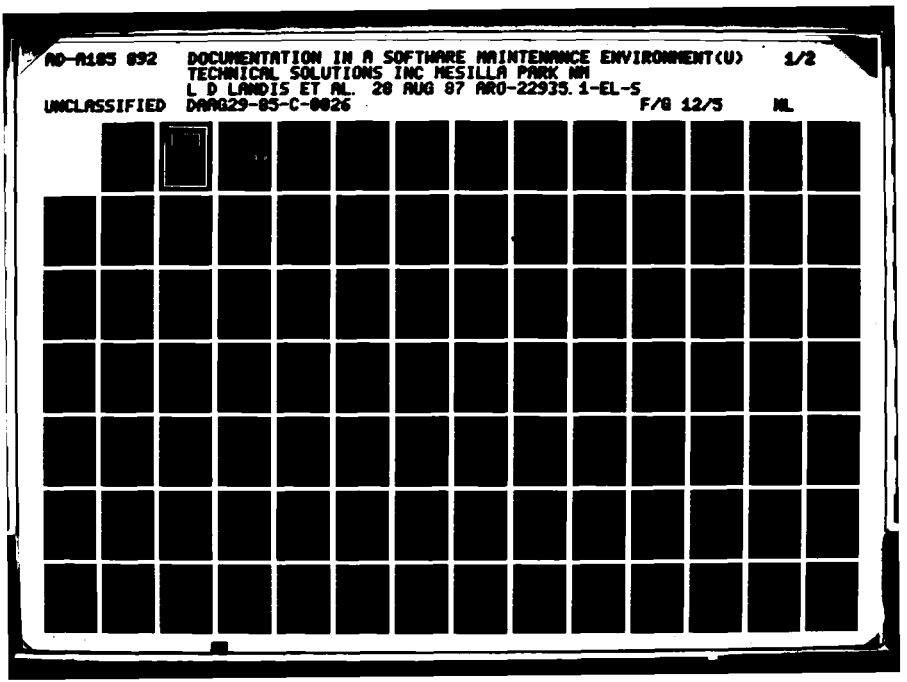
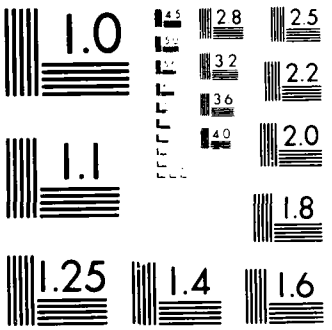

AD-A185 892
DOCUMENTATION IN A SOFTWARE MAINTENANCE ENVIRONMENT(U)
1/2
TECHNICAL SOLUTIONS INC MESILLA PARK NM
L D LANDIS ET AL. 20 AUG 87 ARO-22935.1-EL-S
UNCLASSIFIED
DAAG29-85-C-0026
F/G 12/5
NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

ARO 22935.1-EL-S

AD-A185 892

DTIC FILE COPY (2)

Documentation In A Software Maintenance Environment

DTIC
ELECTE
OCT 2 8 1987
S D
C D

DISTRIBUTION STATEMENT A
Approved for public release
Distribution Unlimited

TECHNICAL SOLUTIONS, *Inc.*

SALES & TRAINING
465 N. RESLER, SUITE A
EL PASO, TEXAS 79912
(915) 581-1819

TECHNICAL CENTER
P.O. BOX 1148
MESILLA PARK, NM 88047
(505) 524-2154

(2)

Documentation In A Software Maintenance Environment

L. D. Landis
A. J. Fine
P. M. Hyland
W. L. Hembree
A. L. Gilbert

DTIC
ELECTE
OCT 2 8 1987
S D
D

Technical Solutions, Inc.
PO Box 1148
Mesilla Park, NM 88047
(505)-524-2154
08/04/87

DISTRIBUTION STATEMENT
Approved for public release
Distribution Unlimited

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

A185 892

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: ARO 22935.1-EL-S	2. GOVT ACCESSION NO.: N/A
3. RECIPIENT'S CATALOG NUMBER: N/A	
4. TITLE (and Subtitle): Documentation in a Software Maintenance Environment	5. TYPE OF REPORT & PERIOD COVERED: Interim Technical 1 Aug 85 thru 31 July 87
	6. PERFORMING ORG. REPORT NUMBER:
7. AUTHOR(s): P.M. Hyland, L.D. Landis, W.L. Hembree, A.J. Fine, A.L. Gilbert	8. CONTRACT OR GRANT NUMBER(s): DAAG29-85-C-0026
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Technical Solutions, Inc. P.O. Box 1148 Mesilla Park, NM 88047	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: N/A
11. CONTROLLING OFFICE NAME AND ADDRESS:	12. REPORT DATE: 28 Aug 87
	13. NUMBER OF PAGES: 90
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office):	15. SECURITY CLASS. (of this report): Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

N/A

18. SUPPLEMENTARY NOTES

The view, opinion, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

computer languages, compilers, formal language theory, documentation

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Software has a limited lifetime of usefulness, because as existing software ages support becomes more difficult. Major factors in determining when to replace rather than maintain software are the cost of and to the time required to train new personnel to provide maintenance support. A primary source of information in training new personnel, making modification or repair easier, is accurate documentation.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20.

By providing automatic techniques to generate documentation of existing software, the life-cycle of software may be extended through accurate information about the current state of the software.

The objective of this effort was to explore and select methods of documentation that automatically extract and display program logic from source code, thereby extending the life-cycle of software. The domain of problems our research examined involved looking into the various weaknesses and vulnerabilities of software and documentation that ages.

Table of Contents

DTIC COPY INSPECTED 1

Accession For
NTIS GRA&I
DTIC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Dist Special
A-1

Documentation In A Software Maintenance Environment

Abstract

Software has a limited lifetime of usefulness, because as existing software ages support becomes more difficult. Major factors in determining when to replace rather than maintain software are the cost of and to the time required to train new personnel to provide maintenance support. A primary source of information in training new personnel, making modification or repair easier, is accurate documentation.

By providing automatic techniques to generate documentation of existing software, the life-cycle of software may be extended through accurate information about the current state of the software.

The objective of this effort was to explore and select methods of documentation that automatically extract and display program logic from source code, thereby extending the life-cycle of software. The domain of problems our research examined involved looking into the various weaknesses and vulnerabilities of software and documentation that ages.

Keywords

Computer Languages, Compilers, Formal Language Theory, Documentation

Introduction

There are two distinct classes of documentation that need to be provided with a software package: user level documentation and system/internal documentation.

User level documentation defines the scope of the problem solved, and how to use the package. Typically this documentation is done to the degree that the user community requires, and is not a substantial problem. Also, functional or user level documentation does not change significantly over time, since users do not want to relearn how to use their software; i.e, upward compatibility usually exists. Large functional changes generally result in new user documentation, which then is not changed significantly until another major revision.

On the other hand, system or internal documentation can become a problem because it details how the package solves the problem. These internal details may change significantly, without changing what the user sees or how the system is used.

This research concentrates on the problem of documenting the internal mechanisms of a package (from the production source code) and representing this detail to the maintenance programmer for training and for verification of functional processes as changes are made to the software. Documentation, as used in this report, is to be understood as that documentation that is of an internal or systems nature rather than user level documentation.

Overview of the Problem

Internal documentation has historically had a low priority in a project, usually not by design, but rather due to lack of time or other considerations. If other phases of a project were poorly estimated, or the emphasis was to get it out the door before the competition makes it obsolete, the internal documentation is typically the effort that suffers.

Even if documentation was done when the project was initially completed, over time it inadequately defines the working or production system. People seldom accept responsibility for updating documentation, hence documentation seldom conforms to new changes. Internal documentation falls into disuse if maintainers discover that they have been misled by inaccuracies or, worse, by not recognizing the inaccuracies they place more programming flaws in the system.

Maintenance is more difficult as systems become larger. Large systems can become too complex for one person to understand and team maintenance further complicates the problem. Significant amounts of old software exist, much of it written before *structured programming*, without the benefit of modern software engineering techniques. This software often suffers from such maladies as unstructured code, machine hardware dependencies, operating system dependencies, and large amounts of suspected dead code. Performance and maintenance efforts on these large systems are barely manageable, and tend to become less manageable over time. Several generations of changes can even transform what was a well defined structure into a confusing and error-filled patchwork.

These problems may be unavoidable, and there are situations that make the problems much more difficult to solve, e.g., external factors involving system organization or available personnel. Size and system complexity can aggravate software problems. Sensitivity of software to small modifications rapidly increases with software size since large software systems have more interfaces and exceptions, leading to effects that one person might not completely understand. Effects begin to look organic, in that a software system can respond to a small correction in one area with large compensations in other areas.

A further contribution to maintenance problems is that software developers prefer to develop new software rather than maintain old software. Development assignments are often reserved for those who have paid their dues and many creative, highly skilled programmers have prejudices against maintenance assignments.

Generally, developers of new software are reluctant to maintain software for very long after the product has been accepted by the end user. For these reasons, original developers of the system are usually unavailable because they have moved on to something new. The result is that maintenance programming is where most programmers begin their careers. Less experienced programmers are more likely to introduce program flaws, worsening the overall poor condition of an aging application.

Even if all these problems and vulnerabilities are avoidable, all software comes to one of three ends:

- Modified for new or changed tasks
- Rewritten when modification is no longer considered cost effective
- Destroyed when no longer needed

Modifying programs is preferable to deleting or rewriting them as most of the application need has been (or is perceived to have been) solved. The problem is: how can the software life-cycle be extended so modification remains cost effective?

Our Motives

This company has interests in various areas closely related to software development:

- Large commercial applications
- Documentation and diagnosis of large applications
- User libraries
- Languages; development, implementation, and compiler support
- Integration of standard routines into current applications
- Aids for data usage evaluation

- Theory development for translators that convert algorithms to natural language descriptions

- Combat simulation models

In addition to training our programming personnel in the basic modeling, graphics and user library software, we incorporate libraries written by others into existing software. As an example of the need for maintainance documentation, a portion of our business currently use combat simulation models originally developed in FORTRAN by several different programmers over a period of several years. The maintainance problem is compounded when the COMBIC and EOSAEL atmospheric obscurants model from the Atmospheric Sciences Laboratory (ASL) are integrated with several weapons effectiveness evaluation models. Knowing how these routines work, without having to extensively read the source code, helps in reducing the training time.

Users of models and other software systems may not be programmers. In many cases it is useful to these users if they can examine the methods being used in the program to validate the results given by the model. Provided documentation of the program logic, in the form of diagrams and charts, these personnel can more effectively participate in the design and implementation of the system they are using.

Approach

Our approach to automatically generating maintenance related documentation consists of two aspects: selection of useful documentation methods for the maintenance environment; and definition of a documentation language with an initial implementation of a compiler for the documentation language and display processors.

Our objective was to build a working documentation generator using existing compiler technology and software engineering methods.

Our Research

This company was awarded a contract [1] to do basic research in automatic documentation generation on existing maintenance intensive, software environments.

The goal of our research is to investigate methods aiding software life-cycle extension. Modification remains a cost effective alternative so long as it involves less effort and expense than rewriting the software system. Maintainers can make repairs in a more cost effective manner when provided adequate documentation, and the software life-cycle may be extended by rejuvenating the software.

Also, techniques for extending the life of the software can be developed by redocumenting software as it ages; delaying the time when modifications are no longer cost effective. These techniques then become tools, improving maintenance efficiency, allowing software to age more gracefully.

Our investigation plan has been to:

- Develop a documentation language, DL, that supports existing third generation languages (i.e. FORTRAN, Pascal and C) and the newer third generation languages such as Ada

[1] Army Research Office Contract: DAAG29-85C-0026

[2]Ada is a registered trademark of the U.S. Government (Ada Joint Program Office)

- Investigate existing techniques of documentation and determine which techniques are most suitable to the maintenance process
- Develop new techniques and algorithms for automatic documentation generation

Project Scope

Research emphasis is on high level, sequential, algorithmic languages. Examples of these are FORTRAN, C, Pascal, and Ada. Assembly languages were not considered since a general solution across systems is impractical [3].

Goal directed programming languages, such as Prolog and Icon, are not being considered as their use is limited. Parallel programming problems are also not being considered since little software yet exists. These and other fourth generation languages (4GL's) are too new, and the experience with maintenance of large software systems in 4GL's is too limited to evaluate documentation requirements for this class of languages. The radically different nature of 4GL's (from the third generation languages like FORTRAN and COBOL) supports the suspicion that substantially different maintenance documentation is required from that needed in the third generation language environment.

Evaluation of existing documentation techniques was the initial task, since many forms of documentation exist. Investigation revealed that while many helpful techniques exist for the software development phase, there are not enough useful documentation techniques for the maintenance phase.

Next, a DL was developed that was capable of representing a wide variety of programming languages. Since the majority of in-house software is written in FORTRAN and C, emphasis on handling those languages has been paramount, but the approach has been left open-ended to allow for expansion of the automatic documentation generation system to include other languages.

[3] Note: large degrees of machine/implementation dependency in higher level languages are possible. FORTRAN serves as a case in point, but this does not usually apply.

Automatic Documentation Generation

The initial focus has been on development of automatic documentation tools for FORTRAN. Since our sponsors have large holdings of software written in FORTRAN, and FORTRAN is still widely accepted for standard use, specifically with currently developed software. There also exists a sufficient base of aging, problem software in this language. A compiler for DL was developed to the point that all constructions in FORTRAN and C were handled.

Since our tools are to eventually operate on software written in other languages (Ada, C, COBOL, Pascal), the implementation strategy was necessarily open ended allowing for incorporation of structured programming environments.

Documentation Techniques

Large amounts of maintenance phase software were examined. Languages represented in the sample were: FORTRAN, DIBOL, C, and BASIC. All applications examined have existed for several years, and are constantly used and updated.

Maintenance programmers were interviewed to help clarify the types of problems occurring in the real-world maintenance environment. Also consulted were members of our staff who work on FORTRAN programs. These programmers range from entry level to research associates, and are in our informal maintenance training program. Many of our new hires work on the large combat simulation models, which are written in FORTRAN. There, they deal immediately with problems in system complexity, system testing, and system modification.

Several standard documentation techniques were examined with a view to usefulness in a maintenance environment. In general, it was found that most techniques are not appropriate for the maintenance portion of the software life-cycle. The primary reason for this is that there is a significant amount of detail present in the actual production code.

Elimination of detail requires too much in the way of understanding the effect of the code to avoid such things as the Turing Halting Problem. Mechanisms must be developed to allow a human to manually eliminate the low level detail, thus making possible documentation of an overview nature. The decision to document code at the low level was made.

Existing Documentation Techniques Evaluation

The following published documentation techniques were evaluated from the perspective of providing documentation in the maintenance environment:

- Pretty Printing
- Warnier-Orr Diagrams

- Michael Jackson Diagrams
- Flowcharts
- Pseudocode / Structured English
- Nassi-Shneiderman Charts
- Action Diagrams
- HOS Charts

Evaluations of each of these documentation techniques are given in Appendix A.

Software Maintenance Evaluation Results

The evaluation criteria used to evaluate suitability of documentation methods for the maintenance environment were:

- Overview of the system
- Program architecture display
- Detailed program logic
- File and database structure

These methods have properties generally suitable by all our criteria:

- Warnier-Orr diagrams can graph program, system and data structure in hierarchical format
- Action diagrams are designed to show all details of an organization at all levels
- HOS charts can describe mathematically rigorous tasks regardless of task size

These methods are considered suitable by the criterion of detailed program logic. They satisfy no other criteria.

- Nassi-Shneiderman charts
- Flowcharts
- Pseudocode / Structured English

Each is specialized for describing procedures, with different aspects of flexibility and elegance. However, they all have considerable problems attempting to describe anything large scale. Hence they are best only for describing the detailed program logic of a single routine.

Tools Initially Selected

Research is being focused on a few tools for the initial development. Although not a documentation technique, many requests were made for good cross references of the code. The decision was made to include a cross-reference listing capability. These conclusions were reached after research was completed on available tools, and the initial tools selected were:

- Cross reference generation
- Nassi-Shneiderman charts for documenting detailed program logic
- Action diagrams for documentation and maintenance of overall systems, programming, and data structures

Applications Anticipated

Good documentation should provide an abstraction of the program at every level of the program. These abstractions provide information useful to programmers and analysts alike, depending on the level of detail desired and the code to be examined. One way to envision this is to contemplate beginning with the individual blocks of code and documenting, or abstracting, them. Then move up one level and abstract all code blocks below. Continue moving up a level at a time, the documentation for the current level being the abstraction of everything at the next lower level in the program. When finished, there exists documentation for the entire program, and an abstraction of the program, level by level.

These abstractions are of use to programmers, checking the individual operations for errors, and to analysts, looking for correctness of the implementation of the program design, assuming there was a design in the first place.

Of the tools selected for initial development, action diagrams and Nassi-Shneiderman charts appear appropriate for abstracting various levels of program. Action diagrams can provide a semi-pictorial representation of the documentation tree structure. Nassi-Shneiderman charts provide a block-structured representation of program structure and may not be appropriate for more than selected blocks of code at a time.

Applications anticipated as useful extractions of relevant information from the compiled DL include:

Cross reference generation

- A list of all identifiers with similar names, to check for typing errors in languages like Fortran that allow variable usage without a prior declaration.
- A list of assignments to variables declared to be of a different type, in languages like C with weak type checking.
- A list of what subroutines are called by a selected piece of code, useful for tracking the side effects of a given change.

Action diagrams

- An interactive system that would allow the user to supply an abstraction for a given piece of code, in order to see less detail and more the overall purpose of the code.
- A documentation "tree", showing the various levels within the program either execution order or lexical order.
- An abstraction of a piece of code, consisting of the inputs, outputs, and all data modified within that piece of code.

Nassi-Shneiderman charts

- Block-structure diagrams that provide a pictorial representation of a block of code. This is useful for programmers new to the language used, as well as programmers new to a given piece of software.
- An abstraction of a piece of code, consisting of the inputs, outputs, and all data modified within that piece of code.

Also useful would be a pictorial representation of the various data structures used in the program. This is especially important in languages that employ data-hiding, where there may be several levels of data abstraction to weave through before grasping the actual structure involved.

Documentation Language Requirements

DL provides notation capable of representing programs written in FORTRAN, Pascal, C and eventually, Ada.

These languages (especially C, Ada and Pascal) support structured programming techniques and are present on the current generation of systems. Additionally, they are similar enough to easily combine into a language that can represent them all.

DL was required to represent programming constructions of several languages to include FORTRAN, Pascal, C and in the future, Ada. Specific needs were: block structure (localized declarations, scope of name bindings), standard data types, user defined data types, aliasing, operator overloading, nested procedures/functions, and encapsulation (packages).

Block Structure

Each block defines an environment and definitions that are local to that block, are bound to the block and inaccessible outside of the block. This allows data hiding. Blocks may be named (as in Ada) or unnamed.

Standard Types

Provisions of basic data types and data structuring allows the source language to be translated into DL. Although based on C, DL is different from C in that several extensions were necessary. Basic data types include enumerations and scalar data (including character, signed/unsigned short integer, signed/unsigned long integer, float, double, complex, and double complex).

User Defined Types

Users may define structured types using structures (Pascal/Ada records). A special class of *structure*, the *union*, may be used to map several declarations to the same space (Pascal/Ada variant records).

Aliasing

A *typedef* facility allows the user to create new names for new or existing types. FORTRAN *equivalences*, another form of aliasing, is handled by defining unions, which provide mapping several variable names to the same storage locations.

Operator Overloading

Control over operator overloading is given to accommodate differences in type promotion/demotion rules for different languages. The most common overloading is for arithmetic operators like "+", "-", "*" and "/". In these cases, the promotion of characters to short integers, short integers to long integers, long to float, float to double, and double to complex are understood so that expression result types are known. Consideration has been given to allowing arbitrary user specified operators so that DL can accommodate fully overloaded and user defined evaluation rules.

Function and Procedure Overloading

Overloading of function and procedure names is required for Ada but not for FORTRAN, so work in allowing this form of overloading has been deferred.

Nested Procedures and Functions

Languages like Pascal and Ada allow nesting of procedures and functions. This allows for controlling the scope (i.e. accessibility) to functions.

Encapsulation

Creation of modules (Ada packages) was considered, but is not implemented. This will allow the developer of code to define interfaces to functions without giving direct access to the internal workings of the functions. Access to the exported functions, procedures and variables is controlled by the module using *struct*-like construction, where the *struct* is the name of the package, and the members of the *struct* are the functions, procedures and variables of the module interface.

Implementation Strategy

Developing an open ended, extensible approach required designing a general purpose documentation language, named *DL*, and developing a compiler for it. Maintenance tool output generators and translators from specific programming languages can then be developed, which use the information derived by the compiler. The following sections present a conceptual overview of the parser/documenter system. Appendix B presents the system-level implementation of the system while Appendix C provides a detailed look at the internal symbol table structures used in the parser/documenter.

Documentation Language

The initial definition of DL was chosen to be the X3J11 standard of the C programming language. C provides an abstract representation which combines data structures at their most primitive level, operators for these primitive structures, and function/subroutine calls. Also, C has sufficient flexibility to represent itself and higher level programming languages. C compilers are established on many computers, and have nearly no special requirements for running on a specific machine.

The new standard of C, developed by the ANSI X3J11 committee, contains revisions addressing problems left unresolved by the older standard, as defined by Kernighan and Ritchie. Translation programs convert software written in the different programming languages and express them in DL. DL syntax is given in Appendix D as a series of flowgraphs.

Documentation Language Compiler

The DL compiler transforms DL source code into a symbol table specifically developed to represent the complete set of semantics available in DL. This transformation is reversible, since the complete set of semantics are preserved in the transformation.

Preprocessors for conversion of standard languages such as FORTRAN and Ada to DL are planned which allow for support of large bodies of software currently in maintenance.

Development is being done under 4.2bsd Unix running on a VAX 11/750. The Unix environment provides a rich set of utilities that aid in the development of compilers and preprocessors. Unix utilities employed were *lex* (for lexical analysis), and *yacc* (for syntax analysis).

Regular expressions, representing the *tokens* of the goal language, are used by *lex* to represent (i.e., *tokenize*) a language's lexical characteristics. Context-free expressions representing a language's syntax are used by *yacc* to generate an LALR(1) parser (look ahead one token left to right). Both representations use *action code* to perform semantic interpretations. The term *action code* refers to the portions of the *lex* or *yacc* source designated to perform syntax directed translation.

