the abstract syntax tree; the meanings are combined to give the meaning of the entire program.

The abstract syntax might be viewed as an ambiguous context free grammar. Each rule will specify a logical group of the language, and we can determine the phrase structure of the language easily.

SEMANTIC SPECIFICATION

Roughly, semantic specification can be divided into two parts: the semantic domains and the functions mapping abstract syntax pieces to members of the domains. A domain may be first order (primitive) or higher order (a set of functions).

A semantic domain is a partially ordered set [Gratzer] [Stoy] and is in fact a form of lattice known as a continuous lattice [Stoy]. The lattice structure is used for mathematically handling recursively defined functions. For example, the lattice for the Integer domain can be diagrammed as



where "$\top$" ("top") an overdefined value, indicating the attribute of an object has been overdefined (such as a type mismatched arithmetic expression), and "$\bot$" ("bottom") is an undefined value, indicating insufficient information about an object. Domains can be of the following four basic forms:

<1> Primitive, e.g., Integer, Boolean, Character

<2> Sum (disjoint union), e.g., Integer + Character

<3> Cartesian product, e.g., Integer X Boolean

<4> Function space, e.g., Integer -> Boolean

Domains can be recursively defined (e.g., as in LISP : LISTS = ATOMS + (LIST X LIST)). All these domains can be realized as continuous lattices [Scott1] [Scott2].

The meanings of program constructs are (continuous) functions mapping arguments to answers in the semantic domains. The functions are represented using a lambda-calculus-like notation known as LAMBDA [Church] [Scott2] [Stoy]. Normally, the *meaning of a program* is defined to be a mapping from an initial program state to a final one. The following is an example using denotational semantics to describe a simple language:

ABSTRACT SYNTAX

P ∈ Pgm ::= begin S end

S1, S2 ∈ Stmt ::= I := E | if E then S1 else S2 | S1 ; S2

E ∈ Exp ::= V | I

V ∈ Boolean ::= true | false

I ∈ Identifiers

SEMANTIC DOMAINS

$\sigma \in$ Store = Identifier -> Truth_value

Truth_value = {t, f}

SEMANTIC FUNCTIONS

$\mathcal{M}$: Pgm -> (Store -> Store)

$\mathcal{M}$ ⟦ begin S end ⟧ = $\lambda \sigma .\ \mathcal{C}$⟦ S ⟧$\sigma$

$\mathcal{C}$: Stmt -> (Store -> Store)

$\mathcal{C}$ ⟦ I := E ⟧ = $\lambda \sigma .\ \sigma$[$\mathcal{E}$⟦ E ⟧(σ)/⟦ I ⟧]

$\mathcal{C}$ ⟦ if E then S1 else S2 ⟧ = $\lambda \sigma .\ \mathcal{E}$⟦ E ⟧$\sigma$ -> $\mathcal{C}$⟦ S1 ⟧σ, $\mathcal{C}$⟦ S2 ⟧σ

$\mathcal{C}$ ⟦ S1 ; S2 ⟧ = $\lambda \sigma .\ \mathcal{C}$⟦ S2 ⟧($\mathcal{C}$⟦ S1 ⟧($\sigma$))

$\mathcal{E}$: Exp -> (Store -> Truth_value)

$\mathcal{E}$ ⟦ V ⟧ = $\lambda \sigma .$ ⟦V⟧ = ⟦true⟧ -> t , f

$\mathcal{E}$ ⟦ I ⟧ = $\lambda \sigma .\ \sigma$(⟦ I ⟧)

A meaning function is defined for each abstract syntax form. For example, the meaning function for Stmt, $\mathcal{C}$, maps a Stmt form to its meaning, which is a function mapping from Store to Store. The set of Stores is a semantic domain-- a store is a function from Identifiers to Truth_values, where Identifier is a "syntax domain" and Truth_value is a primitive semantic domain.

There are some conventions used in describing the semantic functions. The brackets ⟦ ⟧ are used to enclose syntax forms, so that they are not confused with semantic forms. Greek letters, like σ, are used to stand for representatives of a particular semantic domain. Here σ always stands for an arbitrary member of Store. The following notation is commonly used to define functions:

(1) $\lambda\,\sigma.\,\alpha$ stands for a function which needs an argument of type Store to provide answer α.

(2) f(a) stands for function application. Note that when we delete parentheses, we assume "left associativity". For example, ((f(a))b) is abbreviated to fab. But in defining semantic domains, when deleting parentheses, we assume "right associativity". For example: A -> (B -> (C -> D)) is abbreviated to A -> B -> C -> D. Also, a domain such as A -> B -> C is lattice isomorphic (structually the same) as (A X B) -> C.

(3) a->b,c stands for a conditional expression: b if a = true, c if a = false.

(4) $\sigma[\alpha/\beta]$ stands for $(\lambda\, j.\ j=\beta \rightarrow \alpha\ ,\ \sigma(j))$, an update of function σ at argument β by α. The details can be found in [Stoy].

2. TEMPO-- A LANGUAGE WITH LATE BINDING TIMES

2.1 General Description

TEMPO is a programming language defined by Jones and Muchnick [Jones] as a teaching aid. This language emphasizes very late binding times so that it can provide many programmer conveniences such as omitting type declaration for user-defined identifiers (the types are determined during runtime) and providing dynamic program text generation (we can create a new procedure while the program is running and then execute the new procedure).

A binding is an association between a variable and its attributes. Binding time is the time period that the correspondence is fixed. Some bindings, such as identifier-data_type, are fixed very early in PASCAL-like languages, while identifier-storable_value bindings are made during runtime. Generally, the programmers of TEMPO are rid of the burden of specifying many attributes before using identifiers and so have more flexibility than with strongly-typed programming languages like PASCAL. This means that TEMPO has uniformly late (runtime) bindings. Work such as storage allocation and deallocation, static semantic (e.g., context sensitive syntax) processing, and sometimes even the parsing, is done during

runtime.

A compiler for TEMPO can not really do much in the area of "semantic processing". Also, there can be little optimization at compiling time since there is insufficient information to aid the process. Like other late-binding languages such as SNOBOL and LISP, TEMPO is best evaluated using an interpreter.

2.2 Important Features Of TEMPO

The control structures of TEMPO are similar to conventional block-structured programming languages. They include block, if-then and while-do statements (we will not study the goto statement). It has common arithmetic operations, logical operations and concatenation for strings. It also has the ability to define procedures and do input-output operations. Assignment is included, too.

The basic data types of TEMPO are integers, character strings and structured objects. There is also a special undefined value. Each variable can have any one of these data types during execution. The "structured" object can be best viewed as a tree with integer indices pointing to its levels of subtrees or leaves. Here is a sample TEMPO program to insert an element (integer) into a sorted (ascending) array:

```
(* COMMENT : ARRAY-- A SORTED ARRAY OF INTEGERS, *)
(* ELEMENT-- THE INTEGER ELEMENT TO BE INSERTED, *)
(* LENGTH-- LENGTH OF THE ARRAY. *)
        INSERT := 'parameters ARRAY, LENGTH, ELEMENT;
          begin scope INDEX, J;
            INDEX := 1;
            LENGTH := LENGTH + 1;
            ARRAY[LENGTH] := ELEMENT;
            while ARRAY[INDEX] < ELEMENT do INDEX := INDEX + 1;
            if INDEX < LENGTH then
              begin
                J := LENGTH - 1;
                while J ≥ INDEX do
                  begin
                    ARRAY[J + 1] := ARRAY[J];
                    J := J - 1;
                  end
                ARRAY[INDEX] := ELEMENT;
              end
          end';
```

