UC Irvine
ICS Technical Reports

Title
The Distributed BASIC Interpreter System

Permalink
https://escholarship.org/uc/item/0582d5t9

Author
Levin, Steven L.

Publication Date
1973-06-01

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

The Distributed BASIC Interpreter System

Steven L. Levin

TECHNICAL REPORT #33 - JUNE 1973

Department of Information and Computer Science
University of California, Irvine

The Distributed BASIC Interpreter System

by

Steven L. Levin

June, 1973

ABSTRACT: This paper presents a design for a translator system to be used in a distributed computing environment. The concept of a language service for such an environment is discussed and the distribution of the system's processes is examined. A technique for moving interrupted interactive computations among processors is presented. Detailed design specifications are provided for the translator system processes.

The work reported here was supported by National Science Foundation Grants GJ-36414 and GJ-1045.

UCI Technical Report 33, XDA-4, DCS Memo No. 33

ACKNOWLEDGEMENTS

Professor David Farber supplied the initial idea of a distributed BASIC system. Dr. Peter Freeman suggested studying the process used to obtain the design for the BASIC system and aided in defining the format of this paper. Richard Burton, Elaine Gord and Marsha Hopwood provided many ideas and suggestions in the course of this work. Sandra Mass assisted with the editing and typing.

FOREWORD

The design described here was undertaken to explore the issues of a distributed language system and the process of software rationalization. The design process was documented by maintaining a diary in which the path and alternatives of the design were recorded. A conscious effort was made to examine all alternatives and record the reasons for rejection.

Design alternatives resulting from this process of software rationalization are included in this document. The lists of alternatives are not complete. Indeed, exploring how to examine more fully the alternative space for a software designer partly motivated this effort. The utility of providing implementors with alternative designs is being studied as the system is undergoing implementation. Thus, this document is primarily an instance of a rationalized design and does not provide any analysis of the process.

Section 1 is an introduction that explains the goals, motivation and overall design of the language system. The remainder of this document is an application of these ideas to a particular language. Sections 2, 3, and 4 are detailed design specifications for the processes comprising the

implementation of a BASIC system. The technical specification in each section is preceded by an introduction and overview. Section 5 describes the techniques used for moving computations amongst processors.

CONTENTS

Abstract

Foreword

Acknowledgements

Section One: Introduction

1.1 Introduction to the Distributed BASIC Interpreter System 1-1

1.2 The Distributed Computing System 1-1

1.3 DBIS: A Language Service 1-2

1.4 Distributed Processes in the DBIS 1-3

1.4.1 Load Sharing 1-4

1.4.2 Processor Specialization and Hierarchical Computing 1-6

1.5 The Design of the DBIS 1-7

1.5.1 The Input Handler 1-8

1.5.2 The Translator 1-8

1.5.3 The Interpreter 1-9

1.6 Program Mobility 1-12

1.7 System Portability 1-12

Section Two: Input Handler Process

2.1 Introduction 2-1

2.2 Overview 2-1

Section Three: Translator Process

3.1 Introduction 3-1

3.2 Overview 3-1

3.3 Interpretive Expressions 3-3

3.4 Translator Error Messages 3-7

3.5 Technical Specification 3-9

3.5.1 Technical Overview 3-9

3.5.2 Translator Tables 3-9

3.5.3 Translator Procedure descriptions 3-11

3.5.3.1 Lexical Analysis 3-12

3.5.3.2 Translation Procedures 3-21

3.5.3.3 Expression Analysis 3-53

Section Four: Interpreter Process

4.1 Introduction 4-1

4.2 Overview 4-1

4.3 Technical Specification 4-3

4.3.1 Technical Overview 4-3

4.3.2 Interpreter Tables and Data Structures 4-5

4.3.2.1 Program Representation 4-5

4.3.2.2 Symbol Table 4-9

4.3.2.3 The R-stack 4-21

4.3.2.4 The S-stack 4-21

4.3.2.5 The T-stack 4-22

4.3.2.6 Transfer Table 4-22

4.3.2.7 User Information Table 4-23

4.3.3 Storage Allocation 4-24

4.3.3.1 Introduction 4-24

4.3.3.2 Pointer Cells 4-24

4.3.3.3 Floating Point Cells 4-26

4.3.3.4 Strings 4-27

4.3.3.5 Storage Reservation and Liberation 4-28

4.3.3.6 Consolidated Storage Allocation 4-30

4.3.3.7 Design Alternatives 4-31

4.3.3.8 Design Alternatives for String Storage 4-32

4.3.3.9 Implementation Alternatives 4-34

4.3.3.10 Summary of Allocation Functions 4-35

4.3.3.11 Storage Allocation Procedures 4-36

4.3.4 Interpreter Procedure Descriptions 4-43

4.3.4.1 Internalization 4-43

4.3.4.2 Evaluation 4-58

4.3.4.3 String Handling 4-74

4.3.4.4 Stack Accessing 4-84

4.3.4.5 Buffer Accessing 4-91

Section Five: Program Mobility

5.1 Introduction 5-1

5.2 Computation Interruption 5-1

5.3 Context String Formats 5-2

5.3.1 Statement Forms 5-3

5.3.2 Symbol Table 5-4

5.3.3 R-stack 5-9

5.3.4 S-stack 5-9

5.3.5 T-stack 5-12

References

Appendix A: Procedure Index

Appendix B: Partial Procedure Descriptions

Section One

Introduction
to the
Distributed BASIC Interpreter System

1.1 Introduction

The Distributed Basic Interpreter System (DBIS) is an interactive language service designed to provide incremental compilation and execution of BASIC programs in the environment of the Distributed Computing System (DCS). The main design goals are to allow distribution of the processes composing the language system and permit movement of incomplete interactive computations from one computer to another without interruption of service to the user. The techniques in the DBIS are applicable to languages other than BASIC and to environments employing a delayed binding of names to locations.

1.2 The Distributed Computing System

The Distributed Computing System is a computer network under development at the University of California, Irvine [2]. Hardware consists of a collection of computing system elements (processors/memory, secondary storage, peripherals) connected to a digital communications ring. The network software is composed of processes which are addressed by

name and interact by sending and receiving messages.

1.3 DBIS: A Language Service

Users of the DBIS view BASIC as a language service available on the DCS. Specifically, users are provided with a service and in general have no concern or control over how that service is provided. The DBIS provides the network's full range of computational power to a user who need not know the particulars of the processors providing the service. A user becomes familiar with the properties, facilities and protocols of the DBIS and not the machines providing the service.

Since part of the power of the DBIS is the ability to move an incomplete computation to another processor (as when a user's resource demands exceed those of the current processor) users may encounter some computational anomalies. The anomalies arise in executing the same computation at different points in space and time. Different processor architectures result in varying word sizes, arithmetic algorithms and test conditions.

This may lead to different overflow conditions and different roundoff and truncation errors. If these differences are significant, facilities could be provided

within either DBIS or the computing system to allow users to specify required arithmetic precision and arithmetic algorithms.

1.4 Distributed Processes in the DBIS

The DBIS is composed of three component processes that communicate by sending messages. The component processes (input handler (IH), translator (TR) and interpreter (INT)) and their communication paths are shown in figure 1-1.

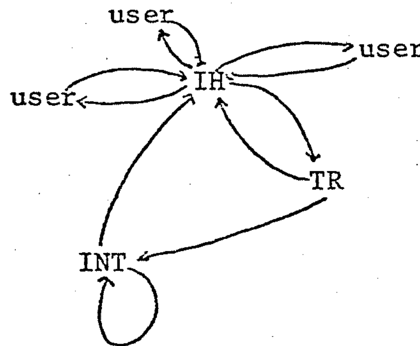


Figure 1-1: DBIS Processes and Communication Paths

The input handler process controls input/output between the user and the DBIS. It establishes a user information table with the user's identification, processing requirements and input handler and interpreter process names. Since a user's BASIC statement may be translated into interpretive format by any available translator process and all processes operate asynchronously, the input handler

must assign sequence numbers to user inputs to insure the correct order of statement evaluation by the interpreter process.

The translator process performs lexical and syntactic analysis on the user input and translates the input into interpretive format. Detected errors result in a message to the user via the input handler process. Translated input in the format of an interpreter context string (CS) is sent to an interpreter process. User information accompanies messages sent from the input handler to a translator, from a translator to an interpreter, and from an interpreter to an interpreter.

The interpreter process internalizes a context string and then may continue the execution of an interrupted computation or initiate a new computation. An interpreter process may generate a context string which is a canonical representation of the information needed to move a partial computation to another interpreter process. Interpretation errors are reported to the user via the input handler process.

1.4.1 Load Sharing

One advantage of a network architecture is the

possibility of load sharing, dynamically distributing the load so that each processor receives some portion of the total work [9].

When a hardware processor becomes unavailable, then any DBIS component process active on that processor may be moved or restarted on another processor.

Most interactive language systems do not attempt dynamic load sharing because of the difficulties in moving computations from one processor to another and the need to adjust communication paths when processes are moved [8,9,10].

The problem of transferring computations is partially solved in the DBIS by uncoupling the user's context from the physical processor. Adjusting communication paths is unnecessary in the DCS since messages are addressed to a process name rather than a physical location.

Another consequence of having distributed process components is that the number of input handlers, translators and interpreters may be varied to meet user demand. An example is shown in figure 1-2 where a large number of users require high levels of computation but very little input/output and statement translation.

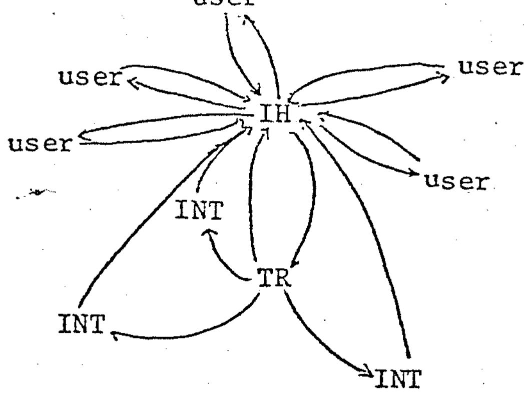


Figure 1-2: Balanced Process Service in the DBIS

1.4.2 Processor Specialization and Hierarchical Computing

The processors that compose a network may have differing architectures. That is, those processors in the system may vary in speed, resources, word sizes and order codes. Given that both computations and processes can migrate in the network, this suggests that processes might be located on 'specialized processors' and that computations with increasing resource demands could be moved to acceptable processors.

A process is located on a 'specialized processor' if the efficiency of executing that process on that particular processor is greater than might be obtained on another processor in the network under the same conditions. If the network had only one processor with hardware floating point arithmetic instructions it would be a good candidate as an

'interpreter processor.' Similarly other processors might be specialized with respect to the DBIS in input/output (input handler) and character handling (translator process). The allocation of processors to processes is considered a function of the network.

If the DCS were augmented by the addition of more powerful processor/memory components then a computation might be moved from processor to processor as its resource demands outstripped those of its current environment.

In a university environment this means students would have most of their computing needs satisfied economically on the smaller more specialized processors while programs that exceed the resources of these processors would be moved to larger processors without interrupting service to the user. Thus, users are supported by a powerful range of computing resources which are accessible through the same language service.

1.5 The Design of the DBIS

This section presents an overall view of the DBIS. Extended discussion of program mobility, system portability and technical design specifications are found in later sections.

1.5.1 The Input Handler

Input/output between the user and the DBIS is performed by the input handler. Statements input by the user are assembled according to DCS message protocols and then sent over the network to be translated by any available translator process. Information identifying the user accompanies the statement as it proceeds to a translator process.

1.5.2 The Translator

The translator transforms user statements into a statement canonical form which is shown below in figure 1-3. A statement form is a list that contains a statement number (if present), a statement type, a statement and a series of interpretive expressions.

(statement number statement type source
interpretive expressions)

Figure 1-3: Statement Canonical Form

The expressions are represented in "Cambridge Polish" notation, a notation in which a function call is represented as a parenthesized list whose first element is the name of the function and whose remaining elements are its arguments

[7].

The statement form and user information constitute a context string (see figure 1-4). This string is sent to the interpreter process on which the current user context is established.

user information # statement canonical form

Figure 1-4: Context String

1.5.3 The Interpreter

Examination of the user information portion of the context string determines whether a user's context is already established on a particular processor. A context includes program, data values and computational status. If the context string originates from a translator it contains user information and a statement form. If the context string originates from an interpreter it consists of user information and canonical forms for the statements, symbol table, and interpreter stacks. The format of a complete context string is given in figure 1-5.

user information # statement forms # symbol table
result stack # expression stack # transfer stack #

Figure 1-5: General Context String

Regardless of origin, context strings are internalized using identical interpreter procedures. A program is a list of statement forms as shown in figure 1-6.

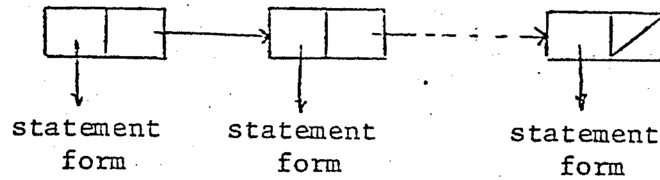


Figure 1-6: Program List

References to functions and variables in the interpretive expressions are to relative entries in the symbol table as shown in figure 1-7.

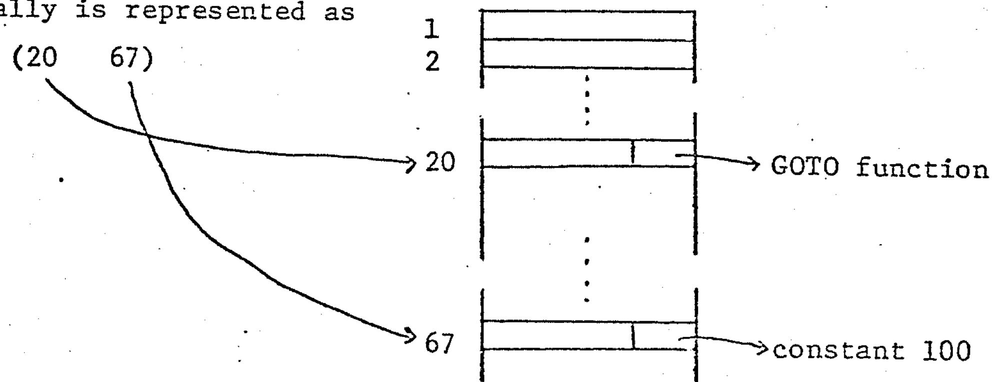


Figure 1-7: Symbol Table References

In this manner, addresses in the interpretive expressions are not bound to physical locations on any processor. Address bindings are made at interpretation time. This practice is extended to the interpreter result stack (R-stack) and transfer stack (T-stack).

Intermediate values obtained in evaluating function arguments and expressions are stored as temporary entries in the symbol table and referenced by relative addresses placed on the R-stack.

Control information (return addresses) is divorced from physical locations by placing relative transfer addresses on the T-stack. These addresses map into a table that contains the physical addresses. As before, binding of relative addresses to physical locations is delayed until execution of the transfer.

On the S-stack (expression stack) actual machine addresses are used as pointers to portions of the program list and its component structures, statement forms and interpretive expressions. A canonical representation for the S-stack is discussed in section 5.3.4.

All interpreter values (user and system) and function addresses are accessed through the symbol table. The symbol

table contains descriptive information on the symbol's name, symbol type (simple numeric, simple string, dimensioned numeric, function), data type (integer, floating point, string) and value. Values are pointers to defined data type structures.

1.6 Program Mobility

Programs may be moved in the DBIS by forming a canonical character string representation (context string) as shown in figure 1-6.

The structure and data of the program, symbol table and stacks are preserved in the format of the string. The DBIS system mandates that all interpreters accept and generate context strings in the same format. Furthermore, the points at which computations may be interrupted are identical in all interpreters.

Detailed format descriptions of the context string are discussed in section 5.

1.7 System Portability

The DBIS processes as implemented on one network processor must perform identically to similar DBIS

components on any other processor. To be feasible, successive implementations of the DBIS on other processors should require a minimum of recoding. This goal is supported by 1) implementing the DCS version in a high level ALGOL-like language [5] and 2) confining machine dependent operations to a set of 'primitive' procedures.

In a limited way, the primitive procedures help to uncouple the design from processor architecture by permitting the implementor to choose suitable data representations without propagating design changes throughout the system. Other procedures requiring information from data structures know only the name and outputs of the appropriate accessing 'primitive.'

Portability in the DBIS is not gained by standardizing on a virtual processor. Implementors are free to tailor their data structures to a particular processor as long as the performance of the accessing procedures meet design specifications.

Section Two

Input Handler Process

2.1 Introduction

The input handler process controls all input/output between the user and the DBIS. It establishes a user information table which may be moved to other processors as may the input handler process. Technical specification will not be available until details in this area are finalized in the DCS design.

2.2 Overview

The input handler accepts input from the user and forms the message string shown in figure 2-1.

user information # user input

Figure 2-1: Message from Input Handler to Translator

This message is sent to any translator process on the network. The user information contains a sequencing number for the interpreter and may include the name of an interpreter process. The sequencing number is used to insure correct order in evaluating user statements.

Synchronization is not guaranteed since all DBIS component processes operate asynchronously.

Design Alternatives

Part of the overhead in the DBIS not found in a traditional language system is the time and code spent assembling and sending messages. Collapsing the two processes into one removes the overhead but eliminates the use of specialized I/O or translation processors.

Section Three

Translator Process

3.1 Introduction

The translator process attempts to generate a statement form from the user input. If errors are detected in translation then a diagnostic message is issued to the user through the input handler, otherwise a context string is formed and sent to an interpreter process.

3.2 Overview

Translation into interpretive format begins by lexically processing the user input to eliminate blanks (except in quoted strings), reduce multiple symbol operators to one token and detect trivial errors. In particular, unequal numbers of quotation marks and unequal numbers of parentheses are detected.

Statement types are identified by comparison with a key word table and individual translation procedures being invoked to process each statement type and produce the interpretive expressions that constitute part of the statement form. Figure 3-1 shows the statement form generated by the translator.

(statement number statement type user input

interpretive exps)

Figure 3-1: A Statement Form

A context string as shown in figure 3-2 is sent to an interpreter process.

user information # statement form

Figure 3-2: Context String

3.3 Interpretive Expressions

Interpretive expressions are lists of actions which must be evaluated to execute a BASIC statement. The number of interpretive expressions generated for a single BASIC statement depends on the type and composition of the statement. Some statements, like the RETURN, generate a fixed number of expressions (i.e. (SET :P: (POP))) while other statement types like INPUT will generate a variable number of expressions depending on their parameter list.

In almost all cases these expressions are interpreted sequentially within each statement form. The exceptions are DATA and DEF statements.

All expressions are represented in "Cambridge Polish" notation, a notation in which a function call is represented as a parenthesized list whose first element is the name of the function (primitive or special in the DBIS) and whose remaining elements are its arguments. The primitive functions manipulate stacks and system status variables (enclosed in colons) while the special functions carry out the execution of BASIC operations.

Design Alternative

An alternative coding was considered for almost every

BASIC statement. The general alternative was to compose the interpretive expressions using only a small set of primitive functions. While this reduces the number of functions that require implementation, reduces some table sizes and lessens the implementations relation to any particular language, it is cumbersome and produces unwieldy expressions. A mixture of primitive and special functions is a middle choice.

Interpretive Expressions Generated By BASIC Statements

1. GOTO exp

```
(GOTO exp)
```

2. READ v1,v2,...,vn

```
(ASSIGN v1 (READD))
(ASSIGN v2 (READD))


(ASSIGN vn (READD))
```

3. GOSUB exp

```
(PUSH :P: (POP))
(GOTO exp)
```

4. RETURN

```
(SET :P:)
```

5. LET v = exp

```
(ASSIGN v exp)
```

6. FOR v = exp1 TO exp2 STEP exp3

```
(ASSIGN v exp1)
(FORTEST v exp2 exp3)
(PUSH :P:)
```

7. NEXT v

```
(SET :FORF: :TRUE:)
(SET :P: (POP))
```

8. STOP

```
(HALT)
```

9. END

    (HALT)

10. DIM v1(exp1), v2(exp2,exp3)

    (DIM v1 exp1 :NIL:)

    (DIM v2 exp2 exp3)

11. DATA exp1,exp2,...,expn

    (exp1 exp2 ... expn)

12. INPUT v1,v2,...,vn

    (ASSIGN v1 (INPUT))

    (ASSIGN v2 (INPUT))

    (ASSIGN vn (INPUT))

13. IF exp1 THEN exp2

    (IF exp1 exp2)

14. RESTORE

    (SET :DATAP: :PROGP:)

15. PRINT exp1, exp2;

    (PRINT exp :COMMA: exp2 :SEMICOLON: :noprint:)

16. DEF FN<letter>(p1,p2,...,pn) = exp

    (p1 p2 ... Pn)

    (exp)

17. FNEND

3.4 Translator Error Messages

An error detected during translation is announced to the user by invoking the procedure TRERR. TRERR has one argument which is the number of the error message to be issued.

TRERR prepares and sends a message to the user via the input handler. The text of the message is kept in an incore table.

Design Alternatives

1. Store error messages on secondary storage.

   If core is at a premium and/or extensive messages are contemplated then the text may be kept on secondary storage and retrieved indirectly or via a hash function. The easiest alternative is to store the text of messages in core.

2. Issue error messages from the procedure that has the most information regarding the error.

   If we had been concentrating on a product where there would be regular use in a teaching environment I would have pushed the idea of having error messages originate from the context where the most

information was available as to what caused the error and how it could be corrected.

3.5 Technical Specification

3.5.1 Technical Overview

The user input is deposited into the buffer SRC and the procedure LEXICAL used to lexically process the string. The output of LEXICAL goes to the buffer labelled NSRC. STMTN is a copy of the statement number formed by LEXICAL.

PARSE identifies the user input by comparison with a keyword table and invokes the appropriate translator procedure to syntactically analyze the statement and produce interpretive expressions which are pushed on the S-stack. TRNCNTL creates a statement form and context string. The context string is sent to an interpreter process.

3.5.2 Translator Tables

A) KEYWRD - Key Word Table

This table contains statement key words ordered by increasing length. An entry consists of a key word string and its symbol table location. NKEYWRDS is the number of keywords in the keyword table.

B) PRECTB - Precedence Table

C) BINOPTB - Binary Operator Table

BINOPTB is an association table for binary operators and function addresses in the symbol table.

3.5.3 Translator Procedure Descriptions

Many of the procedures used in the translator process are duplicated in the interpreter and are described in other sections. A complete index to all procedures is found in appendix A.

Symbols in procedures prefaced 'S:' or ':' are equated to symbol table locations (i.e. They are strings that correspond to the symbol table entry for that function or variable).

3.5.3.1 Lexical Analysis and Statement Recognition

LEXICAL	- Lexical Analyzer
PARSE	- Statement Recognition
GETKEY	- Accessing Procedure for the Keyword Table

1. Name: LEXICAL

2. Function:

This module accepts as input the user's source statement and outputs in a new buffer a character string that is more easily translated by the remaining routines. A limitted amount of error checking is performed.

3. Description:

After initializing the buffer pointers SRCP and NSRCP for the buffers SRC and NSRC respectively the string in SRC is scanned and the user identification information (USER-INFO) is extracted (the user information may be removed by the routine which calls LEXICAL, in this case the pointer SRCP is positioned at the beginning of the user source statement).

Processing continues with the scanning of SRC and the creation of a new source string (NSRC) in which all extraneous blanks have been removed and the multi-character symbols ("<=", ">=", and "<>") have been reduced to single characters.

In parallel, tests are made for matching parentheses and quotation marks. Upon encountering a delimiter the original source statement is extracted (SRCS). The compressed source string is used to obtain the statement number if present. If there is no statement number then STMTN is set to the null string. Control is then passed to the parser (PARSE).

4. Design Alternatives:

A) General table driven lexical analyzer.

BASIC requires only very simple lexical analysis and this particular design is not sufficiently general to require a general analyzer.

B) Process statements by replacing keywords, literals and constants with tokens.

Some token replacement is done as with multiple character symbols but in general it is unnecessary since the type and composition of variables and numbers is simple as is the expression analyzer.

5. Functions Called:

GETCHR PUTCHR OUTSTR DIGIT MATCH

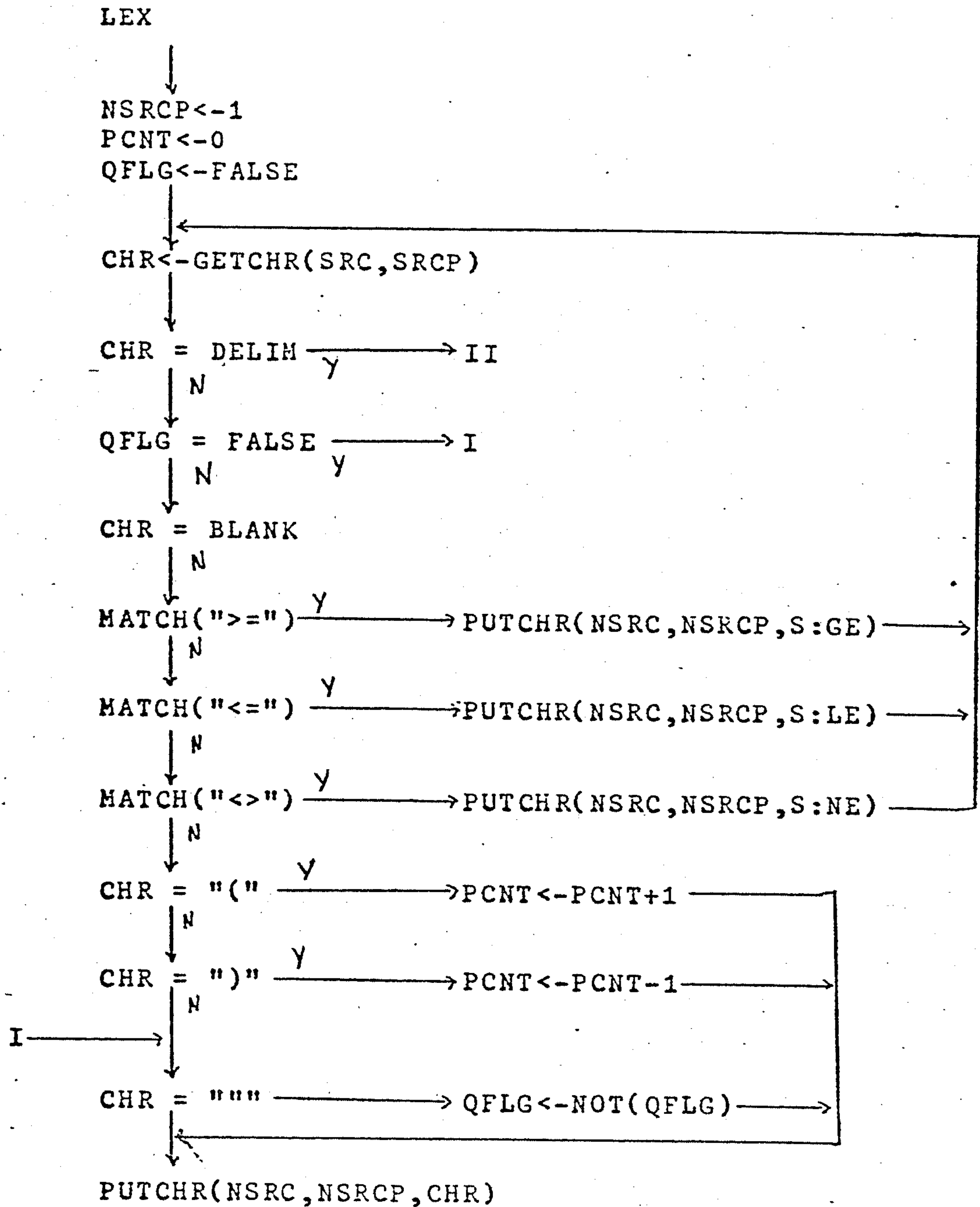
6. Called By:

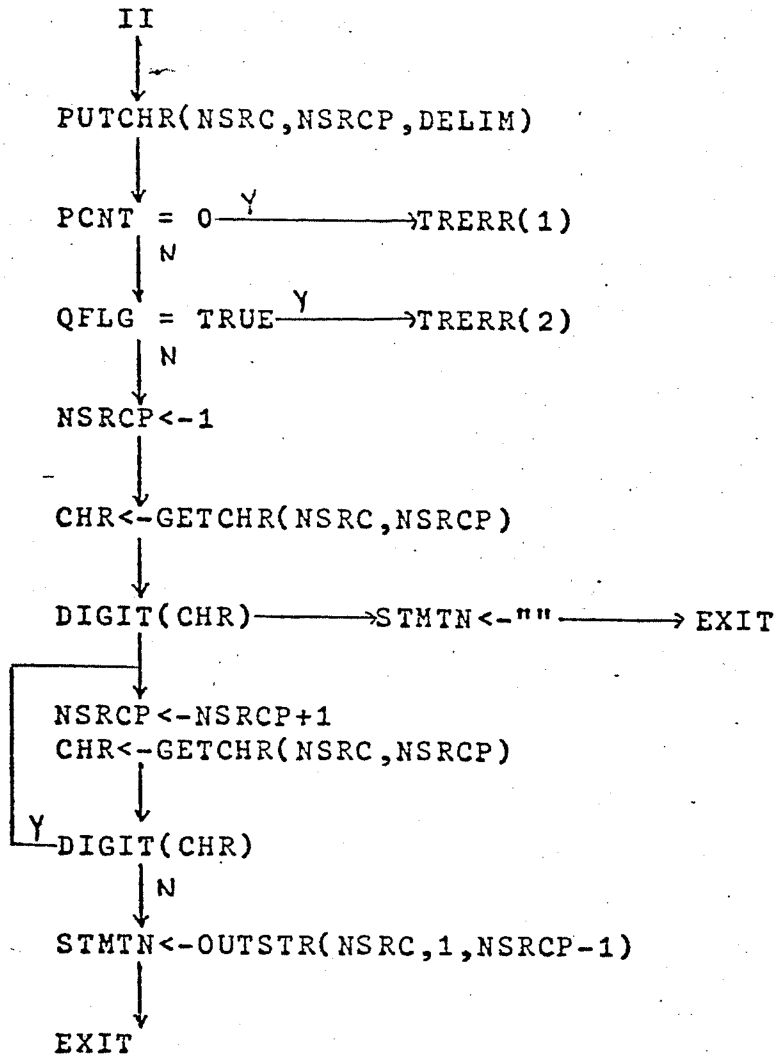
TRNCNTL

7. Error Calls:

TRERR(1): unequal number of parentheses
TRERR(2): unequal number of quotation marks

8. Flowchart:




II
PUTCHR(NSRC,NSRCP,DELIM)
PCNT = 0
Y
TRERR(1)
N
QFLG = TRUE
Y
TRERR(2)
N
NSRCP<-1
CHR<-GETCHR(NSRC,NSRCP)
DIGIT(CHR)
STMTN<-""
EXIT
NSRCP<-NSRCP+1
CHR<-GETCHR(NSRC,NSRCP)
Y
DIGIT(CHR)
N
STMTN<-OUTSTR(NSRC,1,NSRCP-1)
EXIT

1. Name: PARSE

2. Function:

   This procedure identifies the statement type and transfers control to the appropriate coding routine.

