2-12-1979

The AL Compiler

Kenneth Reek

T H E A L C O M P I L E R

Submitted by
Kenneth A. Reek
February 12, 1979

C O N T E N T S

1.
Overview

The AL compiler was designed and implemented on the PDP-11/34 minicomputer system during the summer of 1978. The project was undertaken to fulfill the Master of Science Thesis requirement in the Graduate School of Computer Science and Technology at Rochester Institute of Technology in Rochester, New York by Kenneth A. Reek. This report describes the internal operation of the compiler, its design considerations, and the history of its development.

HOST SYSTEM

All work for this project was done on the School's PDP-11/34 minicomputer system using the UNIX (c) operating system. UNIX was chosen partly for practical reasons (it is the most versatile of the operating systems that are used on the PDP-11, hence it is scheduled to run more often than the others), but mainly because of the software tools it offered. UNIX is an excellent environment in which to develop software due to its many useful utility programs, flexible directory structure, and brief, concise command format. Development of sections of the compiler was simplified by placing groups of related modules into their own directories. This facilitated the use of some special tools. For example, after changing the definition of a parameter, the "grep" program aided the analysis of the change by pinpointing all places in the compiler where that parameter was used. When components of the compiler were finished, they were put into a directory that could be easily accessed from all of the development directories.

LANGUAGE DESIGN GOALS

AL is an acronym for "Algorithmic Language". As the name suggests, AL was originally designed for clean, well structured expression of algorithms. The syntactic structures used closely resemble those in many of the textbook pseudo-languages. Because of this, once an algorithm is specified in pseudo-code, most of the changes needed to produce a working AL program are minor syntax corrections, such as adding semi-colons. Also, when reading an AL program, the algorithms used in it are quite ap-

(c) UNIX is a registered trademark of Bell Telephone Laboratories, Inc.

parent because they are not hidden by clumsy syntactic structures.

LANGUAGE FEATURES

AL includes several features not found in most languages, even some of the recent, modern ones. Many of these features were added after programming experience in the new language demonstrated their desirability.

An AL program consists of a main program and zero or more procedures and functions. Procedures and functions may be included in the source file or may be compiled in separate source files. They may be recursive and reentrant, if desired. Data may be passed between routines with arguments or global variables.

All variables that are declared in a procedure or a function are considered local to that routine, and are created each time the routine is called and destroyed when it exits. Global variables are never destroyed, and may be accessed by any routine in the program.

Program constants may be defined using expressions which contain previously declared constants. This feature, commonly found only in assembly languages, greatly simplifies the creation of tables and data structure offsets, as illustrated in the AL module "c.quad".

Another assembly language feature implemented in AL is conditional compilation. When constants are used in "if" statements, the peephole optimizer performs the comparison at compile time. If the test is true, the "if" statement is removed, allowing the statements it controls to be executed. If the test is false, the "if" statement and the statements it controls are all removed. No object code is produced to test the condition at execution time, and if the condition is false, no object code is produced for the statements within the "if", either. This is commonly referred to as conditional compilation. That name is not perfectly accurate here since the statements are all compiled, but the optimizer discards the undesired statements before any object code is produced, so the net effect is the same.

One common use of conditional compilation is to allow debugging statements, controlled by a constant, to be left in the production version of a program without incurring any penalty in the size of the object program. When future modifications to the program need testing, the constant can be changed to allow compilation of the existing debug statements; the statements themselves do not have to be reentered into the program.

An argument to a function or a procedure has two characteristics, "type" and "mode". Type can be integer, array, or string, and depends on how the argument (or, if it is an expression, the components of it) were declared. AL supports two different modes, call by reference and call by value. Since the compiler saves the mode and type of the formal arguments in procedure and function declarations, the type of actual arguments used in calls to these routines can be checked for validity at compile time. This

prevents mode conflicts such as matching an expression with a call-by-reference formal, and type conflicts such as matching a string with an integer formal.

Single character substrings may be included in arithmetic expressions, and the results of an arithmetic expression can easily be converted to a character.