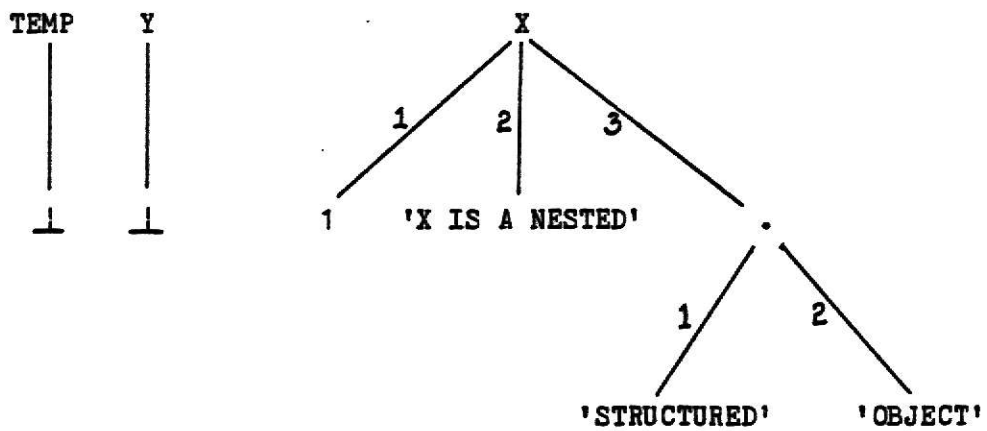
Besides those structures described above, some implicit features of TEMPO deserve attention here.

<1> DYNAMIC DATA STRUCTURES

Since TEMPO does not require type declaration for user-defined identifiers, the contents of identifiers may change in shape or expand/shrink in size at runtime. This will add much burden to the system since there is a lot of overhead for type checking, allocation and deallocation of storage dynamically. The following program is an example:

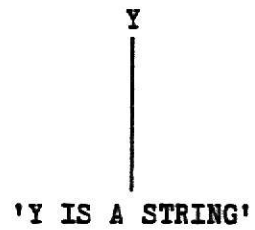
```
begin scope X, Y, TEMP;
      X := <1,'X IS A NESTED', <'STRUCTURED', 'OBJECT'>>;
      Y := 'Y IS A STRING';
      X[2][1] := Y;
      TEMP[2] := X;
      X := 0;
end
```

After the first assignment statement, the identifier-value bindings are:

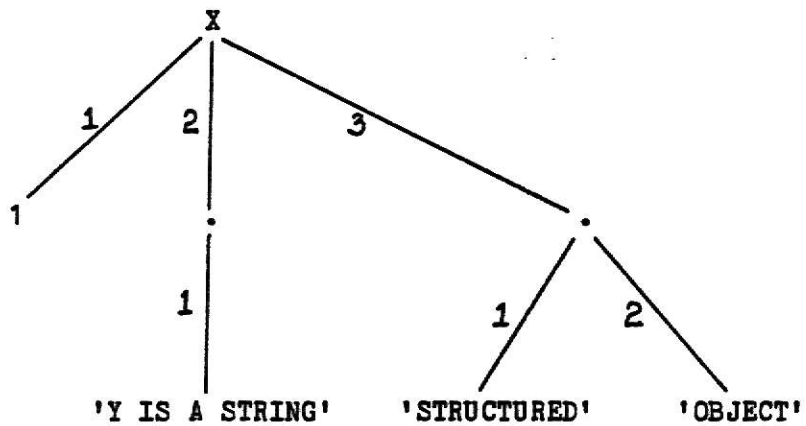


All declared variables are initialized with the undefined value ⊥.

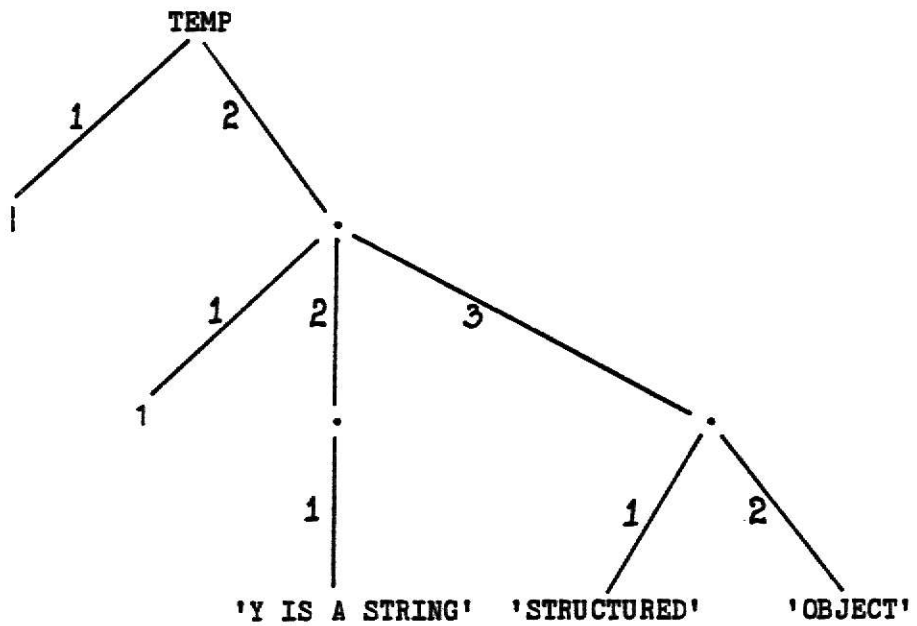
After the second assignment statement, the identifier-value binding for Y becomes:



After the third assignment statement, the identifier-value binding for X becomes:



After the fourth assignment statement, the identifier-value binding for TEMP becomes:



Note the creation of a dummy element for TEMP[1].

After the fifth assignment statement, the identifier-value binding for X becomes:

From the changes of identifier-value binding, we can see that TEMPO supports dynamic data structures, and the runtime system of

TEMPO has to handle the storage management during program execution.

<2> SYMBOLIC INDIRECT ADDRESSING

This feature can provide the programmer the facility to address the runtime value of the variable symbolically while generating the program text. It is especially useful in the execution of the dynamically generated procedures. The following is a program showing this feature:

```
begin scope P, X, Y;
      P := 'X';
      call (P catenate ' := "Y";');
      P := 'Y';
      call (P catenate ' := "Y";');
end
```

In this example, we first assign a character 'Y' to X by executing the called statement 'X := "Y";' and then assign 'Y' to Y by calling 'Y := "Y";'. The call statements are identical.

<3> DYNAMIC GENERATION OF PROGRAM TEXT

In TEMPO, we can create new program text during execution time by building up a character string and then calling it. The character string will be executed as long as it is a syntactically (and semantically) correct procedure. It was seen in the example

above. This will require some extra work from the system because we have to do the parsing for the newly generated program text at runtime to make sure it is syntactically correct. Under such circumstance, the maintenance of the storage allocation/deallocation is also necessary.

<4> PROCEDURE PARAMETER SUBSTITUTION

Call by text parameter passing is used in TEMPO for binding actual parameters to the formal parameters of a called procedure. For instance, in the following example:

```
begin scope P, A, B, C;
      P := 'parameters X, Y; X := Y + 2;';
      A := 1;
      call  P(B, A);
      call  P(C, A+B);
      output := <A, B, C>;
end
```

the first call causes X to be replaced by B, Y replaced by A and results in the program text-- 'B := A + 2;'. The second call causes X to be replaced by C, Y replaced by A+B and results in the program text-- 'C := A+B + 2;'. This feature makes the system very inefficient since it has to textually substitute the parameters in the procedure to produce new program text first, parse it, and then execute the new string. Consider also this

call-- 'call P(A+1, B+1)'. It is an erroneous statement after the text substitution of parameters.