3. Description:

   The process of identifying the statement type is performed by extracting strings of increasing length from the source string and comparing them against entries in the key word table (KEYWRD).

   The key word matching process is terminated upon 1) encountering a delimiter, 2) exhausting entries in the key word table, or 3) a successful match.

   If no match is found then a test is made to determine if the statement is an implicit LET, if not an error is generated.

4. Design Alternatives:

   A) Tree structured dictionary of key words.

      Rather than blindly searching and matching on string length this technique would follow a path through the dictionary dictated by the source string. Although certainly faster in matching statements the dictionary is complicated to set up and questionable in what its storage overhead would be.

   B) Partial matching using a key word table.

      This requires looking at less characters and reduces the storage overhead of the table. It is undesirable as some statements are not completely identified and this job is pushed onto later routines.

   C) Table driven parser.

      BASIC does not require the generality.

5. Functions Called:

```
    GETCHR     OUTSTR    LENGTH    GETKEY    STREQ
    DELSTR     (indirectly calls translation procedures)
```

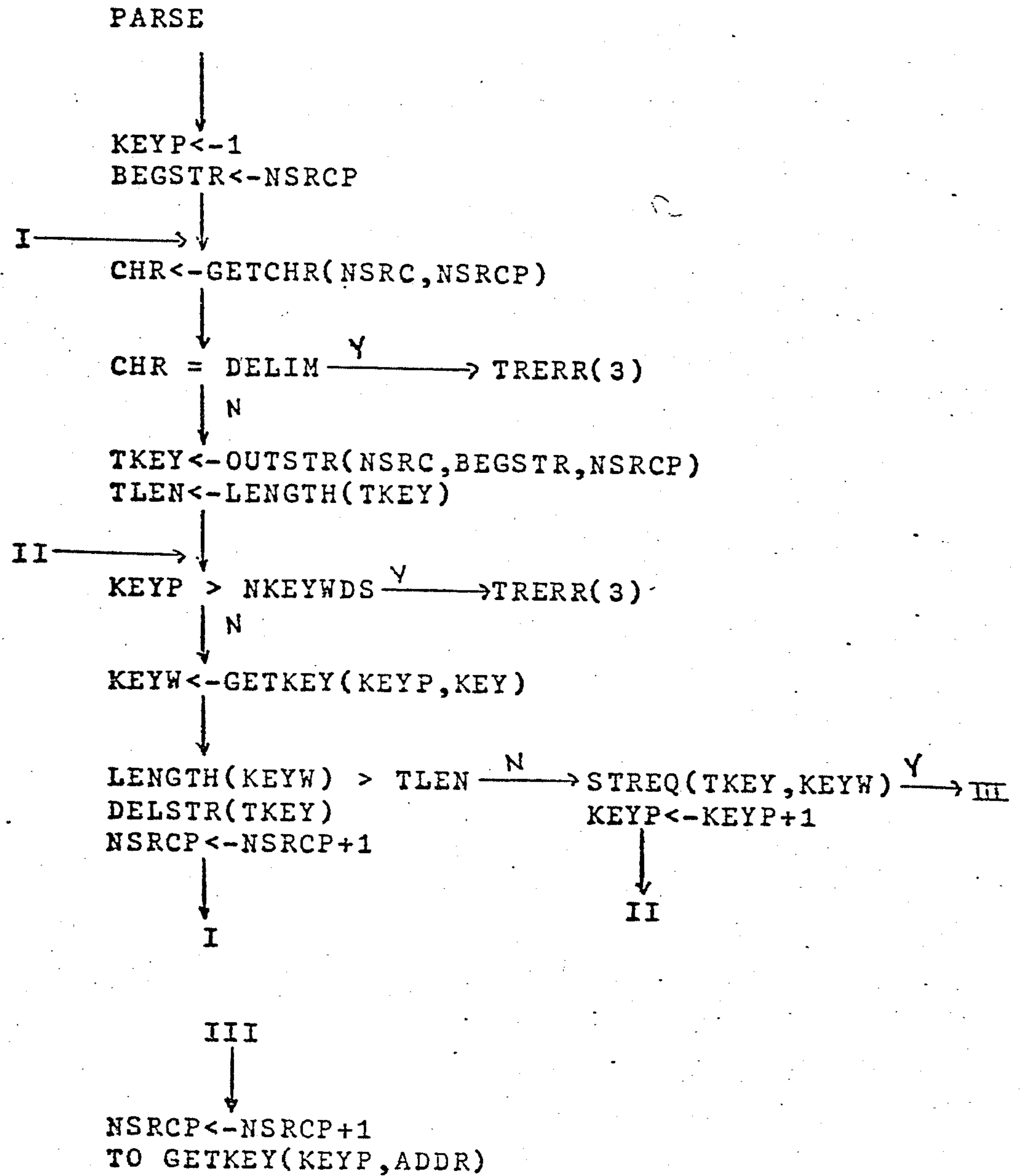
6. Called By:

   LEXICAL

7. Error Calls:

   TRERR(3): unidentifiable statement

8. Flowchart:



1. Name: GETKEY

2. Function:

   Accessing function for the keyword table, KEYWRD.

3. Description:

   The calling sequence is GETKEY(KEYP,ITEM) where KEYP is a relative pointer into the table and ITEM is a flag. If ITEM is 0 then a pointer to a keyword string is returned, if ITEM is 1 then the address of the corresponding translation routine is returned.

4. Design Alternatives: none

5. Functions Called: none

6. Called By:

   PARSE

7. Error Calls: none

8. Flowchart: none

3.3.3.2 Statement Translation Procedures

TLET	- LET statement
TGOTO	- GOTO statement
TFOR	- FOR statement
TNEXT	- NEXT statement
TREAD	- READ statement
TDATA	- DATA statement
TGOSUB	- GOSUB statement
TRETURN	- RETURN statement
TDIM	- DIM statement
TSTOP	- STOP statement
TEND	- END statement
TIF	- IF statement
TINPUT	- INPUT statement
TRESTORE	- RESTORE statement
TPRINT	- PRINT statement
TDEF	- DEF statement
TFNEND	- FNEND statement

1. Name: TLET

2. Function:

   Generate interpretive code for assignment LET statements.

3. Description:

   This procedure generates the interpretive expression (S:ASSIGN var exp).

   S:ASSIGN is pushed onto S-stack and the procedure OPERAND is called to analyze the variable. OPERAND leaves its result on S-stack. If the equal sign is missing an error is generated, otherwise the return address for the call to EXPRESS is stacked and the expression analyzer invoked.

4. Design Alternatives: none

5. Functions Called:

PUSHS	GETCHR	OPERAND	EXPRESS	TRERR
PUSHT	OUTL			

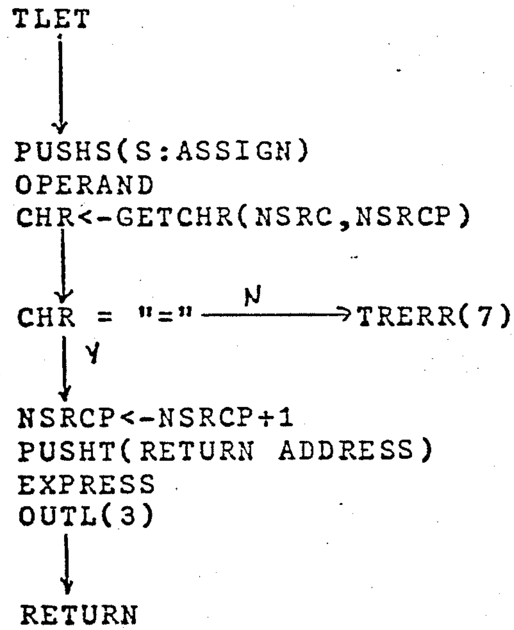
6. Called By:

   PARSE

7. Error Calls:

   TRERR(7): missing equal sign

8. Flowchart:



1. Name: TGOTO

2. Function:

Generate interpretive code for GOTO statements.

3. Description:

The symbol table address of the GOTO function is pushed onto S-stack and the return address for after the call to the expression analyzer is pushed onto T-stack. The expression analyzer is called and the interpretive expression formed by using OUTL.

4. Design Alternatives:

A) Allow only integers as the argument of the GOTO.

B) Use primitive functions to transfer control.

This can be done using a SET :P: and a function that performs a statement number lookup.

5. Functions Called:

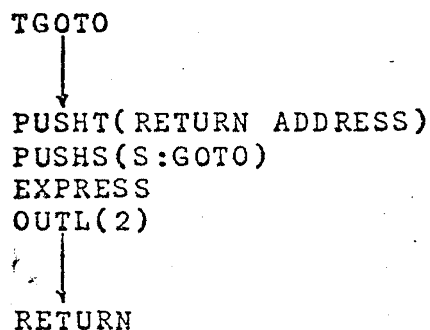
PUSHT PUSHS EXPRESS OUTL

6. Called By:

PARSE TGOSUB

7. Error Calls: none

8. Flowchart:



1. Name: TFOR

2. Function:

Generate interpretive expressions for the FOR statement.

3. Description:

This procedure generates the interpretive expressions

```
(S:ASSIGN var exp1)
(S:FORTEST var exp2 exp3)
(S:PUSH :P:)
```

for BASIC statements of the form

```
FOR var=exp1 TO exp2 STEP exp3
```

The first expression is composed by pushing the address of S:ASSIGN onto S-stack and invoking the procedure OPERAND to analyze the assignment variable. If the equal sign is not present after the variable then an error message is generated, otherwise a return address is stacked and the expression analyzer called. Upon returning from EXPRESS the top three elements of S-stack are now formed into a list expression with a pointer left on S-stack.

A check is made for the prescence of 'TO' and if absent an error message is generated, otherwise a return address is stacked and EXPRESS called to parse the arithmetic bound expression.

If the word 'STEP' is present then a return address is again stacked and EXPRESS used to compile the third expression which is the increment. If no step expression is given then the third expression is set to :NIL:.

The parse of the FOR statement is completed by outputting the top four elements of S-stack and generating a third interpretive expression which pushes a pointer to the current statment onto the S-stack of the interpreter.

4. Design Alternatives:

A) Maintain a FOR/NEXT table to enable linking FOR/NEXT statements when the program is executed.

This adds greater cost in complexity and code since now another table must be in a format for transfer and the requisite routines designed.

B) More intelligence in the interpretive code.

Having the interpretive code push the next expression to evaluate on the stack and having a FINDNEXT function does not reduce the total amount of work done but just distributes it amongst the interpretive code and more special functions.

C) Place a pointer on stack to the expression to evaluate after the initialization code.

This would require either a table of FOR/NEXT entries or scanning the users program. The first alternative is undesirable for program mobility the second because of our desire to limit pre-interpretation overhead.

5. Functions Called:

PUSHS	PUSHT	MATCH	GETCHR	OUTL
EXPRESS	OPERAND			

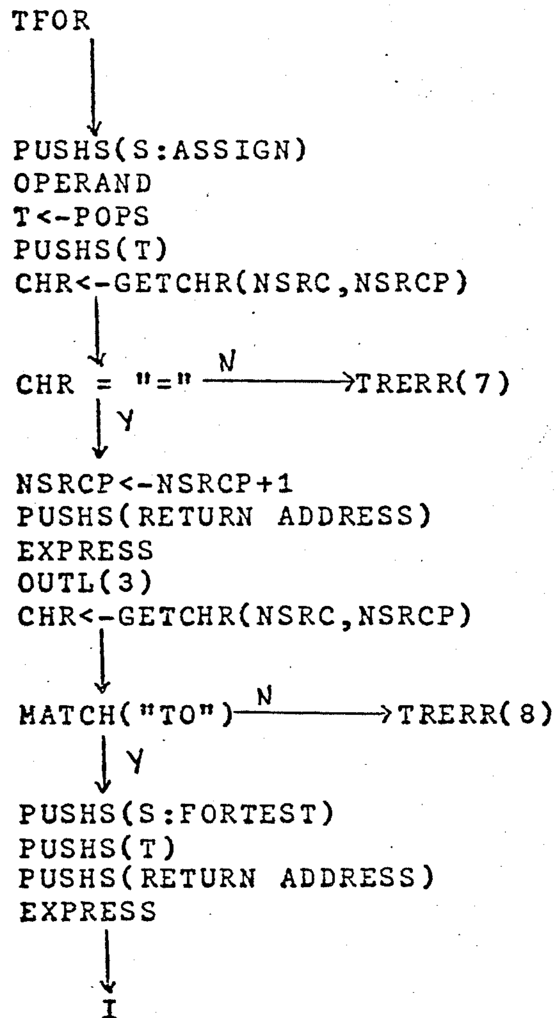
6. Called By:

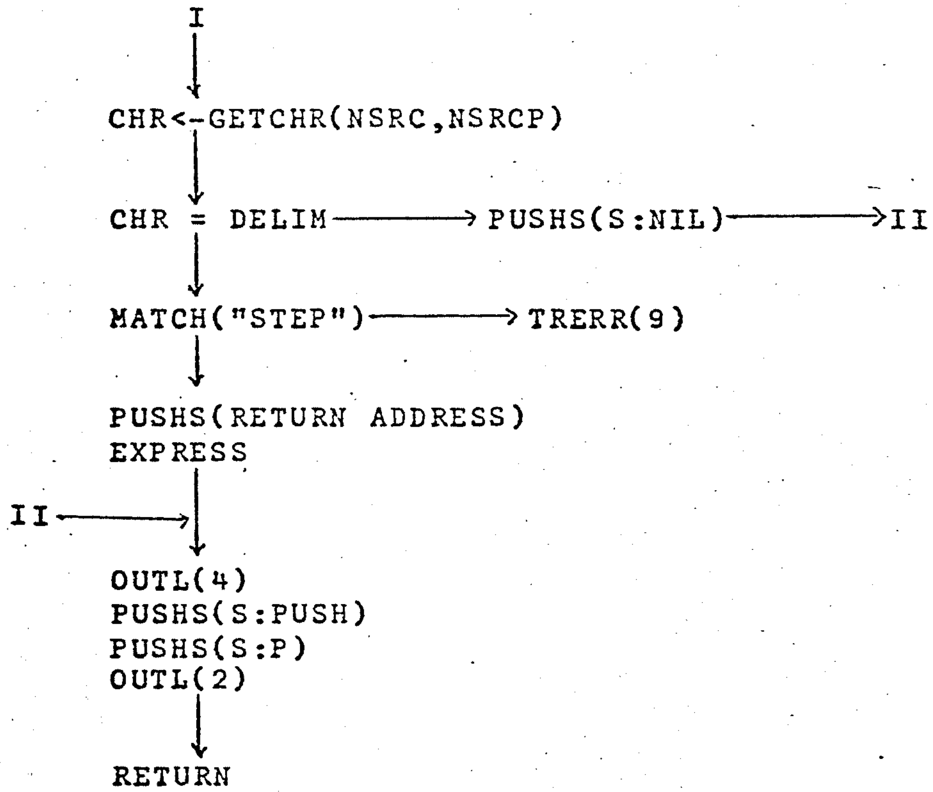
PARSE

7. Error Calls:

TRERR(7): missing equals sign
TRERR(8): missing 'TO' in a FOR statement

8. Flowchart:




I
CHR<-GETCHR(NSRC,NSRCP)
CHR = DELIM
PUSHS(S:NIL)
II
MATCH("STEP")
TRERR(9)
PUSHS(RETURN ADDRESS)
EXPRESS
II
OUTL(4)
PUSHS(S:PUSH)
PUSHS(S:P)
OUTL(2)
RETURN

1. Name: TNEXT

2. Function:

   Generate interpretive code for NEXT statements.

3. Description:

   This procedure compiles interpretive code that sets the flag FORF on and sets the program pointer to the entry on top of the S-stack. FORF indicates whether the loop has been initialized.

4. Design Alternatives:

   A) Use a FOR/NEXT table.

      This allows us to go directly to the correct FOR expression whithout storing any pointers on the stack, it requires saving and moving an additional table and saves negligible computation time.

   B) Eliminate the FOR flag.

      This requires that the pointer on the stack be a pointer to the interpretive expressions that follow the loop initialization. To do this the interpreter must be modified to allow one to jump into the middle of a statement expression, thus bypassing initialization that allows us to transfer context. The restriction on mobility is undesirable.

5. Functions Called:

      PUSHS   OUTL

6. Called By:

      PARSE

7. Error Calls: none

8. Flowchart:

```
T:NEXT
  |
  v
PUSHS(S:SET)
PUSHS(S:FORF)
PUSHS(S:TRUE)
OUTL(3)
PUSHS(S:SET)
PUSHS(S:P)
PUSHS(S:POP)
OUTL(1)
OUTL(3)
  |
  v
EXIT
```

1. Name: TREAD

2. Function:

   Generate interpretive expressions for READ statements.

3. Description:

   Each parameter in the READ statement causes an interpretive expression of the form (S:ASSIGN (S:READD)) to be generated.

   The interpretive expressions are generated by pushing S:ASSIGN onto the S-stack and invoking the procedure OPERAND. S:READD is then pushed on S-stack and OUTL used to form a list. The complete expression is then formed using OUTL(3). If a comma is present, another element is assumed and the procedure repeated.

4. Design Alternatives: none

5. Functions Called:

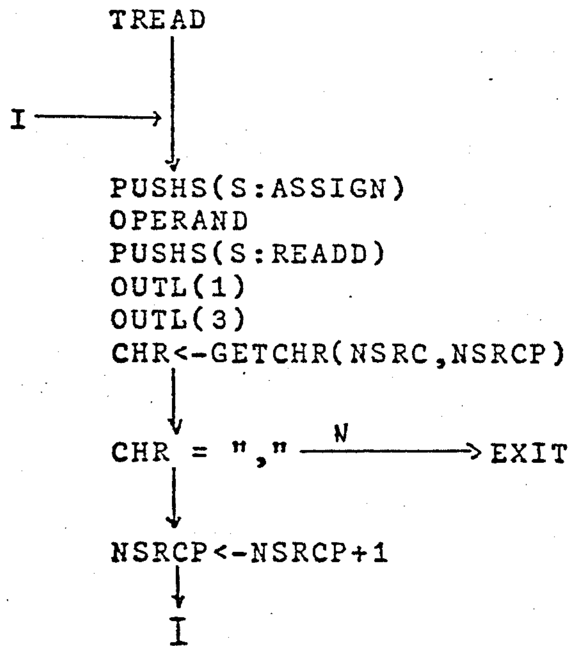
   OPERAND PUSHS GETCHR OUTL

6. Called By:

   PARSE

7. Error Calls: none

8. Flowchart:



1. Name: TDATA

2. Function:

Generate interpretive expressions for the DATA statement.

3 Description:

A DATA statement generates one interpretive expression. The number of elements in the expression corresponds to the number that appeared in the original statement. The items in a DATA statement may be general expressions.

4. Design Alternatives:

5. Functions Called:

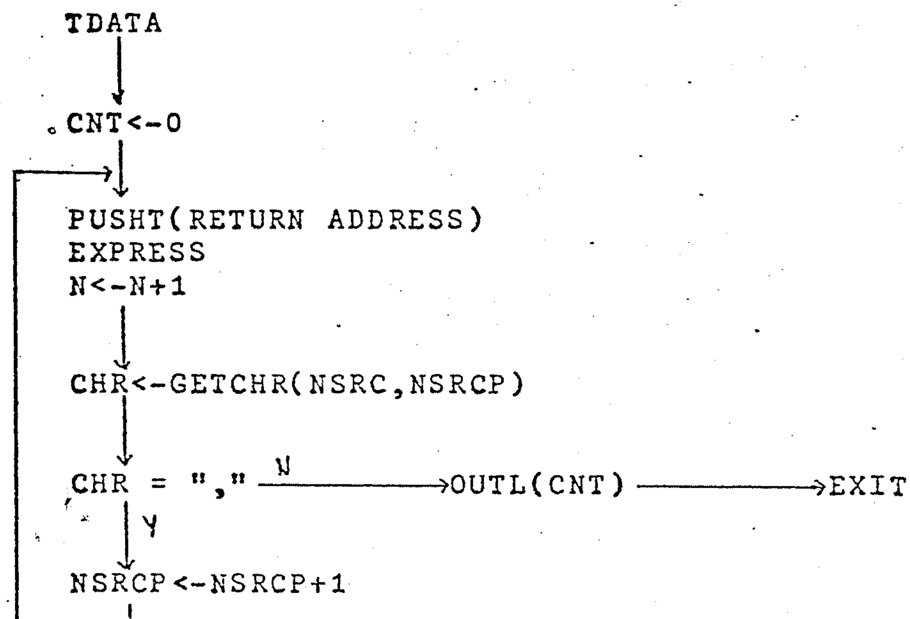
EXPRESS GETCHR OUTL PUSHT

6. Called By:

PARSE

7. Error Calls: none

8. Flowchart:



1. Name: TGOSUB

2. Function:

   Generate interpretive code for GOSUB statements.

3. Description:

   The expression (S:PUSH (S:CDR :P:)) is formed and the procedure TGOTO invoked to generate the transfer expression.

4. Design Alternatives:

   The program list evaluator, EVALP, could be called recursively and modified to terminate upon encountering a RETURN statement.

5. Functions Called:

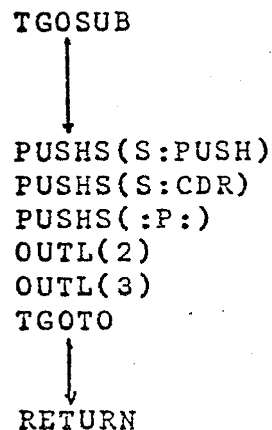
   PUSHS OUTL TGOTO

6. Called By:

   PARSE

7. Error Calls: none

8. FLOWCHART:



1. Name: TRETURN

2. Function:

   Generate interpretive code for the RETURN statement.

3. Description:

   The interpretive expression generated is (S:SET :P: (S:POP).

4. Design Alternatives: see TGOSUB

5. Functions Called:

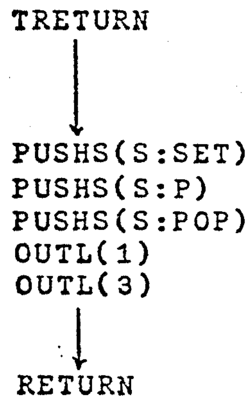
   PUSHS   OUTL

6. Called By:

   PARSE

7. Error Calls: none

8. Flowchart:



1. Name: TDIM

2. Function:

Generate interpretive code for the DIM statement.

3. Description:

An interpretive expression is generated for each variable that occurs in the dimension statement. The DIM function expects three arguments which are the array name and the dimensions for this array. If the second dimension is missing this is indicated by the translator inserting :NIL:.

Since the DIM statement is treated as an executable statement that dynamically allocates storage to the named variable the dimensions may be general expressions.

4. Design Alternatives:

A) Restrict the dimensions to be explicit integer values.

By allowing generality it actually simplifies the number of data types the interpreter must handle. The dimensions are now floating point values. Since we can provide dynamic allocation there is no reason to proscribe its use.

B) Create a table of dimensioned variables.

This is only necessary if a fixed amount of storage is to be allocated. In this case we must know about implicitly dimensioned variables as well as those that appear in DIM statements. If the ADDR function encounters a variable for which no storage is allocated that at execution time it can invoke the DIM function directly to accomplish the allocation.

C) Generate one variadic interpretive expression.

The overhead of multiple expressions is balanced by simplifying the type of functions the interpreter must handle.