Description of *lex*

The input to *lex* consists of a table of regular expressions and corresponding *action code*, which *lex* translates into a C source program. The resulting source program is a software implementation of a deterministic finite-state automaton (DFSA) that recognizes the regular expressions from the input stream, and executes the program fragment to operate on the recognized text.

The DFSA is interpreted, rather than compiled, in order to save space. The automaton interpreter directs the control flow, and the user is allowed to insert additional declarations and statements, or to use external subroutines.

Ambiguous specifications are accepted, in which case *lex* recognizes the longest possible match. If several matches are of equal length, then the first match is used. User supplied action code is then executed and may further refine or reject the match, or do other processing needed by the application. Although it is a very useful tool, *lex* is not a complete language, but rather a recognizer generator that creates a subroutine that processes all input, freeing the user from having to write and rewrite lexical analyzers.

By design, *lex* is particularly suited to integrate with the *yacc* parser generator; where *lex* performs the lexical analysis phase, and the parser recognizes syntactic elements of the language.

Description of *yacc*

Although it is called a compiler compiler, *yacc* is a very flexible and general tool for describing the input expected by a computer program. The *yacc* user specifies the structures of the input, together with code to be invoked as each such structure is recognized. This input is converted by *yacc* into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

The parser subroutine produced by *yacc* calls the lexical analysis routine, previously discussed, that returns the next basic input item, called a token. These tokens are organized according to the input structure rules, called *grammar rules*; when one of these rules has been recognized, then user code supplied for this rule, an *action* is invoked; actions have the ability to return values and make use of the values of other actions.

The heart of the input specification is a collection of grammar rules. In our case the grammar of the ANSI X3J11 C language was expressed using the *yacc* grammar rules. When the input being read does not conform to the specifications, syntax errors are detected as early as is theoretically possible with a left-to-right scan.

While *yacc* cannot handle all possible specifications, its power compares favorably with similar systems. Moreover, the constructions which are difficult for *yacc* to handle are also frequently difficult for human beings to handle.

Actions that do not terminate a rule are actually handled by *yacc* by manufacturing a new nonterminal symbol name, and a new rule matching this name to the empty string. The interior action is the action triggered off by recognizing this added rule.

In our application, a data structure representing the parse tree is constructed in memory, and transformations are applied to it before output is generated. Parse trees are particularly easy to construct, given efficient routines to build and maintain the tree structure desired.

The parser produced by *yacc* consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the *look-ahead token*). The *current state* is always the one on the top of the stack. The states of the finite state machine are given small integer labels; initially, the machine is in state 0, the stack contains only state 0, and no look-ahead token has been read.

The machine has only four actions available to it, called shift, reduce, accept, and error. A move of the parser is done as follows:

- Based on its current state, the parser decides whether it needs a look-ahead token to decide what action should be done; if it needs one, and does not have one, it calls the *lex* entry function *yylex* to obtain the next token

- Using the current state, and the look-ahead token if needed, the parser decides on its next action, and carries it out; this may result in states being pushed onto the stack, or popped off of the stack, and in the look-ahead token being processed or left alone

The shift action is the most common action the parser takes. Whenever a shift action is taken, there is always a look-ahead token. The look-ahead token is cleared as a result of the shift. The reduce action keeps the stack from growing without bounds. Reduce actions are appropriate when the parser has seen the right hand side of a grammar rule, and is prepared to announce that it has seen an instance of the rule, replacing the right hand side by the left hand side. It may be necessary to consult the look-ahead token to decide whether to reduce, but usually it is not; in fact, the default action is often a reduce action.

Reduce actions are associated with individual grammar rules. The reduce action depends on the left hand symbol, and the number of symbols on the right hand side. To reduce, first pop off the top *n* states from the stack (in general, the number of states popped equals the number of symbols on the right side of the rule).

In effect, these states were the ones put on the stack while recognizing the tokens which no longer serve any useful purpose. After popping these states, a state is uncovered which was the state the parser was in before beginning to process the rule. Using this uncovered state, and the symbol on the left side of the rule, perform what is in effect a shift of the left side. A new state is obtained, pushed onto the stack, and parsing continues. There are significant differences between the processing of the left hand symbol and an ordinary shift of a token, however, so this action is called a *goto* action. In particular, the look-ahead token is cleared by a shift, and is not affected by a *goto*.

In effect, the reduce action turns back the clock in the parse, popping the states off the stack to go back to the state where the right hand side of the rule was first seen. The parser then behaves as if it had seen the left side at that time. If the right hand side of the rule is empty, no states are popped off of the stack: the uncovered state is in fact the current state.

The reduce action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack, running in parallel with it, holds the values returned from the lexical analyzer and the actions. When a shift takes place, the external variable value (supplied by the lexical analyzer) is copied onto the value stack. After the return from the user code, the reduction is carried out. When the goto action is done, the external variable is copied onto the value stack.

The other two parser actions are conceptually much simpler. The accept action indicates that the entire input has been seen and that it matches the specification. This action appears only when the look-ahead token is the end marker, and indicates that the parser has successfully done its job. The error action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen, together with the look-ahead token, cannot be followed by anything that would result in a legal input. The parser reports an error, and attempts to recover the situation and resume parsing.

It does this by selecting one of the valid steps wherever it has a choice. A rule describing which choice to make in a given situation is called a *disambiguating rule*. Two disambiguating rules are invoked by default by *yacc*:

- In a shift/reduce conflict, the default is to do the shift
- In a reduce/reduce conflict, the default is to reduce by the earlier grammar rule (in the input sequence)

The first rule implies that reductions are deferred whenever there is a choice, in favor of shifts. The second rule gives the user rather crude control over the behavior of the parser in this situation, but reduce/reduce conflicts should be avoided whenever possible.

Issues of Completeness

In defining Documentation Language (DL) it was necessary to examine all of the languages that we anticipated encountering. This set of languages included BASIC, FORTRAN, Pascal, C and Ada. As previously stated, fourth generation and goal directed evaluation languages were eliminated from consideration (e.g., Prolog and Icon).

DL must both represent the complete source language and facilitate generation of documentation. Designing a language such as DL required analysis of both the programming models of the source languages and the syntax of the language needed to support those models.

Analysis of Languages

Common elements in the languages under consideration include data structures, routines (functions and procedures) and control of reference scope. Some of the source languages did not necessarily require supporting all of these concepts. For example, FORTRAN does not provide the minimal scoping rules (data access control) provided by C, which is also less than that provided by Ada. In this case, the most advanced form of data access control, that of Ada, must be used in DL in order to have a complete representation. Investigation into representations showed that it was possible to represent BASIC, FORTRAN and C data structures using the more complex representations implemented in Ada.

Another issue is the semantics of blocks. For languages such as BASIC there are, typically, no blocks. The bounds of a subroutine in BASIC are from the entry point referenced in the GOSUB until the RETURN(s). In many interpreted BASIC implementations, there is no requirement that the RETURN appear lexically following the entry point. Essentially, a BASIC program is one large, possibly unstructured block. All data is accessible to all parts of the program logic.

FORTRAN, on the other hand, has a limited repertoire of block capabilities. SUBROUTINE and FUNCTION blocks are distinct "capsules" that have a visible entry point(s). Data may be put in capsules using COMMON blocks, but access to data within the COMMON is uncontrolled, allowing unrestricted data conversions.

Blocks are "named" in that the routine or COMMON has a name. FORTRAN does not provide nesting of blocks, so the requirement for FORTRAN is flat, one level blocks. Languages such as C and Pascal allow nesting of blocks. Although different in detail, the semantics for C and Pascal are similar. Both allow for definition of procedures and functions, as with FORTRAN. Pascal allows specification of procedures and functions within (and local to) procedures and functions. In each of these local procedures and functions, declaration of local variables may occur. These variables are private to the declaring routine, and those routines within it (identifier scoping rules).

Similarly, C allows declarations of variables at the beginning of any block, including function-internal blocks. These private variables are accessible only to statements in that block, and generally are created and destroyed on entry to and exit from the block. To handle these block requirements (both C and Pascal), it was determined that each block contain all local declarations in that block. The only difference between a procedure/function and an unnamed block in C is the absence of a name, since parameters are optional.

Ada allows for a merging of the C and Pascal concepts in that blocks like those in C may be named, and have local declarations. Procedures and functions may be nested and have local declarations. These named blocks are executed just like regular blocks (that is control "falls" into the named block), but an extra measure of name scope control is allowed.

Data Structures

Adequacy of any documentation provided by an automatic documentation generator is dependent on the level of detail retained from processing the source program. If the internal intermediate representation of the original source is incomplete, then the documentation process will create inferior documentation.

To protect against the problem of insufficient detail, hand drawn charts were created for significant sample code blocks. From these charts, prototype data structures were created for representing the sample code.

Incorporated in the initial design was the notion that any structure built by the DL compiler should be independently verifiable as to the correctness of the internal linkages. A "tagged memory" scheme was developed such that each element of the compiled structure described itself (in form) so that it is possible to verify that all pointers point to objects of the type they are intended to reference. The general exception to this tagged memory model is that of character strings, in which are stored identifier names, descriptions and in-line comments.

A limited set of common structures were developed to minimize the number of storage managment routines needed. For example, a generic table structure was implemented so that a table can have a dynamic number of elements, all elements being of a specific type. In this way a single routine can add an element to any table, and the table manager is able to verify that the element is of the appropriate type for that table.

Operand Structure

Each structure is called an *operand*, and each operand is assigned a unique *operand type*. All operands have a common prefix tag field containing the type of the operand, the block to which the operand is assigned, the table to which the operand is assigned, and the table entry number at which this operand is assigned. A detailed definition of all of these stuctures appear in Appendix C. DL currently has operand types *block, code, list, mem, init, quad, ref, symbol, table, type, val,* and *expr,* used as follows:

OT_BLOCK

> All code and local definitions to a *block* are stored in an OT_BLOCK operand. A *base block* anchoring the parse tree makes the entire parse tree available to a documentation program.

OT_CODE

Executable statement operations (quadruples) are kept together by code statement, in *code* tables. Sequential processing of the quadruples, in the order of the *code* results in the replication of the execution order of the input source.

OT_LIST

List operands maintain appropriate association (grouping or sequence) of operands as they appeared in the input source.

OT_MEM

Members of data aggregates and unions (both of which are *type* operands) describe the data to be stored in each field of the aggregate or union.

OT_INIT

Initializers for data are stored in *init* operands. The values associated with the *init* operands are *val* operands.

OT_VAL

Values for *init* operands appear in *val* operands. These are used for static and dynamic initialization of variables.

OT_QUAD

Execution operators are stored in quadruples, which have an operator, up to three operands that the operator uses, and a resulting *type* operand.

OT_REF

References are kept in *ref* operands. References include such things as source file, line number, and reference class.

OT_SYMBOL

A *symbol* operand is used to describe variables, function and procedure descriptors, aggregates, unions and enumerations. In the cases of aggregates, unions and enumerations, there may be a name associated with the entry to allow later references. Note that the seperation of *symbol* operands into variable, aggregate, union and enumerator is done by noting which *table* the *symbol* appears in.

OT_TABLE

A *table* of operands is a variable array of operands, all of the same *operand type*.

OT_TYPE

Data type information is captured in the *type* operand, which has several variants, depending on the basic type declaration.

OT_EXPR

This entry is used to keep track of the current type within expression evaluation.

Block Structure

The primary structure that describes the source is the *block*. Each block is given a unique *block identifier number* so that the linkage to all elements belonging to that block can be verified. If the block has a name (function, procedure, or Ada named block) the name is also recorded. A *prototype function block* represents the interface information of a function whose complete formal definition is supplied elsewhere in the input source.

When a formal definition of a block has been completed, then the function block is no longer a prototype. The function block is then LOCKED from further modification. This allows the DL compiler to diagnose attempts to multiply define routines.

Formal parameters, if they exist, are recorded in the *block* so that checks can be made regarding the appropriateness of a *call* to the routine. If the block is a function, the return type of the function is also indicated allowing recognition of inappropriate assignments or uses of the return result.

The currently defined block types in DL are:

BT_BASE

> An initial block is allocated prior to any source being processed. All external definitions and outermost level routines appear in the *base* block. For BASIC, FORTRAN and C, all procedures and functions are found in this level. For Pascal and Ada, the main procedure and all external definitions appear in this *block*.

BT_NONE

> The block type has not been assigned. During the compile, the initial type of a *block* is *none*. Prior to entering a *block* into a table, a block type other than *none* is assigned to the block, else an error is diagnosed.

BT_DATA

Initialized data specifications are stored in *data* blocks. This is where FORTRAN "BLOCK DATA" is stored. FORTRAN "COMMON" blocks are *data* blocks that have the name of the COMMON they represent.

BT_SUBR

Nameless procedures are stored in *subr* blocks. A nameless procedure contains declarations and code that are stored local to the *subr* block, and may contain other blocks. Nameless procedures are activated by fall-through, and do not return a value.

BT_FUNC

Completely defined subprograms are stored in *func* blocks. A subprogram is given a name, optional return value type, and optional formal parameters. A subprogram contains declarations and code that are stored local to the *func* block, and may contain other blocks. Subprograms are activated by invokation. A function is a subprogram that returns a value, a procedure is a program that does not.

BT_PROTO

The *proto* block type is used for the definition of a prototype. A prototype describes an incompletely defined function or procedure. A prototype does not yet contain any code but does describe the name of the block, and optionally, the number and types of parameters to the function or procedure.

Block Table Structures

Each *block* has a group of associated *tables*. Each *table* has the current number of allocated and activated entries in the table. All entries in the table must be of the prescribed type that was specified when the *table* was created. The varieties of tables include:

TBL_AGG

> This table type holds aggregates of data, such as a C *structure* or Pascal /Ada *record*. An aggregate has at least one member element. Local struct definitions are stored in the currently active *block*, when the definition is encountered. Each member of an *aggregate* table is assigned storage following the previous member in the aggregate.

TBL_UNION

> *Unions* differ from aggregates in that each member in the *union table* overlays (is given the same storage offset) the previous union member.

TBL_MEM

> *Mem* tables define member fields within the structure of aggregates or unions.

TBL_BLOCK

> Blocks that are local to the currently active *block* are stored in the *block table*. These *block table* entries may be any of the allowed *block* types, with the convention that a *base block* only occurs at the base of the internal representation. Each entry in the *block table* is a recursive data structure of *blocks*.

TBL_CODE

All executable statements are represented by *code table* entries. *Code* is itself a linked list of quadruples (*quad*).

TBL_QUAD

Executable instructions are stored in quadruples, which consist of an operator, up to three operands (2 sources/1 destination, or 3 sources), and a result *type* operand.

TBL_ENUM

Enumeration definitions local to the current block are stored in the *enum table*. Each *enum* has at least one *enumerator* following in the enumeration definition from the input source.

TBL_ENUMERATOR

A single *enumerator* is assigned an entry in the *enumerator table*. A set of *enumerators* define the range of valid values to which elements of the *enum* type can be assigned.

TBL_TYPE

Local type definitions are stored in the *type table*. Several pre-defined types are installed in the *base block* that allow the user to reference all of the standard supplied types of BASIC, C, FORTRAN, Pascal and Ada.

TBL_VAR

Local variable definitions are stored in the *var* table, with optionally specified associated initialization values. All variables are associated with a *type* entry, that may be in the current *block* or any of the active *blocks* (determined by lexical scope rules).

TBL_REF

Lexical occurance and symbolic references to all elements are stored in *ref table* entries. The *ref table* in a *block* indicates the locations that referenced this currently active *block*.

TBL_LABEL

A *label table* indicates the labels (symbolic addresses) that occur in the block. These labels are referenced by the unconditional *goto* statements in BASIC, C, FORTRAN, Pascal and Ada.

Symbol Table Connectivity

Each *block*, except for the *base block* must be owned by another *block*. In this way, internal compiler failures such as attempting to assign ownership of a *block* to more than one *block* are detected. This eliminates what would otherwise appear as data dependent errors.

In the current implementation a set of simple types is known by the compiler. This set of types includes those necessary for FORTRAN and C. These two languages were chosen because initial development of the compiler was in C, and since we have a large body of software written in FORTRAN. C has a relatively esoteric and sometimes obscure syntax for declaring complex types. The solution of the problems presented by the C syntax was vital to understanding how all data types could be described.

Representation of the scalar types *boolean, char*acter, *short, int, long, float, double, complex, double complex* and *void* are simple types in that there is a fixed allocation for elements of these types. These types are in the set of types known to the compiler. As in C, DL allows a default typing of functions and procedures, called the *default* type. This soft type is aliased to the *int* type and requires the same storage as an *int.*

Types that involve addresses include near pointers, far pointers, arrays, functions and aliases. Both near and far pointers are treated as dimensionless arrays. An alias provides a linkage between two items of the same type, but separated by declarations. Aliases also provide a mechanism for coersion of types and type equivalences.

Structured types *aggregate* and *union* are represented by a table structure that orders their members. Each member of a *aggregate* or *union* is represented by a *mem* which also has a type associated with it. The same table structure also allows for storing enumerators belonging to an enumeration.

Labels into *code* are represented with a *label* type entry. Upon encountering the foward reference to a label, an arbitrary label is created local to the enclosing function block, with the foward reference stored as a pointer in the arbitrary label. When the label is actually encountered, this arbitrary label is moved to the correct block. DL then "backpatches" each foward reference through the appropriate pointer in the corrected label.

During the compilation of the input source, all diagnostic messages are stored in the parse tree at the point where the diagnostic is generated. This feature allows for the compiler to mark all declarations with notification messages as it runs, and then passes this information to the post-processors for further evaluation.

DL Representation Example

The following example is presented to clarify the relationships between data structures in the output of the DL compiler. The example program chosen is factorial, allowing the reader to focus on the data structures, without being concerned with the details of the program. This simple example does not illustrate all features of DL, but does demonstrate the most common data structures.

The reader should be aware that the picture and listing of tables are greatly abbreviated. The intent was to show connectivity rather than showing all of the details. Pruned from the example were all empty tables, and some of the less interesting pointers, such as parent pointers from the various blocks. For a complete representation, see Appendix E.

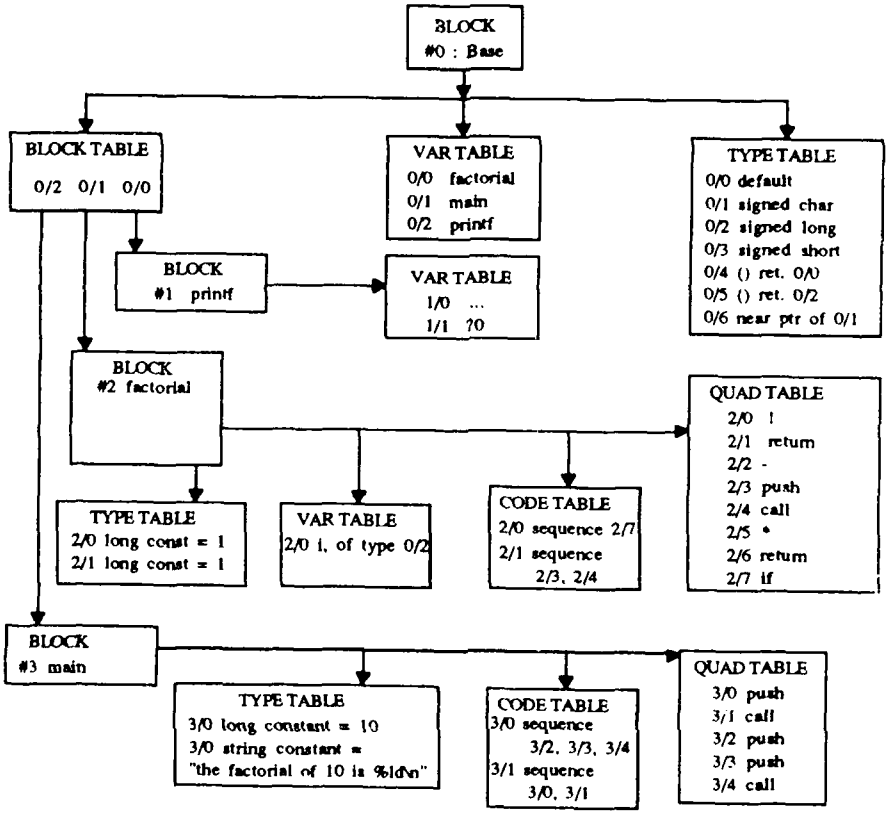
```
printf (char*, ...);

function long factorial (i)
 long i;
{
 i f (!i)
   return 1;
 else
   return (i * factorial (i - 1));
}

function main ()
{
   printf ("the factorial of 10 is%ld\n",
             factorial (10));
}
```

The block index represents the order in which the major programming units, known as blocks, are represented in the program. The block level denotes the degree of nesting a block has. The value 0 represents the outermost level, higher levels indicate deeper nesting.

Factorial Example Data Structures



Block Index

Id#	Level	Name
0	0	external
1	1	printf
2	1	factorial
3	1	main

Block # 0 external (Level:0)
bl_type: BT_BASE, bl_return: <no return type>,
bl_parent: <no parent>

TBL_TYPE (used 8/9 entries) of OT_TYPE
Note: 'I' after Size - full access requires far pointer.

Id#	Name	Class	Size	Reference
0/ 0	default	TY_ALIAS	2	of TBL_TYPE 0/ 3
0/ 1	signed char	TY_CHAR	1	
0/ 2	signed long	TY_LONG	4	
0/ 3	signed short	TY_SHORT	2	
0/ 4	() returning	TY_FUNC	2	TBL_TYPE 0/ 0 ()
0/ 5	() returning	TY_FUNC	4	TBL_TYPE 0/ 2 ()
0/ 6	int	TY_ALIAS	2	of TBL_TYPE 0/ 0
0/ 7	near pointer	TY_NEAR	4	to TBL_TYPE 0/ 1 near/volatile

TBL_VAR (used 3/9 entries) of OT_SYMBOL

Id#	Name	Usage	Type	Attributes/References
0/ 0	factorial	US_DECL	0/ 5	
0/ 1	main	US_DECL	0/ 4	
0/ 2	printf	US_DECL	0/ 4	

DL Representation Example

TBL_BLOCK (used 3/9 entries) of OT_BLOCK

Block # 1 printf (Level:1)
bl_type: BT_PROTO, bl_return: default, bl_parent: external

TBL_VAR (used 2/9 entries) of OT_SYMBOL

Id#	Name	Usage	Type	Attributes/References
1/ 0	...	US_FPARM	0/ 0	
1/ 1	?0	US_FPARM	0/ 7	

Block # 2 factorial (Level: 1)
bl_type: BT_FUNC, bl_return: signed long,
bl_parent: external

TBL_TYPE (used 2/9 entries) of OT_TYPE
Note: 'l' after Size - full access requires far pointer.