In summarizing the above, we can see that these conveniences present many difficulties in implementation and will likely result in a very inefficient system (a great deal of work must be done during program execution). If we compare TEMPO with FORTRAN, which is a highly efficient programming language but with more restrictions on the user, we can discover the reasons for TEMPO being so inefficient. The comparison can be divided into several aspects as following:

(1) STORAGE ALLOCATION

TEMPO: Since the shape and size of the value of the variable may change in the assignment statement, it requires a heap-type storage scheme to support the feature. Some operating system utility programs like garbage collection will be necessary. It will need much more execution time to manage storage.

FORTRAN: The shape and size of the value of variables are not allowed to be changed during the program execution and so the storage needed for the variables can be determined at compile time. The locations of the variables are fixed after the program is loaded in the memory. There will be no extra time spent on storage management.

(2) MAINTENANCE OF THE IDENTIFIERS

TEMPO: Due to the variablity of data type and other attributes of identifiers and program text, the system must keep a runtime symbol table to maintain identifiers during execution. Frequent sorting and searching of such table may be required to locate new variables or access variables.

FORTRAN: Such a table is obviously unnecessary because the attributes of the variables will remain unchanged during the execution; the compiler's symbol table handles this stage of processing.

(3) PROGRAM TEXT GENERATION

TEMPO: The programmer can read in or create new program text during the program execution and cause the new program text to be executed. This is extremely inefficient since the program has to be interpreted or use a runtime parser.

FORTRAN: The programmer is not allowed to generate new program text so that there is no such burden on FORTRAN as on TEMPO.

(4) DATA TYPES

TEMPO: The data type of the value of a variable can only be determined during the execution of the program. Thus no type checking can be performed when the program was

compiled. Runtime type checking is required for each execution of some statements such as arithmetic operations (must verify both operands are integers) or concatenation for strings (operands should be strings). This tremendously increases the execution time (e.g., an arithmetic operation in a loop which will be executed 10,000 times will require that the types of the operands to the operation must also be checked 10,000 times!).

FORTRAN: The data type of the value of a variable is not changable and no such runtime type checking is needed.

(5) PROCEDURE PARAMETER PASSING

TEMPO: Text substitution of the actual argument for each occurrence of a formal parameter is performed. This implies that some syntax checking routines must be present during runtime to check the syntax legality of the substituted text. Runtime checking is also needed for compatability between the arguments and parameters in number.

FORTRAN: Efficient parameter passing methods-- call by value and/or call by reference are used. These are easier to implement and are much more efficient than call by text.

3. FORMAL SYNTACTIC DEFINITION OF TEMPO

3.1 Concrete Syntax

The original grammar defined by Jones and Muchnick has been modified in this report. For example, the goto statement is taken out from the grammar for simplicity. Some functions on strings defined in the original version like "length", "substring" are also omitted because they are not relevant to that which we wish to discuss. We use an extended BNF description for defining the grammar. A BNF description consists of a starting symbol and a set of rewriting rules (productions) composed of nonterminals, terminals and BNF operators. The following is the revised version of the grammar:

note: nonterminals -- represented by all upper-case letters
terminals -- represented by all lower-case letters
BNF operators -> : may be replaced by
| : or
[...] : optional item
{...} : grouping
+ (superscript) : one or more occurrences
* (superscript) : zero or more occurrences
List : list of items with separators

TEMPO'S GRAMMAR

S1. PROGRAM -> BLOCK

S2. BLOCK -> begin [SCOPE] STATEMENT^{+} end

S2.1. SCOPE -> scope IDENT, List;

S3. STATEMENT -> {ASSIGN | IF | CALL | BLOCK | WHILE}

S3.1. ASSIGN -> {VAR | output} := {EXP | input} ;

S3.2. IF -> if LOGEXP then STATEMENT

S3.2.1. LOGEXP -> EXP {< | ≤ | = | ≠ | > | ≥} EXP

S3.3. CALL -> call STREXP [ARGS] ;

S3.3.1. ARGS -> (EXP , List)

S3.4. WHILE -> while LOGEXP do STATEMENT^{+}

S4. EXP -> ATOMEXP | <EXP , List>

S4.1. ATOMEXP -> ARITHEXP | STREXP

S4.2. ARITHEXP -> TERM { + | - } List

S4.2.1. TERM -> FACTOR { * | / } List

S4.2.2. FACTOR -> VAR | NUMBER | (ARITHEXP)

S4.3. STREXP -> STRFACTOR catenate List

S4.3.1. STRFACTOR -> VAR | STRING | (STREXP)

S4.3.2. STRING -> ' { " | NONQUOTE }* '

S4.3.3. NONQUOTE -> LETTER | DIGIT | KEYWORD | . | , | +
| - | * | / | < | ≤ | > | ≥ | = | ≠ |
(|) | [|] | < | >

S4.3.4. KEYWORD -> begin | call | catenate | do | end |
if | input | length | output |
parameters | scope | then | while

S5. VAR -> IDENT SUBSCRIPT*

S5.1. SUBSCRIPT -> [ARITHEXP]

S5.2. IDENT -> LETTER { LETTER | DIGIT }*

S5.3. LETTER -> A | B | ... | Z

S6. NUMBER -> DIGIT^{+}

S6.1. DIGIT -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

S7. PROCEDURE -> [FORMALPARS] STATEMENT

S7.1. FORMALPARS -> parameters IDENT , List ;

S8. DATA -> CONSTANT , List

S8.1. CONSTANT -> NUMBER | STRING | < DATA >

3.2 Abstract Syntax

We can form the abstract syntax from the original grammar directly (intuitively). It is listed as following:

```
P ∈ Pgm                          ::= B

B ∈ Block                        ::= begin S end | begin D ; S end

D ∈ Scope                        ::= scope IL

IL ∈ Idenlist                    ::= I | I , IL

S1, S2 ∈ Stmt                    ::= S1 ; S2 | if LE then S |
                                     while LE do S | call SE (AL) |
                                     call SE | LHS := RHS

LHS ∈ Left_hand_side             ::= V | output

RHS ∈ Right_hand_side            ::= E | input

AL ∈ Argumentlist                ::= E | E , AL

E ∈ Expression                   ::= AE | SE | <AL>

AE1, AE2 ∈ Arith_expression      ::= AE1 Ω AE2 | V | N | (AE)

SE1, SE2 ∈ String_expression     ::= SE1 catenate SE2 | V | ST | (SE)

LE ∈ Logic_expression            ::= E1 ϕ E2

Ω ∈ Arith_ops                    ::= + | - | * | /

ϕ ∈ Logic_ops                    ::= < | ≤ | = | ≠ | > | ≥

V ∈ Variables                    ::= I | I SUB

SUB ∈ Subscripts                 ::= [AE] | [AE] SUB

PRO ∈ Procedure                  ::= S | parameters IL ; S

DA ∈ Data                        ::= C | C , DA

C ∈ Constant                     ::= N | ST | <DA>

I ∈ Identifiers

N ∈ Integers

ST ∈ Char_strings
```

Since any string can be called as a procedure in TEMPO, category Procedure is not used in any production. Parsing of procedures always occurs at runtime. The "DA" is to specify the data format of the input-output operations.

4. FORMAL SEMANTIC DEFINITION OF TEMPO

4.1 Semantic Domains

We begin by stating the sets of values used in defining the meanings of TEMPO programs. These are the semantic domains:

Expressible_Values = Booleans + Storable_Values

Denotable_Values = Locations

$\nu \in$ Storable_Values = Integers + Char_Strings
+ Structured_Objects

$\delta \in$ Locations = { input_buffer_locn_no, output_buffer_locn_no,
L0, L1, L2, ... }

Structured_Objects = Integers -> Storable_Values

$\rho \in$ Env = Identifers -> Denotable_Values

$\sigma \in$ Store = Locations -> Storable_Values

It is assumed that Booleans, Integers, and Char_strings are primitive domains. Again, note that in the functions to follow, the Greek letters (such as σ) stand for arbitrary members of the respective domains.