5. Functions Called:

PUSHS	PUSHT	GETCHR	ALPHA	OUTSTR
CONCAT	EXPRESS	OUTL		

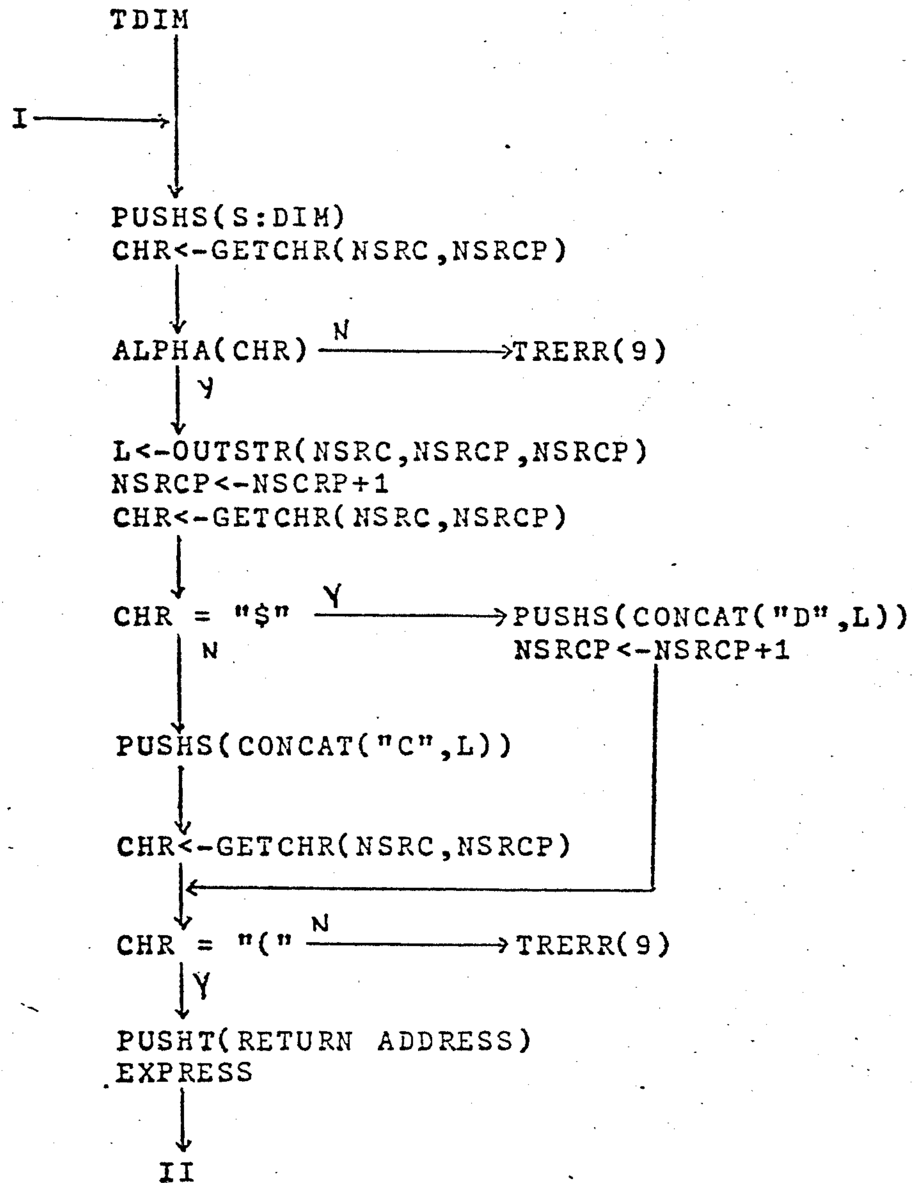
6. Called By:

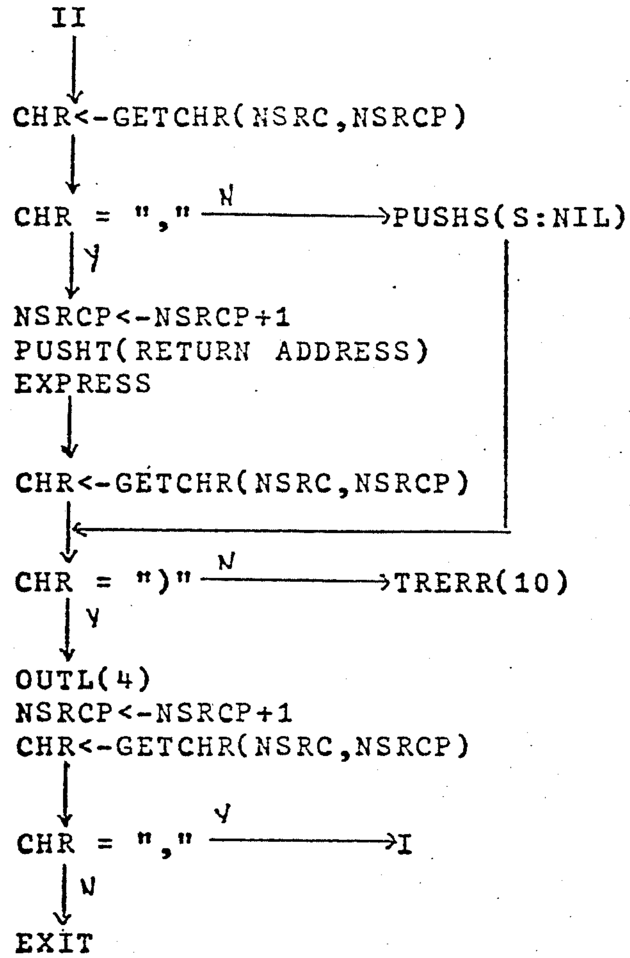
PARSE

7. Error Calls:

TRERR(9): Array names must be a single letter
TRERR(10): Arrays may have no more than two dimensions

8. Flowchart:





II
CHR<-GETCHR(NSRC,NSRCP)
CHR = "," N
PUSHS(S:NIL)
Y
NSRCP<-NSRCP+1
PUSHT(RETURN ADDRESS)
EXPRESS
CHR<-GETCHR(NSRC,NSRCP)
CHR = ")" N
TRERR(10)
Y
OUTL(4)
NSRCP<-NSRCP+1
CHR<-GETCHR(NSRC,NSRCP)
CHR = "," Y
I
N
EXIT

1. Name: TSTOP

2. Function:

Generate interpretive code for STOP statement.

3. Description:

Creates code that causes a program halt.

4. Design Alternatives: none

5. Functions Called:

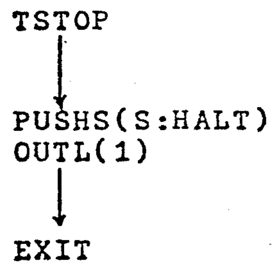
PUSHS OUTL

6. Called By:

PARSE

7. Error Calls: none

8. Flowchart:



1. Name: TEND

2. Function:

   Generate interpretive code for END statements.

3. Description:

   Creates code that causes a program halt.

4. Design Alternatives: none

5. Functions Called:

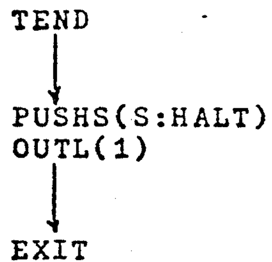
   PUSHS OUTL

6. Called By:

   PARSE

7. Error Calls: none

8. Flowchart:



1. Name: TIF

2. Function:

Generate interpretive expressions for the IF statement.

3. Description:

TIF generates a single expression of the form (S:IF exp1 exp2). Whenever exp1 evaluates to TRUE the IF function sets :P: to the statement form corresponding to exp2, in all other cases :P: is left unchanged and control follows its normal path.

4. Design Alternatives:

Two expressions could be formed, the first a test that causes the second to be evaluated only when the first expression is logically true.

5. Functions Called:

PUSHT EXPRESS MATCH OUTL

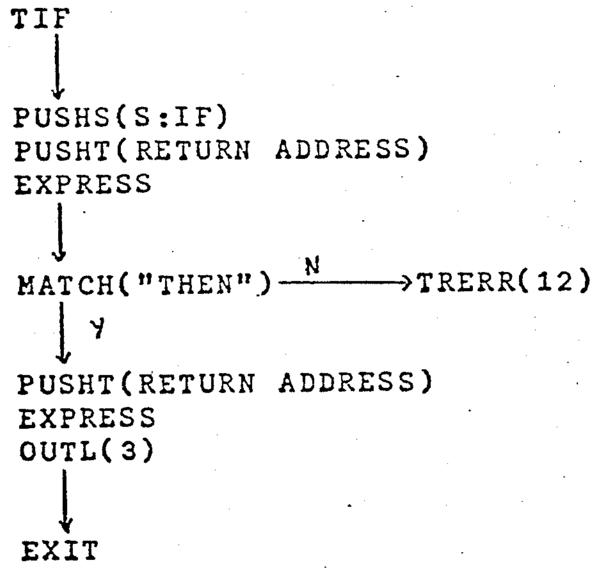
6. Called By:

PARSE

7. Error Calls:

TRERR(12): the word THEN is misspelled or missing

8. Flowchart:



1. Name: TINPUT

2. Function:

   Generate interpretive expressions for INPUT statements.

3. Description:

   Each paramter in the INPUT statement causes an interpretive expression of the form (S:ASSIGN var (S:INPUTD)) to be generated.

   The interpretive expressions are generated by pushing S:ASSIGN onto the S-stack and invoking the procedure OPERAND. S:INPUTD is then pushed on the S-stack and OUTL used to form a list. The complete expression is then formed using OUTL(3). If a comma is present, another element is assumed and the procedure repeated.

4. Design Alternatives: none

5. Functions Called:

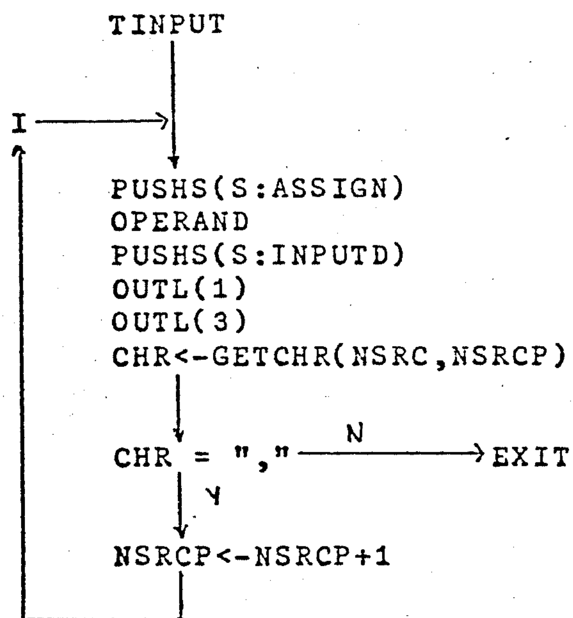
   OPERAND PUSHS GETCHR OUTL

6. Called By:

   PARSE

7. Error Calls: none

8. Flowchart:



1. Name: TRESTORE

2. Function:

Generate interpretive expression for the RESTORE statement.

3. Description:

The expression (S:SET :DATAP: :PROGP:) is formed. It causes the data list pointer, :DATAP:, to be restored to the beginning of the program list.

4. Design Alternatives: see TDATA

5. Functions Called:

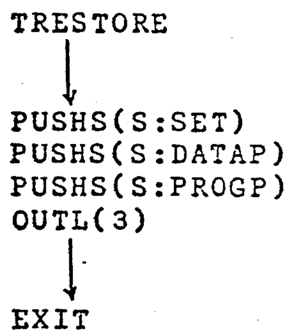
PUSHS OUTL

6. Called By:

PARSE

7. Error Calls: none

8. Flowchart:



1. Name: TPRINT

2. Function:

Generate interpretive expressions for PRINT statements.

3. Description:

TPRINT generates one interpretive expression for each PRINT statement. The number of elements in that expression depends on the parameter list of the PRINT statement.

An element of the list may be an expression or a print character delimiter (comma or semicolon). PRINT statements are evaluated by having the runtime print routine perform its output using the values placed on the R-stack after the PRINT statement's argument list has been evaluated.

The print character delimiters on the R-stack control the formatting of the output for the PRINT statement.

4. Design Alternatives:

A) Generate a separate interpretive expression for each parameter in the PRINT statement.

This causes redundant calls of the runtime print function when all the activity could be accomplished in one invocation.

B) Pass a list of expressions and a string of print control characters to the runtime procedure.

This requires more processing by the runtime procedure to determine the correct formatting.

5. Functions Called:

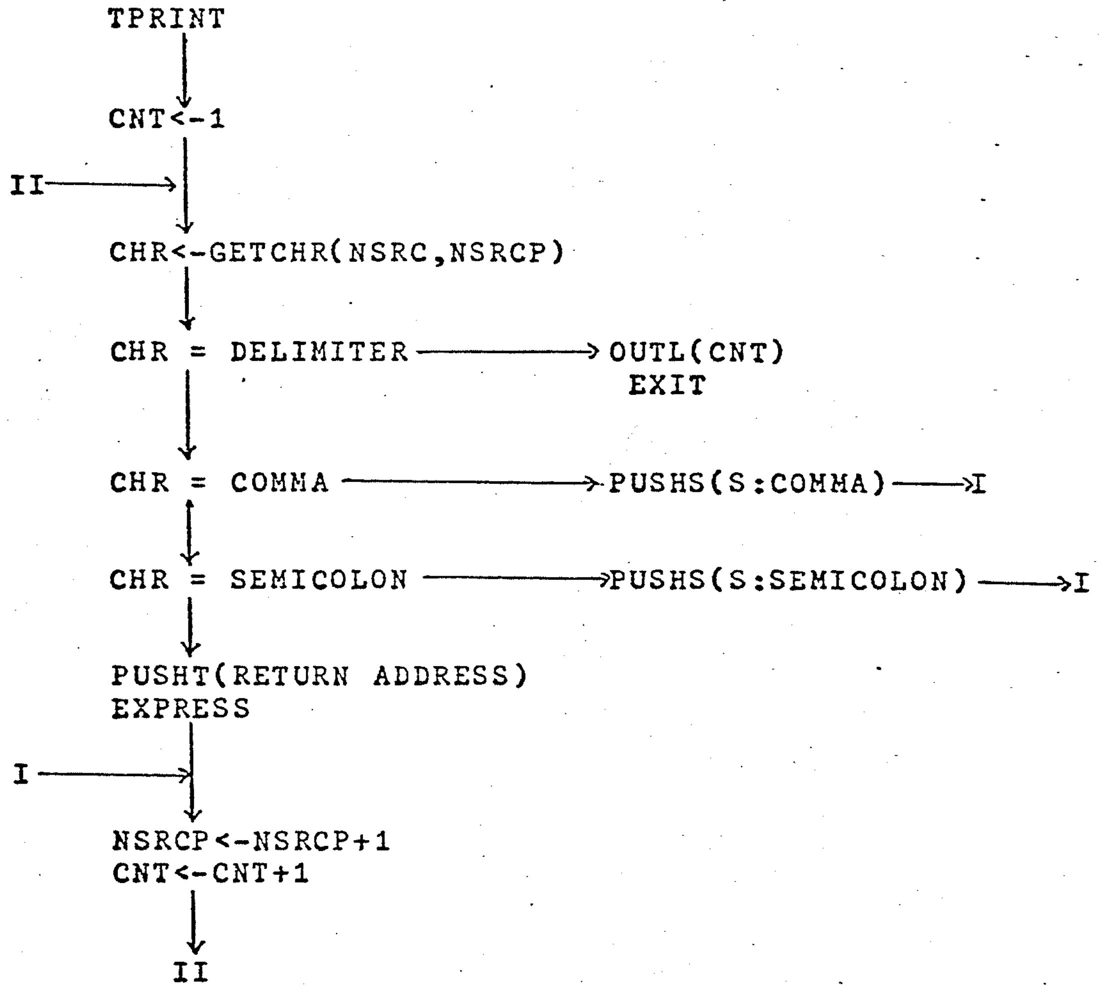
GETCHR PUSHS EXPRESS OUTL PUSHT

6. Called By:

PARSE

7. Error Calls: none

8. Flowchart:



1. Name: TDEF

2. Function:

Produce interpretive expressions for DEF statements.

3. Description:

Single line function definition statements generate two interpretive expressions, a parameter list and a function expression. Multiple line function definitions generate only a parameter list.

The DEF statement form is not directly evaluated but is entered upon encountering an occurence of a function during evaluation of a statement form.

Evaluation of single line functions is accomplished by binding arguments and evaluating the function expression. Multiple line functions are evaluated by binding the argument list and recursively calling the program list evaluator, EVALP. Subsequent statement forms are evaluated until a FNEND statment is encountered at which time variables are unbound and a return executed.

Variable binding and program control is under control of the evaluation routines.

4. Design Alternatives:

Single and multiple line functions can be distinquished by using separate statement types or inspecting the list of interpretive expressions. This design uses both technieques.

5. Functions Called:

MATCH GETCHR PUSHT OPERAND EXPRESS
OUTL

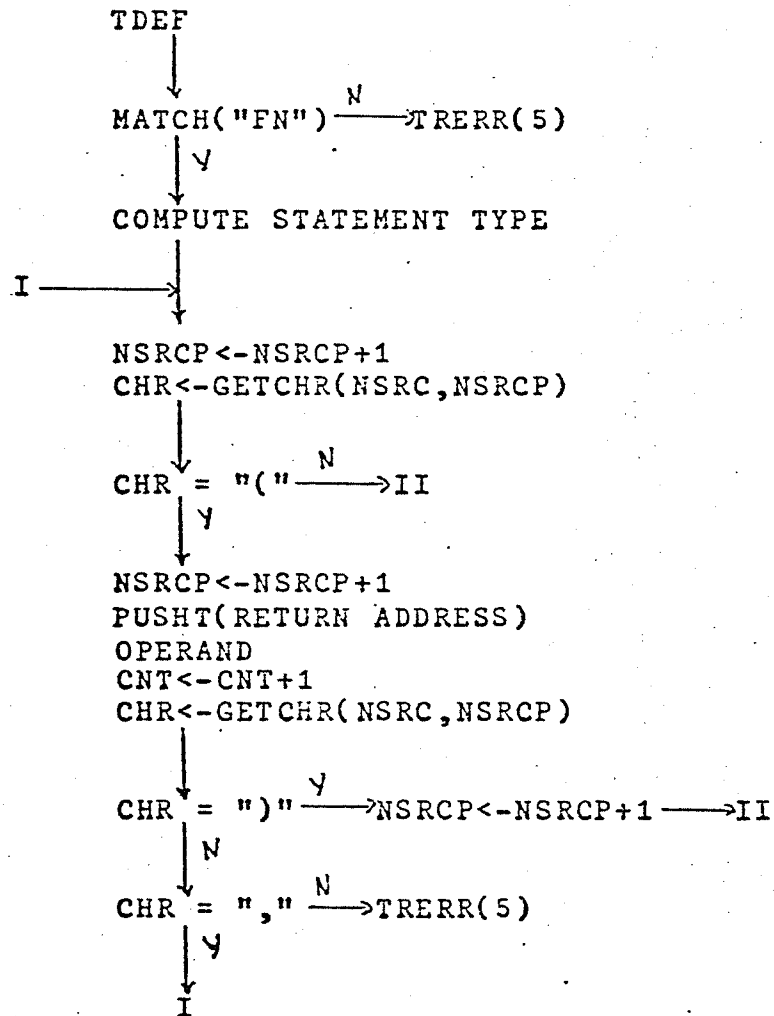
6. Called By:

PARSE

7. Error Calls:

TRERR(5): malformed function header

8. Flowchart:



```
   II
    |
    v
OUTL(CNT)
CHR<-<-GETACHR(NSRC,NSRCP)
    |
    v
CHR = "=" ---N--->TRERR(5)
    | Y
    v
NSRCP<-NSRCP+1
PUSHT(RETURN ADDRESS)
EXPRESS
OUTL(1)
    |
    v
EXIT
```

1. Name: TFNEND

2. Function:

Generate interpretive expressions for FNEND statement.

3. Description:

The FNEND statement does not generate any interpretive expressions. It is recognized by its statement type by the evaluation routines and terminates a program list evaluation.

4. Design Alternatives:

Unbinding of arguments could be made a part of the FNEND statements actions but to be consistent a dummy FNEND would then be required for single line function definitions.

5. Functions Called:

6. Called By:

PARSE

7. Error Calls: none

8. Flowchart: none

3.3.3.3 Expression Analysis

EXPRESS	- Expression Parser
OPERAND	- Operand Analyzer
SUBTRAN	- Subscript Translator
FUNCTRAN	- Function Translator
STRING	- Process String Operands
NUMBER	- Process Numeric Operands
CODE	- Code Selector
BINARY	- Code Generator, Binary Operators
FPREC	- F Precedence Function
GPREC	- G Precedence Function

1. Name: EXPRESS

2. Function:

Parse arithmetic and logical expressions producing the corresponding interpretive expressions.

3. Description:

An operator precedence parser is used to translate expressions into the desired prefix form. This procedure may be called recursively to analyze expressions embedded in subscripts and functions.

The parser uses the precedence functions FPREC and GPREC to compare precedence values of operators on the stack and those in the input stream, respectively.

Upon entering the procedure a delimiter is place on the S-stack. This delimiter is used to flag the completed analysis of an expression. If the first character is either a minus sign, negation operator or a left parenthese it is stacked and the source pointer (NSRCP) advanced. We return to get the next character and test again for these characters.

If any of these operators are present they are stacked, otherwise we stack a return address and invoke the OPERAND procedure. OPERAND returns to us after placing the appropriate operand expression on the S-stack.

After returning from OPERAND the source pointer must be at some operator or a character that will cause us to process the operators and operands on the stack thus completing the parse. A precedence test is performed and either the algorithm continues scanning the string and stacking operators and operands or causes some of the items on the stack to be evaluated and formed into what will become the interpretive expression. The procedure CODE is used to pop an operator off the stack and generate the correct expression.

Expression analysis is ended when the precedence test indicates the stack should be evaluated and the indicated operator is the delimiter.

4. Design Alternatives:

An early design goal of facilitating implementation of a variety of languages in this system prompted considering the value of embedding a general table driven parser in the translator. When this design goal was subjugated we resolved to use a simple and efficient operator precedence parser.

The very tight structure of the parser permitted some deviation from previous standards involving information control. In the expression, parser substructure results are sometimes passed by placing them on the S-stack rather than returning pointers which then could be stacked.

5. Functions Called:

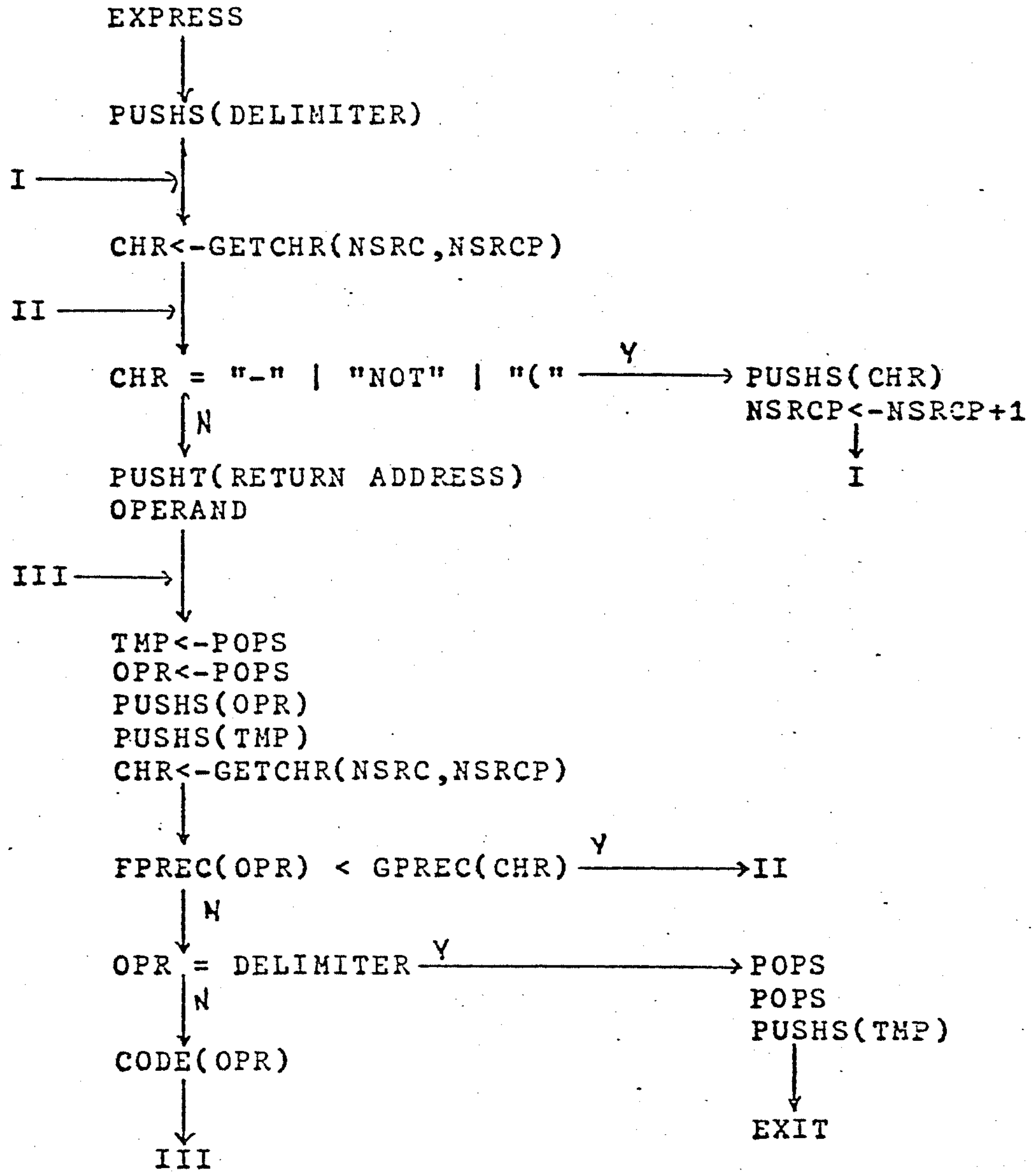
PUSHS	PUSHT	GETCHR	FPREC	GPREC
OPERAND	CODE	POPS	POPT	

6. Called By:

TGOTO	TGOSUB	TLET	TFOR	TIF
TDIM	TDATA	TPRINT	TDEF	SUBTRAN
FUNCTRAN				

7. Error Calls: none

8. Flowchart:



1. Name: OPERAND

2. Function:

This procedure forms the appropriate string expression for all legal BASIC operands. Data and symbol type information is encoded into the expression string.

3. Description:

The operand procedure performs complete processing for simple string and numeric variables and calls other procedures to handle strings, numbers, subscripted variables and function calls after they are identified. The output of OPERAND is always left on the S-stack.

OPERAND is invoked with the source pointer at the beginning of a presumed legal operand. Malformed operands will cause error messages to be issued.