Id#	Name	Class	Size	References
2/ 0	long constant	TY_CONST	4	Value: 1
2/ 1	long constant	TY_CONST	4	Value: 1

TBL_VAR (used 1/1 entries) of OT_SYMBOL

Id#	Name	Usage	Type	Attributes/References
2/ 0	i	US_FPARM	0/ 2	

TBL_CODE (used 2/9 entries) of OT_CODE

Id#	Quadruple Sequence
2/ 0	2/ 7
2/ 1	2/ 3 2/ 4

TBL_QUAD (used 8/9 entries) of OT_QUAD

Id#	Operation	Left		Right		Third	Result	
2/ 0	!	TBL_VAR	2/ 0	< none >		< none >	TBL_TYPE	0/ 0
2/ 1	return	TBL_TYPE	0/ 2	TBL_TYPE	2/ 1	< none >	TBL_TYPE	0/ 2
2/ 2	-	TBL_VAR	2/ 0	TBL_TYPE	2/ 0	< none >	TBL_TYPE	0/ 2
2/ 3	push	TBL_TYPE	0/ 0	TBL_QUAD	2/ 2	< none >	TBL_TYPE	0/ 3
warning: automatic cast to result type, left side resolved via TBL_TYPE alias(es): 0/ 0								
2/ 4	call	TBL_VAR	0/ 0	< none >		< none >	TBL_TYPE	0/ 2
2/ 5	*	TBL_VAR	2/ 0	TBL_CODE	2/ 1	< none >	TBL_TYPE	0/ 2
2/ 6	return	TBL_TYPE	0/ 2	TBL_QUAD	2/ 5	< none >	TBL_TYPE	0/ 2
2/ 7	if	TBL_QUAD	2/ 0	TBL_QUAD	2/ 1	TBL_QUAD 2/ 6	< none >	

Block # 3 main (Level:1)
bl_type: BT_FUNC, bl_return: default,
bl_parent: external

TBL_TYPE (used 2/9 entries) of OT_TYPE
Note: 'l' after Size - full access requires far pointer.

Id#	Name	Class	Size	References
3/ 0	long constant	TY_CONST	4	Value: 10
3/ 1	string constant	TY_CONST	29	Value: "the factorial of 10 is: %ld\n"

TBL_CODE (used 3/9 entries) of OT_CODE

Id#	Quadruple Sequence
3/ 0	3/ 2 3/ 3 3/ 4
3/ 1	3/ 0 3/ 1

TBL_QUAD (used 5/9 entries) of OT_QUAD

Id#	Operation	Left		Right		Third	Result	
3/ 0	push	TBL_TYPE	0/ 0	TBL_TYPE	3/ 0	< none >	TBL_TYPE	0/ 3
warning, automatic cast to result type, left side resolved via TBL_TYPE alias(es): 0/ 0								
3/ 1	call	TBL_VAR	0/ 0	< none >		< none >	TBL_TYPE	0/ 2
3/ 2	push	TBL_TYPE	0/ 7	TBL_TYPE	3/ 1	< none >	TBL_TYPE	0/ 8
3/ 3	push	TBL_CODE	3/ 1	TBL_CODE	3/ 1	< none >	TBL_TYPE	0/ 2
3/ 4	call	TBL_VAR	0/ 2	< none >		< none >	TBL_TYPE	0/ 0
warning, left side resolved via TBL_TYPE alias(es): 0/ 0								

Blocks of the same lexical level are considered children of the block with the next lower block level. In this example, the *external* block is the parent block, and contains child function blocks *printf*, *factorial*, and *main*.

All blocks contain tables to represent information pertaining to internal components. For instance, the *external* block has a type table to represent global types, a variable table to represent global definitions, and a block table to represent subsidiary blocks.

In the example, the function block *factorial* has a type table to represent local types, a variable table to represent local definitions, a code table to represent the order in which quadruples are to be executed, and a quadruple table representing the executable elements of the function.

Type table elements can be described as belonging to any of these sets: simple types, derived types, and constants. Each type has a unique numerical identifier referring to block number and element position, a name, a type classification, size required for each data element of that type, and related information.

A simple type describes a primitive class of data element. DL supplies every simple type. Types 0/1, 0/2,and 0/3 are simple types, as shown by type classifications TY_CHAR (primitive for byte length integral scalar data), TY_LONG (primitive for double word length integral scalar data), and TY_SHORT (primitive for word length integral scalar data) respectively.

A derived type describes a programmer defined class of data element that is either a composite or a direct mapping of other existing types accessible from the lexical level of the programmer's type definition. Type 0/0 is a direct mapping of type 0/3, hence type *default* type is an alias of type *signed short*; the type classification field is TY_ALIAS to denote this. Type 0/5 is a composite type, any function returning data of type 0/0 (*default*) is referenced to the composite type; the type classification field is TY_FUNC to denote this. Type 0/7 is another composite type, where any data element describing a double word address of a location, in which a data element of type 0/1 (*signed character*) is stored, is referenced to the composite type. The type classification field is TY_NEAR to denote this.

A constant type describes a fixed scalar value that is mentioned as a literal part of the program text, and therefore requires special attention for documentation purposes.

Variable table elements correspond to identifiers in the program that represent starting addresses of data. Each identifier, or variable, is associated to an element of a type table accessible from the lexical level of the programmer's variable definition. Each variable has a numerical identifier referring to block number and element position, a name, a variable usage field, the associated type, and related information.

The variable usage field is important, since it determines if the variable is declared within a block (US_DECL) or declared as a formal parameter associated with a function block (US_FPARM).

Variables 1/0, 1/1, and 2/0 are examples of formal parameters. There are three functions that are declared as external block variables, each associated with a function block: *printf* (variable 0/0), *factorial* (variable 0/1), and *main* (variable 0/2).

These functions are considered variables, since a function is represented by a location denoting the starting address of information relating to the executable code and data of the function. Notice that in the absence of a type mentioned with a variable, the variable takes on type *default*; this property is exhibited by variables 0/0 and 0/2.

Variable names can have special properties. Variable 1/1 is a case of an unnamed formal parameter which was mentioned in the prototypical declaration of function *printf*. Since DL was not supplied with the name of the first formal parameter, but only its type, DL creates an identifer for the variable indicating that the name is unknown. The question mark as the first character of the name indicates an invented identifier, followed by the number of the invented identifier thus created. Variable 1/1 is a case of an elliptical formal parameter. An ellipsis (...) tells DL that zero or more actual parameters in the function call may correspond to the same relative position in the function definition; a utility useful in describing a function with an indefinite list of arguments.

Block table elements describe procedural subunits in the program. The external block is an exception in that no executable code is ever described in it, the other blocks may have code and data. Shown are two kinds of function blocks: prototype functions, and complete functions.

A prototype function incompletely defines a function. Prototype definitions require a function name, return type, and formal parameters, so that it can be referenced by the rest of the program. Notice the return type is *default* (type 0/0) since the return type is not mentioned in the prototypical declaration.

Block #1 is an example of a prototype function that is not defined in its entirety by an applications programmer: *printf*. In this case the *printf* function is supplied by the operating system environment of the home compiler through a standard input/output description library at compile time. We represent it here to DL as a prototype so the program can be documented as using the *printf* function, and check argument usage.

Blocks #2 and #3 are examples of complete function definitions. A complete function definition may have a return type, formal parameters, local types, local variables, and executable code.

Code table elements describe the order in which groups of quadruples are executed. A group is a linked list of the bases or roots of expression trees. The first element of the code table, always #0 in the code table of any block, represents the outermost level of code body execution. Other elements represent subsidiary sequences. The quadruples of each sequence are executed left to right.

Code element 2/0 indicates that function block *factorial* is executed by evaluating the expression tree anchored by quadruple 2/7, the *if* statement. The other code elements indicate sequences involving function evaluation. Each of these sequences may involve zero or more *pushes* of actual parameters, followed by the call. The push/call sequence is a simulated entry to a routine's activation record, where placement of the actual parameters occurs in the left to right order indicated by the function evaluation.

Quadruple table elements describe executable code tuples, with an operator, up to three operands, and a result type. Operands can be any variable, constant type, code segment, or quadruple accessible from the lexical level of the programmer's usage of the operator. Expressions and other statements are built from a tree of quadruples .

Most software presently developed by programmers will compile and execute despite obvious misuses, abuses, and mismatches of resulting expression evaluation data types. Some programming languages have what is termed "strong type checking" to prevent most of these kinds of error from passing the compilation stage. Many programming languages have checks that are weaker or even non-existent. Large software systems where routines from different software houses are reused or combined present a potentially interminable problem of interface errors. A fair assumption on our part is that these invisible errors eventually lead to undesireable run-time output under circumstances which are usually unknown until after the fact.

To highlight potential interface problems, DL performs type checking that is much stronger than usually warranted for normal programming purposes, so strong that our term for it is "strict type checking". Part of strict checking involves noting in the symbol dump output referencing of all directly mapped derived types (alias resolution warning), and where warranted implied coercion of non-congruent expression types even if the types involved are conformal (automatic cast warning).

The result type of a quadruple is available to check the result of an expression, and to match the types of subexpressions. Quadruples 2/3, 3/0, and 3/4 illustrate expressions that would have passed not only compilers but also *lint* class code validator utilities, that involve aliases and coercions.

Results

As the example illustrates, retaining all of the details that represent the logic of the program results in a significant connectivity problem. This problem cannot be reduced in size, however, without loss of the original intent of the program. The output generators also require different portions of the DL compiler output. Different degrees of abstraction are possible using the collected information. All details produced by the DL compiler must be retained for the final documentaion application. In other words, abstraction is deferred until the output of the documentation application, and this is a user directed process.

We have identified and resolved the internal representation issues, and are now investigating appropriate maintenance programmer documentation, generated from these internal representations. The next steps in these investigations will determine the applicability and usefulness of the documentation using real world code.

Four applications currently are under development. The first is a highly sophisticated and flexible cross referencing system. The second is a Nassi-Shneiderman generator, primarily for lower level documentation and limited abstraction. The third application will use action diagrams to provide higher levels of abstraction, up to the program level. Fourth, pictorial representation of data structures as an adjunct to the first three documentation generators may prove, beyond acting as a programmer aid, to provide a useful level of documentation to non-programmer users of the systems being documented. The utility and feasibility of interactive systems will also be investigated.

Appendix A

Review of Documentation Techniques

The following documentation methods were evaluated with a view to their applicability in the maintenance environment. Each is discussed with its relative strengths (advantages) and weaknesses (disadvantages).

Pretty Printing

A pretty printer is a stylizer. By reformatting the original program into a "standard" format, it is possible that misleading indentation (in the original) can highlight errors. This technique may be quite useful with old, unsightly source code.

Advantages:

Pretty printing can be implemented for any language. Its output is in a consistent, indented style for a specific language. The style reveals the essential elements and structure of code. The term for this organizational viewpoint is *lexical scope highlighting*.

Disadvantages:

Every language has different dialects known as standards. A standard is some set of deviations from the original definition of the language which serves the convenience of a particular group of programmers. Of course, the original definition is also a standard that deviates from all the dialects.

An extreme example of a language with a very large set of dialects is BASIC. Differing dialects of BASIC exist to control robots, search and maintain database files, support business functions and control graphics displays. Worse, several dialects of BASIC exist for some computers. BASIC is not the only language that shows these symptoms. Different dialects of a programming language have different coding styles. A pretty printer must be re-tailored to every dialect of a language.

Warnier-Orr Diagrams

Warnier-Orr diagramming offers a single technique to show functional decomposition and hierarchical data structures. The techniques's primary strength is in the design phase.

Advantages:

Warnier-Orr diagramming is a system that offers a single technique to show functional decompositions and hierarchical data structures. The system is modular, allowing the user to specify a design over multiple levels of detail. This modularity also makes such a design easier to read, draw, and change.

Disadvantages:

Warnier-Orr diagrams are a human-oriented communications system rather than a computer aided program design tool. Specific fonts, printing conventions, and even special forms may also be needed. Bottom levels of a large diagram often degenerate into a form of pseudocode, making the intent of the diagram difficult to understand. Warnier-Orr diagrams are therefore more suitable for small programs, or high levels of large programs.

Warnier-Orr diagrams do not help reveal the extent of coupling and cohesion between modules. Input-output paths for procedural components are not shown. Conditions and variables that control procedural flow are not shown. Relationships between procedure and data in a program are also not shown. Analysts may have difficulties using a Warnier-Orr diagram to diagnose program flaws during either the design or the maintenance phase.

Michael Jackson Diagrams

Michael Jackson diagrams show data coupling and cohesion. Logic is not represented in this technique, which thus is of limited value.

Advantages:

Michael Jackson diagrams are similar to Warnier-Orr diagrams, since the diagramming technique provides constructs appropriate to structured programming languages. However, the constructs are designed to also show data related coupling and cohesion aspects of a program, something that Warnier-Orr have no provision for. The pictorial relationships and structured text within the icons can be automated, which enable this method to be used for computer aided program design and maintenance.

Disadvantages:

This diagramming technique shows some of the same weaknesses as Warnier-Orr diagrams. The lowest levels of detail in a Michael Jackson diagram can degenerate into a form of pseudocode, thus degrading comprehension. Also, there is no provision for showing conditions and variables that control procedural flow.

Michael Jackson diagrams can become overloaded with detail as the programs they describe become more complex. Regardless of intended program size, the descriptive structures are more wordy than the program it represents. The average programmer can become frustrated with representations of even a small program using this method.

Flowcharts

Flowcharts are an elementary technique easily understood by programmers and non-programmers. The method is fairly reasonable for maintenance, provided the flowchart is correct. Unless automated, it is very difficult to maintain accuracy.

Advantages:

Flowcharts enjoy a rich and ancient history. At the dawn of modern computer science, flowcharts were the only known method of diagramming program structure. Flowcharting is an elementary technique of organization which is known to the general population. It also is the first technique of organization taught to students in computer literacy or some computer science courses. They are widely used, and recognized almost anywhere. At their best, flowcharts are simple, elegant, and flexible.

Disadvantages:

There is a dark side that has practically condemned flowcharting as an obsolete technique: flowcharts have an infinite potential for abuse! Some of the more serious problems are:

Flowcharting perpetuates the "spaghetti code" approach to programming, the flow of control is unrestricted and unstructured. As such, flowcharting is not easily adapted to structured programming concepts or techniques. Worse, there are no agreed upon extensions for structured programming languages.

Flowcharts make it far too easy to confuse high level and low level operations. A group of interconnected, internally detailed program modules can become too convoluted to read. Program logic and program modularity could then be hidden in a maze of detail. General system overviews are likewise difficult. Flowcharts encourage the programmers who use

them to think in terms of older programming styles, exemplified in traditional FORTRAN and assembly language programming. This slows down the desired trend of more modern languages becoming acceptable to the community of programmers, systems analysts, and managers.

Flowcharts are not easy to draw. Enough loops or subroutines can send the drawing off the paper, to hop between sheets drawing either jump dots or lines leading off the paper. Once an agreeable flowchart has been drawn, it is very hard to modify and still keep the same integrity as the original flowchart. Making room on a page for more constructs becomes a major chore.

Pseudocode / Structured English

Pseudcode, or Structured English, is a narrative form of program logic which is difficult to produce automatically, and suffers from many of same problems of standard documentation, if manually generated.

Advantages:

Both these documentation methods behave as a narrator to actual code. Pseudocode refers to narratives in formal notation resembling the actual code, yet not as rigorously expressed. "Structured English" refers to narratives informally styled so nonprogrammers can immediately understand them. Both methods can be helpful to depict overall program structure and architecture.

As an example, a common technique of program design is to start with a structured English specification of the task to be performed. Through stepwise refinement, portions of the specification are given increasing levels of detail until the specification resembles pseudocode. The pseudocode may then be acceptable as input to a fourth-generation language compiler, or the pseudocode can be further refined by the programmer into code acceptable by any of the compilers for high level languages, even ultimately refined into code acceptable by an assembler.

Disadvantages:

Wrongly used, the advantages of pseudocode or structured English become disadvantages. Narratives can become lengthy, difficult to read, even use obscure language or terminology. Narratives can contain a structure or content that requires rote memorization to use effectively. They tend to not be updated, thereby suffering the same end as initial documentation.

Nassi-Shneiderman Charts

Nassi-Shneiderman charts are a relatively modern replacement for flowcharts that is easily automated, especially with *structured programming* techniques.

Advantages:

Nassi-Shneiderman (N-S) charts were invented to replace flowcharting and pseudocode with a method that offers a more organized view of programming, with constructs appropriate to structured programming languages. This technique is deliberately designed to be graphically appealing and also easy to read, learn, and teach.

Disadvantages:

Like flowcharts, N-S charts are time consuming to draw and change. This method is best for showing detailed logic only, it tends to have problems when trying to coordinate the execution of many programs. N-S charts are procedure oriented, not data oriented. This method has no provision for organizing data structures, and in general cannot be linked to data models.

It has a further problem with data similar to Warnier-Orr diagrams: extensions are needed to show input and output data paths between procedures. Cross-checking for some forms of coupling is less possible without the extensions, and non-standardized with them.

Action Diagrams

Action diagrams were designed specifically to overcome many of the disadvantages of older techniques. They are not dependent on specialized output devices, and can be superimposed on the languages under consideration (especially in connection with a *pretty printer*).

Advantages:

Action diagrams are easily hand drawn, as such they are easy to teach and learn.

They are easily computerized and as such do not require hardcopy output forms.

They are well adapted to modern programming techniques. Action diagrams extend across multiple levels of data structure and procedure design, provide constructs appropriate to structured languages, and allow cross-checking of input and output data paths.

Action diagrams are a good interface with actual programming languages. Action diagrams can be tailored to a specific fourth generation language. Graphical relationships in an action diagram are also decomposable into low level code.

Disadvantages:

Action diagramming is a relatively new and unknown process, and has not yet gained the degree of acceptance and number of adherents as older methods have.

HOS Charts

HOS (Higher Order Software) charts guarantee *provably correct* programs when no modules external to their immediate control are used. They have a highly mathematical rule-based orientation, and needs much patience to learn and use correctly. They are not considered an appropriate mechanism since most (or all) code was developed "outside" of the HOS environment.

Advantages:

HOS is one of the most mathematically rigorous development methods ever devised. HOS notation extends over multiple levels of program design, and binds data and procedures to each other intrinsically. Therefore HOS is valuable in creating highly complex specifications with little debugging, in a setting insensitive to a specific language.

Disadvantages:

HOS is very different from other, more widely known techniques. It uses a highly mathematical rule based orientation, requiring much patience to learn and use correctly. HOS is not easy to use, and can create specifications more complex than one person can properly manage. This is a team-oriented tool, useful primarily to any professional analyst who can understand the potential power of the method.

Cross Reference Listings

An adequate cross reference listing with derived information can provide the programmer with a quick method of determining where to look for changes to variables of interest.

Appendix B

Operating System Level Organization

The parser/documenter executes in two phases:

In the first phase (Figure B-1), the documentation pre-processor (dpp) converts the raw source code into a file containing two major classes of information:

> The first class of information is preprocessed source code, expressed in documentation language. The second class of information is an augmented overload matrix of operators. The standard overload matrix of operators for documentation language is combined with additional overloaded operators that are present in the source code and recognized by the preprocessor.

The parser/documentor converts the pre-processor output into an intermediate data file containing the entire meaning of the raw source code (complete semantics). The intermediate datafile is machine readable, and human checkable.

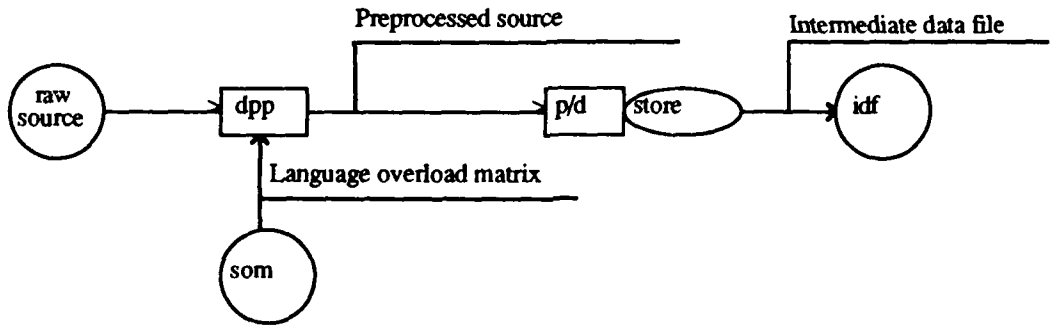


Figure B-1

The second phase (Figure B-2) uses the intermediate data file as input to an application, creating a user-readable output.

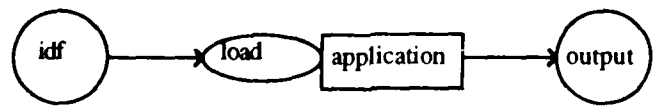


Figure B-2

Available applications include: Symbol Table Dump, Cross Reference Generator, Source Code Pretty Printer, and the Nassi-Shneidermann Diagrammer.

Appendix C

Symbol Table

Block Level Organization

The parser/documentor converts major program units into subsidary symbol tables, called blocks. All blocks have the same general structure, and have specific information related to the specialized purpose of each block. Figure C-1 shows the general organization of a block.

All symbol tables have an external block. The external block anchors the entire symbol table as a base pointer, and contains the primitive type definitions, such as integer, long, float, character, and so on. The external block contains the external definitions (type, variable, enumerator, aggregate) and anchors the first-level program units.

Program units are either compound statements (nameless function blocks), or functions (named function blocks). A function block is a compound statement with additional name, return type, and parameter information associated with the function declaration.