The "exit" and "next" statements provide the programmer with complete flexibility in overriding the normal operation of any loop. The "exit" statement allows premature termination of a loop, and the "next" statement begins the next iteration of the selected loop. By default, the statements apply to the innermost loop that encloses them, but they can also refer to any enclosing loops by referring to the label of the desired loop.

Since label symbols used for loops are not associated with an absolute location in the program, they can be reused as often as is convenient for the programmer.

The "loop" statement is a purely grammatical construct that produces no object code of its own. It can be used to define blocks of statements just as with "begin" and "end" in other languages. Any "exit" or "next" statements that appear within the block will refer to the loop statement. This allows the programmer to construct loops whose iteration criteria are too complex to express well with any of the other looping statements.

The "if" statement has the capacity for any number of "elseif" clauses, which are useful in situations that would normally be coded with a "case" statement. Because arbitrary and possibly overlapping conditions can be tested in the "elseif" clauses, it is more flexible than the typical "case" statement.

The compiler ignores control characters that appear in the source program, but allows "newline" and "null" control characters to be included in string literals. This is done by entering "\n" for a newline, and "\Ø" for a null. The construct "\?" is interpreted as "?", where the question mark represents any character other than "n" or "Ø". This last form is most often used to enter the backslash itself.

DEVELOPMENT PROCESS

The first version of the compiler was written in the C programming language, and used a parser generated by YACC (Yet Another Compiler Compiler). This approach was taken to simplify the development process; the YACC/C interface is direct and easy to work with. This compiler is called ALc, and took a total of about six weeks to become operational. Initially, ALc was tested by compiling small programs. In order to execute these programs, the runtime library had to be written. The library includes the argument set-up routine, which passes arguments from one program to another, the UNIX interface routines for input/output and program execution, and several other routines. The library was written in assembly language in order to avoid incurring the overhead involved in

the C run-time environment, and because the registers that AL must preserve are casually destroyed by C programs. The run-time library took about one week to write, and was debugged along with the compiler.

When ALc began to show signs of life, another version of the same compiler was written. Because the modules in this version were written in the AL language itself rather than in C, they are called the AL modules. Their original purpose was to provide practical experience with the AL language to determine what changes or improvements were needed, and also to debug the ALc compiler. Improvements were added to ALc up to the time when the grammar was frozen about three weeks later for the final version of the compiler.

Since the AL modules were written using the ALc modules as a guide, the gestation period of the new compiler, called AL1, was much shorter than that of ALc. The AL modules are also much clearer and have better structure than their C ancestors. There are many other differences between the two compilers, suggested by experience with the language. The peephole optimizer in AL1 includes several additional transformations; the lexical analyzer includes better handling of escaped characters, file inclusion, and reserved word recognition; and the parser's action routines are more complete and more concise. The major difference between the two compilers other than the source language is AL1's arithmetic expression optimizer, which eliminates redundant sub-expressions within basic blocks of a program. The AL modules, with all of their improvements, took about four weeks to write.

To test and debug AL1, it was used to compile the AL modules, producing another version of the compiler called AL2. Errors in the AL1 compiler were discovered by trying to run AL2. When it showed the symptoms of a bug, the real problem was assumed to be a fault in AL1. Tracking down this fault involved isolating the bug in AL2 to a specific module and recompiling that module with AL1 to determine how and why it was erroneously compiled. The cause of the compilation error was then corrected in the appropriate AL module, and a new AL1 was compiled. This AL1 was used to compile all the AL modules again, producing a new AL2 with which to repeat the debugging process. Since there are around 8000 lines of code in the AL modules, this debugging proved to be a time-consuming process, requiring an average of two hours to fix the a bug, and about three weeks total.

When AL2 worked correctly, it was used to compile the AL modules again, producing AL3. Because AL1 and AL2 were compiled from the same source programs, AL2 and AL3 should have been absolutely identical. This fact provided another debugging tool, since differences in AL2 and AL3 could only be the result of errors in the AL modules themselves.