4. Design Alternatives: none

5. Functions Called:

PUSHS	PUSHT	POPS	POPT	GETCHR
DIGIT	NUMBER	STRING	ALPHA	OUTSTR
CONCAT	SUBTRAN	FUNCTION	FUNCTRAN	OUTL

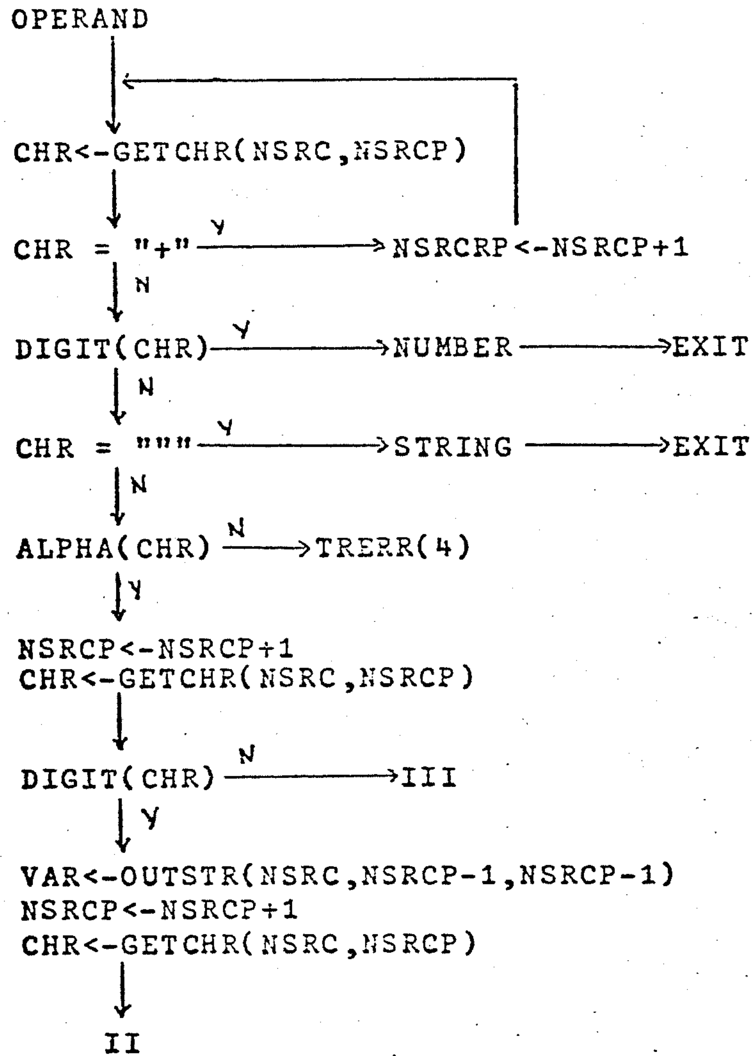
6. Called by:

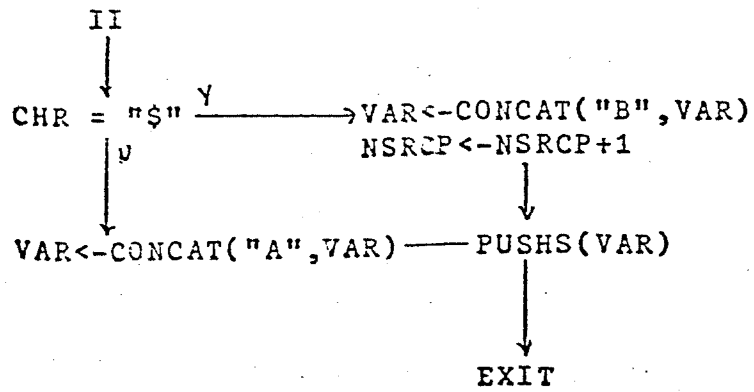
EXPRESS	TREAD	TLET	TFOR	TNEXT
TINPUT	TDEF			

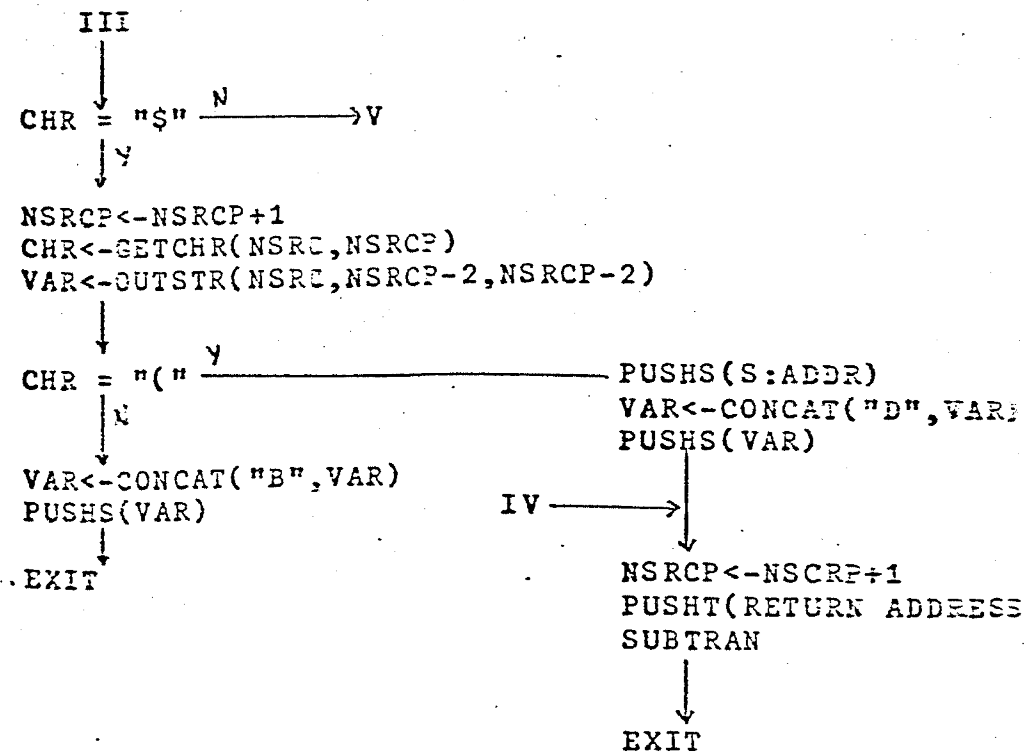
7. Error Calls:

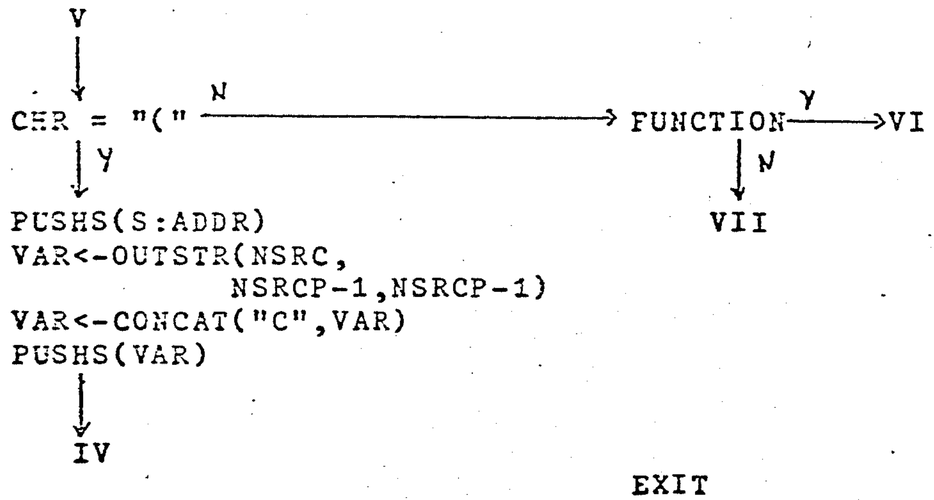
TRERR(4): malformed operand

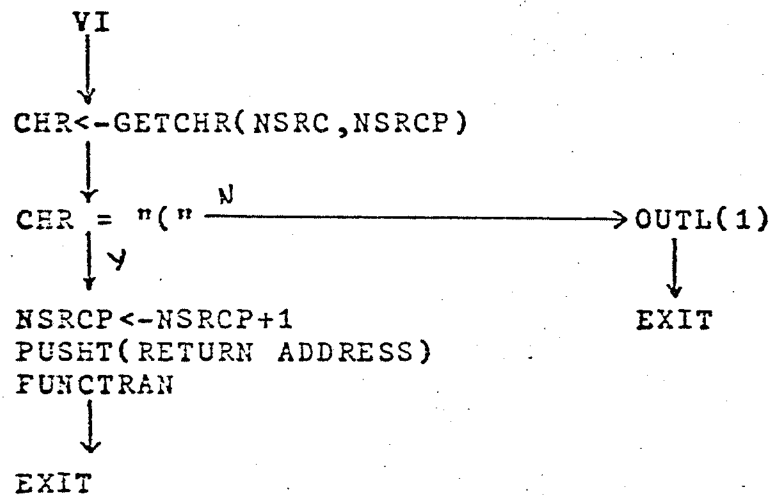
8. Flowchart:

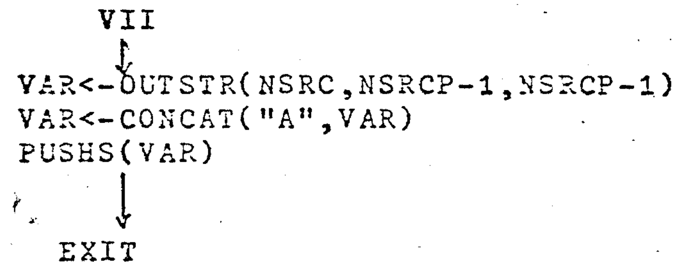



II
CHR = "$"
Y
VAR<-CONCAT("B",VAR)
NSRCP<-NSRCP+1
N
VAR<-CONCAT("A",VAR)
PUSHS(VAR)
EXIT


III
CHR = "$"
N
V
Y
NSRCP<-NSRCP+1
CHR<-GETCHR(NSRC,NSRCP)
VAR<-OUTSTR(NSRC,NSRCP-2,NSRCP-2)
CHR = "("
Y
PUSHS(S:ADDR)
PUSHS(VAR)
N
VAR<-CONCAT("B",VAR)
PUSHS(VAR)
EXIT
IV
NSRCP<-NSCRP+1
SUBTRAN
EXIT


V
CHR = "("
N
FUNCTION
Y
VI
Y
N
PUSHS(S:ADDR)
VAR<-OUTSTR(NSRC,
NSRCP-1,NSRCP-1)
VAR<-CONCAT("C",VAR)
PUSHS(VAR)
VII
IV
EXIT


VI
CHR<-GETCHR(NSRC,NSRCP)
CHR = "("
N
OUTL(1)
Y
EXIT
NSRCP<-NSRCP+1
PUSHT(RETURN ADDRESS)
FUNCTRAN
EXIT


VII
VAR<-OUTSTR(NSRC,NSRCP-1,NSRCP-1)
VAR<-CONCAT("A",VAR)
PUSHS(VAR)
EXIT

1. Name: SUBTRAN

2. Function:

   Analyzer and code generation procedure for subscripted variables.

3. Description:

   SUBTRAN is invoked with the pointer NSRCP positioned at the first character following the left parenthese. The symbol table address of the ADDR function and the name of the subscripted variable are already on S-stack.

   The result of the subscript translation is left on S-stack and NSRCP is advanced beyond the subscript's matching right parenthese. Since SUBTRAN may be called recursively through EXPRESS it is necessary to save a return address when invoking it.

   Subscripts are translated by invoking EXPRESS to analyze the first argument. If the next character on return is a comma then EXPRESS is invoked a second time, if not a symbol table pointer to :NIL: is stacked. The subscripted expression is completed by outputting the top four elements on the stack as the completed list expression.

4. Design Alternatives:

   The ADDR function could be modified to accept a variable number of expressions. If this were done then we would not have to supply a dummy parameter. This requires implementing function calls that do not evaluate their arguments in the interpreter. The savings in work in not implementing this type of function call is balanced against less flexibility for future extensions.

5. Functions Called:

   EXPRESS GETCHR PUSHT OUTL PUSHS

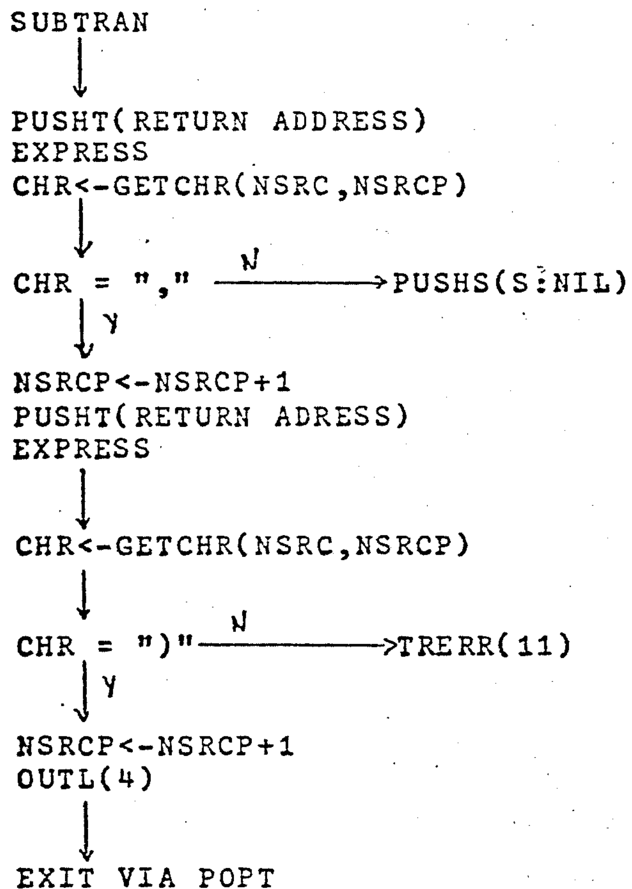
6. Called By:

   OPERAND

7. Error Calls:

TRERR(11): malformed subscripted expression

8. Flowchart:



1. Name: FUNCTRAN

2. Function:

   Form interpretive expressions for functions.

3. Description:

   FUNCTRAN is invoked with NSRCP positioned at the first character of the first argument. The function name has already been placed on S-stack. Since FUNCTRAN analyzes its arguments by calling EXPRESS a return address must be stacked in case of recursive calls.

   Functions can have an arbitrary number of arguments. The count for the number of arguments is kept on S-stack during parsing. The argument count, NARGS is pushed onto the stack before EXPRESS is called and then updated and replaced upon return.

4. Design Alternatives:

5. Functions Called:

PUSET	PUSHS	GETCHR	EXPRESS	OUTL
POPT	POPS			

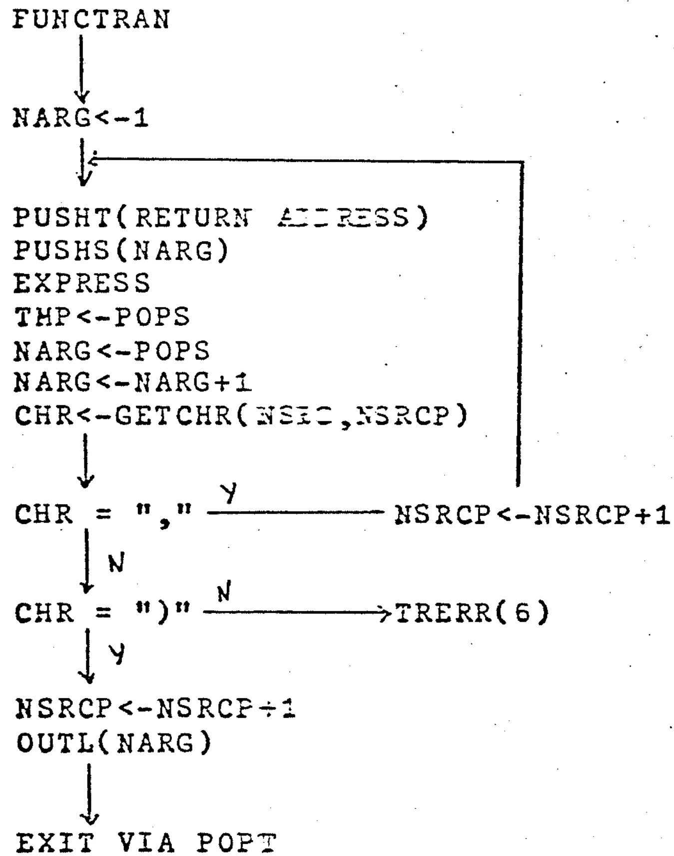
6. Called By:

   OPERAND

7. Error Calls:

   TRERR(6): malformed function expression

8. Flowchart:



1. Name: STRING

2. Function:

   Process string constants and leave a pointer to the operand on S-stack.

3. Description:

   The procedure STRING is invoked with the pointer NSRCP at a quote mark which delimits the beginning of the string. All characters between the initial quote mark and the delimiting quote are formed into a string and prefaced with the letter G (to flag a string constant). A pointer to the string is pushed on S-stack.

   No error checking is performed. The procedure assumes the delimiting quote will be present.

4. Design Alternatives:

   The alternative to including the quote marks surrounding the string in the processed operand was to eliminate them but this necessitates either including some information on the length of the string or a delimiter character. The first option is unwielding and the quote marks already perform the second function.

5. Functions Called:

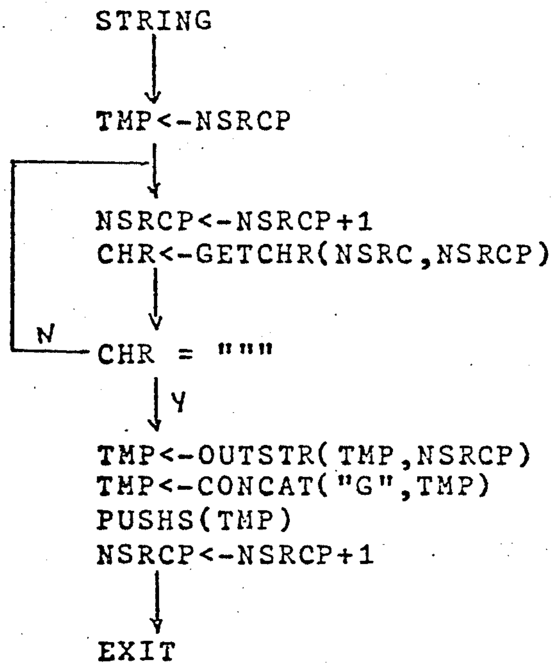
   GETCHR PUSHS OUTSTR CONCAT

6. Called By:

   OPERAND

7. Error Calls: none

8. Flowchart:



1. Name: NUMBER

2. Function:

Process operands that are floating point constants.

3. Description:

The procedure NUMBER is invoked with the pointer, NSRCP at the first digit of the number. The procedure accepts numbers as being syntactically correct if they are of the form <flonum>[E<integer>], where <flonum>=<integer>[.<integer>]. A string representing the number is output and left on the stack. The output string is prefaced with an "F" to flag it as a floating point constant to the interpreter's READ procedure.

4. Design Alternatives:

5. Functions Called:

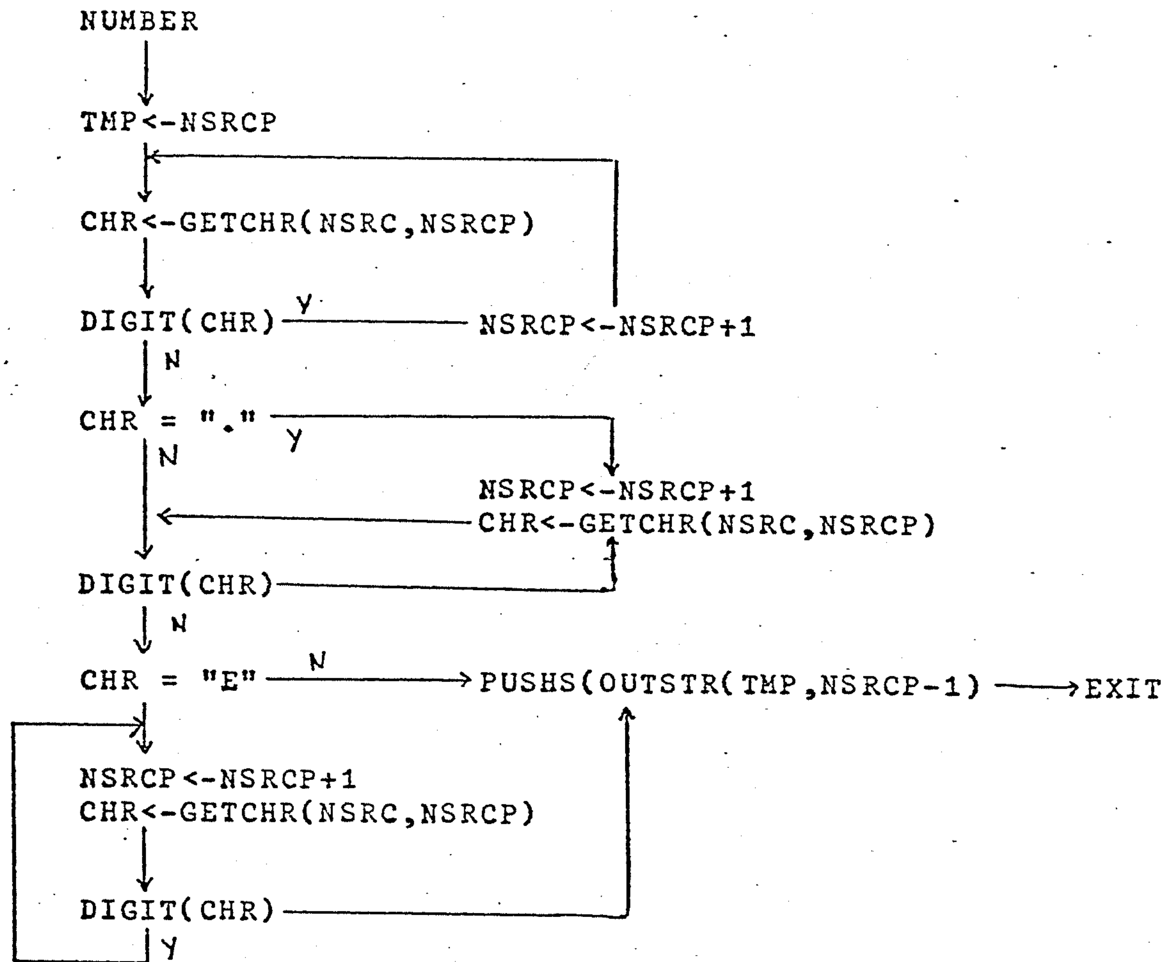
GETCHR CONCAT CUTSTR DIGIT PUSHS

6. Called By:

OPERAND

7. Error Calls: none

8. Flowchart:



1. Name: CODE

2. Function:

   Forms the appropriate list expression based on the operator being coded.

3. Description:

   CODE has one argument, OPR, which must be a legal arithmetic or relational operator.

   CODE uses the contents of the S-stack to form expressions. If OPR is not a unary operator then BINARY is used to determine the correct symbol table name.

4. Design Alternatives: none

5. Functions Called:

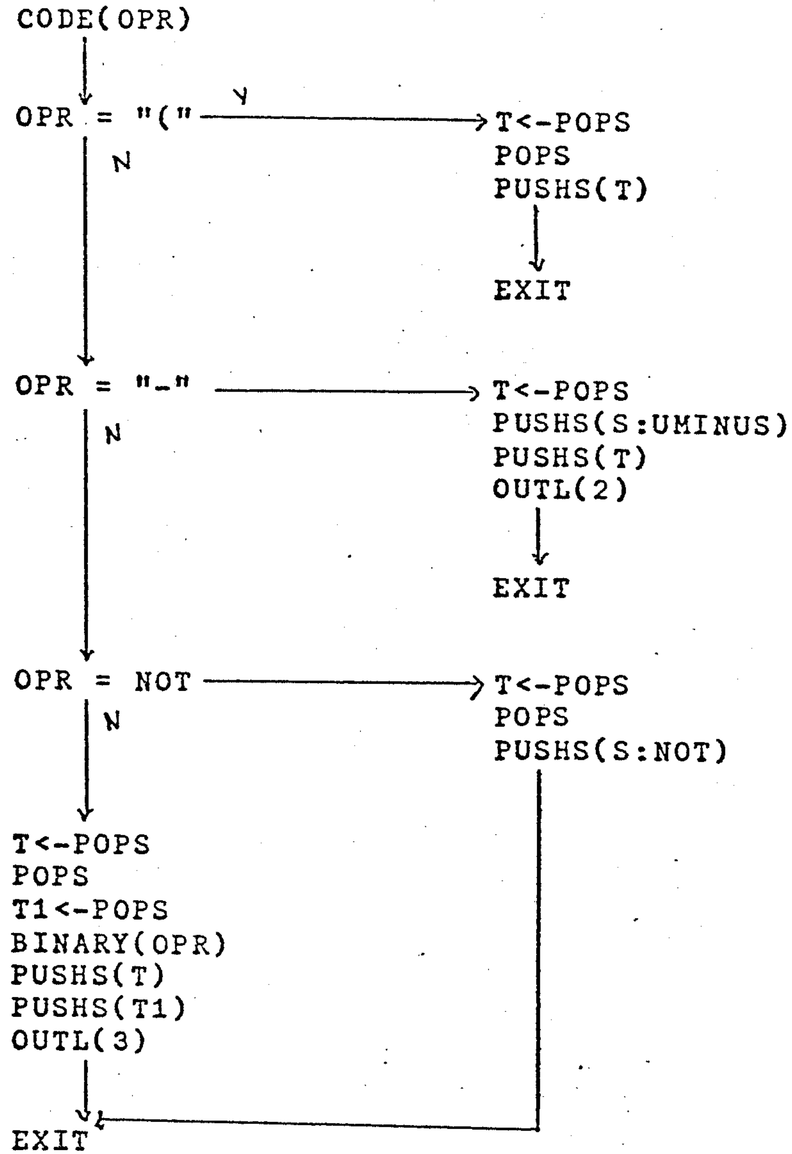
   POPS PUSHS OUTL BINARY

6. Called By:

   EXPRESS

7. Error Calls: none

8. Flowchart:



1. Name: BINARY

2. Function:

   Return the symbol table function (location) for a binary operator.

3. Description:

   BINARY takes one input, a code for a legal binary operator. The binary operator table, BINOPTB, is searched and a pointer to the symbol table location string is returned.

4. Design Alternatives:

   By choosing appropriate token values for the operators they can be used either 1) to directly access the operator table or 2) their value can be the location, thus they are converted to a string.

5. Functions Called: none

6. Called By:

   CODE

7. Error Calls: none

8. Flowchart: none

1. Name: FPREC

2. Function:

   Return precedence value of an operator when stacked.

3. Description:

   FPREC takes one argument, the operator whose precedence value is requested.

4. Design Alternatives:

   The precedence may be determined by searching a table, or possibly direct lookup.

5. Functions Called: none

6. Called By:

   EXPRESS

7. Error Calls: none

8. Flowchart: none

1. Name: GPREC

2. Function:

   Return precedence value of an operator when in the input stream.

3. Description:

   GPREC takes one argument, the operator whose precedence value is requested.

4. Design Alternatives:

   The precedence may be determined by searching a table, or possibly direct lookup.

5. Functions Called: none

6. Called By:

   EXPRESS

7. Error Calls: none

8. Flowchart: none

Section Four

The Interpreter Process

4.1 Introduction

The interpreter process builds a user context by internalizing context strings sent from translators and interpreters. List evaluation procedures are used to interpret the internal program structure. The interpreter environment is designed to separate program and data from physical processor bindings thus allowing user contexts to be moved from processor to processor. The structure of the interpreter list processer is based on a design found in [11].

4.2 Overview

Interpreter processing begins with internalizing a context string. If the user information in the string indicates that a computation has been interrupted then it is resumed from its break point. New computations are begun by interpreting the internalized contents of the context string. Communications with the user are done through an input handler process. An overview of the interpreter flow is shown in figure 4-1.

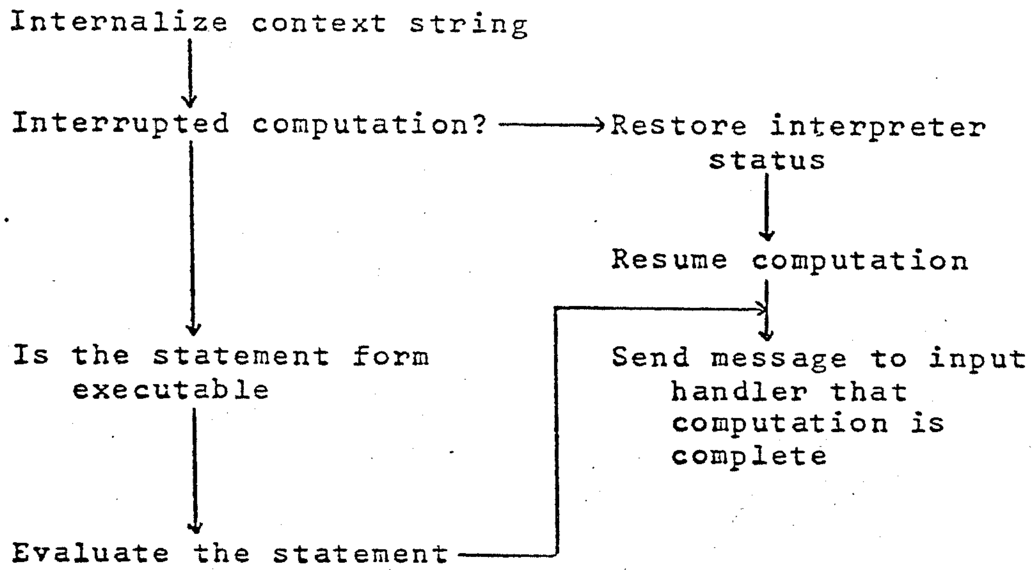


Figure 4-1: Interpreter Overview

A general storage allocator provides dynamic storage allocation during program interpretation. Garbage collection is unneccessary since procedures explicitly release unneeded storage.

The program list, symbol table, stacks and transfer table constitute the major interpreter data structures. All interpreter control and user values are referenced through the symbol table.

Symbol table entries are referenced by relative location to eliminate binding of data references to physical addresses. Similarly, values on the R-stack and T-stack are

relative references to the symbol table and transfer table, respectively.

Three stacks, R-stack, S-stack and T-stack are used by the interpreter. The R-stack stores intermediate computation values, the S-stacks contains pointers into the program list and the T-stack holds relative references to the transfer table. All three stacks have canonical representations in the context string.

4.3 Technical Specification

4.3.1 Technical Overview

INTCNTL, the top level interpreter procedure, invokes the procedure INTERNCS to build the internal structures and tables from the information in the context string. User information, statement forms, the symbol table, and the stacks are internalized by the procedures INTERNUIT, INTERNSF, INTERNSTB, and INTERNSTK respectively.

If this is an interrupted computation (determined by examining the user information) then the interpreter stats is restored from the values in the symbol table and the computation resumed. These activities are carried out by

the procedure RESTART.

If this was not an interrupted computation then a test is made to determine if the statement form from the translator is executable. If so, then EVAL is used to evaluate the form. This may in turn lead to the program list being evaluated by EVALP.

Most of the run time functions have not been described in detail. This is true in part because they have obvious definitions and that many correspond to similarly named procedures already defined in the translator and interpreter. As this design is implementeted these functions will be specified and included in the design document.

4.3.2 Interpreter Tables and Data Structures

4.3.2.1 Program Representation

A program is represented as a list whose elements are statement forms. Each statement form corresponds to a BASIC statement that has been input by the user. A statement form (SF) is itself a list that contains the following elements.

<u>Statement Form</u>

First element:	Statement Number
Second element:	Statement Type
Third element:	Original Source Statement
Fourth Element:	
.	
.	
.	Interpretive Expressions
.	
.	
Nth Element:	

Internally a program is represented as a one way linked list, as in figure 4-3.

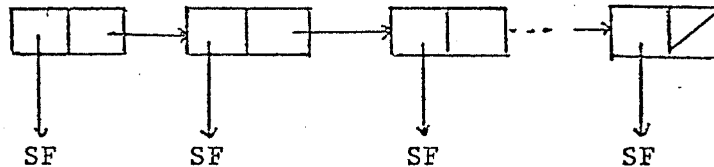


Figure 4-3: The Program List

Each statement form is also a variadic one way linked list, figure 4-4.