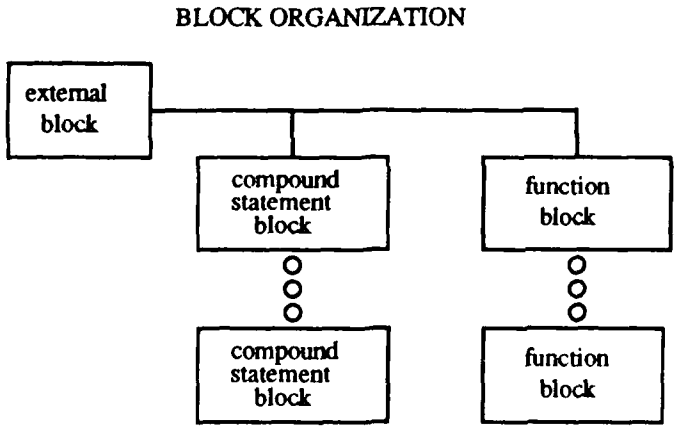


Figure C-1

Block Internal Detail -- Block Operand

Each block contains a set of data and pointer fields. The data fields specify the block's serial number, name, block type (e.g., compound statement, or function), and lexical level. Figure C-2 shows the relationships between data and pointer fields in blocks and between a block and other data structures.

The pointer fields link blocks to their parent blocks, return types, lists of formal parameters, and definitions tables. The definitions table is a table of pointers to tables of definitions. The blocks table links a block to the corresponding subsidiary blocks. The other tables link this block to each defined operand pertaining to this block.

Definitions include code segment operands, symbol operands (members, variables, enumerators, enumerations, structs, unions), type operands (type definitions), and reference operands (reference definitions), as well as block operands (block linkages).

The return type linkage is a link to a type operand, the formal parameters list linkage is a link to a list of symbol operands, and the parent block linkage is a link to a block operand.

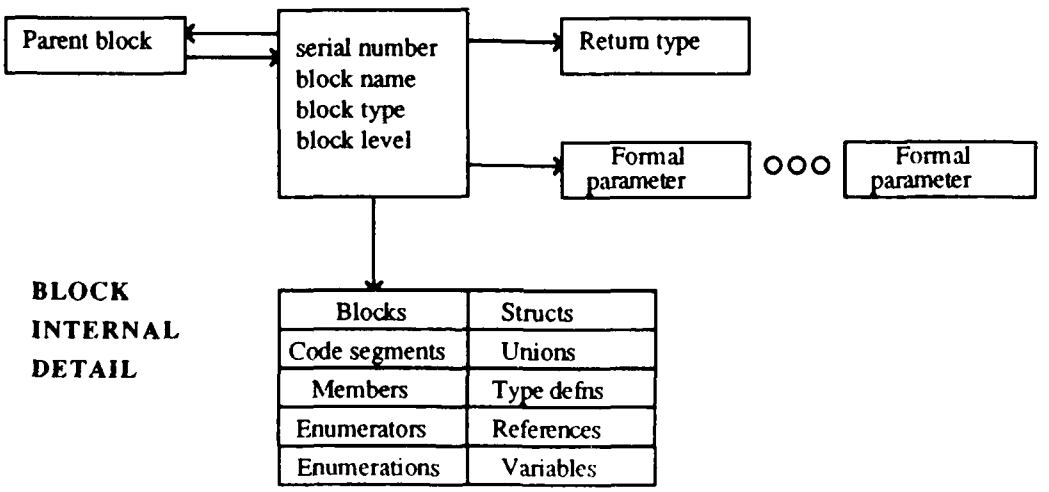


Figure C-2

Symbol Operand

The symbol operand, represented in Figure C-3, is used to represent variables, members, aggregate and enumerator tags, and enumerations. The operand class specifies either a member, or a symbol. The usage class specifies a formal parameter, or other declaration. The parameter number denotes the order of the parameter in a function block's declaration list. Other data fields indicate the symbol's name, size, and storage class attributes.

The symbol always has a link to a type operand of some kind, to denote the type declaration of the symbol. Variables and members may have an initial value as indicated by an optional initializer. Enumerations have an ordinal value. Either of these values may be specified via the symbol's value operand.

Aggegrate and enumerator tags have an owner link. The owner link is a type operand that connects the aggregate or enumerator tag to other symbols that may require the tag as a type definition.

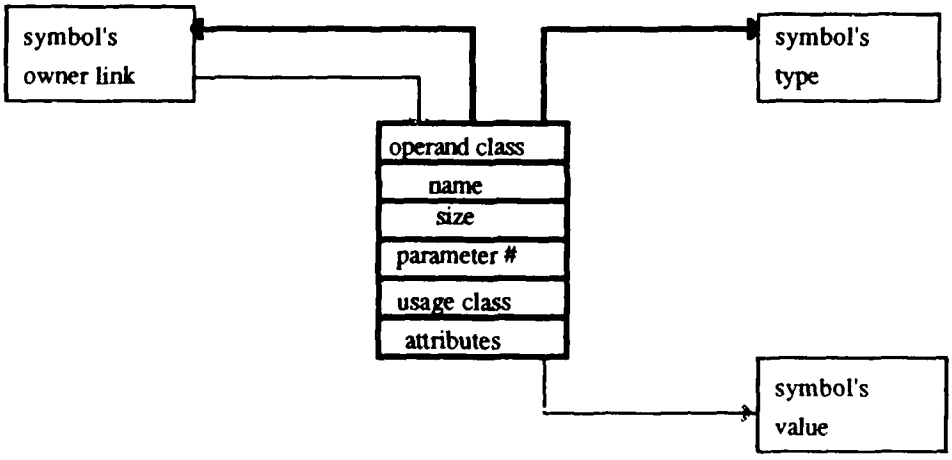


Figure C-3

Type Operand

The type operand, represented in Figure C-4, is the most general purpose operand in the entire symbol table. The type operand serves a number of purposes, depending on the type variant being used. Any primitive type has no variant. All types have a link to their parent blocks. A type may have a table of references. The type has data fields corresponding to the operand class (type operand), name, storage class attributes, and size.

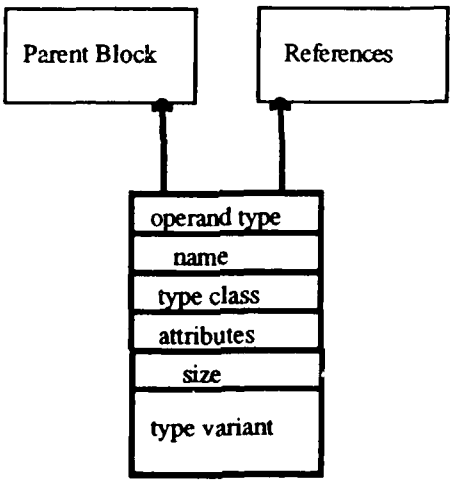


Figure C-4

The type variant in use depends on how the type operand is utilized. Near, far, function, array, and alias variants all have pointers to a subsidiary type, but the array variant also has high and low bound data fields. Aggregate and enumerator variants have an index to the member table in the same block, and the number of entries starting with that index.

The link variant serves as a connective element by linking the structure, union, and enumerator tag to any type using the link as a subsidiary type, and any symbol using the link as the symbol's type. The compiler variant serves as a temporary type, an intermediate result brought up from lower to higher production rules. The label variant serves as a statement label, and refers to a code operand.

Value Operand

The purpose of the value operand, represented in Figure C-5, is to hold initial values. The operand class is *value operand*. The value class indicates which value variant is in use. Value variants are structures that hold constants. Examples of simple constants are the integer, character, floating point number, or string.

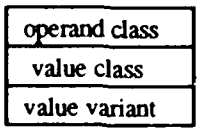


Figure C-5

Reference Operand

The function of the reference operand, represented in Figure C-6, is to hold reference information for any identifier. The operand class is *reference operand*. The file reference and line reference indicates the file and line in which the identifier was mentioned, the usage type indicates the context in which the identfier was mentioned for that file and line.

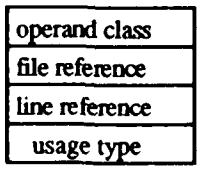


Figure C-6

Code Segment Operand

The purpose of the code segments operand, represented in Figure C-7, is to store quadruples of executable code in a linear array. The linear array represents the order in which the quadruples are to be executed. The operand class is *code operand*. There is a field for the number of quadruples in the array, and then the quadruple table, which is a pointer to the first quadruple in the array.

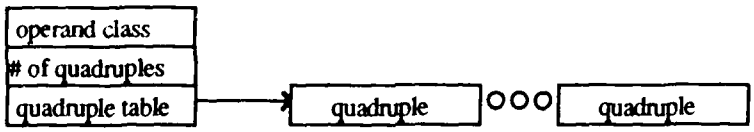


Figure C-7

Quadruple Operand

The function of the quadruple operand, represented in Figure C-8, is to store a unit of executable code. The operand class is *quadruple operand*. There is a field to indicate the operator, and pointer fields for up to three operands to be affected by the operator. These three operands can operands of any class. Hence value operands serve to hold constants, quadruples serve as intermediate expressions, code segments serve as statements or as actual parameter lists.

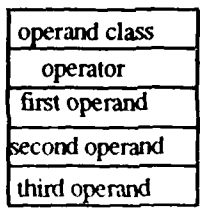


Figure C-8

Symbol Table Example

Consider the external declaration statement "register int i". It parses into the structure shown in Figure C-9. The external block has all the primitive types, including "int". The first level block is the external block, so "i" is a symbol operand in the variable table of the external block. "i" points back to the "int" primitive type.

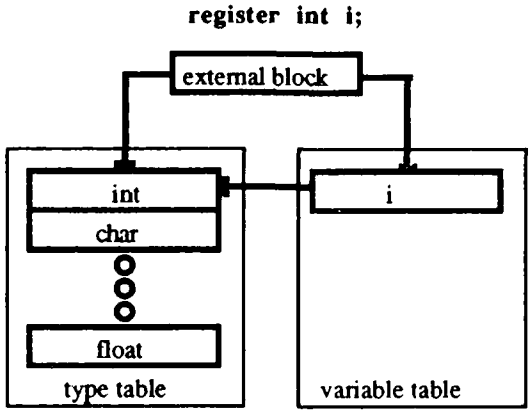


Figure C-9

The primitive type, "int" in the example shown in Figure C-10, is a type operand named "int" with: no type variant (because it is a primitive type), subsidary type of integer, a size of 4 bytes, and has signed, long, and integer attributes. The parent block of "int" is the external block. "int" has no references (being a program keyword) and the null pointer indicates that.

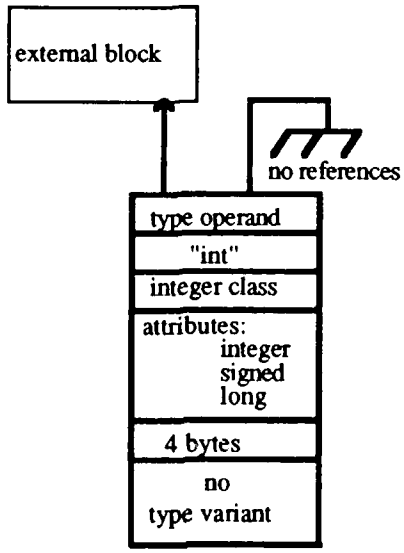


Figure C-10

The variable "i", in Figure C-11, is a symbol operand named "i" with: no owner link (because it is not a member of an aggregate or an enumeration of an enumerator), the symbol's usage is *declared*, no initial value, type "int" (pointer to that type is present), and the same attributes as type "int" but also with the register attribute. The parameter number was set to zero, but this fact is irrelevant since the symbol is not used as a parameter.

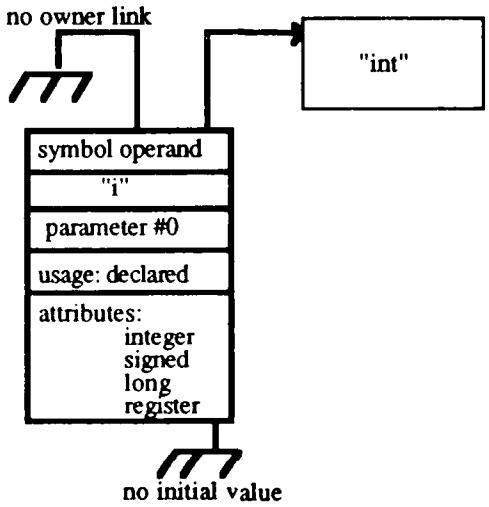
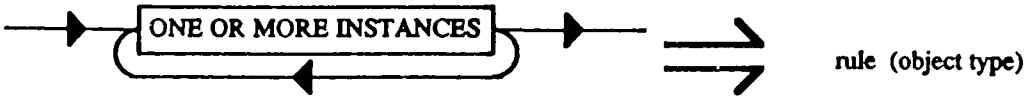


Figure C-11

Appendix D

Documentation Language Flowgraphs

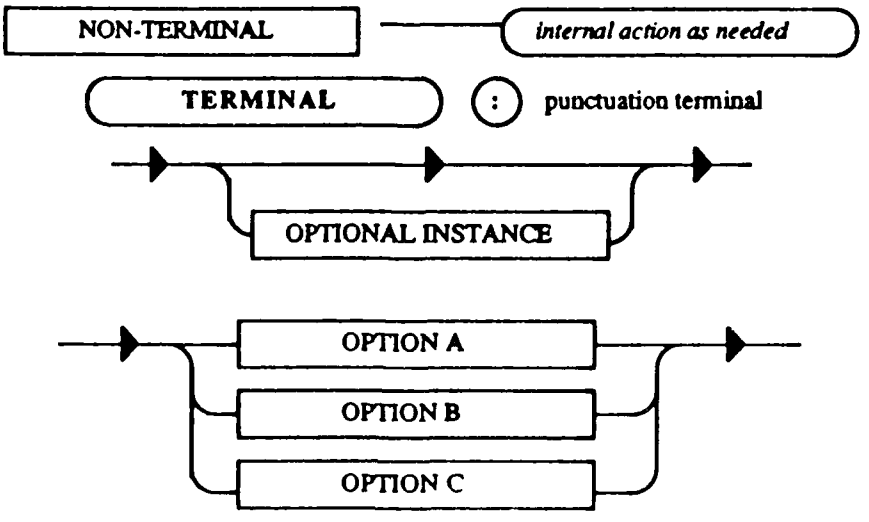
Appendix D shows graphical representations of the Documentation Language mentioned in the main body of this report. The style shown here, frequently refered to as a railroad diagram, indicates the order of acceptance of identifiers, punctuation, and keywords by the language via a top-down series of productions. A production indicates how a series of parsed symbols becomes another production. The series of parsed symbols is accepted in the order indicated by the production's flowgraph. The information is then transformed into a data structure, which is carried by the resultant production.



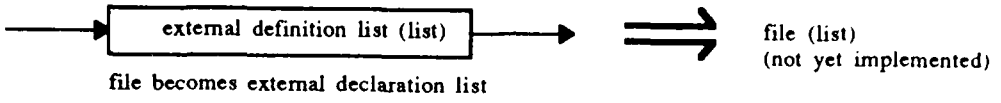
rule (object type) becomes ONE OR MORE INSTANCES

In the above example, the symbols comprising **one or more instances** of "something" is transformed into an **object type** carried by the production **rule.** Flow of control generally proceeds in the direction indicated by the arrows. The large arrow indicates that a transformation has taken place, with the abstraction on the left indicating what symbol or symbols were involved, and the abstraction on the right indicating the result of the tranformation. The phrase invoking the word "becomes" indicates the specific nature of the tranformation, which usually is by assignment but may be the result of a function call.

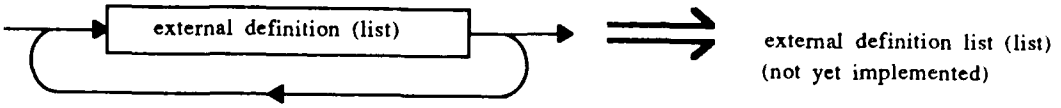
Supplemental Legend



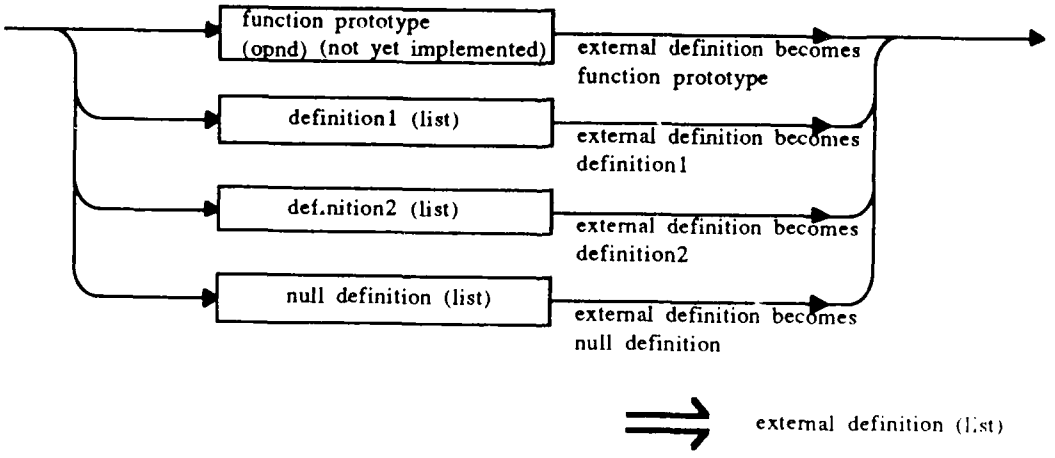
Appendix D

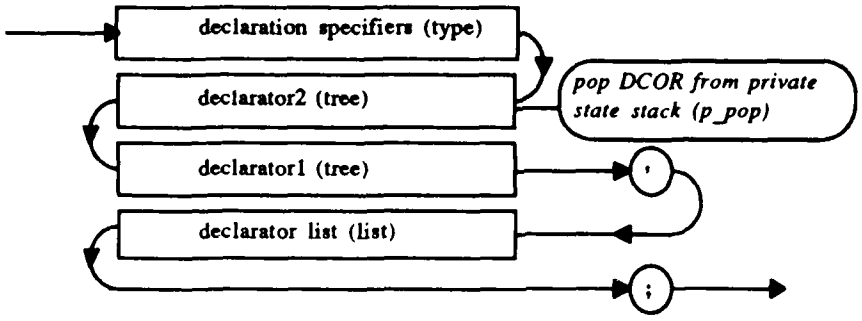


file becomes external declaration list



external definition list becomes:
one external definition : external definition
more than one : concatenation (catlst)
of external definition
to external definition list



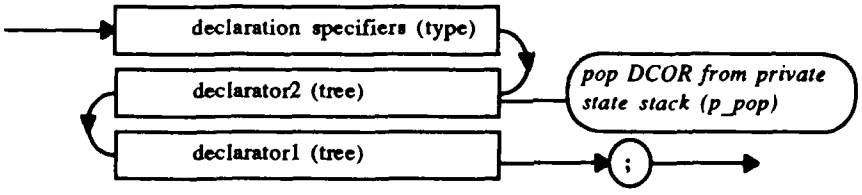


definition1 activates by side-effect

```
p_declare (
              declaration specifiers
            , p_define (
                        p_dchain (
                                    declarator2
                                  , declarator1
                                  )
                        )
            , declarator list
            )
```

⟹ definition1 (list)

(this conversion is used as a side-effect)

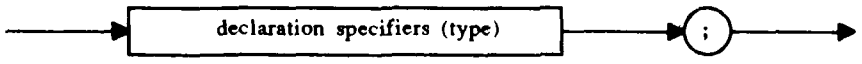


definition2 activates by side-effect

```
p_declare (
              declaration specifiers
            , p_define (
                        p_dchain (
                                    declarator2
                                  , declarator1
                                  )
                        )
            , NULL (no declarator list)
            )
```

⟹ definition2 (list)

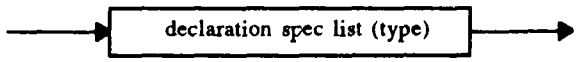
(this conversion is used as a side-effect)



null definition becomes
nulldcl (declaration specifiers)

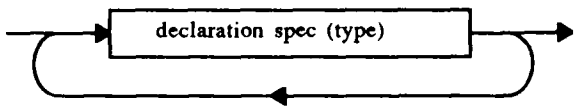
⟹ null definition (list)

Appendix D



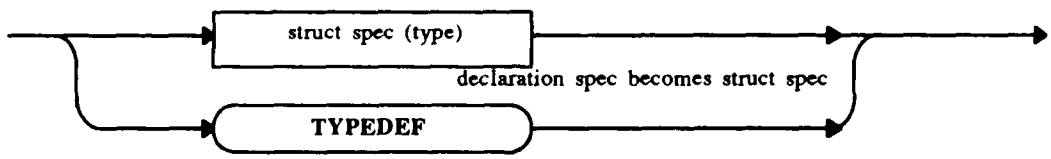
declaration specifiers becomes declaration spec list

⟹ declaration specifiers (type)



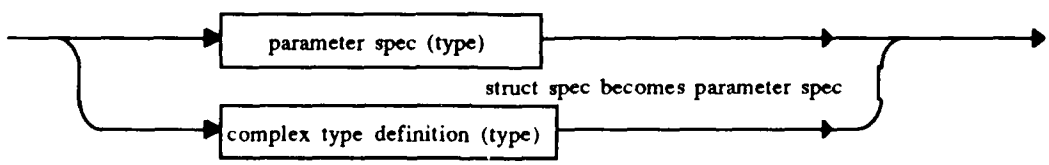
⟹ declaration spec list (type)

declaration spec list becomes:
one declaration spec : declaration spec
more than one : p_atrchk (
declaration spec
, declaration spec list
)
(merge declaration specs if possible)