The Expressible_Values domain contains the values which expressions can evaluate to. For example, a test like ($4 \geq X$) in a if statement has a boolean value although booleans are not explicitly assigned to identifiers.

Denotable_Values are the objects identifiers can stand for.

In this language, the identifiers can only stand for locations. If we would introduce constant declarations or constant procedure labels, then the identifiers can have more options, and the Denotable_Values might contain Integers, (Store -> Store) functions, etc.

Storable_Values are the values that can be stored in one location. Our definition will need unconventional and powerful memory cells which can hold whatever integer, character string or structured object is assigned to an identifier. This viewpoint is similar to that taken when understanding SNOBOL. In reality, the computer system uses heap storage to model high level cells like those needed. Because this is too implementation-oriented, we will not get deeply into it and will only have the above assumption.

Locations in TEMPO will have three "kinds"-- input_buffer_locn_no, output_buffer_locn_no and ordinary memory cell locations-- {L0, L1, L2, ...}. The input_buffer_locn_no and output_buffer_locn_no are special locations for input-output operations' use. Their cells hold input and output buffers. Input and output are handled differently from ordinary operations. The input_buffer and output_buffer models will be explained more clearly later.

The Env (environment) domain contains functions mapping identifiers to their corresponding denotable_values-- locations. An environment can be viewed as the symbol table of the compiler.

The Store domain contains functions mapping locations to the

storable_values associated with the locations. A store object might be viewed as the runtime memory.

The reader may wonder where the data type information for a storable object is kept. The data type "tags" are implicitly included within the disjoint sum of the Storable_Values domain. Recall the formal definition of disjoint sum(union):
$A + B = \{(1, a) \mid a \in A\} \cup \{(2, b) \mid b \in B\}$ -- type tags are attached to A's and B's objects. Similar tags exist for the Storable_Values domain. These tags can be checked upon evaluation; we will use the $\in$ symbol to do just that. Formally, for an object a:A to become a member of A+B it must be <u>injected</u> into the domain (the tag must be attached), and for a member of A+B to be used as an A-object, it must be <u>projected</u> into A (the tag must be removed). We will assume that the <u>injection and projection occur</u> "<u>automatically</u>" where needed in the semantic functions. This will reduce the notation. It is straightforward to manually insert these operations (by checking the domains and ranges of the functions), and the reader is welcome to do so.

4.2 Semantic Functions

We give the semantic functions for each abstract syntax group. Detailed explanation of the functions and their relationship to an implementation of TEMPO are delayed until the

next chapter. We will use the following "primitive functions" upon these data domains. These functions correspond to low level operations and so will not be formally defined.

Update_env : Identifiers X Locations X Env -> Env

This function is to update the Location attributes of the Identifier in the Environment, just like updating the symbol table in the real implementation.

Update_store : Storable_Values X Locations X Store -> Store

Update_store changes the Storable_Value contents associated with the Locations in the memory Store.

Allocate : Store -> (Locations X Store)

"Allocate" allocates a new Location from the Store when needed. Both the new Location and an updated Store result.

Deallocate : Store -> Store

This function frees the memory space used by the local variables in the block just executed.

In addition, let the form D^* stand for the set of lists whose members come from D. Formally, $D^* = <> + D + DXD + DXDXD + \ldots$. We use

head : D^* -> D

tail : D^* -> D^*

cons : D X D^* -> D^*

append : D^* X D^* -> D^*

to extract the head member of a D^* list, determine a D^* list less

its head, add a D member to a D^* list, and append two lists, respectively. An ordered n-tuple object from D^* is represented as <d1, d2, ..., dn>.

We now provide a semantic function for each abstract syntax form defined in chapter 3.

(1) $\mathcal{M}$: Pgm -> Store -> Store

$\mathcal{M}[\![B]\!] \sigma = \mathcal{B}[\![B]\!] \rho_0 \sigma$

where $\rho_0 = \lambda I. \perp_{Denotable\_Values}$

The meaning of a program is the meaning of its body. Here, σ contains the initialized input_buffer, output_buffer and ordinary memory cells. ρ_0 is an initialized environment, just like an empty symbol table in which everything is undefined. $\perp_{Denotable\_Values}$ means the undefined value-- $\perp$ in the Denotable_Values domain.

(2) $\mathcal{B}$: Block -> Env -> Store -> Store

$\mathcal{B}[\![$ begin S end $]\!] \rho \sigma = \mathcal{C}[\![S]\!] \rho \sigma$

$\mathcal{B}[\![$ begin D ; S end $]\!] \rho \sigma =$ Deallocate($\mathcal{C}[\![S]\!]^* (\mathcal{D}[\![D]\!] \rho \sigma)$)

The meaning of a block follows from the meaning of its body S following the processing of its declarations D. The * when used after a function stands for "multiple subscripting". This provides convenience in applying a function to its arguments, for example, $f^*\langle a, b\rangle = ((fa)b)$.

(3) $\mathcal{D}$: Scope -> Env -> Store -> (Env X Store)

$\mathcal{D}[\![$ scope IL $]\!]\rho\sigma = \mathcal{IL}[\![$ IL $]\!]\rho\sigma$

(4) $\mathcal{IL}$: IL -> Env -> Store -> (Env X Store)

$\mathcal{IL}[\![$ I $]\!]\rho\sigma = (\lambda\delta.\ \lambda\sigma'.\langle$Update_env(I, δ, ρ),

Update_store($\perp_{Storable\_Values}$, δ, σ') $\rangle$)

*(Allocate(σ))

$\mathcal{IL}[\![$ I, IL $]\!]\rho\sigma = \mathcal{IL}[\![$ IL $]\!]$ * ($\mathcal{IL}[\![$ I $]\!]\rho\sigma$)

Each identifier I has a new location allocated for it, and the value is initialized to undefined.

(5) $\mathcal{C}$: Stmt -> Env -> Store -> Store

$\mathcal{C}[\![$ S1 ; S2 $]\!]\rho\sigma = (\mathcal{C}[\![$ S2 $]\!]\rho \cdot \mathcal{C}[\![$ S1 $]\!]\rho$)σ

$\mathcal{C}[\![$ B $]\!]\rho\sigma = \mathcal{B}[\![$ B $]\!]\rho\sigma$

These functions decompose the composition of statements and produce the meaning of a block by using the block semantic function.

$\mathcal{C}[\![$ if LE then S $]\!]\rho\sigma$

$= \mathcal{LE}[\![$ LE $]\!]\rho\sigma$ -> $\mathcal{C}[\![$ S $]\!]\rho\sigma$, σ

$\mathcal{C}[\![$ while LE do S $]\!]\rho\sigma$

$= \mathcal{LE}[\![$LE$]\!]\rho\sigma$ -> $\mathcal{C}[\![$while LE do S$]\!]\rho$ ($\mathcal{C}[\![$S$]\!]\rho\sigma$), σ

The first function processes the selection operation. The action taken will depend on the result of evaluation of LE. The second is to process the while-do loop. Note that the function is

recursively defined. If lattice domains are used, the "least fixed point" of the definition is used to define a noncircular form of the function [Stoy].

$$\mathcal{C}[\![\ \text{call SE}\ ]\!]\rho\sigma = (\lambda\ \text{e:Procedure.}$$
$$e \in [\![\ S\ ]\!] \rightarrow \mathcal{C}(e)\ \rho\sigma, \top_{Store})$$
$$(\text{parse\_procedure}(\mathcal{E}[\![\ \text{SE}\ ]\!]\ \rho\sigma))$$

$$\mathcal{C}[\![\ \text{call SE (AL)}\ ]\!]\rho\sigma$$
$$= (\lambda\ \text{d:Char\_strings.}\ \mathcal{C}(\text{parse\_statement(d)})\rho\sigma)$$
$$((\lambda\ \text{e:Procedure.}$$
$$\text{call\_by\_text\_substitution}$$
$$(\text{(extract\_stmt\_string(e))},$$
$$\text{(extract\_formal\_parm\_list(e))},$$
$$(\mathcal{A}[\![\text{AL}]\!]))$$
$$(\text{parse\_procedure}(\mathcal{E}[\![\text{SE}]\!]\ \rho\sigma)))$$

These definitions are complex and will be explained in detail later. Basically, a call statement requires its argument SE to be parsed into a legal procedure. If any parameters AL are specified, they must be matched to the formal parameter names and textually substituted into SE. The result is evaluated as a statement.