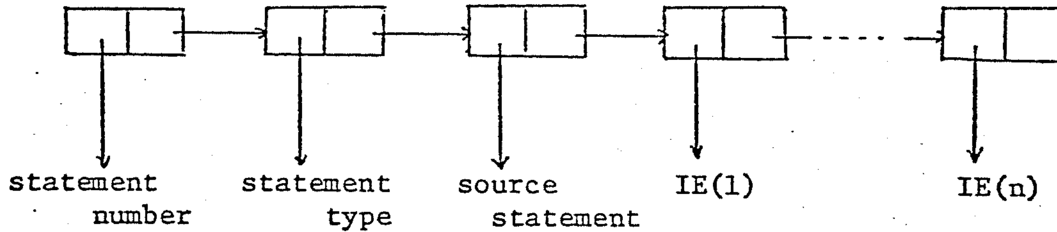


Figure 4-4: A Statement Form

The interpretive expressions are lists in "Cambridge Polish" notation and are described in section 3.4.

A complete internal representation of the statement

110 FOR I = 1 TO LEN(K$)

is given in figure 4-5.

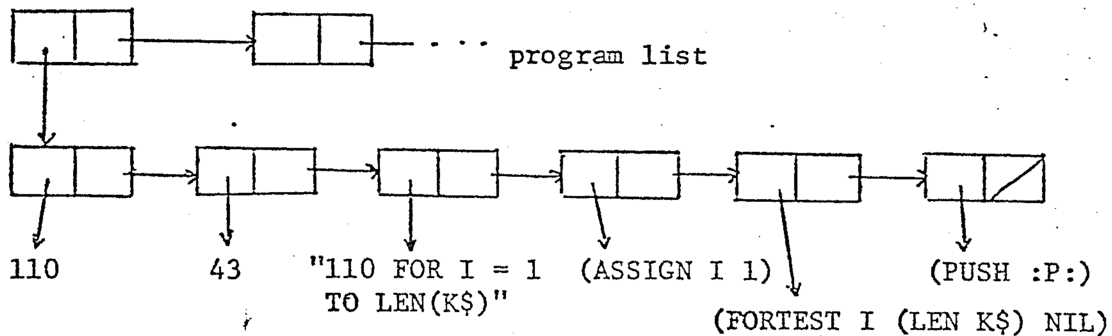


Figure 4-5: Complete Statement Form

Design Alternatives

1. Represent statement forms as fixed length blocks (see figure 4-6).

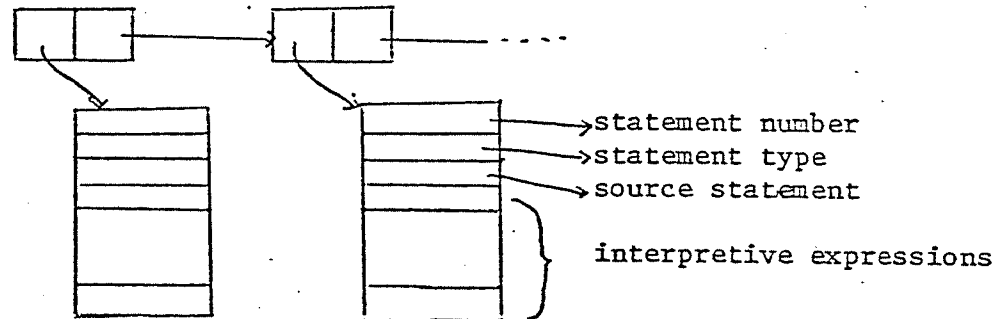


Figure 4-6: Fixed Length Statement Forms

This alternative reduces some storage overhead if the upper limit on the number of interpretive expressions is two or three. Since READ and INPUT statements may have up to 20 interpretive expressions, fixed size blocks represent too great a storage overhead.

2. Represent statement forms as variable length blocks.

This eliminates the problem of alternative one but it is now necessary to determine the size of the block to be allocated which may involve a complicated lookahead problem. A block structure complicates the problem of expression movement since internal pointers on any particular object machine are ambiguous with respect to cell versus word size.

3. Uncompile interpretive code to reconstruct source statements instead of storing original source statement.

   A) The storage costs are not too extreme in preserving the original source string.

   B) It is difficult to recompile the interpretive code due to the use of primitive functions used in representing the source statements.

4. Do not explicitly store the statement type.

   A) It is difficult to determine the statement type from examining the statements interpretive expressions.

   B) This alternative increases the interpretation speed, although not significantly.

Implementation Alternative

The statement number and type can be placed in the location used as a pointer to these items in the statement form instead of pointers if the cell size is adequate.

4.3.3.2 Symbol Table

General Organization

When represented as a table (other representations are discussed in this section under design alternatives), the symbol table has four divisions which correspond to system functions, user functions, system variables and user variables. While the total number of symbols is a function of the user program, a large percentage of the symbol in the table are fixed.

system functions
user functions
system variables
user variables

Figure 4-7: Symbol Table Organization

The system functions comprise all procedures appearing in the interpretive code with the exception of user defined functions (i.e. user functions have the form FN<letter>). System variables are critical interpreter values that are used in preserving context for program mobility (i.e. :DATAP:, :PROGP:, etc.). User variables are all the variables that appear in the program in addition to function

values, constants, and temporary values.

Temporary values are those that are created in the process of evaluating expressions. They are stored in the symbol table by treating the remaining symbol space as a stack during program execution. Thus, all values are accessed through the structure of the symbol table, a feature that facilitates error checking and mobility.

To achieve program mobility it is necessary to maintain complete information on symbol entries and separate the data representation of symbol values from the table format.

Symbol Entries

A symbol has four descriptive entries but not all of them may be applicable depending upon the type of symbol (i.e. system functions do not have name components). These entries include:

SNAME: symbol name
DTYPE: data type
STYPE: symbol type
SVALUE: symbol value

The list of symbol types and the possible data types they might assume is given in table 4-1. The presence of a

symbol name is also indicated.

	STYPE	SNAME	DTYPE
simple numeric variable	A	yes	1
simple string variable	B	yes	2
dimensioned numeric variable	C	yes	1
dimensioned string variable	D	yes	2
function value	E	yes	1,2
floating point constant	F	no	1
string constant	G	no	2
temporary value	H	no	1,2
system variable	I	no	0,1,2,3
user function	J	no	-
system function	K	no	-

Table 4-1: Symbol Table Data Types

The data type codes are:
0 - integer
1 - floating point
2 - string
3 - list expression

Symbol Values

No values are stored in the symbol table proper because there can be no assumptions about the amount of storage

required for the different data types. To achieve this machine independence the symbol value entry is always a pointer to the appropriate data structure (value). In some cases, as with undefined functions, this may be a pointer to :NIL:. The value structures differ depending on the symbol type. The storage decisions for numbers and strings are discussed in section 4-6. For now it is sufficient to have the concept of having a pointer to a floating point cell or a string to express the remaining structure of the symbol table.

A diagrammatic representation of the various symbol table entries is given in figure 4-8.

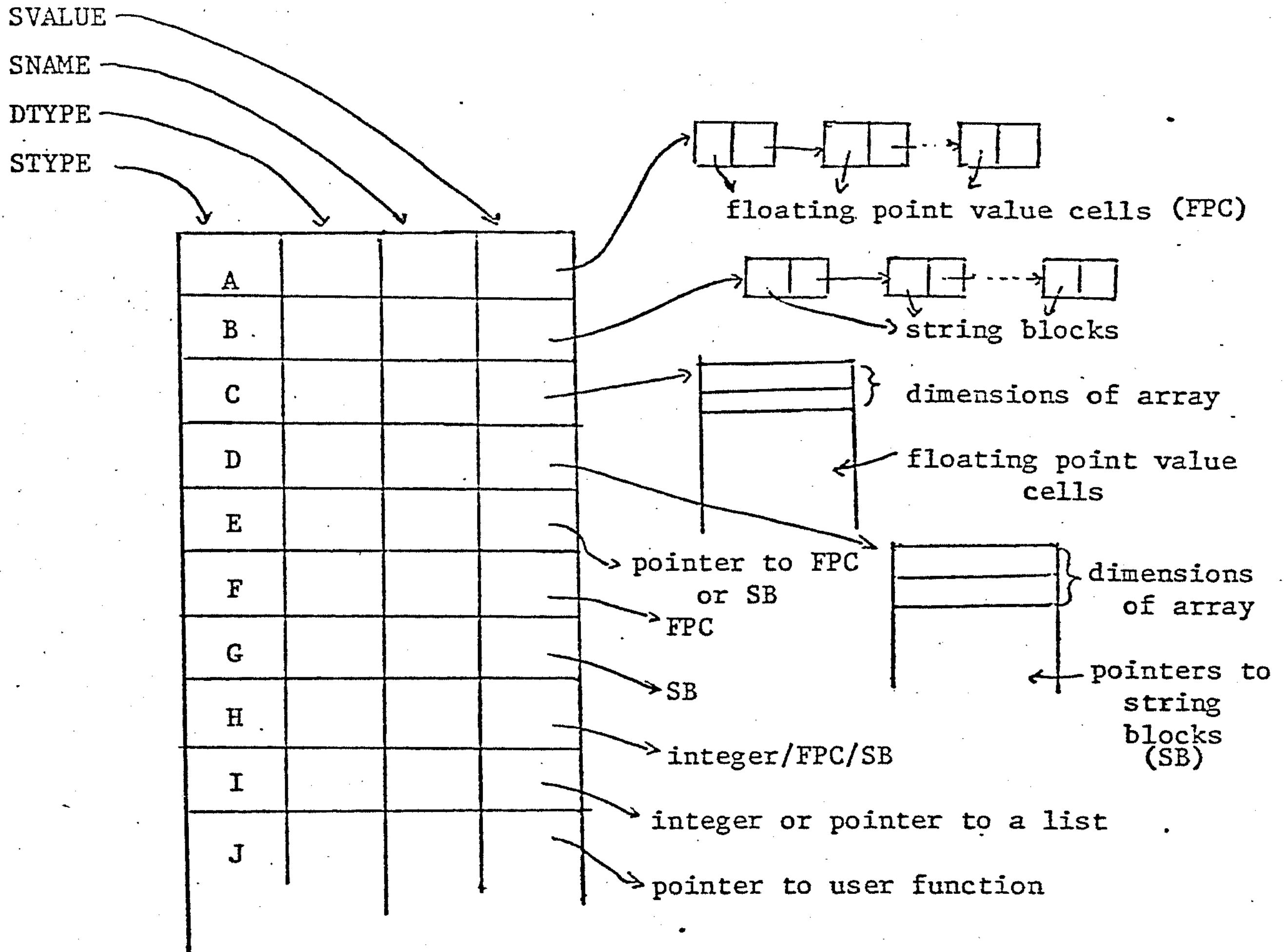


Figure 4-8: Symbol Table Value Representations

Symbol Table Value Representations

1) Simple numeric and string variables (types A and B)

Since BASIC allows for functions with arguments it is necessary to have a mechanism for saving the values of variables that appear as parameters in function calls. This is achieved by having multiple value cells in the form of a one way linked list. The cell at the head of the list is always the most current value for that variable.

The CAR of the cell is a pointer to a floating point cell for numeric variables and a pointer to a string block for string variables.

2) Dimensioned numeric and string variables (types C and D)

For dimensioned variables the value pointer is to a block of storage in which the first two components are the dimensions of the array. The remainder of the block is composed of floating point cells if it is a numeric array or regular pointer cells if it is a string array. In the latter case the cells then contain pointers to the appropriate string blocks.

3) Function value (type E)

The pointer may be to a floating point value cell or a string block.

4) Floating point constant (type F)

The pointer is to a floating point value cell.

5) String constant (type G)

The pointer is to a string block.

6) Temporary value (type H)

The pointer is to a floating point value cell or a string block.

7) System variable (type I)

The pointer is either an integer value or a pointer to a floating point value cell, a string block, or a list structure.

8) User function (type J)

The pointer is to the statement expression that defines the function (i.e. the corresponding DEF statement).

9) System function (type K)

The pointer is the address of a function.

Table Accessing Functions

All accesses to the symbol table are performed with the primitive functions GETS and PUTS. For the purposes of this design these functions are assumed to operate on a table such that we can directly accesss symbols by their relative position (i.e. third symbol, tenth symbol, etc.).

Other functions, such as FINDSYM, access the symbol table but always by using GETS and PUTS.

1. Name: GETS

2. Function:

Accessing function for retrieving information from the symbol table.

3. Description:

The calling sequence is GETS(POS,ENTRY) where POS is an integer value indicating the symbol to access and the value of ENTRY an indicator for which part of the entry to retrieve.

4. Design Alternatives: see symbol table description

5. Functions Called:n none

6. Called By:

   FINDSYM

7. Error Calls: none

8. Flowchart: none

1. Name: PUTS

2. Function:

   Accessing function for depositing information into the symbol table.

3. Description:

   The calling sequence is PUTS(POS,ENTRY,VAL) where POS is an integer indicating the symbol to access, ENTRY is an indicator for which part of the symbol entry to access and VAL the value to be deposited.

4. Design Alternatives: see symbol table description

5. Functions Called: none

6. Called By:

   FINDSYM

7. Error Calls: none

8. Flowchart: none

Design Alternatives

1. Direct access table

   This alternative meets all of the design requirements but requires reserving storage for all the system functions, system variables and 650 entries for all the BASIC variables (simple variables-286 + dimensioned variables-52 + function values-26 = 650). The cost in storage requirements is mediated by eliminating symbol lookup time in internalizing expressions. This does not seem to be sufficient justification since the one time lookup cost represents only a small portion of the interpreter's worktime.

2. Hashed and sorted symbol tables

   Both of these techniques are inappropriate for a system implementing BASIC since the size of the variable name space (i.e. variables are limitted to a letter or a letter followed by a digit) is so restricted. The number of symbols in a users program is usually very low and the variable names do not hash well.

   In addition, the sorted symbol table would require

either looking up each symbol at interpretation time or passing over the interpretive code linking it with the symbol table.

3. Storing function parameter values on a stack

   This strategy does not significantly reduce processing or storage requirements but instead increases the amount of bookkeeping to insure program mobility.

4. Eliminate the data type entry

   Elimination of the data type entry would require more worktime at interpretation time to determine the data types of symbols that may have multi type values.

5. Maintain a separate structure for temporary values

   All of the information required for processing regular symbols and moving them is needed for temporaries. Using the same accessing functions and structure reduces unnecessary overhead and maintains a uniform structure.

4.3.2.3 The R-stack

Temporary values created in evaluating argument lists or expressions are placed on the R-stack. An R-stack entry is either a delimiter or a relative reference to an item in the symbol table. Delimiters indicate the bounds of an argument list and should have a value that could not be mistaken as a symbol table reference.

Access to the R-stack is by the procedures PUSHR and POPR which are discussed in section 4.3.4.4. The context string format for the R-stack is found in section 5.3.2.

4.3.2.4 The S-stack

Internalization and evaluation procedures use the S-stack for storage of pointers (physical addresses) to the list structures composing the program list. During interpretation the stack contents can be viewed as sets of pointers to particular statement expressions. A mapping back to the appropriate statement expression is made by marking the limit of each set with a delimiter and the statement number of the statement expression for the set. Figure 4-9 shows a typical composition of the S-stack during interpretation.

S-stack

statement number
delimiter
pointer(1)
pointer(2)
.
.
pointer(n)
statement number
delimiter
pointer(1)
.
.

Figure 4-9: S-stack Composition

4.3.2.5 The T-stack

Return addresses for procedure calls are stored on the T-stack. An address is a relative reference to a transfer table entry which in turn contains the physical location of the procedure. Thus, return transfers are indirect and the addressing information on the T-stack is independent of the procedure actual hardware location.

4.3.2.6 Transfer Table

The transfer table, TRNTB, is a fixed length address vector for mapping references on the T-stack in physical locations. Any address that may occur on the T-stack at a computation interruption point appears in the table. The

table access procedure is GETTRN.

4.3.2.7 User Information Table

The user information table (UIT) is a fixed length table containing the internalized data from the user information portion of the context string. Access to the table is with the procedures GETUIT and PUTUIT.

4.3.3 Storage Allocation

4.3.3.1 Introduction

There are three functional types of storage that the allocator must dynamically allocate. Storage types are differentiated by their use. Thus, the storage allocator might respond differently in a request for pointer cells than in a request for floating point value cells. The motivation for the separation is to eliminate from consideration any one object matchine architecture.

Subsequent sections describe the types of storage, accessing functions for the different storage types and general reservation and liberation mechanisms. The last section shows how the storage allocation for all of the storage types may be handled by one set of functions and structures.

4.3.3.2 Pointer Cells

1. Description

   Pointer cells are the basic elements from which list structures are built. A pointer cell (see figure 4-10) must be able to contain two address pointers and be addressable.

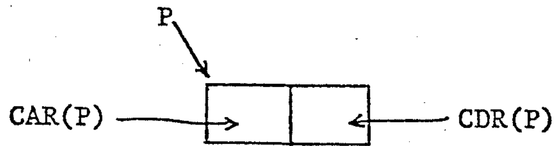


Figure 4-10: pointer cell

2. Accessing Functions

Pointer cells are accessed by the primitive functions CAR and CDR. If P is a pointer to a cell then CAR(P) is the left hand component of the cell and CDR(P) is the right hand component.

3. Allocation Functions

Pointer cells may be allocated individually or in blocks. A block of pointer cells may be used in representing a dimensioned string array.

GETCELL	value returned is a pointer to a new cell
GETBLK(N)	returns pointer to a block of N pointer cells
FRCELL(X)	returns pointer cell pointed at by X

to the available storage pool

FRBLK(X,N)	returns block of N pointer cells pointed at by X to the available storage pool

4.3.3.3 Floating Point Cells

1. Description

A floating point cell must be able to represent a floating point number on a given object machine.

2. Accessing Functions

Floating point cells are addressable and directly accessed.

3. Allocation Functions

Similiar to pointer cells, floating point cells may be allocated individually or in blocks. Blocks are used to represent numeric arrays.

GETFCELL	returns pointer to a floating point cell
GETFBLK(N)	returns pointer to a block of N floating point cells
FRFCELL(X)	returns the floating point cell pointed to by X to the available storage pool
FRFBLK(X,N)	returns the block of floating point cells pointed at by X to the available

storage pool

4.3.3.4 Strings

1. Description

A string is represented in the form of a string block. String blocks have two components, the first is the length of the string in characters, the second is the characters composing the string.

The length component should be large enough to avoid arbitrary restrictions on the size of strings.

2. Accessing Functions

Strings and their string blocks are accessed by the primitve functions GETC and PUTC.

3. Allocation Functions

The only representation for a logical string is in the form of a string block. The allocation functions are:

GETSBLK(N)	returns a pointer to a block of sufficient to represent a string of N characters (includes storage for the length component)
FRSBLK(X,N)	returns the string block of size N pointed at by X to the available

storage pool

4.3.3.5 Storage Reservation and Liberation

Available Storage Lists

The available blocks of storage are maintained in a one way linked list which is ordered by memory addresses. Depending on the object machine architecture the storage allocator may maintain separate availability lists for different types of storage [6]. The lists are represented as in figure 4-11.

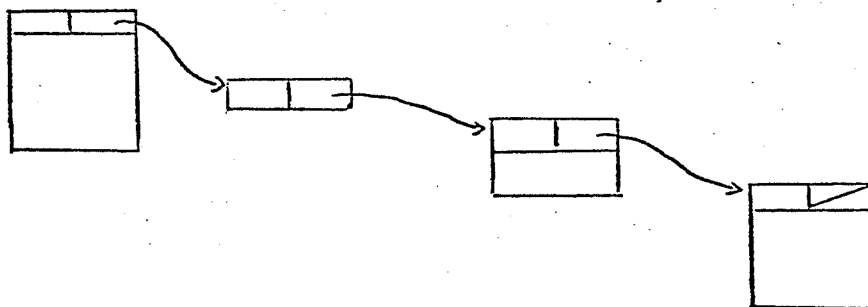


Figure 4-11: Available Storage Lists

In each storage block is kept the size of the block and a pointer to the next available block.

Reservation

A block of storage is reserved by searching down the appropriate storage list until the first block which is large enough is found. The required amount of storage is

then allocated from that block.

Liberation

The appropriate storage list is searched until the insertion position for the freed block is found. The freed block is inserted into the list or is compacted with abutting upper or lower blocks if possible.

4.3.3.6 Consolidated Storage Allocation

Unless the object machine has an architecture that would make separate storage lists desirable the pointer cell, floating point cell and string block type storage can all be obtained from the same availability list. This list can describe storage in words, bytes, or bits, whatever is best suited for that machine.

The storage allocation functions thus become mapping functions for the general storage allocator. If the allocator manipulated words as its basic elements then the following functions could be defined.

WFCELL(N)	number of words to represent N floating point cells
WPCELL(N)	number of words to represent N pointer cells

These functions can then be used in calls to the storage allocator. Thus the allocation function, GETBLK(N) is defined as

```
GETBLK(N): RETURN(ALLOCATE(WPCELL(N)))
```

The allocation functions for this design are specified for a word oriented storage allocator.

4.3.3.7 Design Alternatives

By placing the burden on the users for informing the storage allocator of freed storage the choice of allocation schemes is simplified.

1. Garbage collection and reference counts

   The 'freedom' inherent in these systems is unnecessary for this BASIC system since it is always known when structures are available to be freed. These systems like others may be more or less attractive given a particular machine that has an extra bit or two to make the particular scheme efficient.

2. Buddy-system

   The requirement of a free bit in each block restricts this techniques generality. In addition, since most blocks allocated in blocks of size 2 there can be considerable wasted space.

Given the adopted allocation system a number of alternatives are available for the available storage lists organization and reservation and liberation algorithms.

1. Order available storage list by block size

This strategy insures a best fit on block allocation but complciates memory compaction.

2. Random list ordering

   This strategy makes the liberation and reservation algorithms both slow and complicated.

3. Use best-fit as a reservation algorithm

   This algorithm is much slower but presents the best alternative to first-fit.

4.3.3.8 Design Alternatives for String Storage

The major alternative to allocating blocks for strings was to maintain a fixed size string storage table [11]. This system eliminates the need for searching an available storage list yet still provides dynamic storage for strings.

Externally all of the string handling functions would remain unchanged. Internally string storage would be allocated as needed from a fixed size storage area (it is possible that this area could be enlarged dynamically). To free a string would be accomplished by 'nulling' that string out in storage (replacing all characters with a defined null character). When the string storage would be exhausted then the area could be compacted by eliminating the aggregate

null areas.

The difficulty arises in compacting the storage area, since the organization of pointers to strings is not always well defined. In compaction, the address of the strings are changed and the original pointer must be modified. To do this the pointer must be identifiable and addressable.

There are a number of alternatives that can be used to handle the compaction problem.

1. Trace the list and symbol table structures to build a table that identifies all string pointers.

   This technique is very inefficient and time consuming.

2. Maintain a two way pointer system (have a back pointer in string storage).

   This does not completely disambiguate the source pointer since multiple pointers may exist in a single addressable word on some machines. It is then necessary to compare addresses within the word. This may be impractical without storing further information in string storage to help identify the object pointer.

3. Maintain a string descriptor table.

This causes all string references to be indirect through the descriptor table. This alternative increases the amount of context information which must be savable and adds to storage overhead.

Allocating blocks from the regular storage mechanism has the advantage of solving the garbage collection problem and eliminates the need for a string storage table which might arbitrarily restrict string sizes.

Another side effect is to eliminate the need for another movable data structure.

4.3.3.9 Implementation Alternatives

Any of the functions like GETCELL, which call a more general function with a fixed parameter may be replaced by the more general function.

Similiarly all of the functions may be replaced by using invocations of ALLOCATE and RELEASE.

4.3.3.10 Summary of Allocation Functions

General

ALLOCATE(N)
RELEASE(X,N)

Pointer Cells

GETCELL
GETBLK(N)
FRCELL(X)
FRBLK(X,N)

Floating Point Cells

GETFCELL
GETFBLK(N)
FRFCELL(X)
FRFBLK(X,N)

Strings

GETSBLK(N)
FRSBLK(X,N)

Primitives (for a word machine)

WFCELL
WPCELL
WSCELL

4.3.3.11 Storage Allocation Procedures

ALLOCATE	- Storage Allocation
RELEASE	- Storage Liberation
GETCORE	- Core Expansion
DELSTR	- Release Strings

1. Name: ALLOCATE

2. Function:

Returns a pointer to a block of N words.

3. Description:

ALLOCATE(N) searches down the available storage list, AVSTORE, until the first block which is large enough is found or the list is exhausted. If a block is found, it is allocated with any excess storage left on the list. If the list is exhausted more storage is requested from the system.

4. Design Alternatives: see Storage Allocation

5. Functions Called:

CAR	CDR	LOC	GETCORE

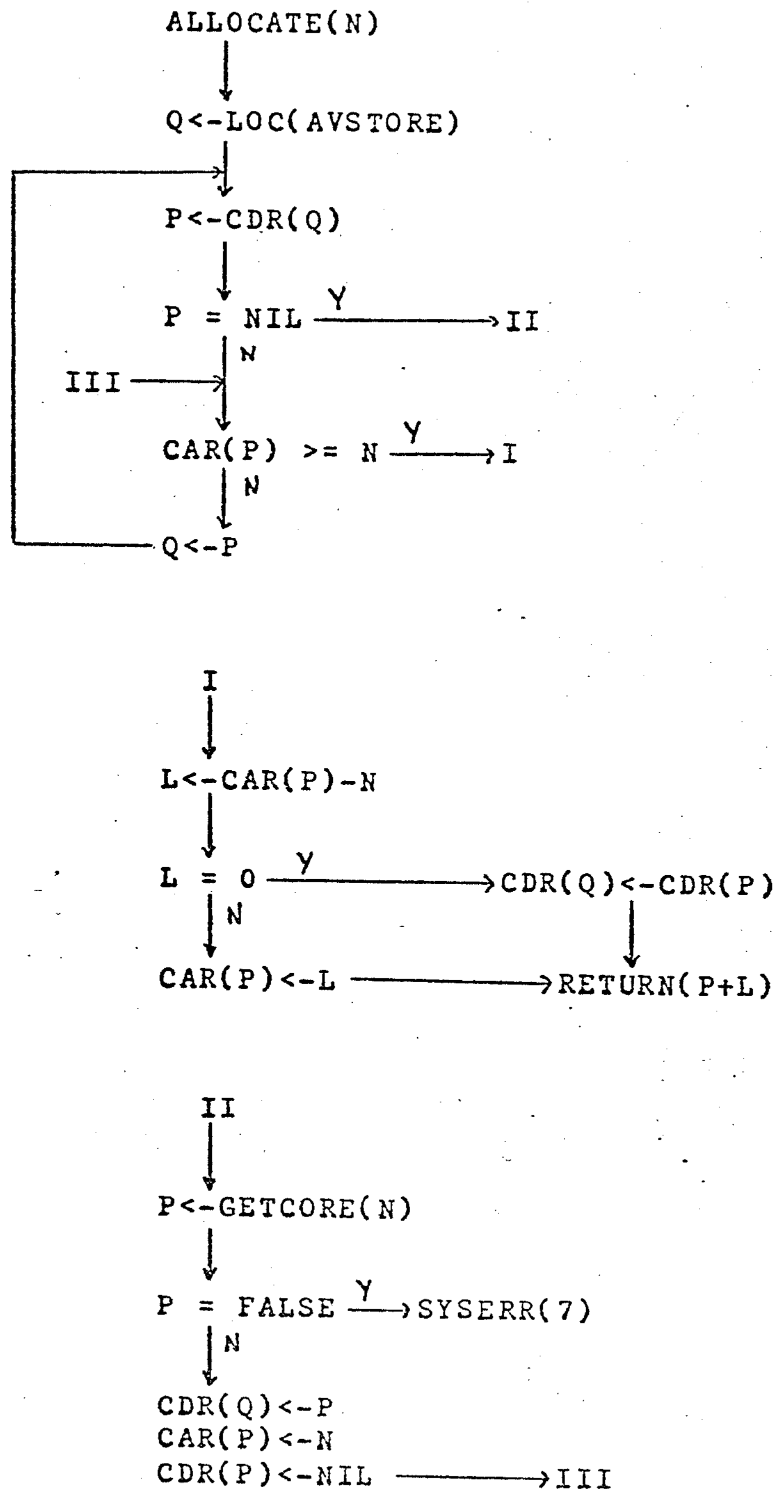
6. Called By:

GETCELL	GETBLK	GETFCELL	GETFBLK	GETSBLK

7. Error Calls:

SYSERR(7): insufficient core available for expansion

8. Flowchart:



1. Name: RELEASE

2. Function:

Returns the block of N words pointed at by X to the available storage pool.