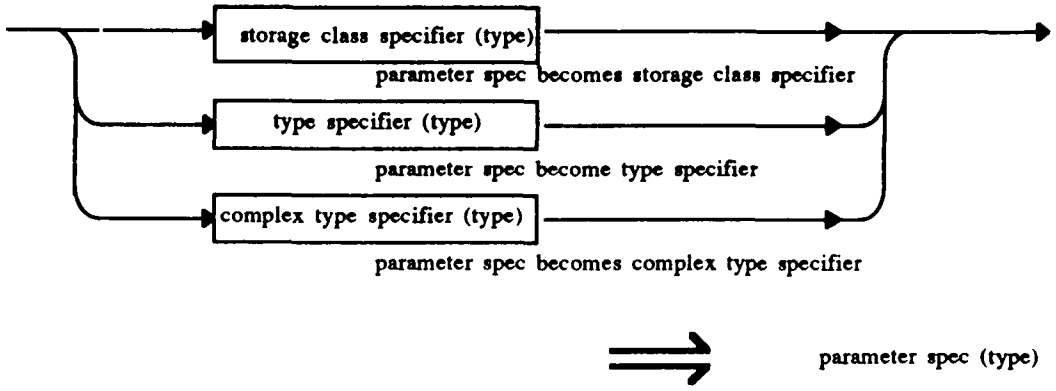
declaration spec becomes p_tamark (TA_TYPEDEF)
(set TYPEDEF attribute flag)

⟹ declaration spec (type)

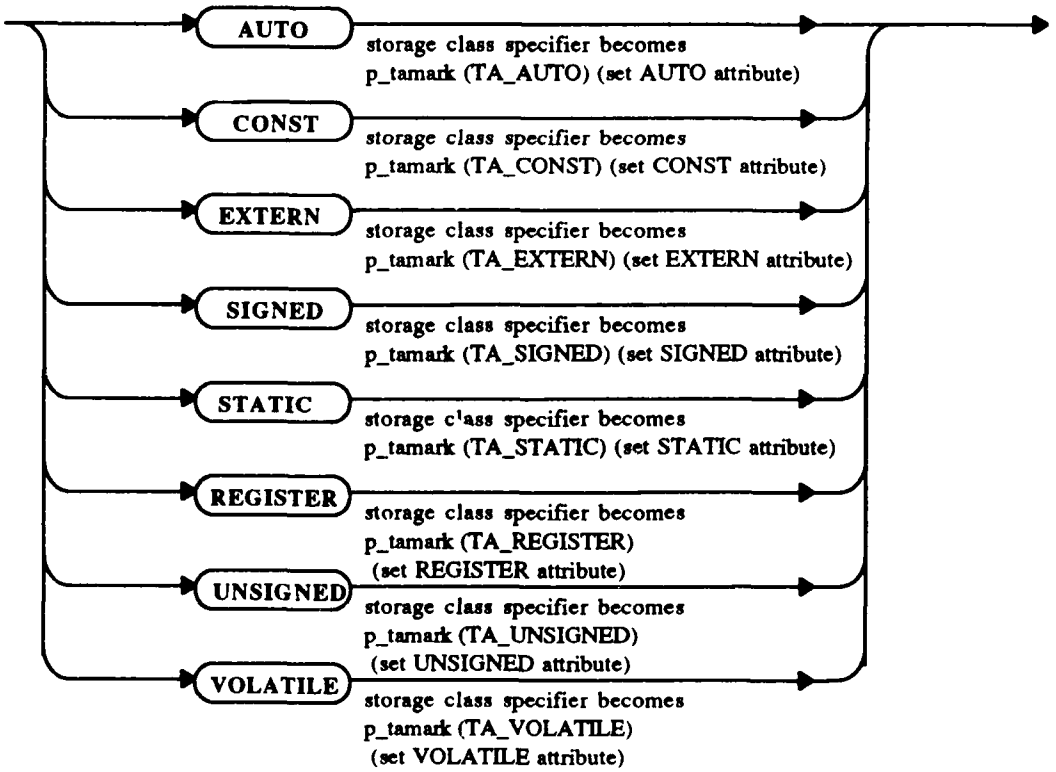


struct spec becomes complex type definition

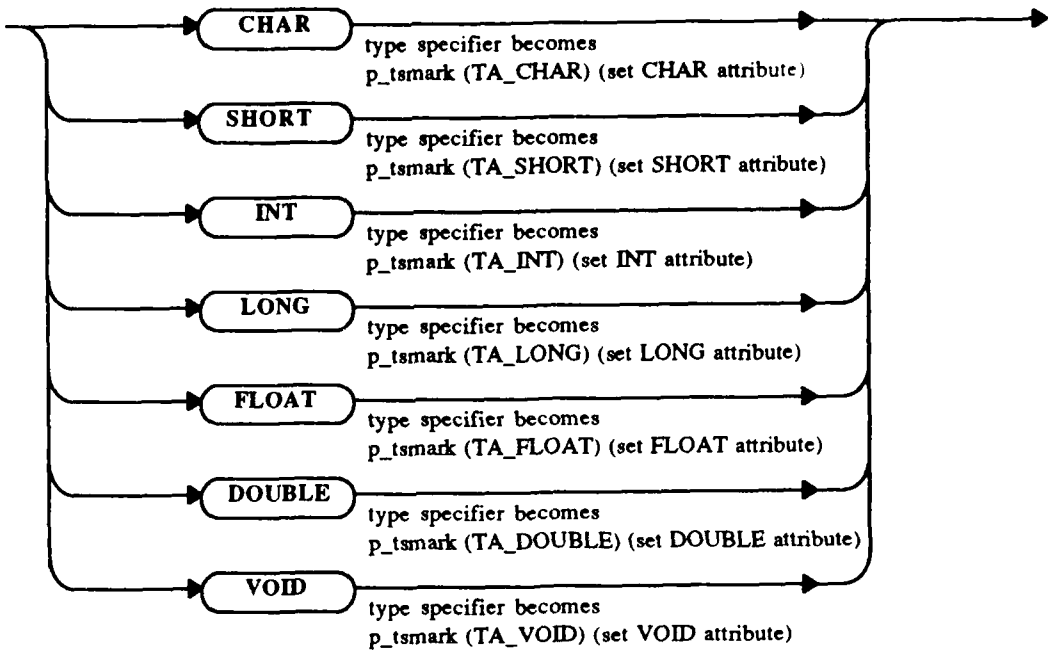
⟹ struct spec (type)



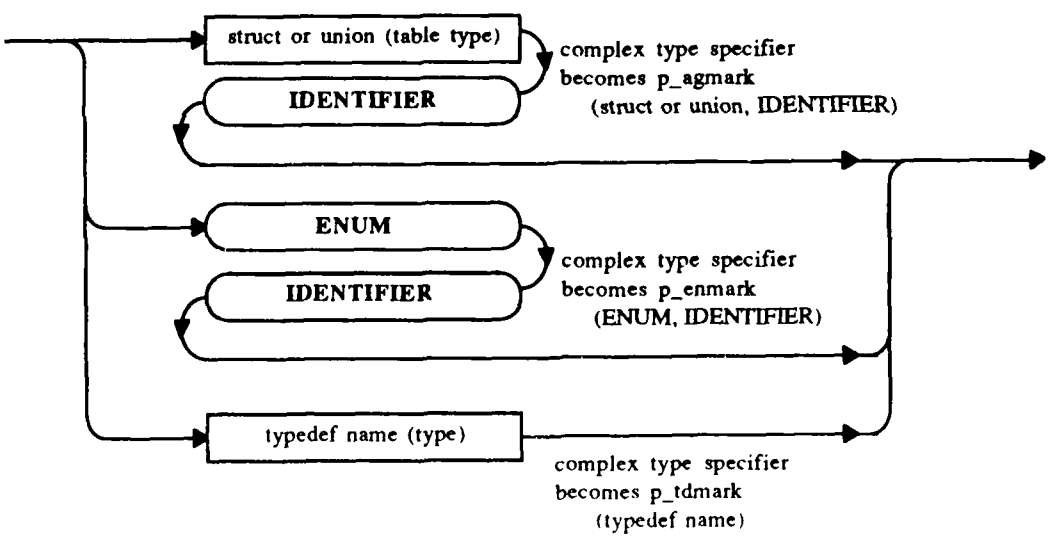
⇒ parameter spec (type)



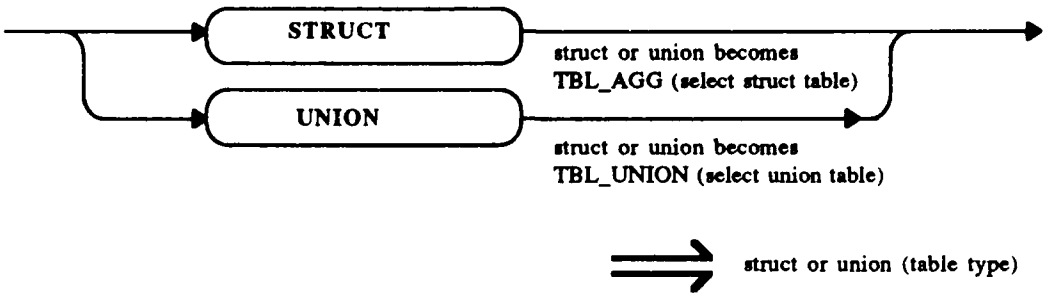
⇒ storage class specifier (type)

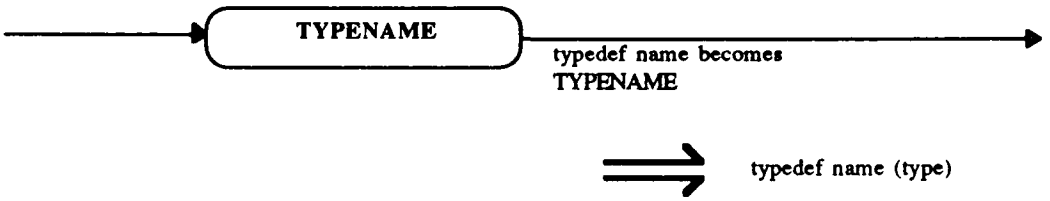


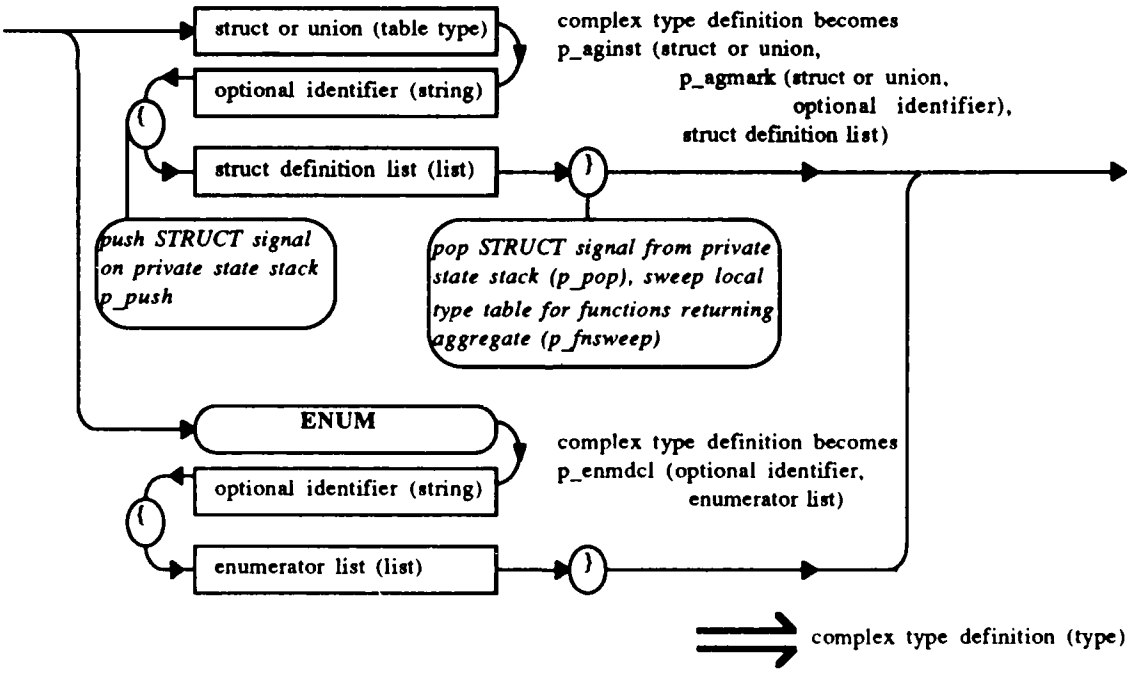
⇒ type specifier (type)

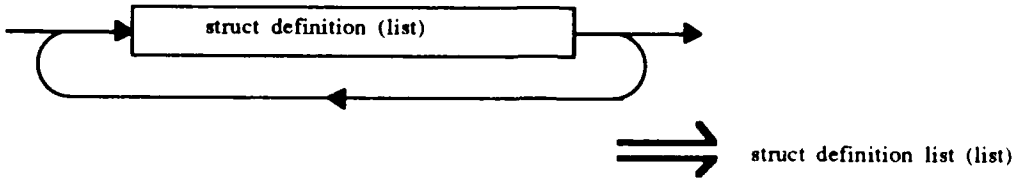


⇒ complex type specifier (type)

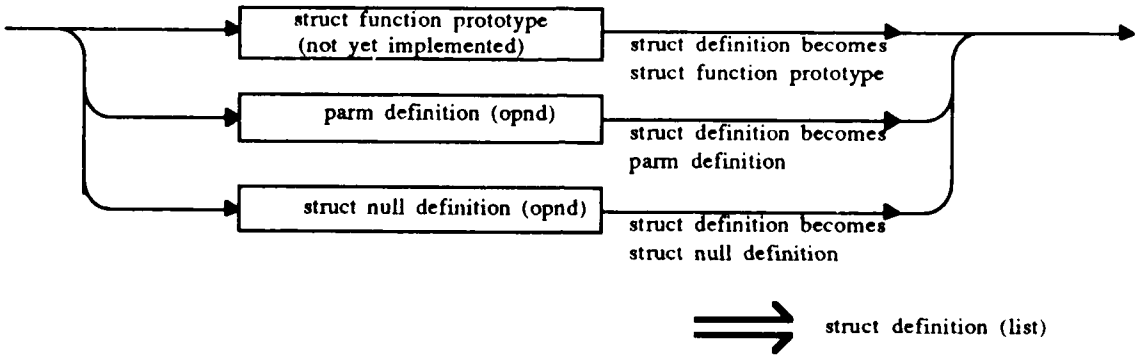

STRUCT
UNION
struct or union becomes
TBL_AGG (select struct table)
struct or union becomes
TBL_UNION (select union table)
struct or union (table type)

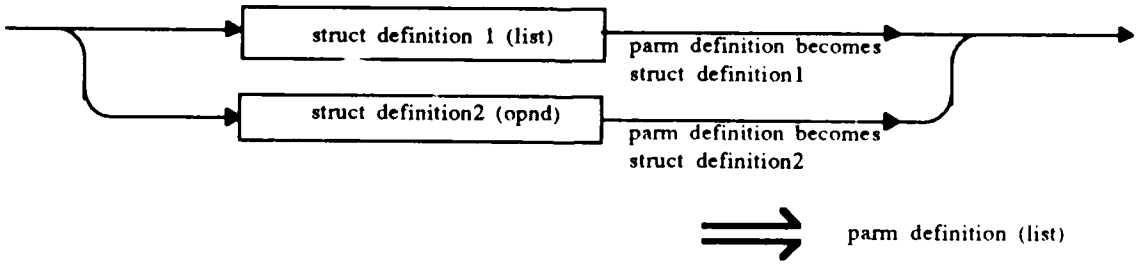

TYPENAME
typedef name becomes
TYPENAME
typedef name (type)

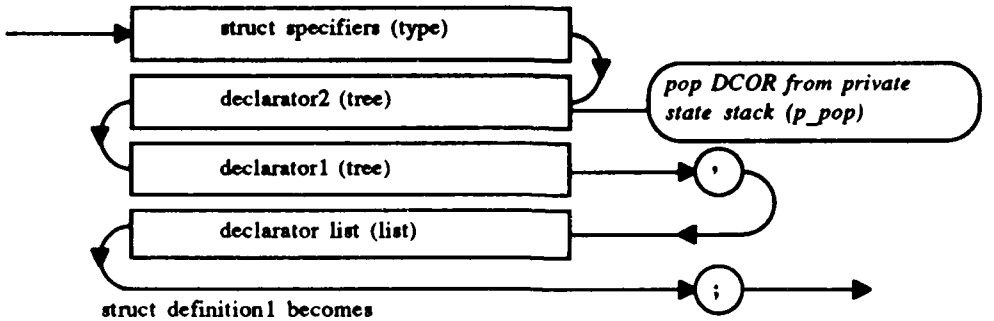

struct or union (table type)
optional identifier (string)
(
struct definition list (list)
)
complex type definition becomes
p_aginst (struct or union,
p_agmark (struct or union,
optional identifier),
struct definition list)
push STRUCT signal on private state stack p_push
pop STRUCT signal from private state stack (p_pop), sweep local type table for functions returning aggregate (p_fnsweep)
ENUM
optional identifier (string)
(
enumerator list (list)
)
complex type definition becomes
p_enmdcl (optional identifier,
enumerator list)
complex type definition (type)



struct definition list becomes:
one struct definition : struct definition
more than one : concatenation (catlst)
struct definition
to struct definition list

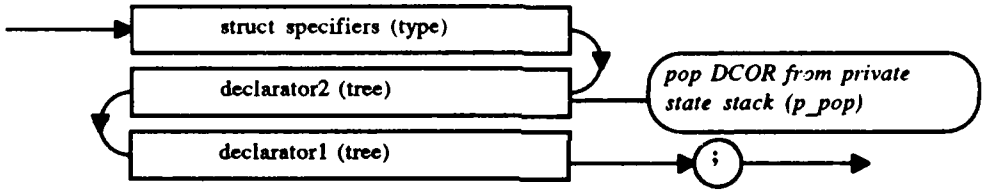






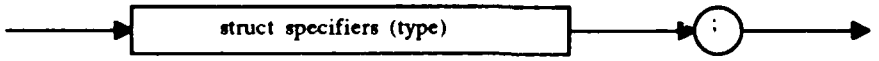
struct definition1 becomes
p_declare (
struct specifiers
, p_define (
p_dchain (
declarator2
, declarator1
)
)
, declarator list
)

⟹ struct definition1 (list)



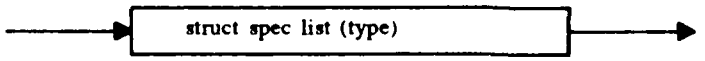
struct definition2 becomes
p_declare (
struct specifiers
, p_define (
p_dchain (
declarator2
, declarator1
)
)
, NULL (no declarator list)
)

⟹ struct definition2 (list)



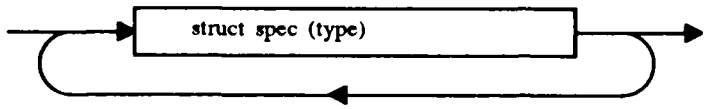
struct null definition becomes
nulldcl (struct specifiers)

⟹ null definition (list)



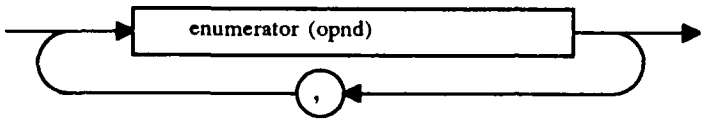
struct specifiers becomes struct spec list

⟹ struct specifiers (type)



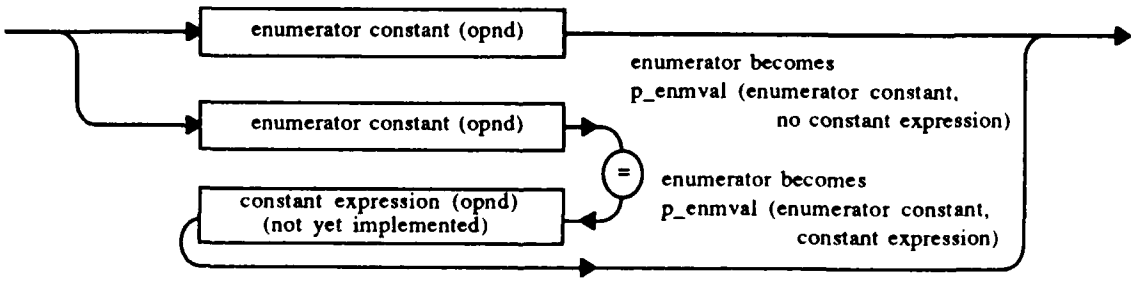
struct spec list becomes:
one struct spec : struct spec
more than one : p_atrchk (struct spec
, struct spec list
)
(merge struct specs if possible)

⟹ struct spec list (type)



enumerator list becomes:
one enumerator : p_list (no list, enumerator)
create single element list
more than one : p_list (enumerator list,
enumerator)
create multiple element list

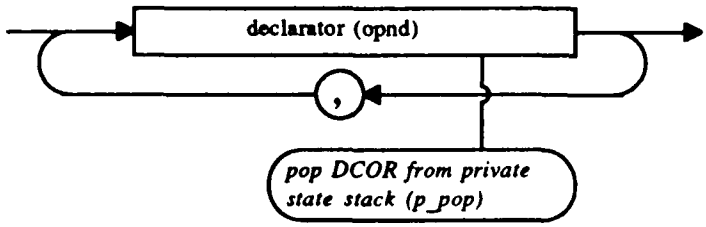
⟹ enumerator list (list)



⟹ enumerator (opnd)

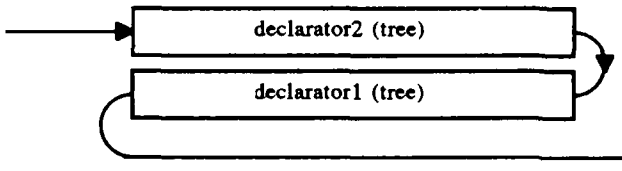
enumerator becomes
p_maken (IDENTIFIER)
(create enumerator operand)

⟹ enumerator (opnd)



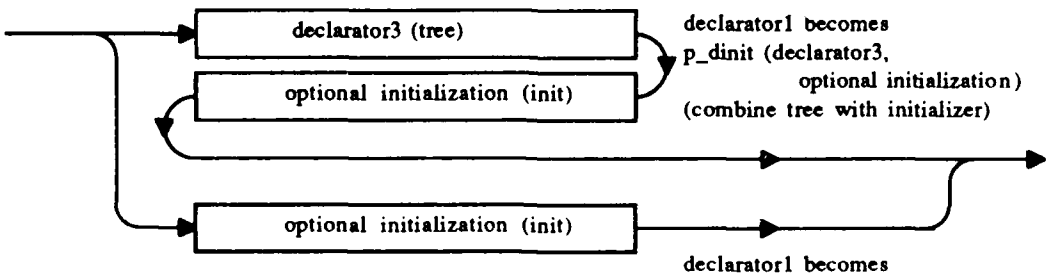
declarator list becomes:
one declaration : p_list (no declarator list,
p_define (declarator))
(create single element list of declarator)
more than one : p_list (declarator list,
p_define (declarator))
(create multiple element list of declarator)