When AL3 was compiled correctly, the production version of the compiler (AL4) was made, using the "fast code" option. Because of this, AL4 is about 20% smaller than the previous versions. As a final

check of the compiler, AL4 was used to compile the AL modules again with the fast option, producing a compiler that should have been an exact duplicate of AL4. It was.

EXECUTION SPEED

Programs written in AL execute reasonably fast on the PDP-11 since they are compiled rather than interpreted. AL programs are slower than C programs, however, for two primary reasons. First, because of the dynamic nature of AL arrays, all subscript calculations include extra instructions to check the subscript value against the upper and lower limits to guard against subscript errors. This takes extra time. The primary cause for the slower execution is the flexible argument structure used in procedure calls, which necessitates calling a rather complicated run-time routine to set up arguments. In programs with many procedure or function calls, much time is spent in this routine. To compare a large AL program with a large C program, the lexical analyzer was compiled with ALc and AL4, and ALc required about 60% less time than AL4. This comparison is only a rough one due to major internal differences between the compilers, but it appears that AL programs, on the average, will take about twice as long to execute as equivalent C programs.

PROGRAMMING CONVENTIONS

Several conventions were used in writing the AL modules. Files containing procedures and functions that were declared external in other routines have names that begin with the characters "k.", followed by a short abbreviation of what kind of routines are in the module and a sequence number. For example, the lexical analyzer is contained in the files k.lx1 and k.lx2, and the symbol table routines are in the files k.st1, k.st2, and k.st3. Files containing main programs have names as described above, except that they begin with "m.", as in m.dst, the program that prints the symbol table dump. Procedure and function names begin with an abbreviation that matches the name of the file that contains them. For example, the lexical analyzer routines are lx_read, lx_get, and lx_lex. Procedures and functions contained in an "m." file may not follow this convention since they are always local to that program.

Files with names that start with "c." contain constant declarations, and the rest of the name specifies what kind of constants are in it. c.lx contains constants used by the lexical analyzer and c.quad contains the constants used in quads. These files are included in each routine needing those constants.

Files containing external definitions have names that begin with "x.", and files whose names begin with "v." contain global variable declarations.

Constants and global variables are prefixed with abbreviations denoting the routines that use them so that they can be located easily. Since AL variables can be up to 12 characters in length, variable names may be quite descriptive. Constants defined by YACC are all upper case. These include the

token keywords and some operators.

REFERENCES

Many of the techniques implemented in AL are described in the text Principles of Compiler Design by Alfred V. Aho and Jeffery D. Ullman (Addison Wesley, 1977), which was used for the course ICSS 760, Compiler Construction. The expression optimizer is based upon techinques found in the text Flow Analysis of Computer Programs by Matthew S. Hecht, which was used as the text in a seminar on data flow analysis.

2.
Lexical Analyzer

The purpose of the lexical analyzer is to convert a stream of input characters into a stream of symbols called "tokens". Each token represents an entity that may have taken several characters to express. While parsing algorithms could be written in such a way as to eliminate the need for a lexical analyzer, they would be far less efficient than an equivalent parser with an analyzer.

TOKENS

The tokens in the AL grammar represent keywords, different types of identifiers, and multiple-character operators such as "<=". Keyword tokens within the compiler are represented with upper case symbols that begin with K_. Other token symbols begin with T_, and lexical entities that consist of a single character, such as the addition and multiplication operators, are given to the parser unchanged. The grammar rules in the parser are constructed from patterns of token symbols and characters, which are matched by the tokens emitted by the lexical analyzer. For some tokens, an additional value is returned to the parser in the global variable yy_lval. For example, when the token for an integer identifier is returned yy_lval will contain a pointer to that identifier in the symbol table, and for numeric constants yy_lval will contain the constant value. It is important to note that while the values in yy_lval are never used by the parser to detect or perform reductions, they are used by the action statements when a reduction takes place.