3. Description:

The calling sequence is RELEASE(X,N) where X is a pointer to the storage to be freed and N is the number of words to be released.

The available storage list, AVSTORE, is searched to find the position to insert the freed storage. If the freed storage abuts either the blocks above or below then they are compacted into one block.

4. Design Alternatives: see Storage Allocation

5. Functions Called:

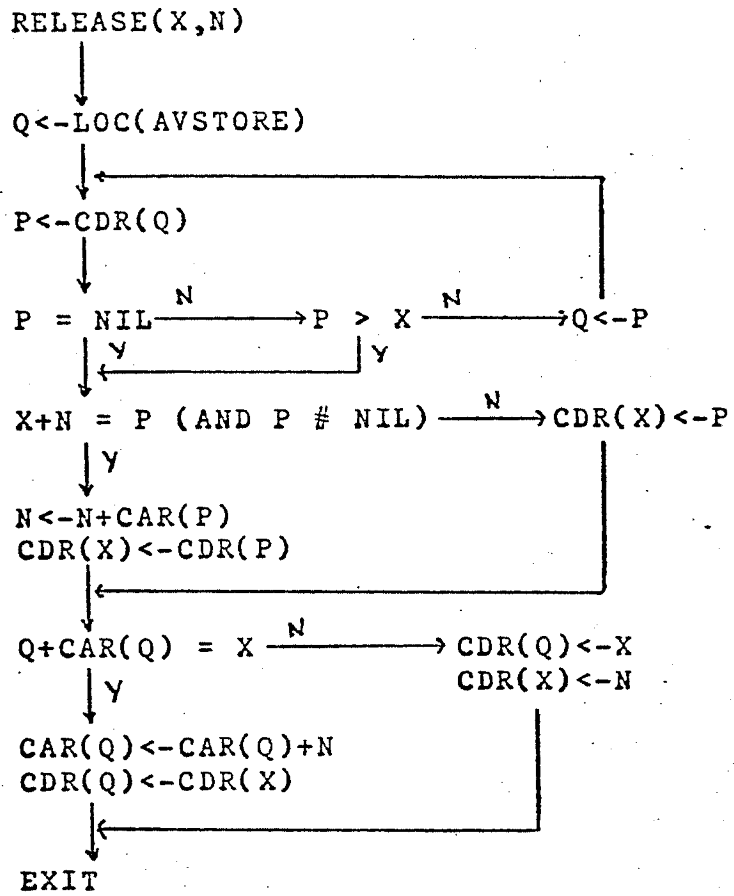
LOC CAR CDR

6. Called By:

FRCELL FRBLK FRFCELL FRFBLK FRSBLK

7. Error Calls: none

8. Flowchart:



1. Name: GETCORE

2. Function:

   Requests more core storage from operating system.

3. Description:

   GETCORE takes one argument, N, which is the amount of storage requested. The function returns FALSE (i.e. 0) if the request cannot be satisfied or a pointer to the requested storage.

4. Design Alternatives:

   In the environments where core expansion is not allowed this function becomes a dummy call that always generates an error message informing the user of insufficient core.

5. Functions Called: operating system

6. Called By:

   ALLOCATE

7. Error Calls: none

8. Flowchart: none

1. Name: DELSTR

2. Function:

   Releases storage of indicated string.

3. Description:

   DELSTR takes one input, a pointer to the string to be released. The string is released by using the string primitive for freeing string blocks, FRSBLK.

4. Design Alternatives:

   Strings can be released by directly invoking the FRSBLK procedure and doing the required calculations.

5. Functions Called:

   FRSBLK

6. Called By:

   PARSE

7. Error Calls: none

8. Flowchart: none

4.3.4.1 Internalization Procedures

INTERNSF	- Internalize Statement Forms
INTERNIE	- Internalize Interpretive Expressions
ARG	- Internalize Arguments
FINDSYM	- Symbol Table Entries
STR2STR	- String to String Conversion
STR2INT	- String to Integer Conversion
FLCNVT	- String to Floating Point Conversion

1. Name: INTERNSF

2. Function:

Internalizes the statement form (SF) portion of the context string.

3. Description:

On entry to INTERNSF the buffer pointer, BFP, is assumed to be positioned at the delimiter that follows the user information segment of the context string. The BFP is repositioned to the first character of the statement form as shown in figure 4-12.

SF

#(stmt no._stmt type_source_IE_...IE)#

Figure 4-12: Segment of the Context String

If the first character is a blank then a flag indicating no statement number is set by setting STMTN to -1, otherwise the procedure STR2INT is used to convert the character string up to the next delimiting character into the machine representation of an integer. Similiarly, the statement type and statement source string are translated into an integer and string respectively.

A partial statement form structure is formed and then the INTERNIE procedure is used to internalize the interpretive expressions associated with the statment. The entire statement form is then inserted into the program list.

If additional statement forms are present the INTERNSF procedure is repeated, otherwise control is returned to the calling procedure. The BFP is positioned at the delimiter following the list of statement forms.

4. Design Alternatives:

Until INTERNSF was redesigned to handle context strings as received from the interpreter as well as the translator this procedure would initiate program

interpretation.

5. Functions Called:

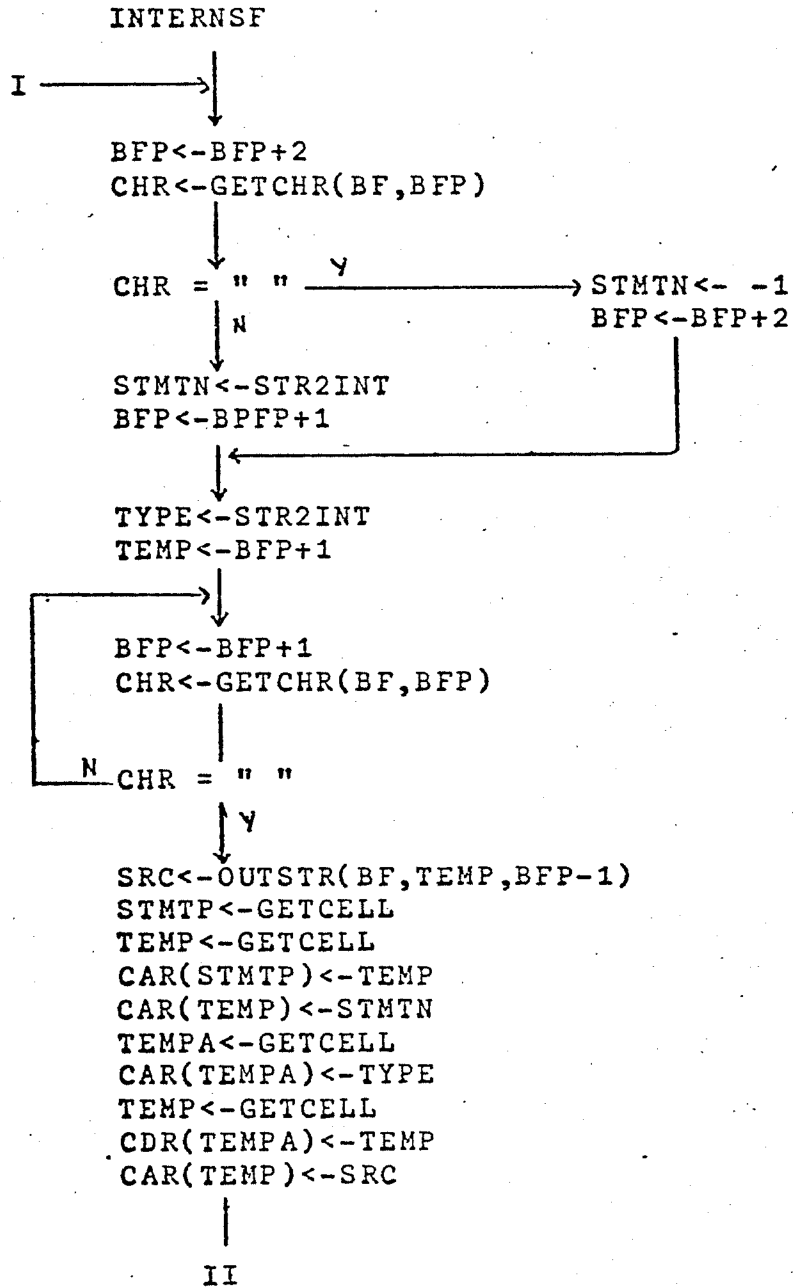
GETCHR	GETCELL	STR2INT	OUTSTR	CAR
CDR	INTERNIE	PUTS	PUSHT	INSERTS

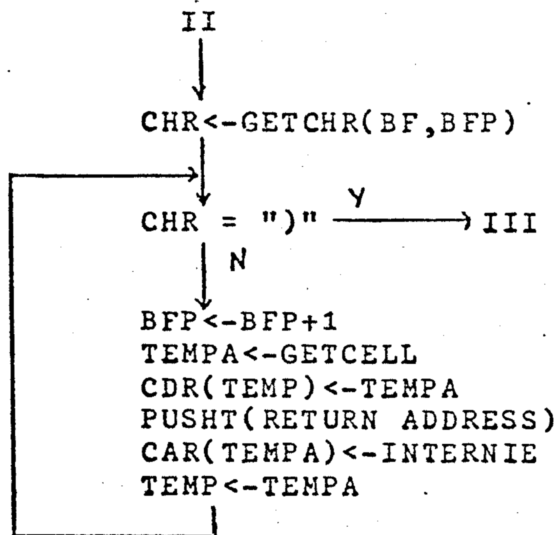
6. Called By:

INTERNCS

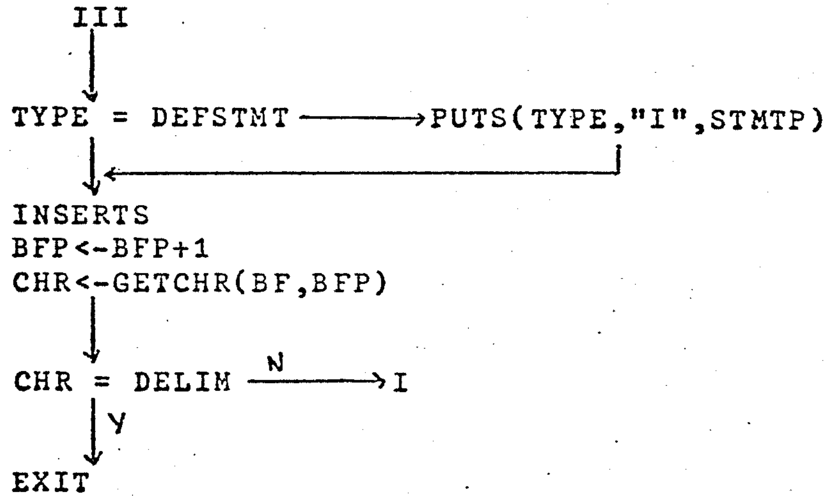
7. Error Calls: none

8. Flowchart:





II
CHR<-GETCHR(BF,BFP)
CHR = ")"
Y
III
N
BFP<-BFP+1
TEMPA<-GETCELL
CDR(TEMP)<-TEMPA
PUSHT(RETURN ADDRESS)
CAR(TEMPA)<-INTERNIE
TEMP<-TEMPA



III
TYPE = DEFSTMT
PUTS(TYPE,"I",STMTP)
INSERTS
BFP<-BFP+1
CHR<-GETCHR(BF,BFP)
CHR = DELIM
N
I
Y
EXIT

1. Name: INTERNIE

2. Function:

   Internalize interpretive expressions to produce internal list representations.

3. Description:

   The internalization of a list expression is accomplished by recursively invoking INTERNIE to process embedded lists. The value returned by INTERNIE is a pointer to an internalized list structure.

   The list is processed by analyzing the first character of an element. If the character is a left parenthese then INTERNIE is called recursively, if it is a right parenthese INTERNIE returns a pointer to a internalized list, otherwise ARG is used to process the element.

4. Design Alternatives: none

5. Functions Called:

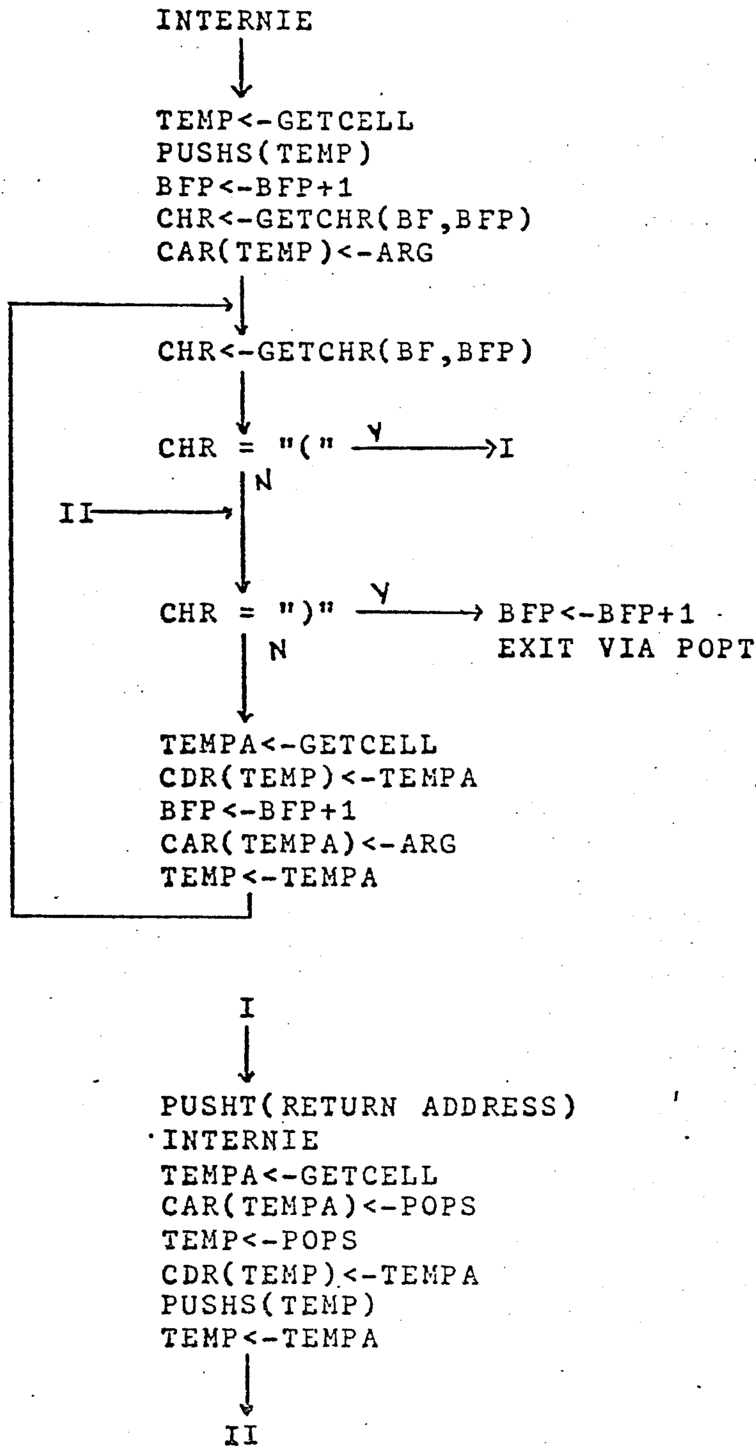
PUSHS	PUSHT	GETCELL	GETCHR	ARG
POPS	POPT			

6. Called By:

   INTERNIE

7. Error Calls: none

8. Flowchart:



1. Name: ARG

2. Function:

Processes list elements and returns the symbol table position of the element.

3. Description:

All list elements, except symbol table positions, are prefaced by a character that defines the elements symbol type. If the element is an address, STR2INT is used to convert the string representation of the address to an integer and this value is returned.

If the symbol type is 'A','B','C','D', or 'E' then the symbol name packed into NAME and FINDSYM invoked. The value returned by FINDSYM is returned.

If the symbol type is 'F' then the remaining string is converted into the machine representation of a floating point number, otherwise it is assembled as a string constant. In both cases FINDSYM is then used to enter the argument into the symbol table.

4. Design Alternatives:

Part of the ARG procedure consists of packing a symbol name into NAME (i.e. storing the one or two character variable name in NAME). This is a machine dependent strategy and may be avoided by processing the symbol name as a string and correspondingly storing the name as a string (string block) in the symbol table.

5. Functions Called:

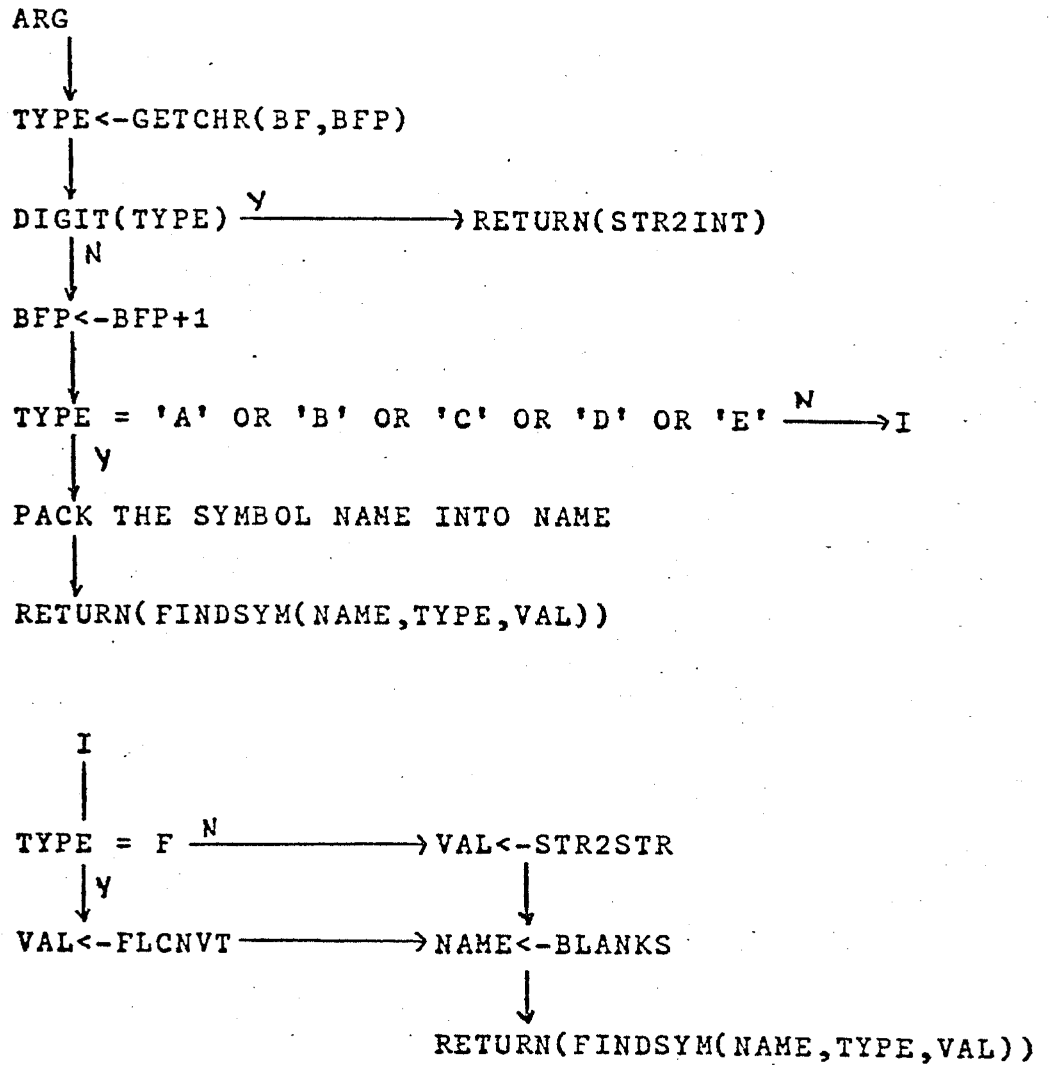
GETCHR DIGIT FINDSYM FLCNVT STR2STR
STR2INT

6. Called By:

INTERNIE

7. Error Calls: none

8. Flowchart:



1. Name: FINDSYM

2. Function:

Returns the symbol table address of a symbol. If the symbol is not present an entry is created, else the location existing symbol is returned.

3. Description:

The calling sequence is FINDSYM(NAME,TYPE,VAL) where NAME is the symbol name (possibly null), TYPE is the symbol type, and VAL is the symbol value (null unless TYPE is a string or floating point constant).

Unless the symbol is a string literal (type G) the current symbol table is searched for an already existing entry. If present the location of the existing entry is returned.

If no entry is present a new entry is made in the symbol table and its location is returned. NXTSYM contains the value of the next available symbol table position.

4. Design Alternatives: none

5. Functions Called:

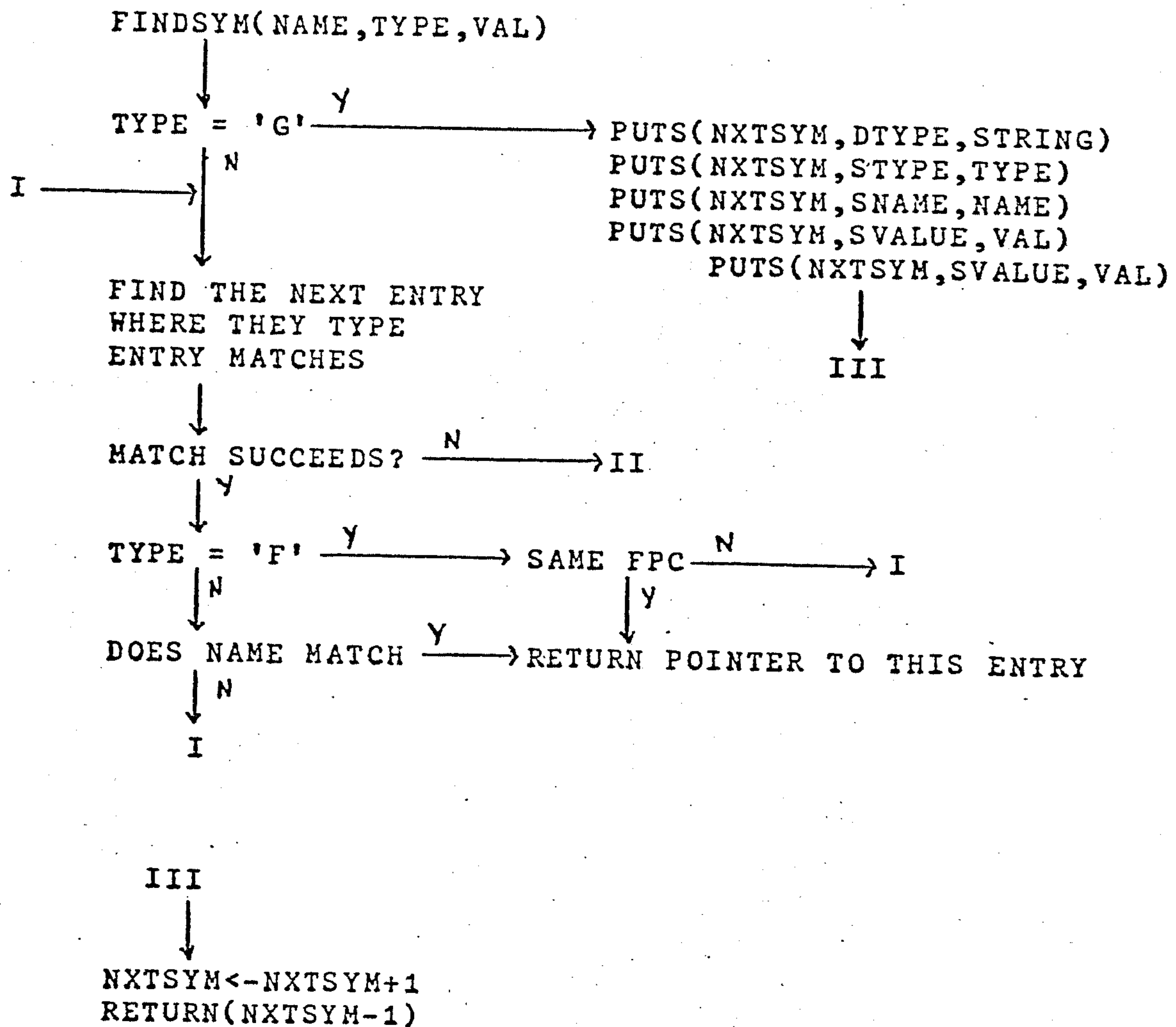
PUTS CAR GTSBLK GETCELL
GETS

6. Called By:

ARG

7. Error Calls: none

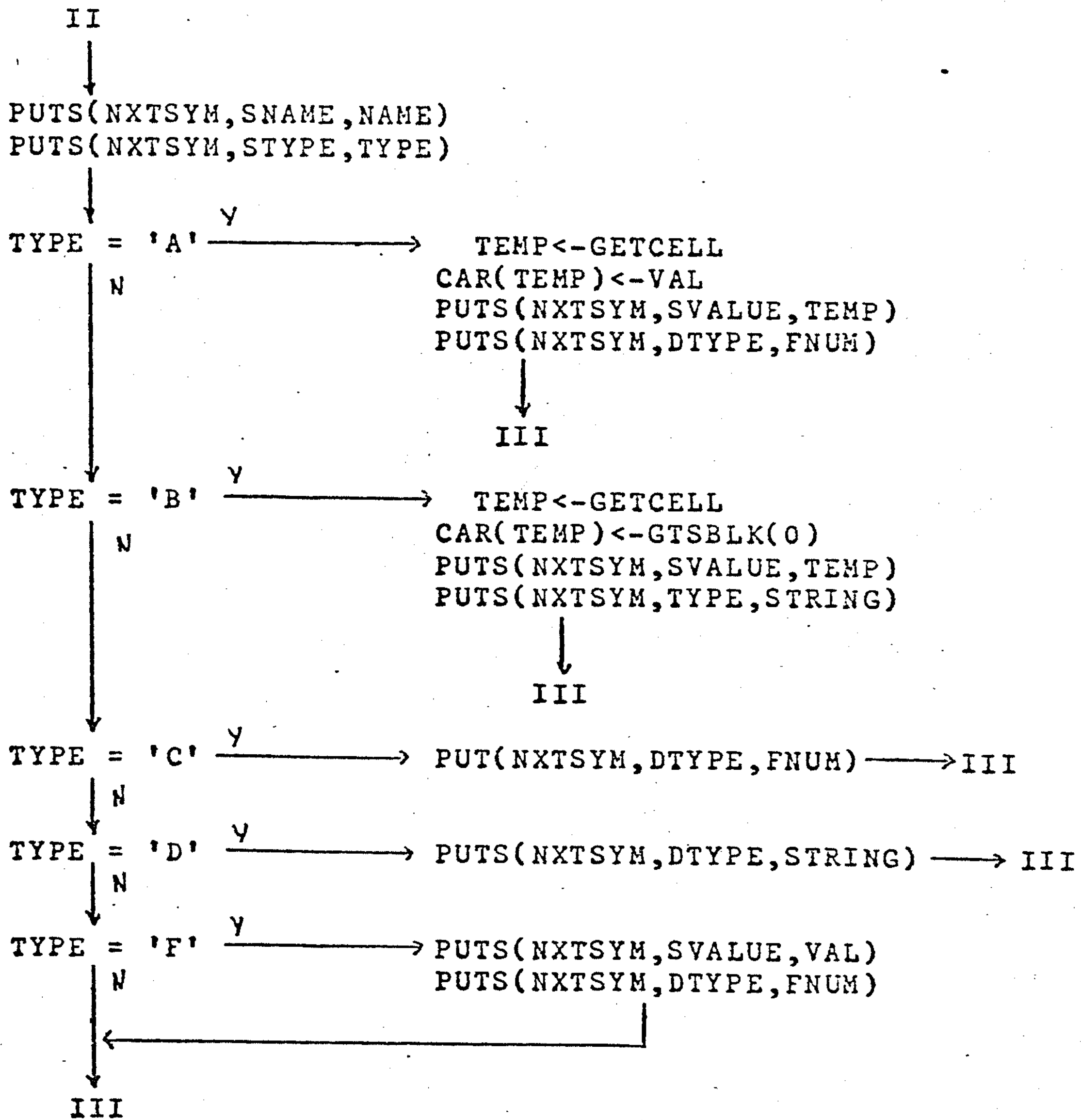
8. Flowchart:



III

NXTSYM<-NXTSYM+1
RETURN(NXTSYM-1)

STRING and FNUM are predefined symbol values that indicate data types of the string and floating point number


II
PUTS(NXTSYM,SNAME,NAME)
PUTS(NXTSYM,STYPE,TYPE)
TYPE = 'A'
Y
N
TEMP<-GETCELL
CAR(TEMP)<-VAL
PUTS(NXTSYM,SVALUE,TEMP)
PUTS(NXTSYM,DTYPE,FNUM)
III
TYPE = 'B'
Y
N
TEMP<-GETCELL
CAR(TEMP)<-GTSBLK(0)
PUTS(NXTSYM,SVALUE,TEMP)
PUTS(NXTSYM,TYPE,STRING)
III
TYPE = 'C'
Y
N
PUT(NXTSYM,DTYPE,FNUM)
III
TYPE = 'D'
Y
N
PUTS(NXTSYM,DTYPE,STRING)
III
TYPE = 'F'
Y
N
PUTS(NXTSYM,SVALUE,VAL)
PUTS(NXTSYM,DTYPE,FNUM)
III

1. Name: STR2STR

2. Function:

Internalizes the sequence of characters enclosed by quotation marks as a string and returns a pointer to the string.