⟹ declarator list (list)



declarator becomes
p_dchain (declarator2, declarator1)
(combine trees into an operand)

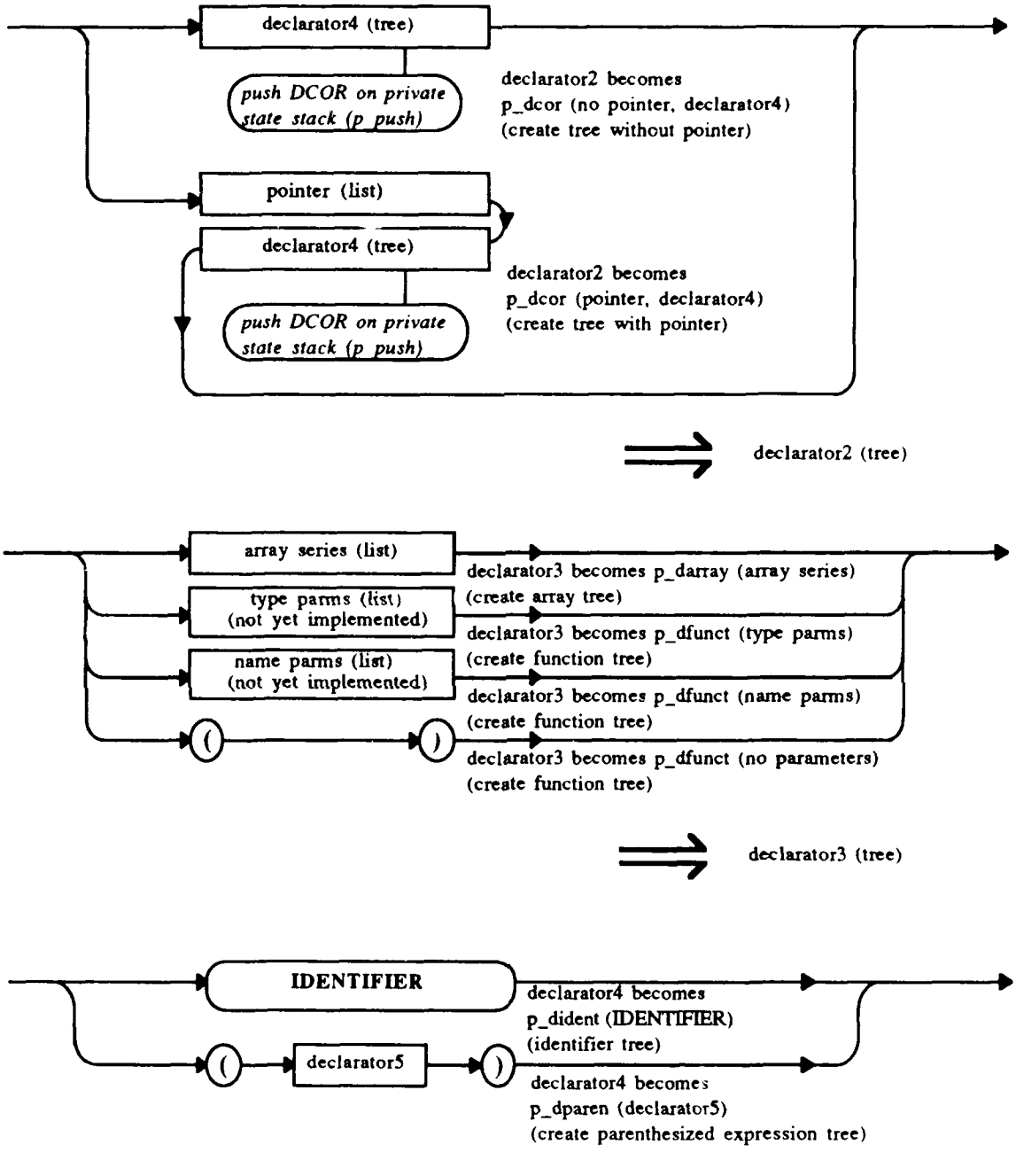
⟹ declarator (opnd)

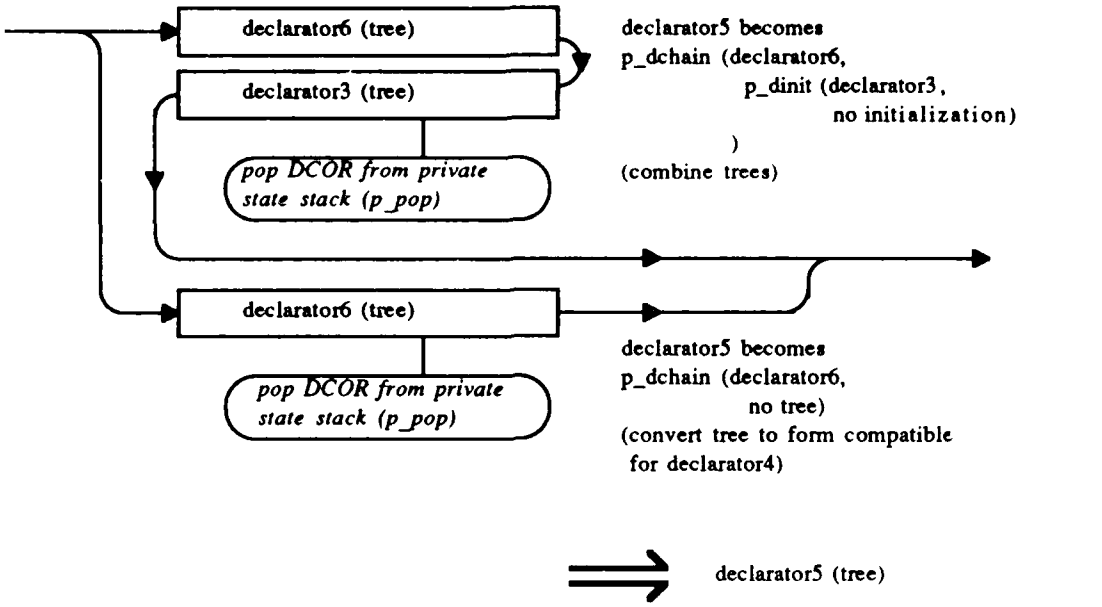


declarator1 becomes
p_dinit (declarator3,
optional initialization)
(combine tree with initializer)

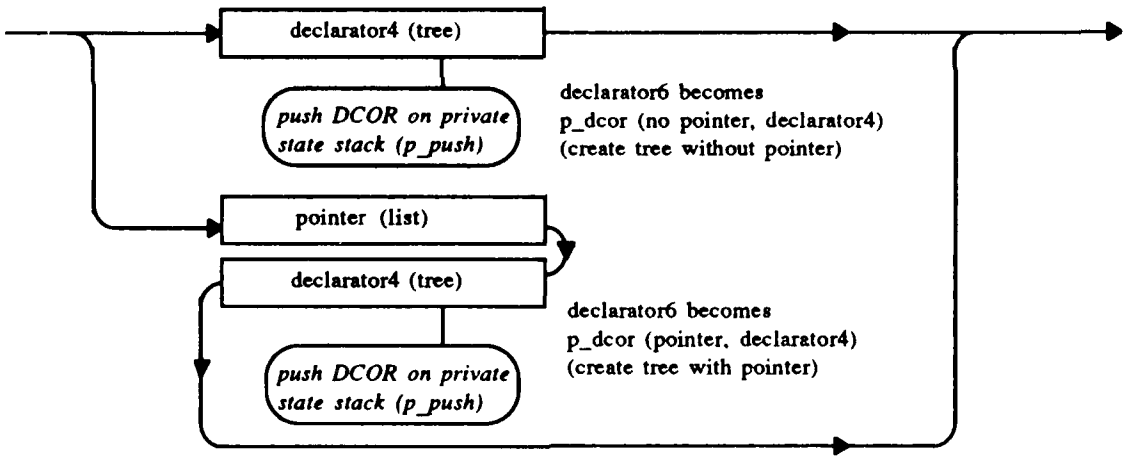
declarator1 becomes
p_dinit (no declarator3,
optional initialization)
(transform initializer into tree)

⟹ declarator1 (tree)


declarator4 (tree)
push DCOR on private state stack (p_push)
declarator2 becomes
p_dcor (no pointer, declarator4)
(create tree without pointer)
pointer (list)
declarator4 (tree)
push DCOR on private state stack (p_push)
declarator2 becomes
p_dcor (pointer, declarator4)
(create tree with pointer)
declarator2 (tree)
array series (list)
type parms (list)
(not yet implemented)
name parms (list)
(not yet implemented)
(
)
declarator3 becomes p_darray (array series)
(create array tree)
declarator3 becomes p_dfunct (type parms)
(create function tree)
declarator3 becomes p_dfunct (name parms)
(create function tree)
declarator3 becomes p_dfunct (no parameters)
(create function tree)
declarator3 (tree)
IDENTIFIER
(
declarator5
)
declarator4 becomes
p_dident (IDENTIFIER)
(identifier tree)
declarator4 becomes
p_dparen (declarator5)
(create parenthesized expression tree)
declarator4 (tree)

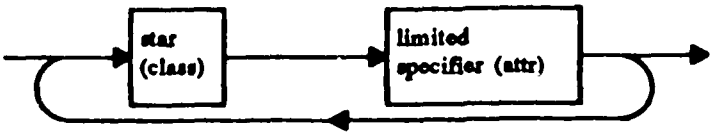


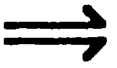
⟹ declarator5 (tree)



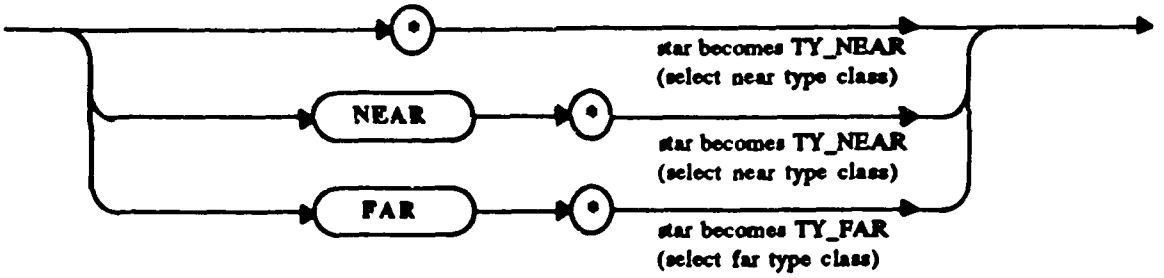
⟹ declarator6 (tree)

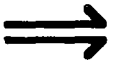
Appendix D



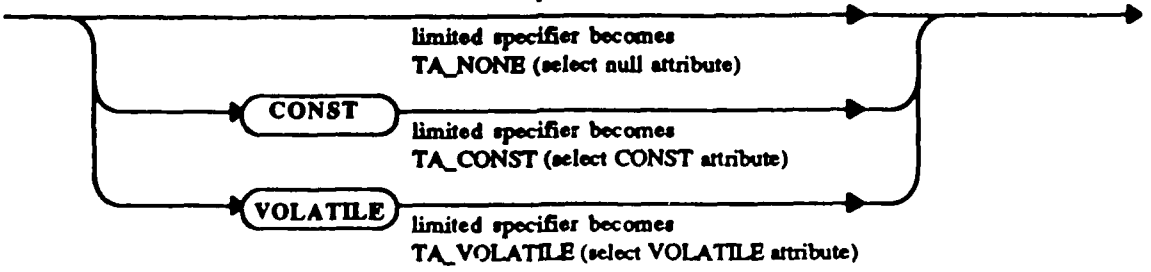


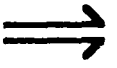
```
pointer becomes:
    one iteration  :  p_list (no pointer,
                              p_star (star,
                                      limited specifier)
                             )
          (create single element list of pointer)
    more than one  :  p_list (pointer,
                              p_star (star,
                                      limited specifier)
                             )
          (create multiple element list of pointer)
```



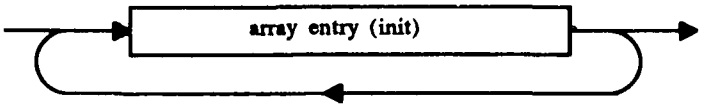


star (class)



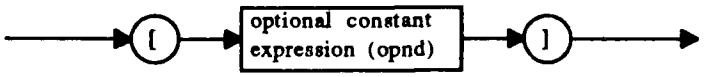


limited specifier (attr)



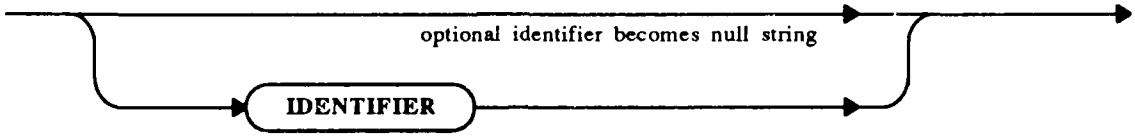
array series becomes:
one array entry : p_list (no array series, array entry)
(create single element list of array entry)
more than one : p_list (array series, array entry)
(create multiple element list of array entry)

⟹ array series (list)

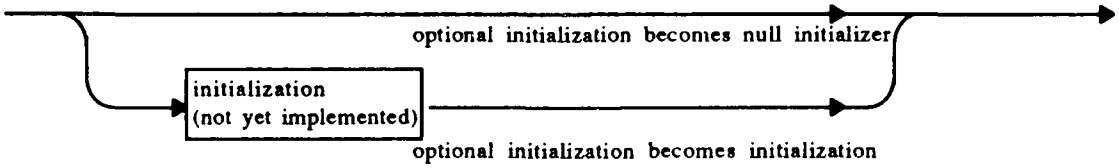


array entry becomes
p_dbound (optional constant expression)
(create array bounds operand)

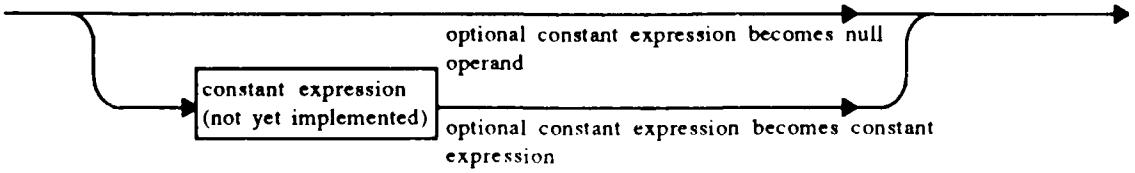
⟹ array entry (init)



⟹ optional identifier (string)



⟹ optional initialization (init)



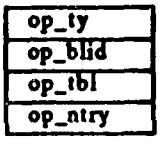
⟹ optional constant expression (opnd)

Appendix E

Compiler Data Structures

Operand Identification Structure – *Opnd*

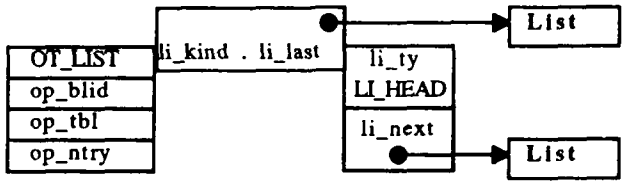
op_ty	:	operand type (OT_BLOCK, OT_CODE, OT_LIST, OT_MEM, OT_INIT, OT_QUAD, OT_REF, OT_SYMBOL, OT_TABLE, OT_TYPE, OT_VAL)
ob_blid	:	owner block identification (int)
op_tbl	:	owner table type (TBL_AGG, TBL_BLOCK, TBL_CODE, TBL_ENUM, TBL_ENUMERATOR, TBL_MEM, TBL_TYPE, TBL_UNION, TBL_VAR)
op_ntry	:	entry number in table (int)



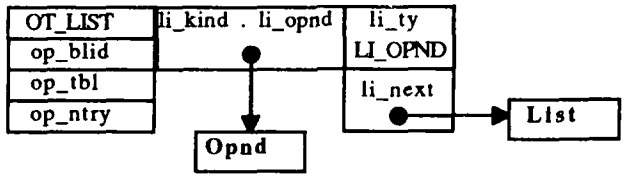
All *operands* (Block, Code, List, Member, Initializer, Quadruple, Reference, Symbol, Table, Type, and Value) have an *operand identification structure*that essentially causes each operand to behave like tagged storage. The operand identification structure is a tag that allows the Documentation Language routines to determine the operand type, owner block number, owner table type, and entry number for the purposes of selection, decision, and verification. In some modules, the operands are visible only by these tags, and the operand type needs to be determined before individual data elements can be accessed. In other modules, the operands are visible by the individual operand type structure; to determine what information exists in the tag the operand can be coerced (cast) into the operand identification structure.

Linked list structure operand – *List*

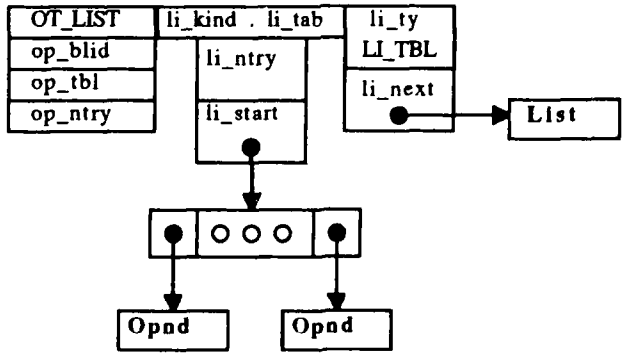
li_ty	:	variant indicator (LI_OPND, LI_HEAD, I.I_TBL)
li_next	:	pointer to next list structure
li_kind . li_opnd	:	pointer to single operand
li_kind . li_last	:	pointer to findal list structure
li_kind . li_tab	:	multiple operand variant
li_kind . li_tab . li_ntry	:	number of operands (int)
li_kind . li_tab . li_start	:	pointer to first operand



List head variant



Single operand *List* variant

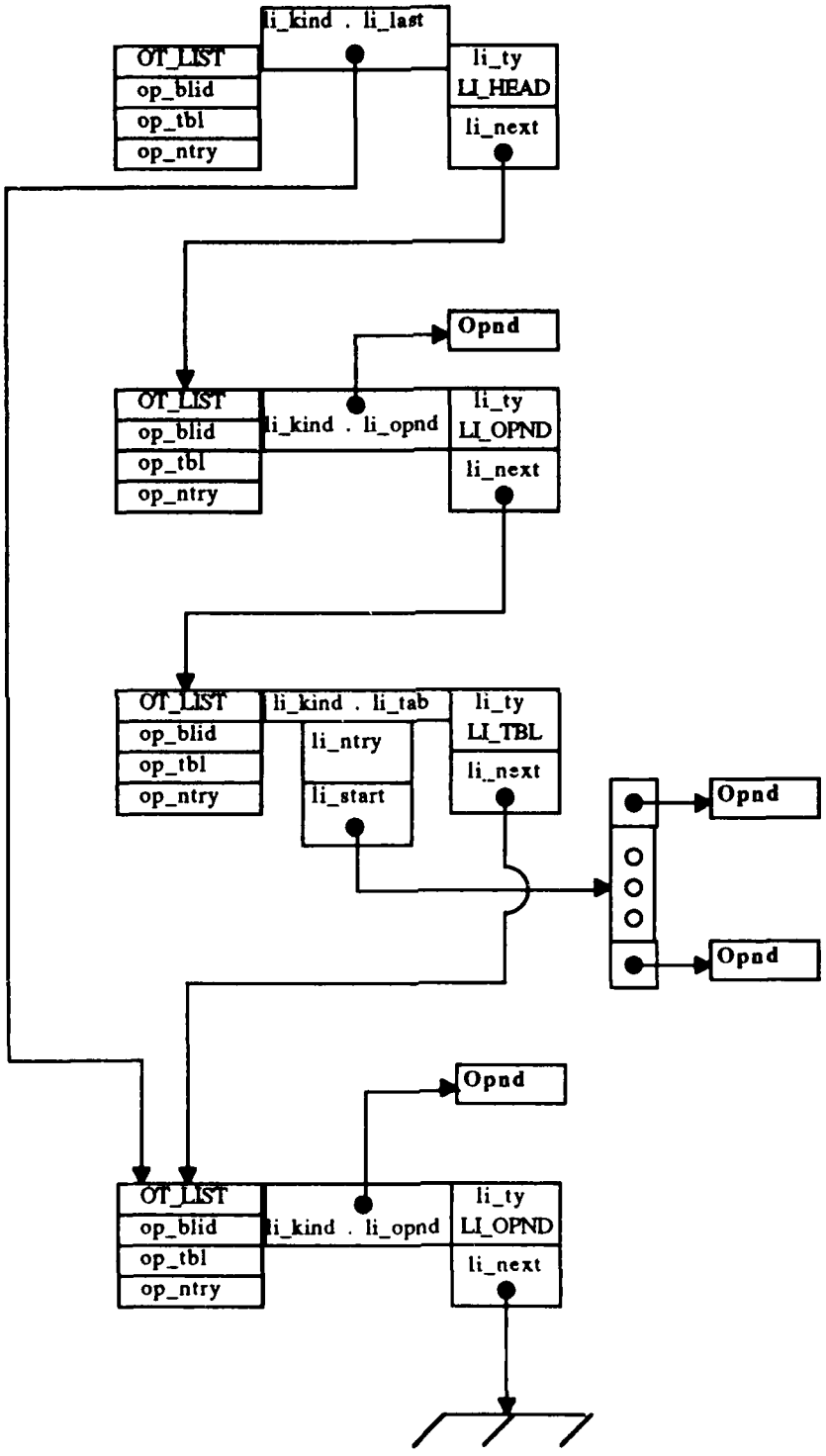


Multiple operand *List* variant

The variants of the linked list structure operand work together to form linked lists. A linked list always starts with a list head variant. Following are a mixed series of single or multiple operand variants. The field *li_kind.li_last* always points to the final list structure operand in the list. The multiple operand variant uses a double pointer sequence to access a linear series of operands. An example of a linked list is shown on the next page.