A number of auxiliary functions have been used. They are:

parse_procedure : Char_strings -> Procedure

Parse_procedure parses a character string into abstract syntax if

the string is a syntactically correct procedure.

parse_statement : Char_string -> Statement

This function parses a character string into statement format if the string is syntactically correct.

call_by_text_substitution : Char_strings X Identifiers* X Expression* -> Char_strings

The purpose of this function is to do the physical text substitution of parameters by calling arguments. Its definition is given below.

extract_stmt_string : Procedure -> Char_strings

This is to extract the "statement"-- S from the procedure [[parameters IL; S]].

extract_formal_parm_list : Procedure -> Identifiers*

This is to extract the formal parameters IL from the procedure [[parameters IL; S]] and make them into a list.

substitute_string : Char_string X Identifier X Expression -> Char_strings

Substitute_string textually replaces the Identifier in the Character String with the Expression argument.

call_by_text_substitution : Char_strings X Identifiers* X Expression* -> Char_strings

Call_by_text_substitution is defined as follows:

```
call_by_text_substitution(c, i, e)
     = i ∈ Identifiers -> (e ∈ Expression ->
                    substitute_string(c, i, e), T_char_strings ),
                    call_by_text_substitution
                    (call_by_text_substitution
                         (c, (head(i)), (head(e))),
                         (tail(i)), (tail(e)))
```

Note that the conditions check for a mismatch in the number of formal to actual parameters.

$$\mathcal{C}[\![\text{LHS} := \text{RHS}]\!]\rho\sigma$$

$$= (\lambda\, \text{rhs\_value}:\text{Storable\_Values}.\ \lambda\sigma'.$$

$$(\lambda\, r:\text{Locations}.\lambda\, \text{subscripts}:\text{Integer}^{*}.\lambda\sigma''.$$

$$\text{Update\_store}(\sigma'', r,$$

$$(\text{subscripts} = \langle\rangle \rightarrow \text{rhs\_value},$$

$$\text{new\_value}((\sigma''(r) \in \text{Structured\_Objects}$$

$$\rightarrow \sigma''(r), \perp_{\text{Structured\_Objects}}),$$

$$\text{subscripts}, \text{rhs\_value})))$$

$$)^{*}\mathcal{L}[\![\text{LHS}]\!]\rho\sigma$$

$$)^{*}\mathcal{R}[\![\text{RHS}]\!]\rho\sigma$$

where

```
new_value : Storable_Value X Integers* X Storable_Values
                                        -> Storable_Values
```

```
new_value(object, subscripts, result)
     = subscripts = <> -> result,
                    (object ∈ Structured_Objects ->
                       (λ j:Integers.
                          j = (head(subscripts)) ->
                            new_value( (object(head(subscripts)))
                                       (tail(subscripts))
                                       result ),
                             object(j) ),
                    ⊥Storable_Values)
```

The assignment statement is somewhat complicated since it allows general updates on structured objects and input-output. Note that the Store must be updated with rhs_value at location r. Since ⟦ LHS ⟧ may contain a subscript list, the rhs_value may need to be embedded into a new_value, formed using the existing $\sigma(r)$. If a new_value must be built, it is always a structured object -- this is what the (λ j:Integers. ...) expression represents.

(6) $\mathcal{L}$: LHS -> Env -> Store -> (Locations X Integers* X Store)

$\mathcal{L}\,[\![\ V\ ]\!]\,\rho\sigma = \langle \mathcal{V}[\![\ V\ ]\!]\,\rho\sigma,\ \sigma\rangle$

$\mathcal{L}\,[\![$ output $]\!]\,\rho\sigma$

```
= (λ ν.<output_buffer_locn_no, <ν(1)>,
        Update_store(σ, output_buffer_locn_no,
                        new_value(ν, <1>, ν(1)+1)) >)
                        (σ(output_buffer_locn_no))
```

This function processes the left hand side information, which may

be an output operation. The output buffer is modeled as a structured object in which the first subscripted element is taken as the counter of the number of objects in the buffer plus two. It is stored in a reserved memory location--output_buffer_locn_no. The result is a triple: the location to be updated, a subscript list giving the subpart of the object to be updated in the location, and the current store. Note that since the output buffer is a structured object, the buffer counter is returned as is the subscript list.

(7)$\mathcal{R}$: RHS -> Env -> Store -> (Storable_values X Store)

$\mathcal{R}[\![E]\!]\rho\sigma = \langle \dot{\mathcal{E}}[\![E]\!]\rho\sigma, \sigma \rangle$

$\mathcal{R}[\![$ input $]\!]\rho\sigma$

$= (\lambda\mathcal{V}.\langle \mathcal{V}(\mathcal{V}(1)),$

Update_store(σ, input_buffer_locn_no,

new_value($\mathcal{V}$, <1>, $\mathcal{V}$(1)+1)) >)

(σ(input_buffer_locn_no))

This function processes the right hand side of an assignment, which may be an input operation. The input buffer is also modeled as a structured object same as the output buffer except it is stored in another reserved memory location--input_buffer_locn_no.

The next group of functions determine the meanings of arithmetic, string, and logical expressions and expression lists.

(8) $\mathcal{A}$: AL -> Expression*

$\mathcal{A}$ ⟦ E ⟧ = ⟦ E ⟧

$\mathcal{A}$ ⟦ E, AL ⟧ = cons($\mathcal{A}$ ⟦ E ⟧, $\mathcal{A}$ ⟦ AL ⟧)

(9) $\mathcal{E}$: Exp -> Env -> Store -> Storable_Values

$\mathcal{E}$ ⟦ AE ⟧ ρσ = $\mathcal{AE}$ ⟦ AE ⟧ ρσ

$\mathcal{E}$ ⟦ SE ⟧ ρσ = $\mathcal{SE}$ ⟦ SE ⟧ ρσ

$\mathcal{E}$ ⟦ <AL> ⟧ ρσ = build_structured_object

($\mathcal{A}$⟦ AL ⟧, $\perp_{Structured\_Objects}$, 1, ρ, σ)

where build_structured_object : Expression* X Structured_Object X

Integer X Env X Store

-> Structured_Object

build_structured_object(e, s, i, ρ, σ) =

e = <> -> s, build_structured_object

(tail(e),

(λ j:Integer. j=i -> $\mathcal{E}$ (head(e)) ρσ, s(j)),

i+1, ρ, σ)

Build_strucutred_object builds a linear structured object from the expression list argument.

(10) $\mathcal{AE}$: Ari_Exp -> Env -> Store -> Integer

$\mathcal{AE}$ ⟦AE1 Ω AE2⟧ ρσ = ari⟦Ω⟧(($\mathcal{AE}$⟦AE1⟧ ρσ), ($\mathcal{AE}$⟦AE2⟧ ρσ))

where ari : Arith_ops X Integer X Integer -> Integer

does the arithmetic operations on integers.

$\mathcal{AE}[\![V]\!]\rho\sigma$ = (λ r:Location. λ i:Integer*.