3. Description:

The buffer pointer, BFP, is advanced to beyond the matching quotation mark and OUTSTR is used to created the string. BFP is left positioned after the matching quotation mark.

4. Design Alternatives:

The pointer marking the end of the string can be moved in the calling procedure and OUTSTR used directly to eliminate the need for this procedure.

5. Functions Called:

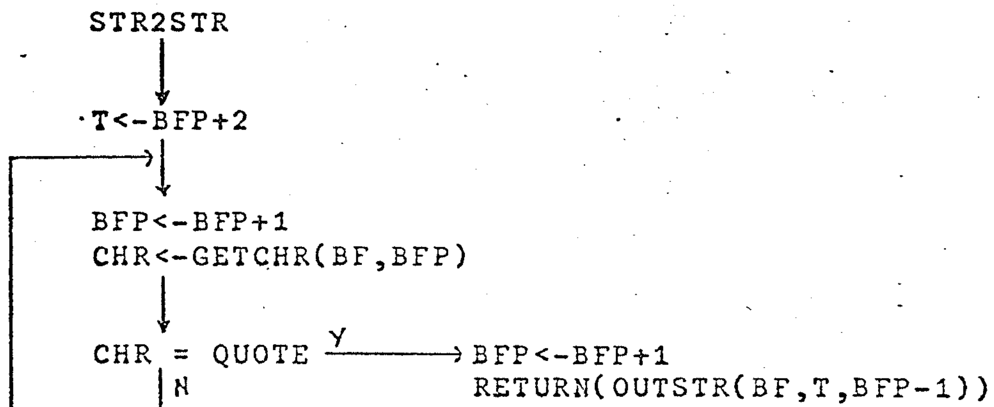
GETCHR OUTSTR

6. Called By:

ARG

7. Error Calls: none

8. Flowchart:



1. Name: STR2INT

2. Function:

   Internalizes the sequence of characters starting at the buffer pointer and to the next non-numeric character as an integer and returns the integer value.

3. Description:

   The buffer pointer, BFP, is advanced to beyond the last numeric character encountered.

4. Design Alternatives: none

5. Functions Called:

6. Called By:

   ARG INTERNSF

7. Error Calls:

8. Flowchart: none

1. Name: FLCNVT

2. Function:

   Internalizes the sequence of characters starting at the buffer pointer and to the next blank as a floating point number. A pointer to the floating point number is returned.

3. Description:

   The sequence of characters in the number are delimited and converted to a floating point number. The result of the conversion is stored in a floating point value cell.

4. Design Alternatives: none

5. Functions Called:

   GETCHR GETFCELL

6. Called By:

   ARG

7. Error Calls:

8. Flowchart: none

4.3.4.2 Evaluation Procedures

EVALP	- Evaluate Program List
EVALE	- Evaluate Interpretive Expressions
EVAL	- Evaluate an Interpretive Expression
ETYPE	- Determine Function Type
ESUBR	- Evaluate System Functions
EEXPR	- Evaluate User Functions
EVLARG	- Evaluate an Argument List
BIND	- Bind Argument List
REBIND	- Unbind Argument List

1. Name: EVALP

2. Function:

Top level procedure that controls the evaluation and interpretation of the program list.

3. Description:

EVALP traverses the program list interpretively evaluating statement forms until either the list terminator is encountered, a FNEND function is found, or a halt function is executed by the program.

User defined functions are conveniently handled by recursively calling EVALP. Thus, upon invocation, EVALP preserves a certain amount of program context (i.e. the current statement number) on the S-stack.

Whenever a statement form is to be evaluated the statement number of that statement and a delimiter are placed on the S-stack. This is done to facilitate moving the S-stack and is discussed in detail in the section on program mobility (section 5).

When the program list's terminator is reached or a FNEND statement encountered then control is returned to the invoking function, otherwise the statement type of the current statement form is examined.

If the type of statement is a REM, DATA, or DEF then the form pointer P is advanced to the next form and this entire procedure repeated. If the DEF statement is a multiple line function (detected by examining the form's interpretive list), the pointer to advanced to beyond the matching FNEND statement. The implementor should include a test for embedded DEF statements.

EVALE is called upon finding an interpretable statement form. The current statement form pointer to updated to the next expression to evaluate before EVALE is invoked.

4. Design Alternatives:

5. Functions Called:

CAR	CDR	EVALE	PUSHT	PUSHS
POPS	POPT			

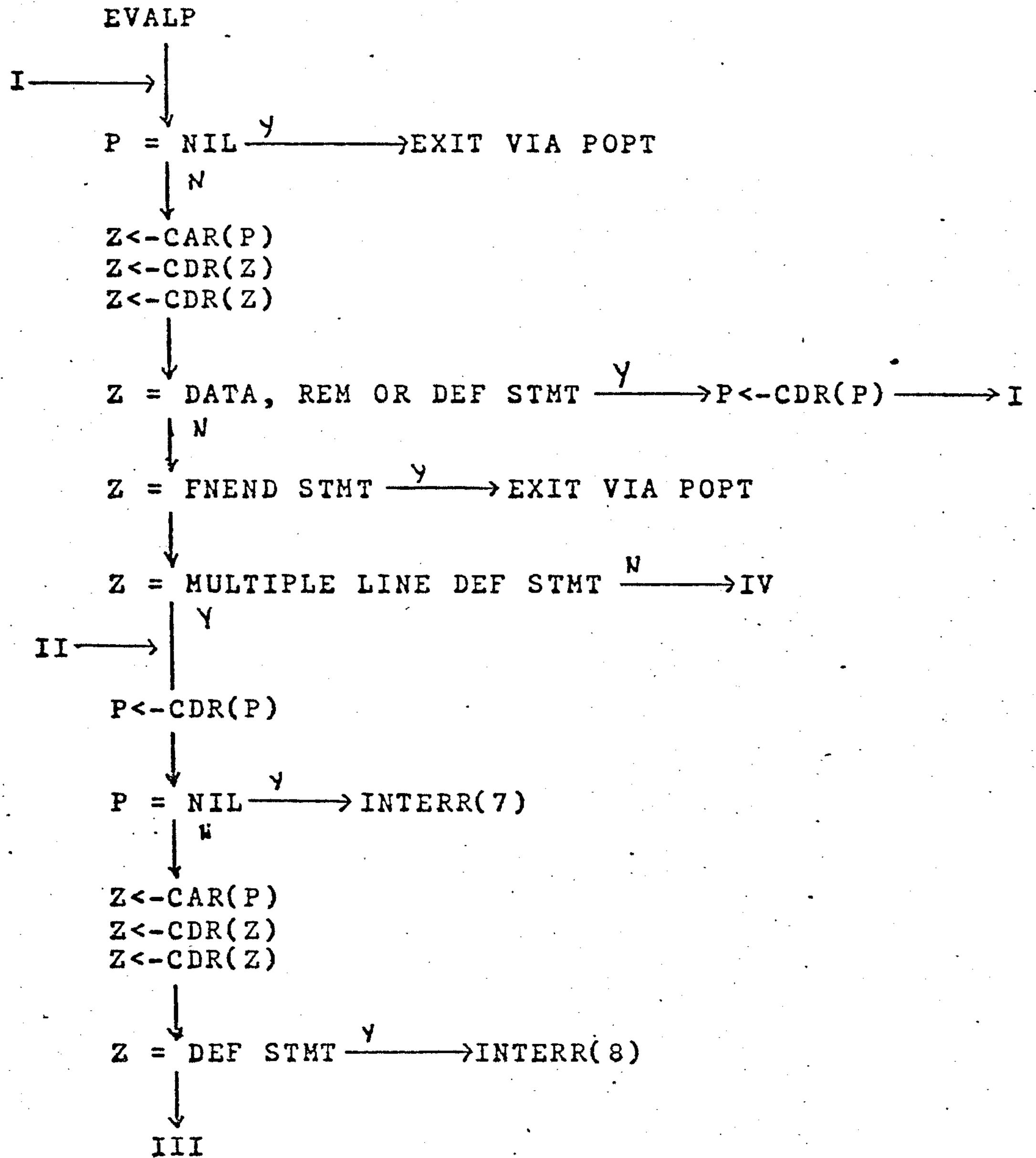
6. Called By:

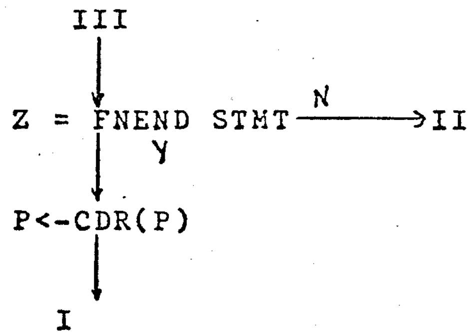
INTCNTL

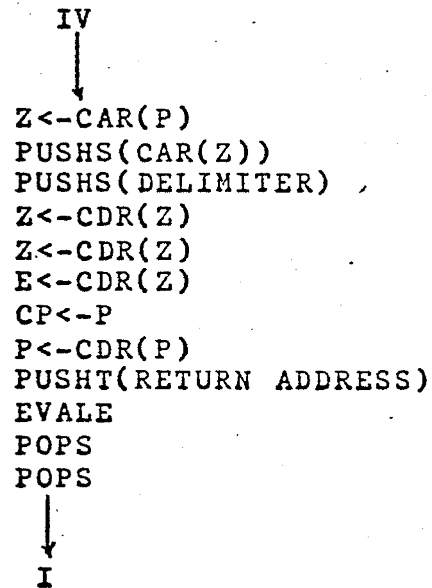
7. Error Calls:

INTERR(7): DEF statement has no matching FNEND
INTERR(8): illegal embedded DEF statement

8. Flowchart:




III
Z = FNEND STMT
N
II
Y
P<-CDR(P)
I


IV
Z<-CAR(P)
PUSHS(CAR(Z))
PUSHS(DELIMITER)
Z<-CDR(Z)
Z<-CDR(Z)
E<-CDR(Z)
CP<-P
P<-CDR(P)
PUSHT(RETURN ADDRESS)
EVALE
POPS
POPS
I

1. Name: EVALE

2. Function:

   Interpret the list of interpretive expressions for a statement expression.

3. Description:

   EVALE sequentially interprets the list of interpretive expressions until the list terminator is encountered, at which time an exit is made to the calling procedure.

   Since EVALE maybe called recursively all local information (such as the pointer to the next interpretive expression) is saved on the S-stack.

   EVAL is called to evaluate individual interpretive expressions.

4. Design Alterntives:

5. Functions Called:

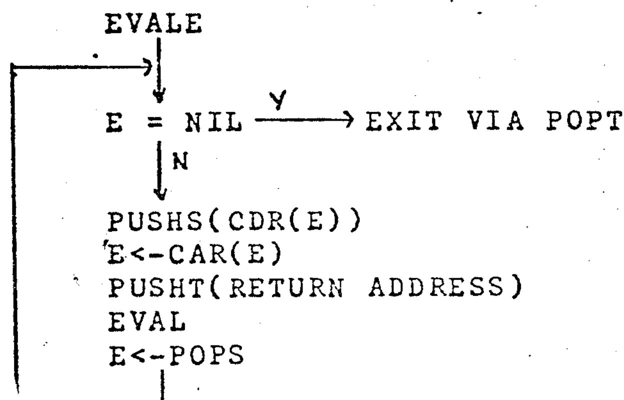
   EVALE POPT POPS PUSHS PUSHT
   CDR CAR

6. Called By:

   EVALP

7. Error Calls: none

8. Flowchart:



1. Name: EVAL

2. Function:

   Evaluate the interpretive expression pointed at by E.

3. Description:

   EVAL assumes that the structure pointed at by E is a list, if its first element is not atomic (i.e. a pointer to a function) then ETYPE is invoked to identify and evaluate the function. Otherwise, the value of the atomic item is retrieved from the symbol table (i.e. a pointer to the symbol table location) and is pushed onto the result stack (R-stack).

4. Design Alternatives: none

5. Functions Called:

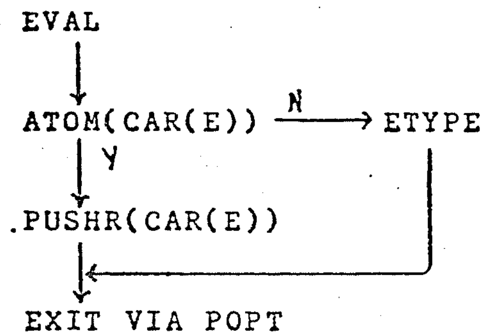
   ATOM CAR PUSHR POPT ETYPE

6. Called By:

   EVALE EEXPR

7. Error Calls: none

8. Flowchart:



1. Name: ETYPE

2. Function:

   Determines the function type for a function and invokes the proper interpreter procedure.

3. Description:

   The function type is determined from the symbol table and control is passed to the appropriate function evaluating procedure.

   Function types may be either SUBR's or EXPR's both of which evaluate their arguments before invoking the function. EXPR's are user defined functions and SUBR's are machine code procedures.

4. Design Alternatives:

   An FSUBR facility (arguments not evaluated) is easily implemented and some interpretive code expressions could be rewritten to take advantage of such a facility.

5. Functions Called:

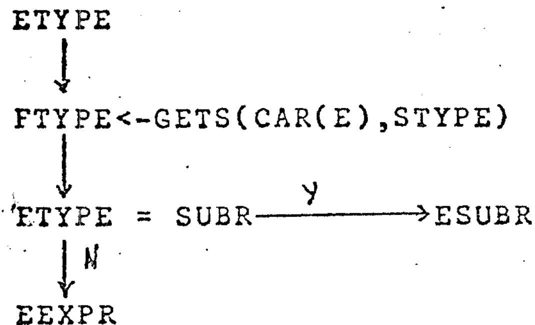
   GETS CAR ESUBR EEXPR

6. Called By:

   EVAL

7. Error Calls: none

8. Flowchart:



1. Name: ESUBR

2. Function:

   Evaluate system functions whose arguments must be evaluated.

3. Description:

   A pointer to the function's symbol table entry is saved on the R-stack until after the functions argument list is evaluated by EVLARG. The function's address is restored after returning from EVLARG and the function is then executed.

   The evaluation procedures are all recursively callable and must save all local information.

4. Design Alternatives: none

5. Functions Called:

PUSHR	PUSHT	POPR	POPT	GETS
CAR	EVLARG	EXECUTE		

6. Called By:

   ETYPE

7. Error Calls: none

8. Flowchart:

```
ESUBR
  |
  v
PUSHR(GETS(CAR(E),SVALUE))
PUSHT(RETURN ADDRESS)
EVLARG
EXECUTE(POPR)
  |
  v
EXIT VIA POPT
```

1. Name: EEXPR

2. Function:

Evaluate user defined functions.

3. Description:

The name of the function is saved on the R-stack and a return address is placed on the T-stack. After the function's argument list has been evaluated by EVLARG then the procedure BIND is used to bind the values on the R-stack to the functions argument list. Error tests are performed to check for a matching number of arguments.

By examining the interpretive expression list it is determined if the function is a single or multiple line definition.

Single line definitions are handled by calling EVAL to evaluate the interpretive expression, rebinding the arguments and exitting.

In the case of multiple line functions the pointer to the next statement expression is saved on the S-stack. The function name is saved to allow returning a value and EVALP called recursively. Upon return the arguments are rebound and the procedure SETVAL used to return the evaluated functions value.

4. Design Alternatives:

5. Functions Called:

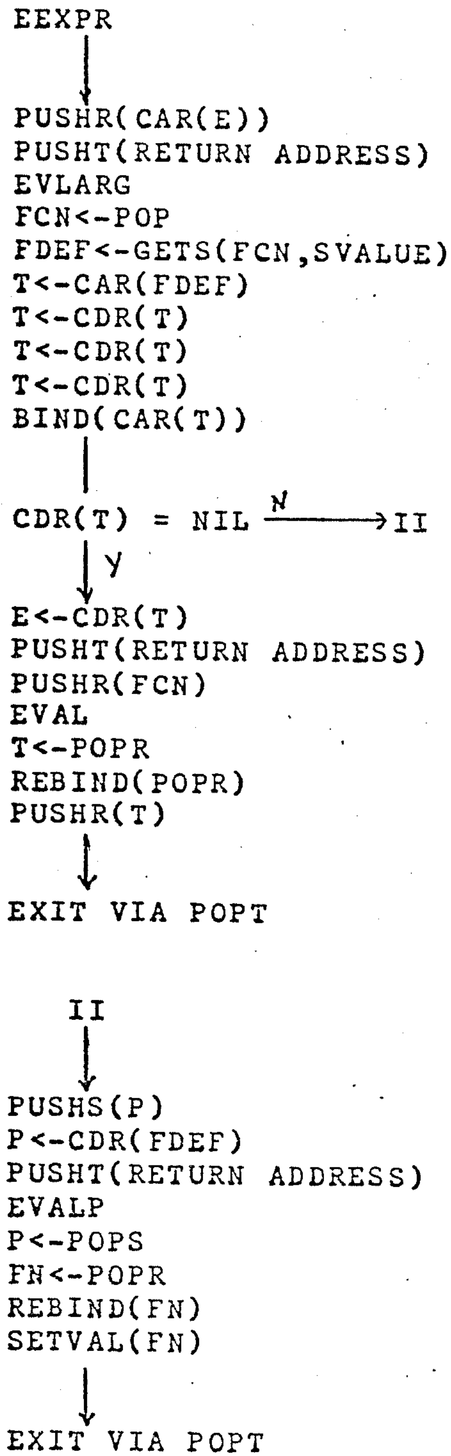
PUSHS	PUSHR	PUSHT	POPR	POPS
POPT	EVLARG	GETS	CAR	CDR
GETCELL	PUTS	BIND	EVAL	REBIND
EVALP	SETVAL			

6. Called By:

ETYPE

7. Error Calls: none

8. Flowchart:



1. Name: EVLARG

2. Function:

   Evaluate a functions argument list.

3. Description:

   A delimiter for the argument list is pushed onto the R-stack as part of the initialization. If no arguments are left to evaluate the calling function is returned to via a POPT, otherwise a return address is stacked, the current expression pointer, E, set and EVAL invoked. The process is repeated until all arguments are evaluated.

4. Design Alternatives: none

5. Functions Called:

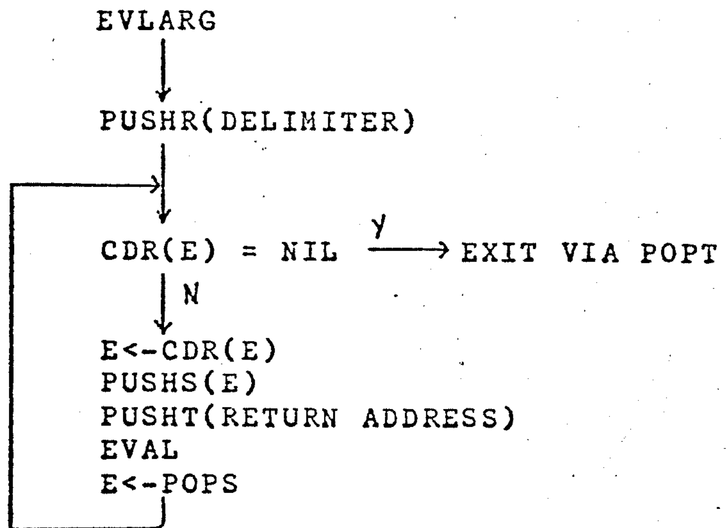
   PUSHR PUSHS PUSHT POPS POPT
   EVAL CDR

6. Called By:

   ESUBR EEXPR

7. Error Calls: none

8. Flowchart:



1. Name: BIND

2. Function:

Saves current context of variables in argument list and binds then with new values from the R-stack.

3. Description:

BIND takes one argument, ARGP, a pointer to the argument list (the elements in turn point to symbol table entries). For each element in the argument list its corresponding symbol table value entry is modified by placing a new cell at the beginning of the value list. The value this new cell points at is obtained from the R-stack.

Error messages are issued whenever the number of arguments used in the function call differs from the definition.

4. Design Alternatives:

The current value of parameter variables can also be saved by pushing a function identifier and the current values onto a stack.

5. Functions Called:

GETCELL CDR GETS PUTS POPR
CAR

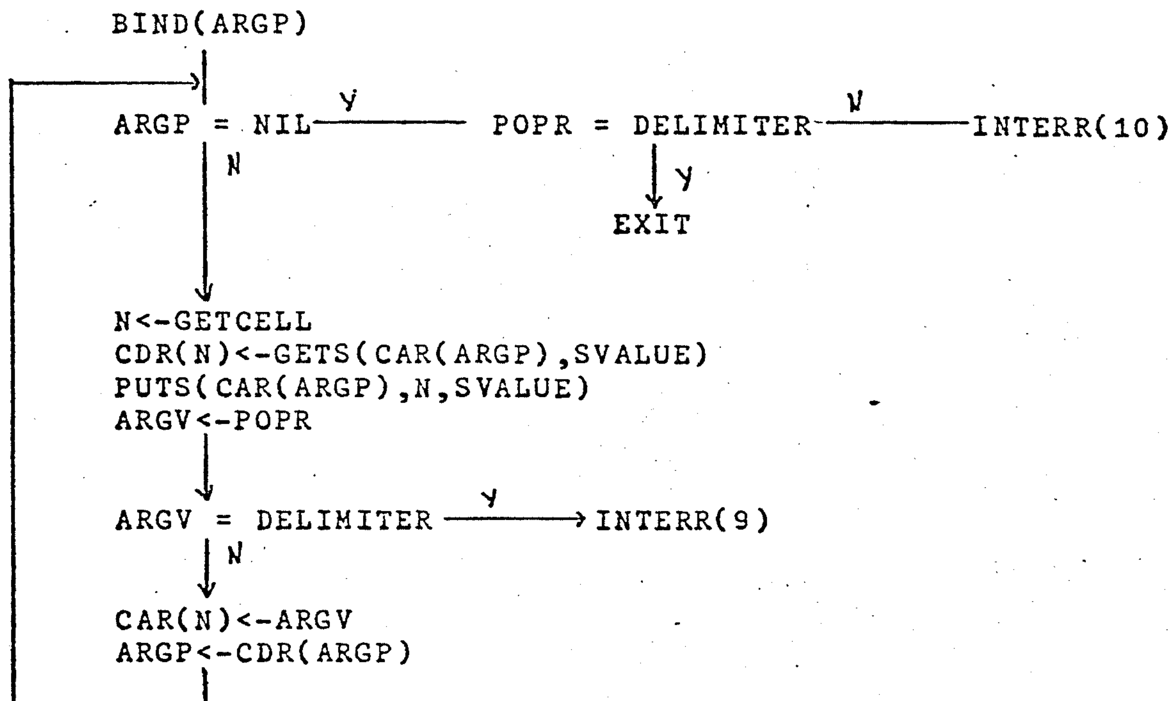
6. Called By:

EEXPR

7. Error Calls:

INTERR(9): too many arguments in function call
INTERR(10): too few arguments in function call

8. Flowchart:



1. Name: REBIND

2. Function:

   Restore previous context for a list of variables.

3. Description:

   REBIND takes one argument, ARGP, a pointer to the argument list. The list pointed at by ARGP is mapped down and each elements symbol table entry has the first value cell deleted.

4. Design Alternatives: none

5. Functions Called:

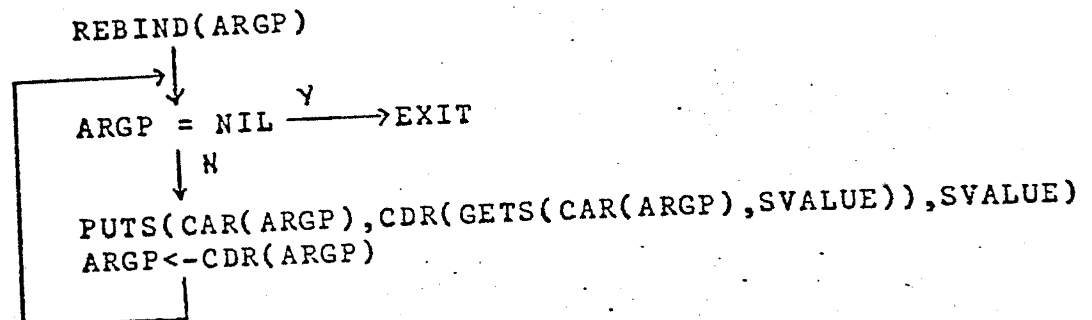
   CAR CDR PUTS GETS

6. Called By:

   EEXPR

7. Error Calls: none

8. Flowchart:



4.3.4.3 String Handling Procedures

These procedures are used in manipulating strings in the translator and interpreter processes.

GETC	- Get Character From a String
PUTC	- Put Character into String
LENGTH	- Determine Length of String
CONCAT	- Concat Strings
STREQ	- Test String Equality
MATCH	- Test Matching Substring

1. Name: GETC

2. Function:

   Returns specified character from a string.

3. Description:

   Calling sequence is GETC(STRP,CHRP) where STRP is a pointer to the string and CHRP is a relative pointer to the character requested. CHRP must be an integer value between 0 and the length of the string.

   If the value of CHRP is zero the request is interpreted to be a request for the length of the string.

4. Design Alternatives: none

5. Functions Called: none

6. Called By:

7. Error Calls: none

8. Flowchart: none

1. Name: PUTC

2. Function:

   Deposit character into a specified position in a string block.

3. Description:

   The calling sequence is PUTC(STRP,CHRP,VAL) where STRP is a pointer to the string, CHRP is a relative string pointer, and VAL is the value to be deposited.

   CHRP must be an integer value between 0 and the length of the string. If the value of CHRP is 0 then the value is deposited as the length of the string(i.e. wherever the length component of strings are stored.

4. Design Alternatives:

5. Functions Called: none

6. Called By:

7. Error Calls: none

8. Flowchart: none

1. Name: LENGTH

2. Function:

   Returns the number of characters in a string.

3. Description:

   LENGTH has one input, a pointer to the named string. The length of the string is determined by invoking the string primitive GETC(pointer to string,LEN).

4. Design Alternative:

   The length of strings can be directly accessed under some implementations by using the string primitives. The use of LENGTH is clean but has some overhead.

5. Functions Called:

   GETC

6. Called By:

   LEXICAL PARSE

7. Error Calls: none

8. Flowchart: none

1. Name: CONCAT

2. Function:

Return a pointer to a new string formed by the concatenation of two strings.

3. Description:

The procedure CONCAT takes two inputs which are pointers to the strings to be concatenated. The lengths of both strings are obtained, summed, and used to allocate a new string block for the concatenated string. The object strings are copied into the new string block character by character using the string primitives GETC and PUTC.

A pointer to the new string block is returned as the value of CONCAT.