Appendix E

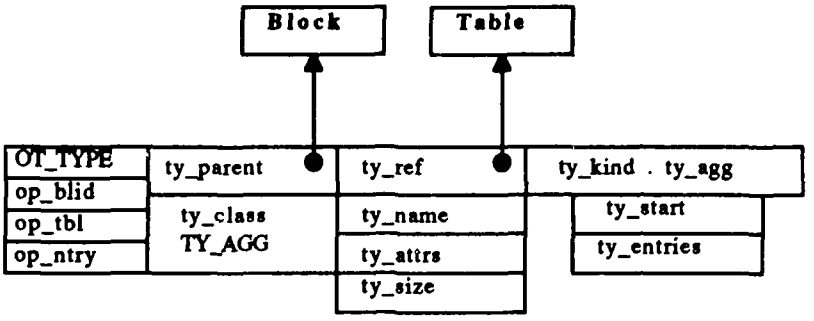
Example of linked list using *List* structure operands



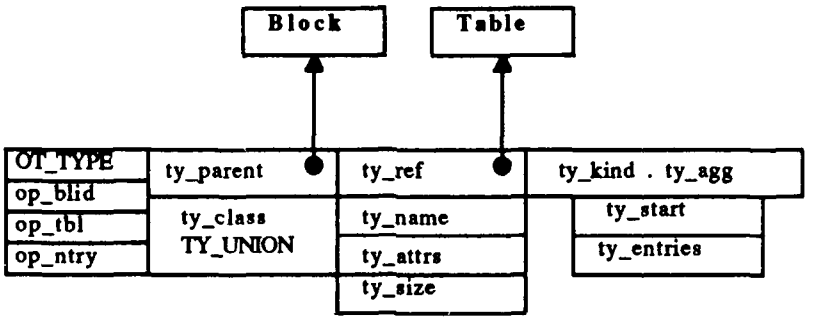
Sequence is: list head, single operand, multiple operand, single operand.

Appendix E

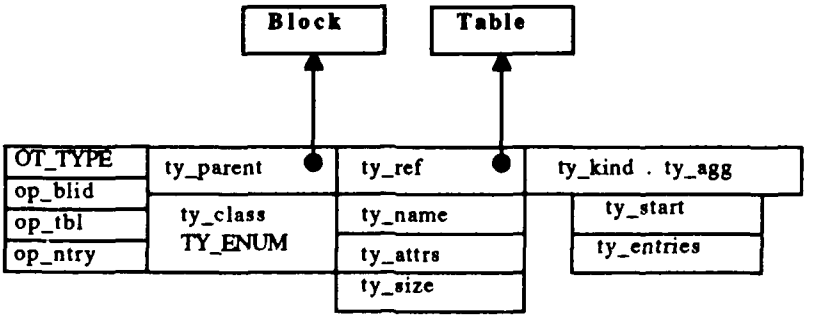
Type Operand Structure – *Type*



Struct *Type* Variant



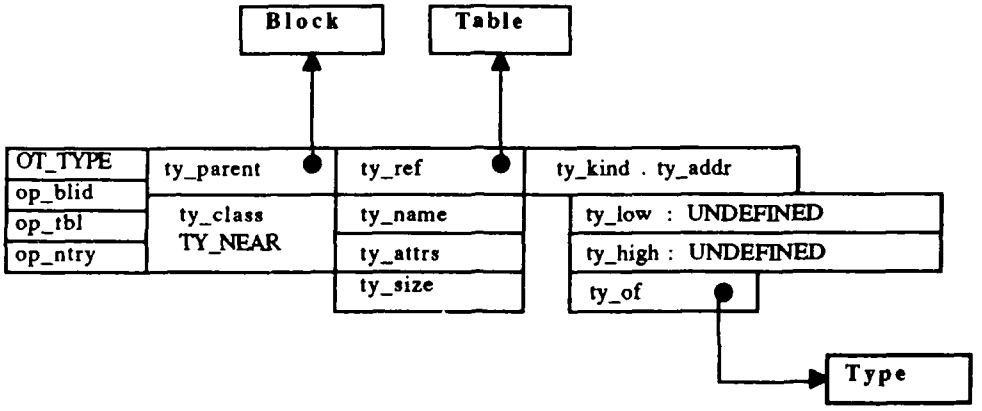
Union *Type* Variant



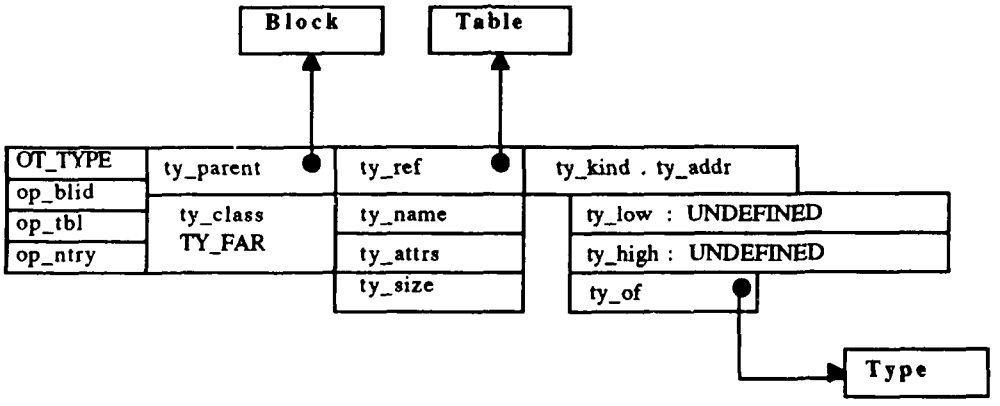
Enumeration *Type* Variant

ty_parent	:	parent block of type (to determine lexical scope of type)
ty_class	:	type class (TY_AGG,TY_UNION, TY_ENUM,TY_NEAR, TY_FAR,TY_ARR,TY_FUNC,TY_ALIAS,TY_LABEL, TY_CONST, TY_LINK,TY_YACC)
ty_ref	:	type's reference table
ty_name	:	name of type (string)
ty_attrs	:	type's attributes (attr)
ty_size	:	size of type (long)
ty_kind.ty_agg	:	table dependent variant (TY_AGG, TY_UNION, TY_ENUM)
ty_kind.ty_agg.ty_start	:	position of first entry in relevant table
ty_kind.ty_agg.ty_entries	:	number of entries in relevant table

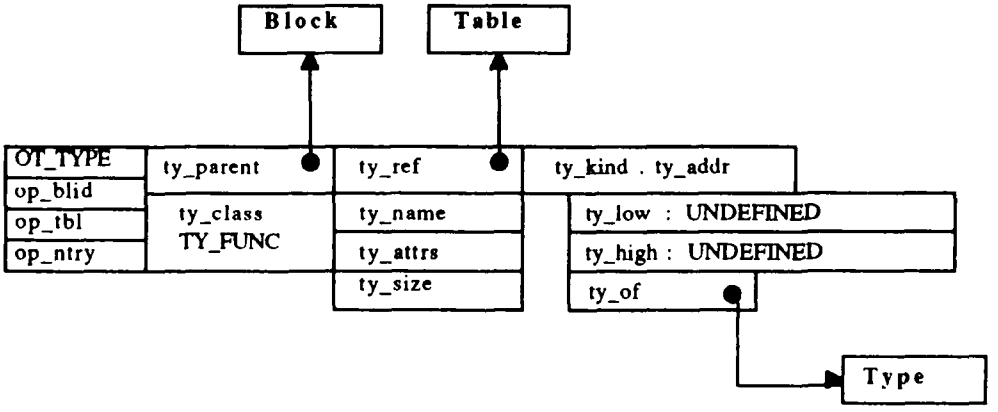
Type Operand Structure (cont.)



Near *Type* Variant



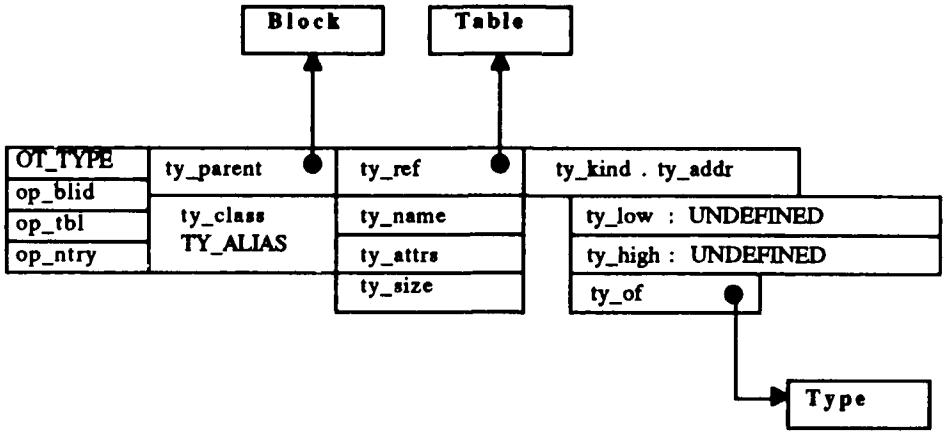
Far *Type* Variant



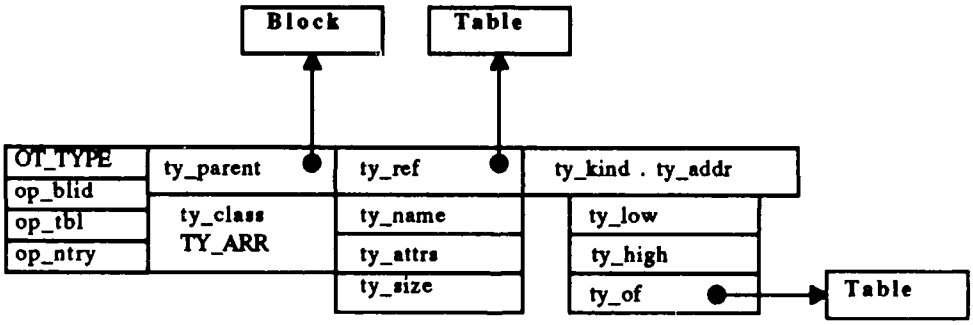
Function *Type* Variant

ty_kind.ty_addr	:	type dependent variant (TY_NEAR, TY_FAR, TY_FUNC, TY_ARR, TY_ALIAS)
ty_kind.ty_addr.ty_low	:	low bound of array subscript (used in TY_ARR)
ty_kind.ty_addr.ty_high	:	high bound of array subscript (used in TY_ARR)
ty_kind.ty_addr.ty_of	:	subsidiary type (type of type)

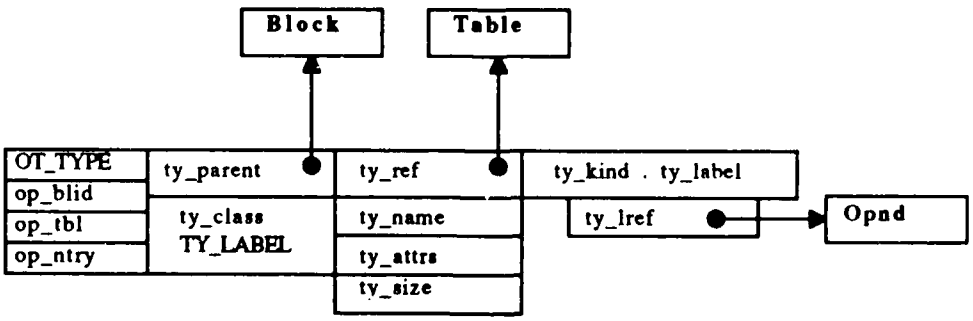
Type Operand Structure (cont.)



Aliased *Type* Variant



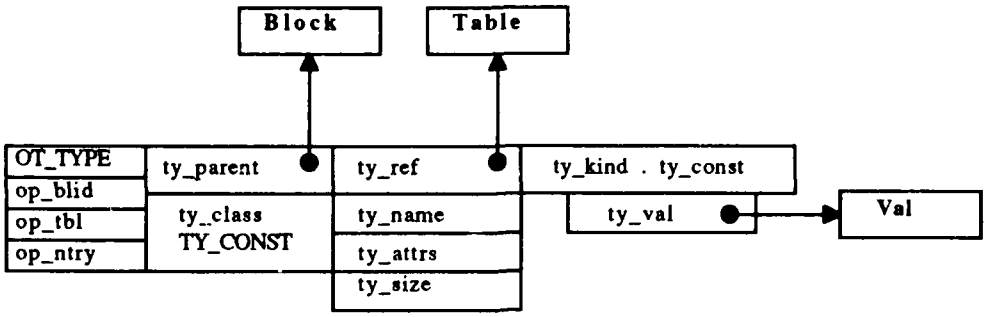
Array *Type* Variant



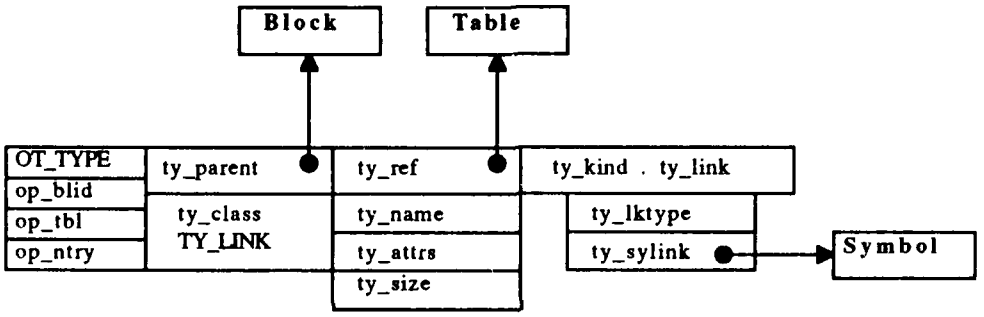
Label *Type* Variant

ty_kind.ty_addr	:	type dependent variant (TY_NEAR, TY_FAR, TY_FUNC, TY_ARR, TY_ALIAS)
ty_kind.ty_addr.ty_low	:	low bound of array subscript (used in TY_ARR)
ty_kind.ty_addr.ty_high	:	high bound of array subscript (used in TY_ARR)
ty_kind.ty_addr.ty_of	:	subsidiary type (type of type)
ty_kind.ty_label	:	statement label (TY_LABEL)
ty_kind.ty_label.ty_lref	:	label reference operand (opnd)

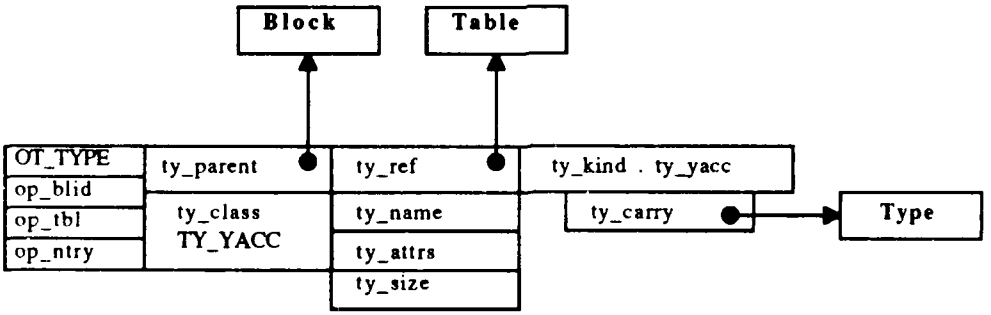
Type Operand Structure (cont.)



Constant *Type* Variant



Link *Type* Variant

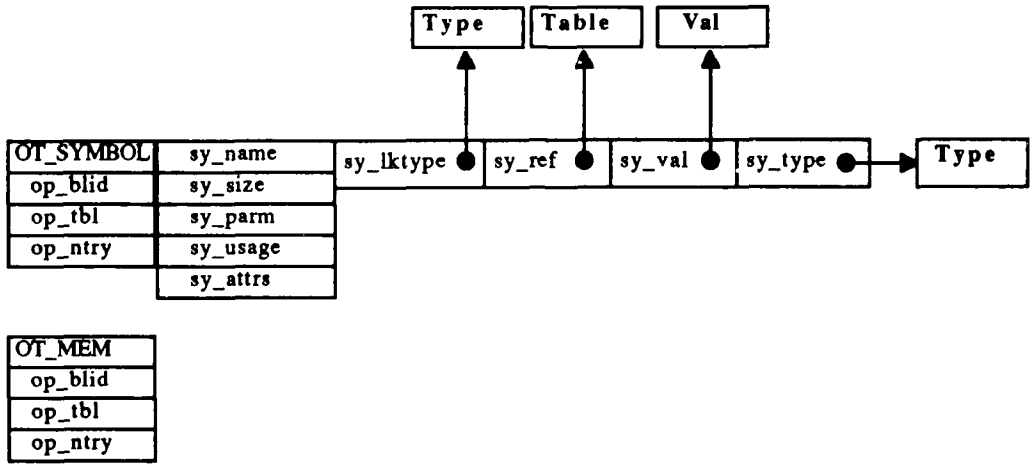


Compiler *Type* Variant

ty_kind.ty_const	:	constant type variant (TY_CONST)
ty_kind.ty_const.ty_val	:	value of constant type
ty_kind.ty_link	:	link type variant (TY_LINK)
ty_kind.ty_link.ty_sylink	:	symbol link for aggregates and enumerator types
ty_kind.ty_link.ty_lktype	:	specify which type is using symbol link (LK_STRUCT, LK_UNION, LK_ENUM)
ty_kind.ty_yacc	:	compiler type variant (TY_YACC)
ty_kind.ty_yacc.ty_carry	:	type being carried through grammer by compiler type

Appendix E

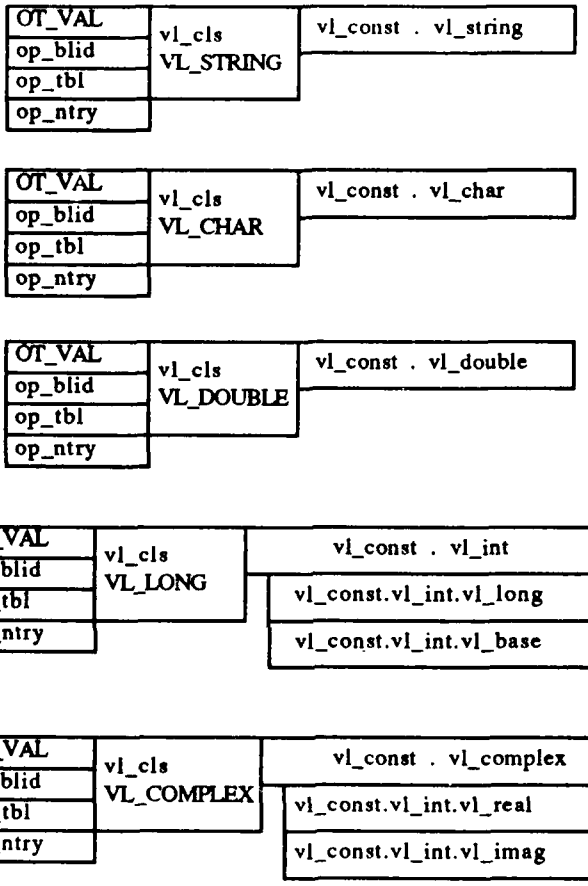
Symbol Structure Operand – *Symbol*
and
Member Structure Operand – *Mem*



sy_name	:	symbol name (string)
sy_size	:	symbol size (long)
sy_parm	:	symbol parameter number (int)
sy_usage	:	symbol usage class (US_DECL,US_FPARM)
sy_attrs	:	symbol attributes (attrs)
sy_lk_type	:	pointer to symbol's owner link
sy_ref	:	pointer to symbol's reference table
sy_val	:	pointer to symbol's initial value
sy_type	:	pointer to symbols's type

The composition of *symbol* and *mem* structure operands is the same, except that the operand type field, op_ty, is different (OT_SYMBOL for symbols, OT_MEM for members). Symbol operands belong in binary searched symbol tables, which are appropriate for random-accessed entities such as struct and union tags, and variables. Member operands belong in sequentially searched member tables, which are appropriate for sequentially accessed entities such as members of structs and unions, and enumerations within an enumerator type. The Symbol Table Variant is the Table Structure Operand in which symbol and member operands are stored.