($\lambda\mathcal{V}$. $\mathcal{V}$ $\in$ Integer -> $\mathcal{V}$, $\top_{Integer}$)

(Access_object(i , σ(r)))

)* $\mathcal{V}[\![V]\!]\rho\sigma$

Access_object : Integer* X Storable_Value -> Storable_Value

Access_object (i , s) =

i = <> -> s, Access_object(tail(i), s $\in$ Structured_Object

-> s(head(i)), $\top_{Structured\_Object}$)

Access_object is a function to perform subscripting, used on structured objects to pull out the desired element.

The arithmetic expressible meaning of an identifier is the value in its corresponding store location after any needed subscripting. If error occurs, return $\top$.

$\mathcal{AE}[\![N]\!]\rho\sigma$ = integer_value_of[[N]]

Integer_value_of is a function which maps the syntactic integer object to the corresponding semantic number.

$\mathcal{AE}[\![(AE)]\!]\rho\sigma = \mathcal{AE}[\![AE]\!]\rho\sigma$

(11) $\mathcal{LE}$: Log_Exp -> Env -> Store -> Boolean

$\mathcal{LE}[\![E1 \phi E2]\!]\rho\sigma$ = log[[ϕ]]($(\mathcal{E}[\![E1]\!]\rho\sigma)$, $(\mathcal{E}[\![E2]\!]\rho\sigma)$)

where log : Logic_ops X Expression X Expression -> Boolean

does the logical operations upon the two expressions.

(12) $\mathcal{SE}$: String_Exp -> Env -> Store -> Char_strings

$\mathcal{SE}$ ⟦ SE1 catenate SE2 ⟧ $\rho\sigma$

= concatenate(($\mathcal{SE}$⟦ SE1 ⟧ $\rho\sigma$), ($\mathcal{SE}$⟦ SE2 ⟧ $\rho\sigma$))

where concatenate : Char_string X Char_string -> Char_string

concatenates two character strings into one character string.

$\mathcal{SE}$ ⟦V⟧ $\rho\sigma$ = (λ r:Locations. λ i:Integer*.

($\lambda\mathcal{V}$. $\mathcal{V}$ $\in$ Char_strings -> $\mathcal{V}$, $\tau_{char\_strings}$)

(Access_object(i , σ(r)))

)* $\mathcal{V}$⟦ V ⟧ $\rho\sigma$

This gives the string expressible meaning of an identifier, similar to $\mathcal{AE}$⟦ V ⟧ above.

$\mathcal{SE}$⟦ ST ⟧ $\rho\sigma$ = value_of_string⟦ ST ⟧

Value_of_string is a function mapping from the syntactic string object to semantic character string concept.

$\mathcal{SE}$⟦(SE)⟧ $\rho\sigma$ = $\mathcal{SE}$⟦SE⟧ $\rho\sigma$

(13) $\mathcal{V}$: Var -> Env -> Store -> (Location X Integer*)

$\mathcal{V}$⟦ I ⟧ $\rho\sigma$ = <Access_env(I, ρ), <>>

$\mathcal{V}$⟦ I sub ⟧ $\rho\sigma$ = <Access_env(I, ρ), $\mathcal{SU}$⟦ sub ⟧>

(14) $\mathcal{SU}$: Sub -> Env -> Store -> Integer*

$\mathcal{SU}$⟦ [AE] ⟧ $\rho\sigma$ = $\mathcal{AE}$⟦AE⟧ $\rho\sigma$

$\mathcal{SU}$⟦ [AE] sub⟧ $\rho\sigma$ = append($\mathcal{SU}$⟦AE⟧ $\rho\sigma$, $\mathcal{SU}$⟦sub⟧ $\rho\sigma$)

$\mathcal{V}$ defines the denotable meanings of identifiers. If the identifier is subscripted, the subscript list is evaluated and returned also.

5. ANALYSIS OF IMPLEMENTATION OF TEMPO

5.1 Original Implementation

As suggested in [Jones], TEMPO can be implemented by using a linked-list data structure. Since TEMPO emphasizes variability of the user-defined identifiers and dynamic program text generation, very late binding times are necessary. The linked-list cells are connected by pointers and need not be in contiguous memory. This characteristic can satisfy the need for changing size and shape of program's data structure during program execution. The "structured" object in TEMPO, which can be viewed as a tree-like structure, may be easily represented by linked-lists.

In implementing TEMPO, the computer memory is divided into two sections: one has a pair of linked-lists and the other has a stack. The linked-list pairs will be called the heap storage area; it holds all the necessary information about the program (program list) and maintains an inventory of currently available list cells (free list). The stack keeps track of procedure return points.

Execution of the program in TEMPO can be modeled by a series of "snapshots", which contain the program text, the location of the currently executing instruction, the values of all variables,

and other information pertinent to the execution of the program. The computer is given first an initial snapshot, which has the program to be executed, and produces successively a sequence of snapshots derived from executing the program text given originally or generated at runtime. The following is a simple snapshot:

```
SS3 = (Controlpoint, [4], Pgm)
   Blockstart (A) (P, 'parameters X; begin scope A; A := X/2;
   output := A * X; X := A; end')
```

which is the snapshot prior to evaluation of the 4th segment in the program below:

```
Pgm = [1] begin [2] scope A, P;
      [3] P := 'parameters X;
            begin scope A;
              A := X/2;
              output := A * X;
              X := A;
            end';
      [4] A := input;
      [5] call P[A];
    [6] end [7]
```

The snapshot listed above has the following information:

(1) Controlpoint - indicates the starting of a level made by

calling a procedure or first time entering the program.

(2) Pgm - the abbreviated form of segmented program text; execution of a segement of the program with previous snapshot will produce the current snapshot.

(3) [4] - marker, pointing to next segment to be executed.

(4) Blockstart - labels the starting of a block.

(5) the variables: A is not bound to any value yet, P is bound with a character string which can be treated as a procedure to be called later.

The following are some examples using the linked-list data structure representation: ([...|...] is a pair of memory cells)

<1> Numbers: Number is represented by cells of the form

[NUMBER | VALUE]

where "NUMBER" is an identifying code and "VALUE" itself is the value of the number.

<2> Strings: the string contains the characters S1 S2...Sn for $n \geq 0$ is represented as

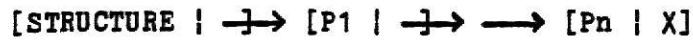
[STRING | ⊣→ [S1 | ⊣→ ⟶ [Sn | X]

"X" is the null code (indicating the end of the string).

<3> Undefined: the undefined value "⊥" is represented by

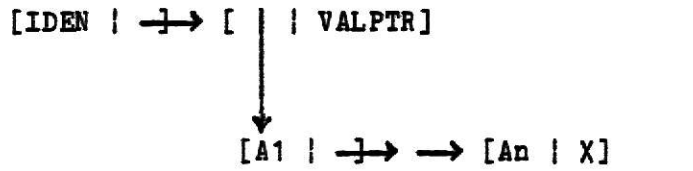
[UNDEFINED | X]

<4> Structured: the structured object <V1,...,Vn> is represented as



where P1 is a pointer pointing to the linked list representation of V1 and Pn is pointing to Vn.

<5> Identifiers: an identifier with name A1...An and value bound to it is



where VALPTR is the value_pointer pointing to another list representing the value. If there is no value bound yet, VALPTR will be null.

A strong similarity exists between the snapshot form and the runtime data structures of LISP [McCarthy1]. The structured objects are implemented similar to SNOBOL arrays [Griswold].

5.2 Realization Through Semantic Functions

As we have described, the denotational semantics approach for a programming language can give a clear description of the features of the language and provide directions in implementation. Let us examine the semantic functions described in 4.2 to catch the ideas revealed.

(1) $\mathcal{M}$ can be viewed as the operations needed to start compiling/executing a program. The program represented as Pgm and

the computer memory (Store) is represented by σ. We will need to initialize a symbol table (Env) which is normally used at compile time. In TEMPO we have to keep the symbol table at runtime to handle the local declarations arising from calls of dynamically generated procedures.