4. Design Alternatives: none

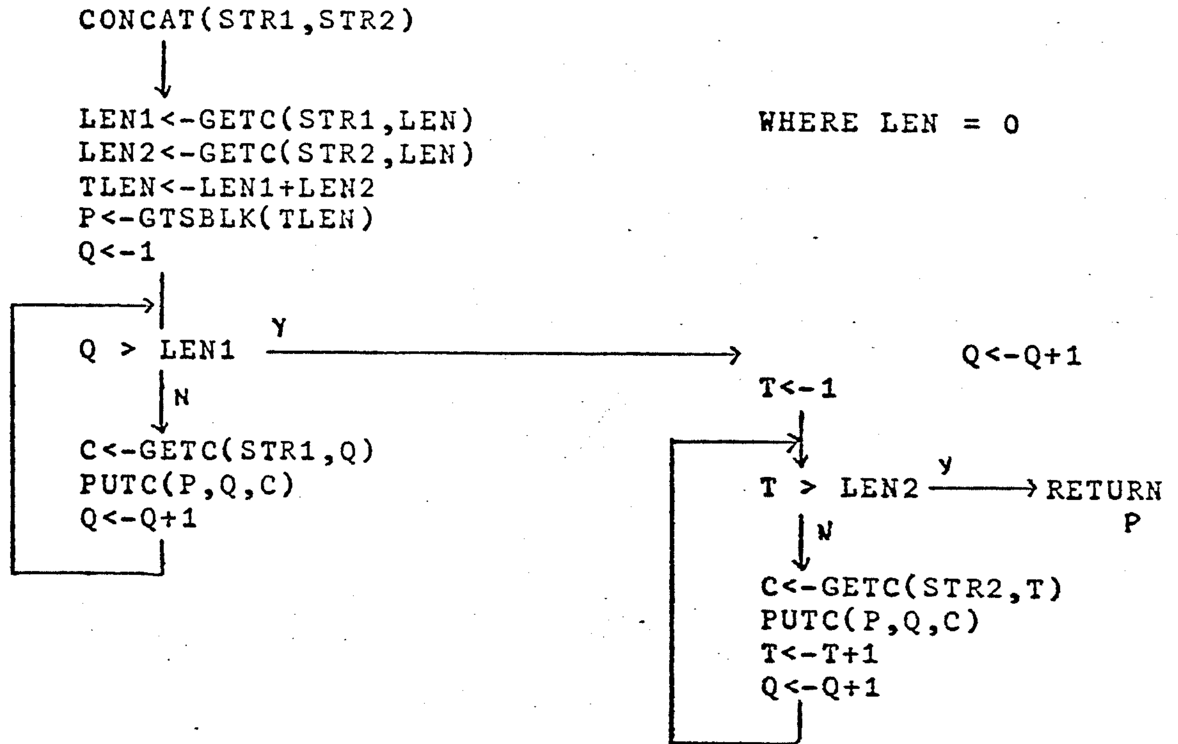
5. Functions Called:

GETC	PUTC	GTSBLK	TDIM	OPERAND
NUMBER	STRING			

6. Called By:

7. Error Calls: none

8. Flowchart:



1. Name: STREQ

2. Function:

   Compares two strings to determine if they are equal.

3. Description:

   The calling sequence is STREQ(STR1,STR2) where STR1 and STR2 are pointers to string blocks. TRUE is returned if equal, else FALSE.

   If the lengths of the two strings are the same they are compared characterwise until either a mismatch is found (and FALSE is returned) or the end of the string is reached (return TRUE).

4. Design Alternatives: none

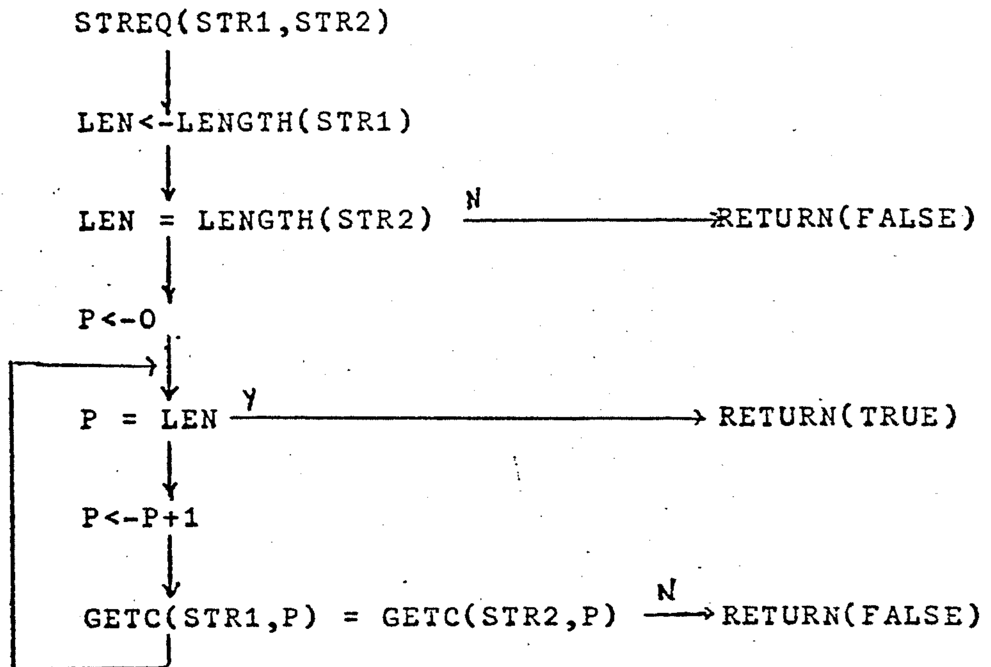
5. Functions Called:

   LENGTH GETC PARSE

6. Called By:

7. Error Calls: none

8. Flowchart:



1. Name: MATCH

2. Function:

Determines if a given string matches a substring starting at the current position of the given pointer and buffer.

3. Description:

The calling sequence is MATCH(BUF,BUFP,STRING) where BUF is a pointer to the start of a buffer, BUFP is a relative pointer for that buffer, and STRING is a pointer to the string to be matched against.

The comparision is done character by character until either 1) the end of the buffer is reached, 2) a mismatch is found, or 3) the match succeeds. In cases 1 and 2 FALSE is returned else TRUE.

4. Design Alternatives:

A matching string of the same length could be created by using the OUTSTR function and then the two strings compared using the procedure STREQ.

This is a cleaner procedure but would require changing OUTSTR to handle requests that cannot be filled, i.e. detecting a delimiter, or prescanning the buffer before the OUTSTR to determine if it can be accomplished.

5. Functions Called:

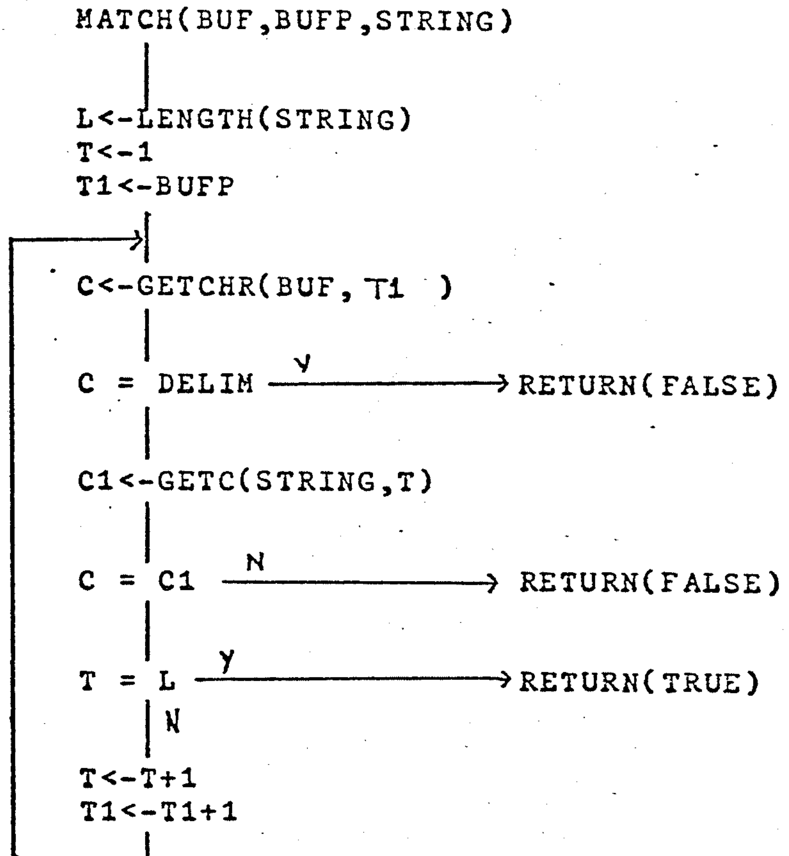
LENGTH GETCHR GETC

6. Called By:

LEXICAL TFOR TDEF

7. Error Calls: none

8. Flowchart:



4.3.4.4 Stack Accessing Procedures

PUSHR	- Push onto R-stack
PUSHS	- Push onto S-stack
PUSHT	- Push onto T-stack
POPR	- Pop off R-stack
POPS	- Pop off S-stack
POPT	- Pop off T-stack

1. Name: PUSHR

2. Function:

   Places a value on the R-stack.

3. Description:

   The calling sequence is PUSHR(VALUE) where VALUE is the item to be pushed onto the stack.

   A test is performed to see if stack overflow will occur, if so a diagnostic is issued through the system error routine.

4. Design Alternatives: none

5. Functions Called: none

6. Called By:

7. Error Calls:

   SYSERR(3): R-stack overflow

8. Flowchart: none

1. Name: PUSHS

2. Function:

   Places a value on the S-stack.

3. Description:

   The calling sequence is PUSHS(VALUE) where VALUE is the item to be pushed onto the stack.

   A test is performed to see if stack overflow will occur, if so a diagnostic is issued through the system error routine.

4. Design Alternatives: none

5. Functions Called: none

6. Called By:

7. Error Calls:

   SYSERR(1): S-stack overflow

8. Flowchart: none

1. Name: PUSHT

2. Function:

   Places a value on the T-stack.

3. Description:

   The calling sequence is PUSHT(VALUE) where VALUE is the item to be pushed onto the stack.

   A test is performed to see if stack overflow will occur, if so a diagnostic is issued through the system error routine.

4. Design Alternatives: none

5. Functions Called: none

6. Called By:

7. Error Calls:

   SYSERR(2): T-stack overflow

8. Flowchart: none

1. Name: POPS

2. Function:

   Return value on top of the S-stack.

3. Description:

   The pointer to S-stack is tested to see if the pop will cause stack underflow, if true then an error message is issued else the stack is popped and the value returned.

4. Design Alternatives: none

5. Functions Called: none

6. Called By:

7. Error Calls:

   SYSERR(4): S-stack underflow

8. Flowchart: none

1. Name: POPT

2. Function:

   Return value on top of the T-stack.

3. Description:

   The pointer to T-stack is tested to see if the pop will cause stack underflow, if true then an error message is issued else the stack is popped and the value returned.

4. Design Alternatives: none

5. Functions Called: none

6. Called By:

7. Error Calls:

   SYSERR(5): T-stack underflow

8. Flowchart: none

1. Name: POPR

2. Function:

   Return value on top of the R-stack.

3. Description:

   The pointer to R-stack is tested to see if the pop will cause stack underflow, if true then an error message is issued else the stack is popped and the value returned.

4. Design Alternatives: none

5. Functions Called: none

6. Called By:

7. error Calls:

   SYSERR(6): R-stack underflow

8. Flowchart: none

4.3.4.5 Accessing Input/Output and Working Buffers

Accessing Functions

The primitive accessing functions for buffers are GETCHR(BUF,RELP) and PUTCHR(BUF,RELP,CHR). BUF is a pointer to the beginning of some buffer and RELP is a pointer relative to the beginning of the buffer which indicates the character to access.

The use of relative buffer addressing has the advantages of 1) allowing us to write higher level string handling procedures independent of machine architecture and 2) of preserving greater process mobility.

Design Alternatives

If we assume the object machine has byte addressing features then a direct addressing pointer can be used. Direct character addressing can also by done if we assume one character/word or construct complex pointers that have word and character components.

1. Name: GETCHR

2. Function:

   Returns indicated character from some buffer.

3. Description:

   The calling sequence is GETCHR(BUFP,CHRP) where BUFP is a pointer to the beginning of some buffer and CHRP is an integer value which is a relative pointer from the beginning of the buffer. Thus the invocation GETCHR(NSRC,4) would return the fourth character from the beginning of the buffer NSRC.

4. Design Alternative:

5. Functions Called: none

6. Called by:

LEXICAL	PARSE	TFOR	TREAD	TDATA
TDIM	TINPUT	TPRINT	TDEF	EXPRESS
SUBTRAN	FUNCTRAN	STRING	NUMBER	OPERAND

7. Error Calls: none

8. Flowchart: none

1. Name: PUTCHR

2. Functtion:

Deposits a character into a buffer.

3. Description:

The calling sequence is PUTCHR(BUFP,CHRP,CHR) where BUFP is a pointer to the buffer, CHRP is a relative pointer in the buffer, and CHR is the value of the character to be deposited.

4. Design Alternatives:

5. Functions Called: none

6. Called by:

LEXICAL

7. Error Calls: none

8. Flowchart: none

Section Five

Program Mobility

5.1 Introduction

A technique for moving partial computations amongst similar processors is discussed in this section. The consequences involved in program transferability were presented in section 1. The following sections consider in detail when a computation may be interrupted and the canonical forms for representing structures and data.

5.2 Computation Interruption

The choice of the level at which a computation can be interrupted and moved is subject to measures of feasibilty and cost. Portions of the interpreter process that involve machine dependent activity cannot be halted. In particular, internalization procedures, storage reservation and liberation, and most run time functions must compute to completion. Procedures involving arithmetic processes (software implemented floating point arithmetic) are not interruptible. Since input/output can be included under run time functions they are not interruptible. The lower the level at which program interruption is allowed the greater

the amount of context information that must be saved.

The locations and number of program break points can be established by the implementor, however, restricting interruptions to the evaluation procedures appears reasonably effective. Within this scope break points may be established at the level of statement forms, interpretive expressions or functions. The size of the symbol table and transfer table are directly related to the level of program interruption.

5.3 Context String Formats

The context string components are canonical representations of the interpreter data structures and permit the transfer of user context without information loss. Section 1.5 described the organization and contents of the various structures. This section describes the formats used in forming the context string whose general overall format is shown in figure 5-1.

user information # statement forms # symbol table
R-stack # S-stack # T-stack #

Figure 5-1: General Context String

In the format descriptions the characters '#', '/' and ',' are used as delimiters and are abitrary choices left to the implementor.

5.3.1 Statement Forms

Each statement form has four components:

1) statement number
2) statement type
3) original source statement
4) interpretive expressions

The statement number and type are integers, the source string is a quoted string and the interpretive expressions are external representations of list expressions. The interpretive expressions are structurally identical to the original expression produced by the translator except that all symbols have been replaced by integers which are symbol table entry positions. When the statement forms are re-internalized it is unnecessary to search the symbol table.

The statement

```
10 LET K1 = LEN(R$)
```

when translated by the translator process generates

(10_statement type_"10 LET K1 = LEN(R$)"
(61_AK1_(92_BR))

The values 61 and 92 represent the symbol table entries for the assignment and length functions. The letters A and B preceding the variable names indicate the symbol types simple numeric and simple string. The underscore '_' represents a blank.

When this statement form is formatted for transfer to another interpreter the variable references are replaced by symbol table references.

(10_statement type_"10 LET K1 = LEN(R$)"
(61 104 (92 123)))

therefore the values 104 and 123 are the table positions assigned for the symbols K1 and R$.

5.3.2 Symbol Table

The symbol table context is preserved by scanning the table from top to bottom and outputting a string form representing all symbol table entries except those for system functions (type K).

Conversion from internal representations to character

formats is done using the PRINT and READ functions found in BASIC.

The order of symbol entries is preserved by the context string, a necessity since interpretive expressions and partial results on the R-stack reference symbol table items by relative location.

The general string format is given below in figure 5-2. Each symbol type and its format is discussed individually. The format of the symbol types is summarized in table 5-1 at the end of section 5.3.

/<symbol type>,[<data type>,][<symbol name>,]
<symbol value> [,<symbol value>]/

Figure 5-2: General Symbol Table Format

The square brackets enclose optional description elements. A symbol description may have multiple value elements depending upon the symbol type. Some of the notation used in describing the string formats is given below.

<symbol type>:= A | B | C | D | E | F | G | H | I
<data type>:= 0 | 1 | 2 | 3
<value>:= <integer> | <floating point> | <string> |

<list pointer>

<name>:= <letter> | <letter> <digit>

Symbol Table Context String Formats

Type A: simple numeric variables

Multiple values may be present from function calls.

/A,<name>,<floating point>,...,<floating point>/

Type B: simple string variables

Multiple values may be present from function calls.

/B,<name>,<string>,...,<string>/

Type C: dimensioned numeric variables

The number of value elements is indicated by the dimensions of the array which are elements three and four.

/C,<name>,<integer>,<integer>,<floating point>, ...,<floating point>/

Type D: dimensioned string variables

The format is similiar to that of the symbol type C except the the value elements are strings.

/D,<name>,<integer>,<integer>,<string>, ...,<string>/

Type E: function values

There is only one value element but it may be either a floating point value or a string.

/E,<name>,<value>/

Type F: floating point constant

/F,<floating point>/

Type G: string constant

/G,<string>/

Type H: temporary value

/H,<floating point>/ or /H,<string>/

Type I: system variables

/I,<data type>,<value>/

Table 5-1: Summary of Symbol Table String Formats

<table>
<tr><th>symbol type</th><th>string formats</th></tr>
<tr><td>A</td><td>/A,<name>,<flo pt>,...,<flo pt>/</td></tr>
<tr><td>B</td><td>/B,<name>,<string>,...,<string>/</td></tr>
<tr><td>C</td><td>/C,<name>,<integer>,<integer>,
<flo pt>,...,<flo pt>/</td></tr>
<tr><td>D</td><td>/D,<name>,<integer>,<integer>,
<string>,...,<string>/</td></tr>
<tr><td>E</td><td>/E,<name>,<string>|<flo pt>/</td></tr>
<tr><td>F</td><td>/F,<flo pt>/</td></tr>
<tr><td>G</td><td>/G,<string>/</td></tr>
<tr><td>H</td><td>/H,<flo pt> | <string>/</td></tr>
<tr><td>I</td><td>/I,<data type>,<integer> | <flo pt> |
<string> | <list pointer>/</td></tr>
</table>

5.3.3 R-stack

Pointers to evaluated arguments, temporary values and the results of function calls are placed on the R-stack. Actual values are never placed on the stack, instead, pointers in the form of relative locations in the symbol table are used.

The R-stack is represented as

#<integer>,<integer>,...,<integer>#

where each <integer> represents a relative symbol table location.

5.3.4 S-stack

The S-stack is composed of pointers (machine addresses) to list expressions that have yet to be evaluated or to which program control will later be returned. These expressions are either entire statement forms or sublists of statement forms which are placed on the S-stack as statement forms are evaluated.

Whenever a statement form is entered for evaluation then the statement number is pushed on the stack followed by a delimiter. Subsequent pointers are all found in that

statement form until the next statement number/delimiter pair occurs. The stack has the general composition given in figure 5-3.

<u>S-stack</u>

Statement number
Delimiter
Pointer(1)
Pointer(2)
.
.
Pointer(n)
Statement number
Delimiter
Pointer(1)
.
.

Figure 5-3: S-stack Composition

Knowing the statement form which the list pointer references allows use to convert the machine address pointers to strings that provide relative list positions similiar in form to the list addressing information as used in the BBN LIST editor [1].

The list form given in figure 5-4 has several pointers which are shown referencing parts of a list structure.

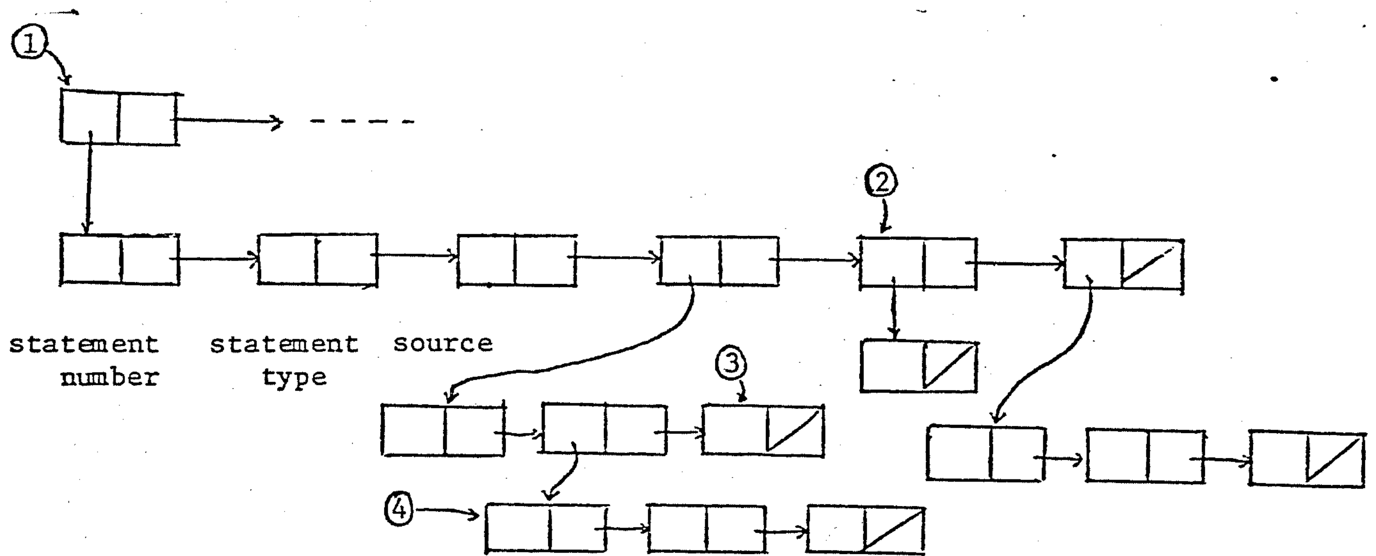


Figure 5-4: Example List Structure

The pointers can be represented in the form of relative list addressing shown below.

Pointer 1: 0
Pointer 2: 5
Pointer 3: 4/3
Pointer 4: 4/2/1

The meaning of the pointers can be seen more clearly by viewing the structure in list form.

```
1            4      3   2
(A B C (D (E F G) H) (I) (J K L))
```

Thus, for pointer 1, 0 is the top most level, or a pointer to the statement form. Pointer 2 is 5 or the 5th element of the list. Pointer 3 is to the 4th element of the

list and then to the 3rd element of that list. The addressing scheme is applied recursively to the list being accessed. The structure referenced by pointer 4 is found by taking the list which is the 4th element, the 2nd element of that list and then the 1st element of that list.

The entire context string would appear as in figure 5-5.

#statement number_0_5_4/3_4/2/4,....#

Figure 5-5: S-stack portion of Context String

Design Alternatives

An elegant and more general solution suggested by Martin Kay is that the relative position of the pointer on the stack be inserted at the position that it references in the program list. As the statement forms composing the list are internalized the pointers are used to reconstruct the S-stack.

5.3.5 T-stack

The information regarding control transfers within the interpreter can be preserved by making such transfers

indirect through the transfer table. In practice this causes all function calls and jumps to be made indirectly through a relative transfer table entry. Address information, expressed as relative transfer table entries, is placed on the T-stack.

The T-stack is represented in the format shown in figure 5-6.

#<iNteger>,<integer>,...,<integer>#

Figure 5-6: Format of T-stack Context String

Each string element delimited by a comma is a relative pointer into the transfer table.

9. Rowe, D.A., Hopwood, M.D., Farber, D.J. "Software Methods for Achieving Fail-Soft Behavior in the Distributed Computing System." Proc. IEEE Symposium on Computer Software Reliability, April 1973.

10. Sattley, Millstein, and Warshall. "On Program Transferability", November 1970, NTIS Document #AD-716-476.

11. Waite, W.M. Implementing Software for Non-Numeric Applications. (Prentice Hall, 1972).

APPENDIX A

Procedure Index

ALLOCATE.................................... 4-37
ALPHA....................................... Appendix B
ARG... 4-50
ATOM.. Appendix B
BINARY...................................... 3-71
BIND.. 4-71
CAR... 4-25
CDR... 4-25
CODE.. 3-69
CONCAT...................................... 4-78
DELSTR...................................... 4-42
DIGIT....................................... Appendix B
EEXPR....................................... 4-67
ESUBR....................................... 4-66
ETYPE....................................... 4-65
EVAL.. 4-64
EVALE....................................... 4-63
EVALP....................................... 4-59
EVLARG...................................... 4-69
EXECUTE..................................... Appendix B
EXPRESS..................................... 3-54
FINDSYM..................................... 4-52
FLCNVT...................................... 4-57
FPREC....................................... 3-72
FRBLK....................................... 4-26
FRCELL...................................... 4-25
FRFBLK...................................... 4-26
FRFCELL..................................... 4-26
FRSBLK...................................... 4-27
FUNCTION.................................... Appendix B
FUNCTRAN.................................... 3-63
GETBLK...................................... 4-25
GETC.. 4-75
GETCELL..................................... 4-25
GETCHR...................................... 4-92
GETCORE..................................... 4-41
GETFBLK..................................... 4-26
GETFCELL.................................... 4-266
GETKEY...................................... 3-20
GETS.. 4-17
GETSBLK..................................... 4-27
GETTRN...................................... Appendix B

GETUIT.................................... Appendix B
GPREC..................................... 3-73
INSERTS................................... Appendix B
INTCNTL................................... Appendix B
INTERNCS.................................. Appendix B
INTERNIE.................................. 4-48
INTERNSF.................................. 4-44
INTERNSTB................................. Appendix B
INTERNSTK................................. Appendix B
INTERNUIT................................. Appendix B
LENGTH.................................... 4-77
LEXICAL................................... 3-13
MATCH..................................... 4-82
NUMBER.................................... 3-67
OPERAND................................... 3-57
OUTL...................................... Appendix B
OUTSTR.................................... Appendix B
PARSE..................................... 3-17
POPR...................................... 4-87
POPS...................................... 4-88
POPT...................................... 4-89
PUSHR..................................... 4-85
PUSHS..................................... 4-86
PUSHT..................................... 4-87
PUTC...................................... 4-76
PUTCHR.................................... 4-93
PUTS...................................... 4-18
PUTUIT.................................... Appendix B
REBIND.................................... 4-73
RELEASE................................... 4-39
SETVAL.................................... Appendix B
STR2INT................................... 4-56
STR2STR................................... 4-55
STREQ..................................... 4-80
STRING.................................... 3-65
SUBTRAN................................... 3-61
TDATA..................................... 3-33
TDEF...................................... 3-49
TDIM...................................... 3-36
TEND...................................... 3-41
TFNEND.................................... 3-52
TFOR...................................... 3-25
TGOSUB.................................... 3-34
TGOTO..................................... 3-24
TIF....................................... 3-42
TINPUT.................................... 3-44
TLET...................................... 3-22
TNEXT..................................... 3-29

TPRINT.. 3-47
TREAD... 3-31
TRERR... 3-7
TRESTORE.. 3-46
TRETURN... 3-35
TRNCNTL... Appendix B
TSTOP... 3-40
WFCELL.. 4-30
WPCELL.. 4-30
WSCELL.. 4-30

Appendix B

Partial Procedure Descriptions

The procedures in this appendix are referenced in the design document but do not have complete descriptions. These procedures are described briefly in this section so that the implementor is made aware of their function.

Function: ALPHA(X)

Returns TRUE if character X is alphabetic.

Function: ATOM(X)

Returns TRUE if item X is atomic.

Function: DIGIT(X)

Returns TRUE is character X is a digit.

Function: EXECUTE(X)

Transfers control to machine subroutine whose address is X. These are usually system functions. This function is invoked in the procedure ESUBR.

Function: FUNCTION

Invoked by the procedure OPERAND to determine if the next characters in the buffer NSRC is a function name (either user or system). FUNCTION returns TRUE if the function name is present and places the symbol table address on the S-stack.

Function: GETTRN(N)

Value returned is the Nth item in the transfer table.

Function: GETUIT(N)

Returns the Nth entry in the User Information Table (UIT).

Function: INSERTS

Inserts the current statement form into the program list.

Function: INTCNTL

Top level procedure for the interpreter process.

Function: INTERNCS

Controlling procedure for the internalization of the context string.

Function: INTERNSTB

Function to internalize symbol table portion of the context string.

Function: INTERNSTK

Function to internalize stack portions of the context string.

Function: INTERNUIT

Function to internalize user information portion of the context string.

Function: OUTL(N)

Returns a pointer to a string composed by forming a list using the top N elements of the S-stack.

Function: OUTSTR(BF,BEGP,ENDP)

Returns pointer to a string block formed from the sequence of characters in the buffer BF from the position BEGP to ENDP.

Function: PUTUIT(N,VAL)

Deposits the value, VAL into the Nth position of the user information table.

Function: SETVAL(FN)

Pops the top value off the R-stack and makes it the value of the function FN.

Function: TRNCNTL

Top level procedure for the translator process.