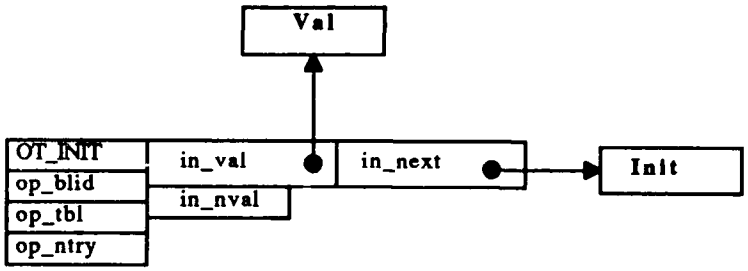
Value Structure Operand – *Val*



vl_cls	:	value class (VL_STRING, VL_LONG, VL_DOUBLE, VL_CHAR, VL_COMPLEX)
vl_const.vl_string	:	string value (string)
vl_const.vl_char	:	char value (long)
vl_const.vl_double	:	floating point value (double)
vl_const.vl_int.vl_long	:	integer value (long)
vl_const.vl_int.vl_base	:	numerical base of integer value
vl_const.vl_complex.vl_real	:	real component of complex value
vl_const.vl_complex.vl_inag	:	imaginary component of complex value

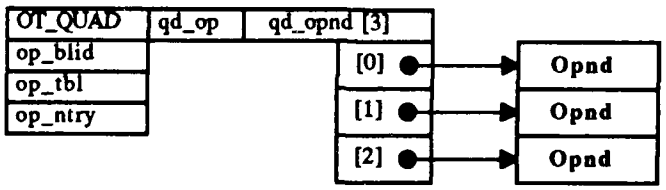
Appendix E

Initializer Structure Operand – *Init*



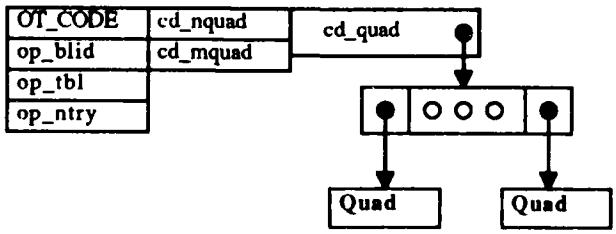
in_nval	:	number of initializer (int)
in_val	:	value of this initializer
in_next	:	pointer to next initializer

Quadruple Structure Operand – *Quad*



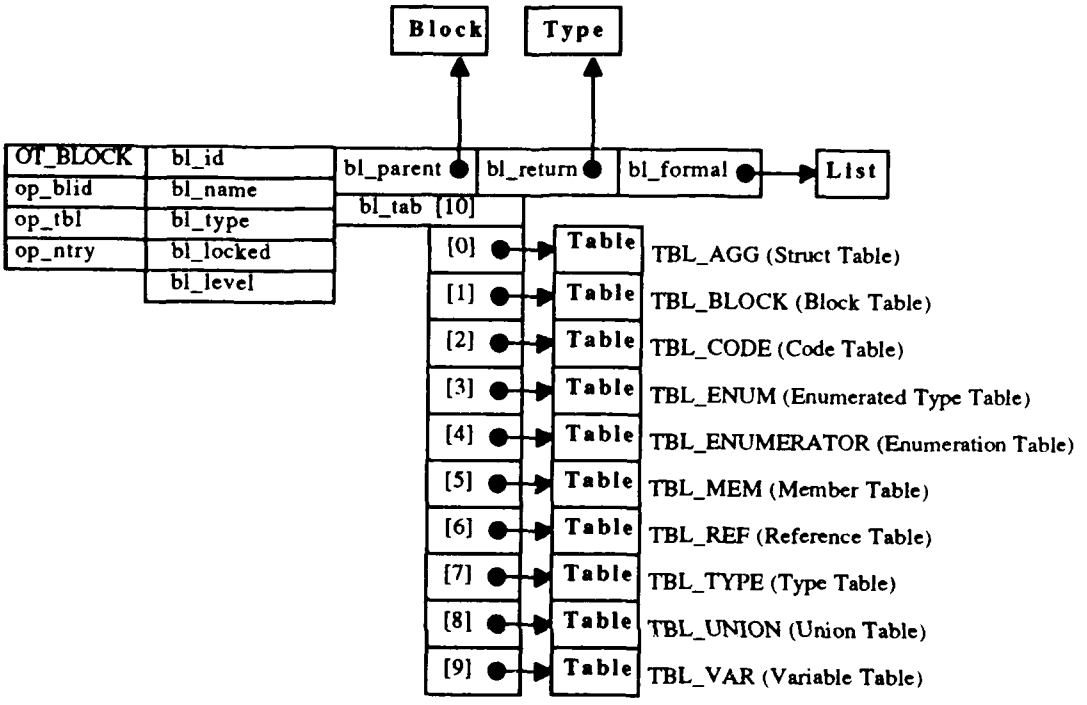
qd_op	:	quadruple operator (opcode)
qd_opnd [3]	:	Up to three pointers to operands

Code Table Structure Operand – *Code*



cd_nquad	:	number of quadruples used in code table
cd_mquad	:	number of quadruple spaces allocated to code table
cd_quad	:	pointer to first quadruple in code table

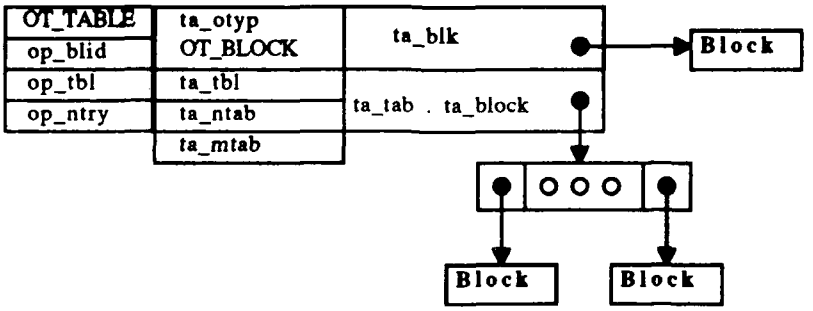
Block Structure Operand – *Block*



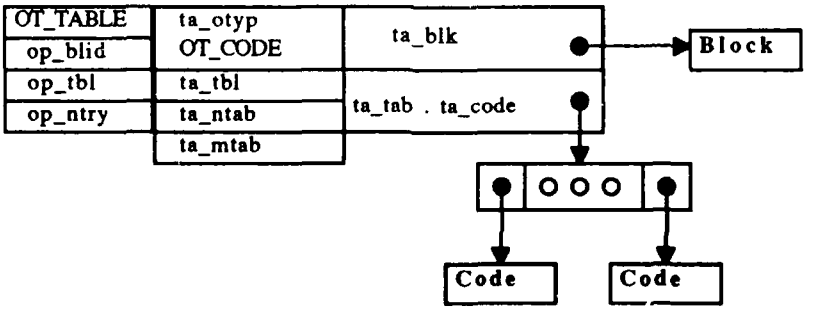
bl_id	:	block identification number (int)
bl_name	:	block name (string)
bl_type	:	block type (BT_NONE, BT_BASE, BT_BLOCK, BT_DATA, BT_FUNC, BT_PROTO, BT_SUBR)
bl_locked	:	was the block described as a function block with parameters and compound statement? (True / False)
bl_level	:	lexical level of block (int)
bl_parent	:	pointer to parent block of this block
bl_return	:	pointer to return type of this block if this block was described as a function block
bl_formal	:	pointer to formal parameter list of this block if this block was described as a function block

Appendix E

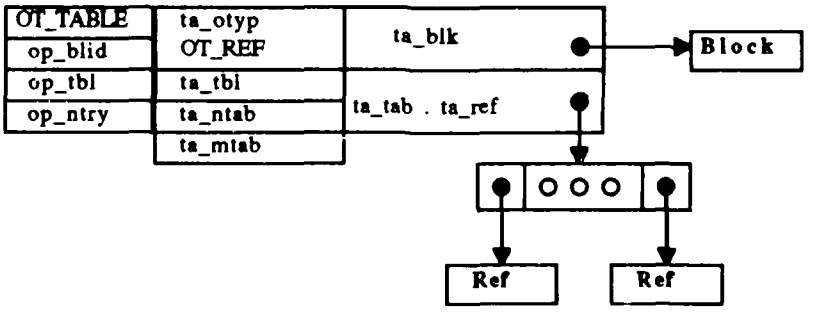
Table Structure Operand – *Table*



Block *Table* Variant ta_tbl : TBL_BLOCK (1)



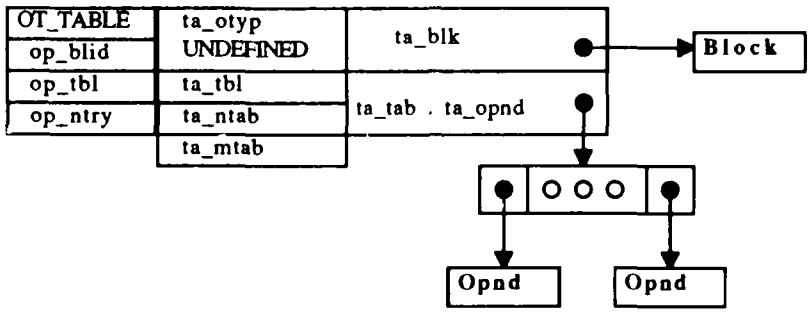
Code *Table* Variant ta_tbl : TBL_CODE (2)



Reference *Table* Variant ta_tbl : TBL_REF (6)

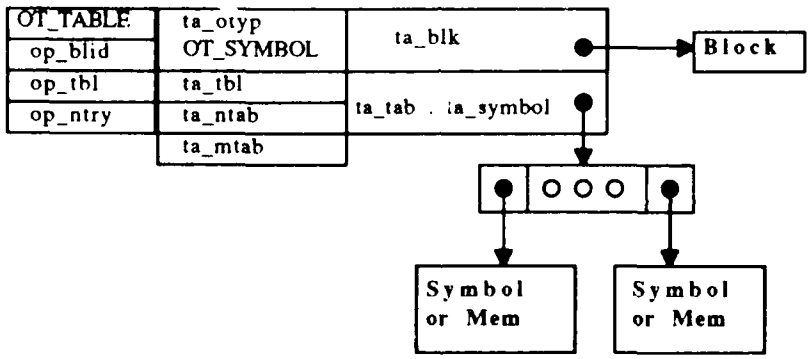
ta_otyp	:	operand type in this table (OT_BLOCK, OT_CODE, OT_REF, OT_OPND, OT_SYMBOL, OT_REF)
ta_tbl	:	specific table in owning block (0 - 9 : see Block Structure Operand)
ta_ntab	:	number of entries in this table
ta_mtab	:	number of entry spaces alocated to this table
ta_blk	:	pointer to block owning this table
ta_tab.ta_block	:	pointer to first block entry in block table
ta_tab.ta_code	:	pointer to first code entry in code table
ta_tab.ta_ref	:	pointer to first reference entry in reference table

Table Structure Operand (cont.)



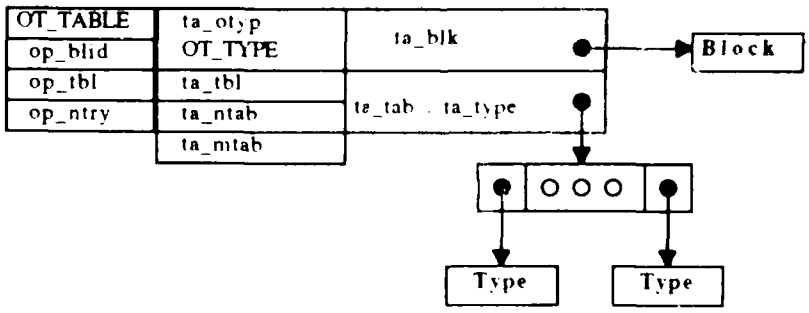
Operand *Table* Variant

Note : This variant exists for convieience in compiler internal operations. All *Table* Variants can be coerced (cast) into this form for the purpose of accessing the Operand Identification Structures of all elements in that coerced table.



Symbol *Table* Variant

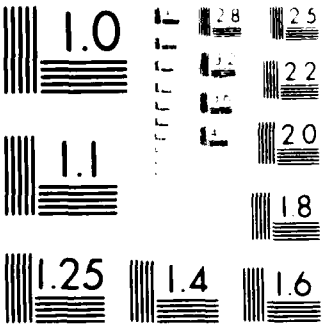
ta_tbl : (Symbol) TBL_AGG (0) TBL_ENUM (3)
TBL_UNION (8) TBL_VAR (9)
: (Member) TBL_ENUMERATOR (4) TBL_MEM (5)



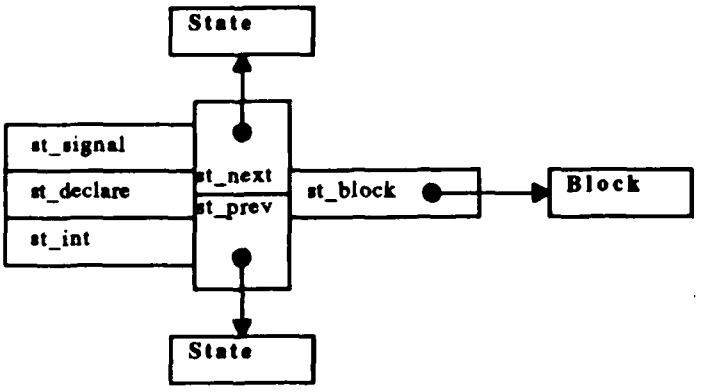
Type *Table* Variant ta_tbl : TBL_TYPE (7)

ta_tab.ta_opnd : pointer to first operand entry in table
ta_tab.ta_symbol : pointer to first symbol (member) entry in symbol (member) table
ta_tab.ta_type : pointer to first type entry in type table

AD-A185 092 DOCUMENTATION IN A SOFTWARE MAINTENANCE ENVIRONMENT(U) 2/2
TECHNICAL SOLUTIONS INC MESILLA PARK NM
L D LANDIS ET AL. 28 AUG 87 ARO-22935.1-EL-S
UNCLASSIFIED DAAG29-85-C-0026 F/G 12/5 NL


1.0
1.1
1.25
1.4
1.6
2.8
2.5
2.2
2.0
1.8

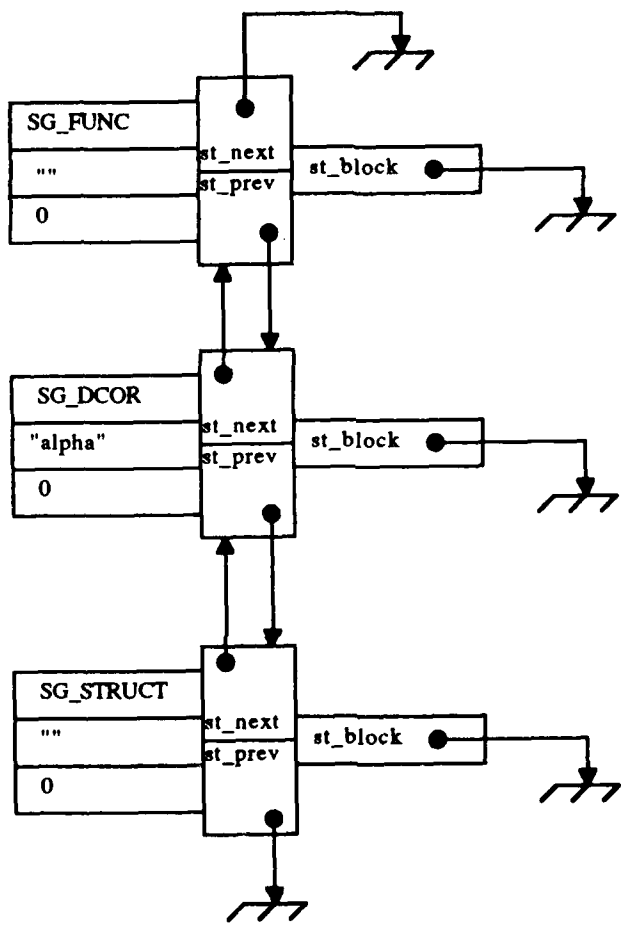
Stack state structure for private stack – *State*



st_signal	:	compiler state signal indicator (Signal)
st_declare	:	string storage for declarations (string)
st_int	:	general purpose integer (int)
st_block	:	pointer to a block for declarations of compound statement, function, or function prototype
st_next	:	next state toward top of stack
st_prev	:	previous state toward bottom of stack

The private state stack is a device that allows the Documentation Language compiler to transmit and receive signals to itself that indicate context-sensitive structures or other special conditions that require handling. The stack is constructed as a doubly-linked list, where each element has the ability to temporarily hold general-purpose information which may be relevant to the compiler when processing the signal or combination of signals.

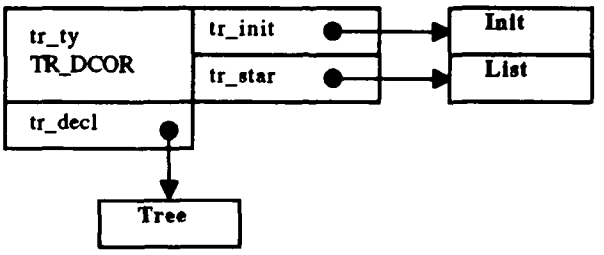
Example of private state stack



This example indicates that the parser is currently inside an aggregate. The identifier name "alpha" has been read by the parser. The declaration "alpha" has been declared to be a function or function prototype. The top of this diagram corresponds to the top of stack, which can be accessed by the function *p_tos()*. The bottom of this diagram corresponds to the bottom of stack, which can be accessed by the by the function *p_bos()*.

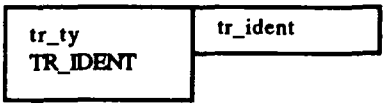
Declarator Tree Structure Elements – *Tree*

The declarator *Tree* is a device that allows the Documentation Language compiler to temporarily store the elements of a declarator in the exact order they were parsed. This form can be manipulated to enforce the precedence of parenthesis, arrays, functions, and pointers, or scanned when the compiler needs to make a decision based on declarator tree elements, such as if the declarator was named or abstract.



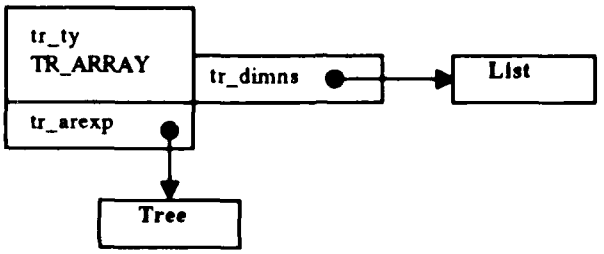
Declaration Variant

tr_ty	:	Tree Type (is TR_DCOR)
tr_decl	:	Declaration Expression Tree
tr_init	:	Optional Initializer
tr_star	:	Optional List of Pointer Types



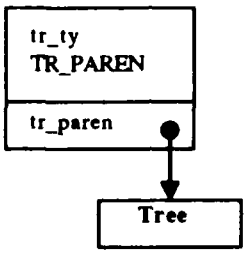
Identifier Variant

tr_ty	:	Tree Type (is TR_IDENT)
tr_ident	:	Identifier name (string)



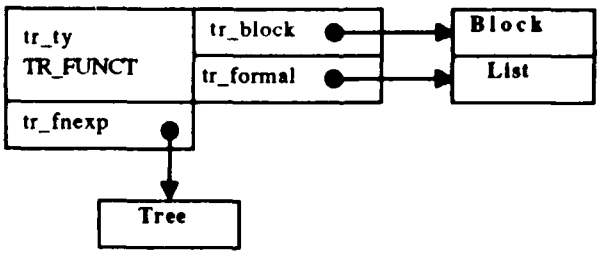
Array Variant

tr_ty	:	Tree Type (is TR_ARRAY)
tr_dims	:	List of array dimensions
tr_arexp	:	Array Expression Tree



Parenthesized Expression Variant

tr_ty	:	Tree Type (is TR_PAREN)
tr_paren	:	Parenthesized Expression



Function Variant

tr_ty	:	Tree Type (is TR_FUNCT)
tr_block	:	Prototype Block for this function expression
tr_formal	:	List of parameter declarator
tr_fnexp	:	Function Expression Tree

Appendix F

Suggestions for Further Reading

Aho, A. V., and Corasick, M. J., Efficient String Matching: An Aid to Bibliographic Search, Communications of the ACM, vol. 18, 333-340, 1975.

Aho, A. V., Hopcroft, J. E., and Ullman, J. D., Data Structures and Algorithms, Addison–Wesley, 1983.

Aho, A. V., and Ullman, J. D., Principles of Compiler Design, Addison-Wesley, 1977.

Barrett, W. A., and Couch, J. D., Compiler Construction: Theory and Practice, Science Research Associates, 1979.

Berg, H. K., Boebert, W. E., Franta, W. R., and Moher, T. G., Formal Methods of Program Verification and Specification, Prentice–Hall, NJ., 1982.

Biggs, C. L., Birks, E. G., and Atkins, W., Managing the Systems Development Process, Prentice–Hall, NJ., 1980.

Birrell, N. D., and Ould, M. A., A Practical Handbook for Software Development, Cambridge University Press, 1985.

Boar, B. H., Application Prototyping, John Wiley & Sons, 1984.

DeMarco, T., Controlling Software Projects, Yourdon Press, NY., 1982.

Freedman, D. P., and Weinberg, G. M., Handbook of Walkthroughs, Inspections, and Technical Reviews, Little, Brown, and Company, 1982.

Feuer, A. R., The C Puzzle Book, Prentice–Hall, NJ., 1982.

Grogono, P., Programming in Pascal, Addison–Wesley, 1980.

Hansen, K., Data Structured Program Design, Ken Orr and Associates, 1983.

Hunter, R., The Design and Construction of Compilers, John Wiley & Sons, 1981.

Jensen, K., and Wirth, N., Pascal User Manual and Report, Springer–Verlag, 1974.

Johnson, S. C., Yacc: Yet Another Compiler Compiler, Computing Science Technical Report No. 32, Murray Hill, NJ., 1975.

Jones, C. B., Software Development, Prentice–Hall International, London, 1980.

Kernighan, B. W., Ratfor: A Preprocessor for a Rational Fortran, Software Practice and Experience, 1975.

Kernighan, B. W., and Ritchie, D. M., The C Programming Language, Prentice-Hall, NJ., 1978.

Lesk, M. E., The Portable C Library, Computing Science Technical Report, Report No. 31, Murray Hill, NJ.

Martin, J., Application Development Without Programmers, Prentice–Hall, NJ. 1987.

Martin, J. Fourth–Generation Languages, Volumes 1 – 3, Prentice–Hall, NJ., 1985.

Martin, J. Recommended Diagramming Standards for Analysts & Programmers, Prentice–Hall, NJ., 1987.

Martin, J., and McClure, C., Action Diagrams: Clearly Structured Program Design, Prentice-Hall, NJ., 1985.

Martin, J., and McClure, C., Structured Techniques for Computing, Prentice-Hall, NJ., 1985.

Purdum, J. J., Leslie, T. C., Stegemoller, A. L., C Programmer's Library, Que Corporation, 1984.

Schreiner, A. T., and Friedman, H. G. Jr., Introduction to Compiler Construction with Unix, Prentice-Hall, NJ., 1985.

Swann, G. H., Top–Down Structured Design Techniques, Petrocelli Books, 1978.

Tausworthe, R. C., Standardized Development of Computer Science Software, Parts 1 and 2, Prentice–Hall, NJ., 1977.

Ulmman, J. D., Principles of Database Systems, Computer Science Press, MD., 1982.

Ada Joint Program Office, Reference Manual for the Ada Programming Language, United States Department of Defense, 1983.

Vick, C. R., and Ramamoorthy, C. V., Handbook of Software Engineering, Van Nostrand Reinhold Company, NY., 1984.

Warnier, J., Logical Construction of Systems, Van Nostrand Reinhold Company, NY., 1981.

Yourdon, E., and Constantine, L. L., Structured Design, Prentice–Hall, NJ., 1979.

Zelkowitz, M. V., Shaw, A. C., and Gannon, J. D., Principles of Software Engineering and Design, Prentice–Hall, NJ., 1979.

END

DATE

FILMED

DEC.

1987