(2) $\mathcal{B}$ is to specify the block structure of the program. A block has two forms: the first one is simply the Stmt-- S. The second one has local variable declarations for the corresponding block. Upon exit of the block, we can deallocate the memory space used by the local variables, which are not needed anymore. This implies that we need a method to mark unneeded cells so that we can collect them later. We might need a scheme like a garbage collector to collect this space.

(3) $\mathcal{D}$ scans the local variables and uses the $\mathcal{DL}$ function to take care of necessary operations.

(4) The main purposes for $\mathcal{DL}$ are to, first, allocate new memory space for newly declared local variables; second, supply the identifiers' names and allocated memory locations to update the environment; third, initialize the memory cells just allocated with the value $\perp$.

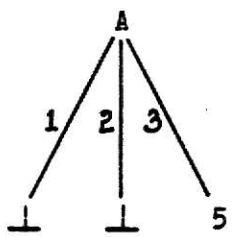
(5) $\mathcal{C}$ defines the control structures of TEMPO. $\mathcal{C}[\![$ S1 ; S2 $]\!]$ can be implemented as a sequence of execution of statements; S1 is

executed first and then S2 is executed with the new memory store produced by execution of S1. In $\mathcal{C}[\![$ if LE then S $]\!]$, LE will be evaluated first to determine the logical expression being true or false. If true, then execute the statement body--S, else skip to next instruction. $\mathcal{C}[\![$ while LE do S $]\!]$ is an iteration and is defined recursively (as described before). One obvious implementation of this function is to use a stack storage scheme. First evaluate LE, if the result is true then push the whole statement--while LE do S onto stack and execute S. After the execution, pop the whole statement just pushed and execute the whole statement again with the new memory store generated by previous execution of the statement body--S. This kind of execution will continue until LE is evaluated being false. Of course, this is hardly efficient-- analysis of the least fixed point solution shows that the usual iterative approach will work correctly (as it normally does for all "tail recursive" function subroutines).

For the call statements, we have to parse the character string generated at runtime (treat it as a Procedure) to see if it is syntactically correct. This is for the sake of dynamic program text generation ability of TEMPO. It implies we should keep a parser around during program execution. For those call statements with parameters, in addition to parsing, we have to textually substitute the parameters by expression arguments. This will involve some work: separate the statements and formal parameters in the procedure, reformat the arguments and formal parameters,

then do the substitution. At the same time, we have to match the amount of the arguments with the parameters. No type checking needed at this time; it is only made during the evaluation of expressions.

The assignment statement is the most difficult construct to define. It is especially true in TEMPO because the programmer can vary the types of variables, and the sizes of the variables may change. A great deal of house keeping is needed. There are some problems encountered because of the special features of structured objects. For instance, we may have to handle new subscripted identifiers implicitly even though the programmer is not asking for them. In A := 'A'; A[3] := 5; we have two statements: the first one is a primitive variable with character 'A' bound to identifier A; the second assignment will have the third element of A be assigned an integer-- 5. If A was not a structured object previously, we have to arrange room for A[1] and A[2] also. The identifier-value binding will appear like



In a sense, A[4] = ⊥, A[5] = ⊥, ... Also, we can assign another structured object to an element of a structured object. Ordinarily, when we are doing the assignment operation, the needed information for a FORTRAN-like assignment includes the location corresponding to the left_hand_side identifier and the storable

value of the right_hand_side expression. But in updating the variables in TEMPO, we need the value of the left_hand_side plus its location. The strategy for evaluating the assignment statement is as follows:

<A> Evaluate the right_hand_side to find the value associated with it; the right_hand_side may be an expression or an input operation. Both the value and a new Store are produced (possibly the contents of the memory have been modified, as in an input operation).

<B> Evaluate the left_hand_side to get the location corresponding to it, a subscripts_list (which may be empty), and a new Store (which possibly has been updated if the right hand side is an output operation).

<C> Using the location provided from the left_hand_side, access the corresponding storable value.

<D> If the subscripts_list is empty, then update the left_hand_side location with the right_hand_side value, else if the left_hand_value is a structured_object, then update the subscripted portion, else convert the left_hand_side to a structured object and update the left_hand_side location using the right_hand_side value.

Here is an example; [N] points to the statement currently executed, <X> means the step in the strategy currently activated:

```
[1]  A := 0;
[2]  A := <0 , 0>;
[3]  A[1] := 'X';
[4]  A := A[2] + 1;
[5]  A[2] := 'Y';
```

[1]-- A := 0;

<A> the right_hand_side value is 0.

<B> the location of left_hand_side A is called L. The subscripts_list is empty.

<C> the value of left_hand_side is ⊥ .

<D> since the subscripts_list is empty, replace ⊥ with 0, the right_hand_side value, in location L.

The identifier-location-value binding becomes

[2]-- A := <0 , 0>;

<A> right_hand_side value = <0 , 0>.

<B> location is L; subscripts_list = <>.

<C> left_hand_side value is 0.

<D> subscripts_list = <>; replace 0 by <0 , 0>

The binding becomes

[3]-- A[1] := 'X';

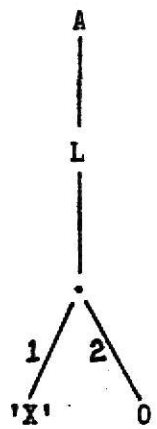
<A> right_hand_side value = 'X'.

<B> location is L; subscripts_list = <1>.

<C> left_hand_side value = <0 , 0>.

<D> subscript is 1; and left_hand_side value is a structured object; change first subscripted value.

The binding becomes:



[4]-- A := A[2] + 1;

<A> right_hand_side value = 1, determined by obtaining location L, extracting <0 , 0> from the cell, subscripting out the 0, and adding 1 to it.

<B> location is L; subscripts_list = <>.

<C> left_hand_side value = <0 , 0>.

<D> subscripts_list =<>; replace <0 , 0> by 1.

The binding becomes

[5]-- A[2] := 'Y';

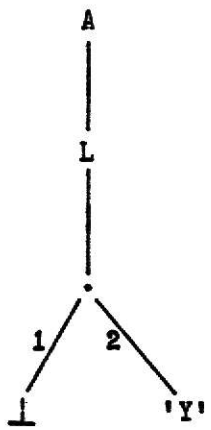
<A> right_hand_side value = 'Y'.

<B> location is L; subscripts_list = <2>.

<C> left_hand_side value = 1.

<D> subscript is 2, and left_hand_side value is an integer; coerce it to an undefined structured object, initialize the first subscripted value with ⊥ and change the second subscripted value to 'Y'.

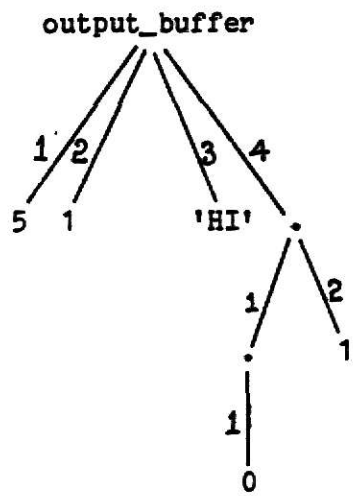
The binding becomes



From the example above we can notice that, though every memory location can contain any kind of data, we must have a way to tell

the data type of the value in the memory location. It can be achieved in implementation by using some bits of each memory location to give such information. We will only assume we have the ability to get such information.

(6) The $\mathcal{L}$ function is to provide necessary information which can be utilized by the assignment operation in the $\mathcal{E}$ function. Also, it deals with the output operation so that we can put data into the output_buffer. In the output_operation, the output_buffer is modeled as a structured object stored in a reserved memory location (output_buffer_locn_no). The first subscripted value is regarded as the counter pointing to next subscript which will be used to store the next output value. It is incremented by one every time the output operation is activated. The following example can make the output_buffer model clear. After executing three statements-- output := 1; output := 'HI'; output := <<0>, 1>, we can diagram the output_buffer as



(7) The $\mathcal{R}$ function is to supply the information to the assignment operation also. It evaluates the right hand side expression of an assignment; this might be an input operation. The input_buffer is modeled in the same way as the output_buffer described before except it is stored in a different reserved memory location--input_buffer_locn_no.

(8) The $\mathcal{A}$ function is to make the argumentlist an expression list so that it will be easier for the $\mathcal{C}$ function to evaluate.

(9) The $\mathcal{E}$ function is to evaluate the expression and return with the Storable_Values. They may be integers, character_strings or structured objects.

(10) The $\mathcal{AE}$ function is to evaluate the arithmetic expression and return with an integer value. This is one of the functions which need type checking. If the operands are not both integer numbers, then $\mathcal{AE}$ should return $\mathcal{T}_{\text{Integer}}$ back.

(11) The $\mathcal{LE}$ function performs logical operations and returns a boolean value.

(12) The $\mathcal{SE}$ evaluates the string operations such as concatenation. This function also needs to do type checking to make certain the operands are all Char_string type.

(13) The $\mathcal{V}$ function evaluates a variable to obtain a location and (a possibly empty) integer subscript list.

(14) $\mathcal{SU}$ takes care of the subscript part needed in the $\mathcal{V}$ function. It evaluates the subscript list of arithmetic expressions and makes the result into an integer list.

5.3 Improvements In Efficiency

From the previous discussion we know the implementation of TEMPO is achievable but will be very inefficient. The following are some suggestions on improving the efficiency of TEMPO. They place more restrictions on the language so that the bindings can be performed earlier, say, at compile time and make TEMPO run more efficiently.

(1) Datatype And Size Of Identifier

In order to make datatype checking and storage allocation more efficient, we can have earlier bindings by altering the "scope" statement in the grammar to specify the datatype and, in the case of structured objects, the size (upper and lower bounds).

```
scope I : integer;
      ARRAY : structured_object [1..10] of char;
```

I is declared as an integer and ARRAY is declared as a structured

object, which actually is an array of characters with upper bound of 10 and lower bound of 1. After this change the compiler can check types and determine storage sizes at compile time. This change also makes the assignment statement much simpler because the storage sizes and locations are static, and the assignments to structured objects are simplified.

(2) Program Text Generation And Parameter Passing

Keeping a parser around during runtime is necessary to support the dynamic program text generation feature of TEMPO. This is one of the main entities which make the system inefficient. To avoid runtime parsing, we have to eliminate dynamic program text generation and force the procedures to be declared before they are used. Furthermore, since the call_by_text parameter passing method also causes inefficiencies, we can replace this method with other more efficient methods such as call_by_value and/or call_by_reference so that runtime text manipulation is not needed. This allows for easier address calculation and storage arrangement. Changes can be made as shown in the following example:

```
procedure P(var A : integer);
          begin
            A := input;
            A := A + 1;
          end
```

With these changes, it is now possible to make TEMPO more block structured and use stack storage management. We can notice that these changes are toward earlier bindings so that the compiler can have sufficient information to do more work. As late bindings are not provided anymore, programming convenience has been traded for system efficiency.

6. CONCLUSIONS

We have studied TEMPO and defined the denotational semantics for the programming language. A very formal and mathematical description for TEMPO has been given so that we have a rigorous way to examine the language.

From the semantic domains and functions defined we can understand the features of TEMPO clearly and realize the problems that we are going to have in implementation. Methods for solving these problems have been suggested through the modeling of structured_objects, input-output buffers, side effects, etc. The main unusual features of TEMPO are that TEMPO allows powerful assignment operations, allows type variance of user defined identifiers, and allows the programmer to generate program text during program execution. These features provide the programmer a great deal of flexibility, but they also add much burden to the runtime system. We have discussed solutions for handling the work. For instance, in assignment operations, we can assign a value to a variable (structured or not) in any way the programmer wishes, such as assigning a structured object to a variable which previously contained an integer. In order to avoid problems caused by side effects, we model structured objects as a function mapping from integer list (subscripts) to storable values. This model implicitly points out we need to create another copy of the

assigned value for the new variable. SNOBOL has similar features but it can not handle the side effects problems: after assigning a variable which contains an array to another variable, if we change one of these two variables, then we will discover that both variables are changed. In APL, everything is regarded as an array but then the arithmetic operations are very complex and overloaded. LISP does not even allow the programmer to assign an array to the element of another array.

Although the problems revealed can be solved as suggested in this report, it can be predicted that the real implementation will be terribly inefficient. The possible improvements are to trade in programmer convenience for efficiency of the runtime system. By adding more restrictions to the language, the compiler can do more and will make the runtime system more efficient.

BIBLIOGRAPHY

[Church] Church, A.; "The Calculi Of Lambda Conversion", Princeton University Press, Princeton, N. J. (1951).

[Gordon] Gordon, M. J.C.; "The Denotational Description Of Programming Languages, An Introduction", Springer-Verlag, New York, NY (1979).

[Gratzer] Gratzer, G.; "Universal Algebra", Van Nostrand, N. Y. (1967).

[Griswold] Griswold, R. E.; "The Macro Implementation Of SNOBOL4", W. H. Freeman & Co., San Francisco, CA (1972).

[Hoare] Hoare, C. A. R.; "An Axiomatic Basis For Computer Programming", Communications of the ACM, 12-10 (1969) 576-580, 583.

[Jones] Jones, N. D., and Muchnick, S. S.; "TEMPO : A Unified Treatment Of Binding Time And Parameter Passing Concepts In Programming Languages", Springer-Verlag, Berlin (1978).

[McCarthy1] McCarthy, J., et. al.; "LISP 1.5 Users' Manual", MIT Press, Cambridge, Mass. (1962).

[McCarthy2] McCarthy, J.; "Towards A Mathematical Science Of Computation", Proc. IFIP Congress, Munich, North-Holland (1972).

[Scott1] Scott, D.; "Continuous Lattices", in Lawrere, F. W. (ed.), "Toposes, Algebraic Geometry And Logic", Springer-Verlag, N. Y., 97-136 (1972).

[Scott2] Scott, D.; "Data Types As Lattices", SIAM Journal Of Computing, 5 (1976) 522-587.

[Stoy] Stoy, J. E.; "Denotational Semantics: The Scott-Strachey Approach To Programming Language Theory", The MIT Press, Cambridge, Mass. (1977).

[Tennent1] Tennent, R. D.; "The Denotational Semantics Of Programming Languages", Communications of the ACM, 19-8 (1976) 437-453.

[Tennent2] Tennent, R. D.; "Principles Of Programming Languages", Prentice-Hall, London (1981).

[Wegner] Wegner, P.; "Programming Language Semantics", in Rustin, R., (ed.), "Formal Semantics Of Programming Languages", Prentice-Hall, Englewood Cliffs, N. J., 149-248 (1972).

Analysis of TEMPO
Using the Denotational Semantics Approach

by

Wan-Hong Cheng

B. S., National Chiao-Tung University, Taiwan, R. O. C., 1980

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1982

ABSTRACT

The denotational semantics description method is used to describe and highlight the characteristics of a nontrivial programming language. The language is TEMPO, a SNOBOL-like language emphasizing late binding times and providing programmer conveniences for data object creation and assignment. The denotational definition of TEMPO points out the difficulties in practical implementations of the language and gives some solution to the problems. An analysis of the language according to the semantic definitions is given. It reveals the structure of the language and suggests some directions which can be followed to improve implementation efficiency.