STRUCTURE OF THE ANALYZER

The lexical analyzer actually consists of three functions. The function lx_read is called when the compiler has processed all the characters that were previously read and refills the input buffer by reading the next chunk of characters. This buffering is faster than reading one character at a time. lx_get gets characters from the input buffer one at a time and performs any character translations that are required. This is where the backslash editing of control characters is performed, and where the ASCII control characters are flagged so that they will be ignored by the analyzer itself. The source program line counter, which is used by the compiler's debug option and in compile-time error messages, is maintained here by simply counting the number of newline characters that have appeared. Since the newline character is a terminator, the routine counts each newline after the analyzer

has processed whatever token it terminated. In this way, any errors caused by that last token will be reported on the correct line.

LEXICAL ANALYZER OPERATION

The function lx_lex is the main part of the lexical analyzer. It takes the characters returned from lx_get and collects them to form symbols, reserved words, and other constructs. Each time the analyzer is called, it begins collecting characters to form a token. Often the type of entity being collected can be determined by looking at its first character, but in many cases it is necessary to examine the following character also. The global variable lx_nextc is used to store the lookahead character when it is not used so that it will be processed correctly the next time lx_lex is called. This variable is global so that its value will be preserved between calls to lx_lex.

File inclusion is supported in lx_lex and is triggered by the character "{". When this character is found, all characters up to but not including the matching "}" are collected, and the resulting string is used as a UNIX pathname in an "open" call. If the open is successful, the current context of lx_read, including the input buffer, column counter, and file descriptor variable, are saved, and the file descriptor variable is changed to point to the newly opened file. The next time lx_read is called, it will access the new file. When end of file is encountered, the original context of lx_read is restored, and input from the main file continues normally.

Currently there is room for storage of only one of these contexts, so nested includes are not allowed. Saving the context on a stack would allow processing nested includes, but would require more memory space.

Since all keywords follow the rules for forming identifiers, the analyzer recognizes keywords and identifiers with the same routine. When an identifier is recognized in the input, the symbol table is searched for it. If it is there, the type of the identifier is returned as the token, otherwise, the token for untyped identifier is returned. Since all keywords are stored in the skeleton symbol table with the correct token values for their types, when keywords are recognized the correct tokens are returned automatically.

Numeric constants are recognized in either of two bases, depending on the first digit of the constant.

Unary operators that preceed constants are returned to the parser exactly as they appeared in the program. The context in which the operator "-" appears determines whether it is unary or binary, but only the parser can determine that context, not the lexical analyzer. If the operator happened to be binary, removing it by negating the constant would result in an illegal expression. For this reason the lexical analyzer returns all operators as if they were binary. The parser then generates a quad for the unary operation, but the peephole optimizer eliminates it by negating the constant it-

self.

String literals in an AL program may be up to 500 characters long. Since a literal of this length cannot possibly fit across a page, format characters such as newline or tab that appear in a string literal are ignored by the analyzer. These strings are then stored in the string literal table by the routine st_estring, which returns a pointer to the correct table entry. This pointer is used when generating accesses to the string.

3.
Symbol Table Routines

The symbol table routines in ALc were designed with one goal in mind: quick implementation. ALc was never intended to be a production compiler, and since symbol table problems are fairly well known, "quick-and-dirty" methods were employed in writing these routines for ALc. The symbol table itself resided in main memory, and was organized as a simple list. It was searched linearly, and had a capacity of about 300 symbols. The linear search was adequate for a table in main memory, and was small in size, fast, and easy to code.

The later versions of AL used a more sophisticated data structure for the symbol table. The remainder of this section describes the symbol table as it is implemented in the production compiler.

SYMBOL TABLE STORAGE

The symbol table is stored in a disk file to minimize the memory space required by the parser. The keywords in the language are recognized by their presence in the symbol table, so the table must be initialized before compilation can begin. This is accomplished by copying an existing symbol table file into the user's directory. This existing file, called the skeleton symbol table, already contains the keywords.

When AL writes a string variable, it writes only the characters currently in the string, but when it reads a string variable, it reads enough bytes to fill it. For this reason, all symbols are padded with spaces before being written into the table, and a field containing the actual length of the symbol is also stored. With this technique, all records in the symbol table file are the same length, and the write/read discrepancy described will not occur.

ACCESS METHOD

A hashing scheme is used to find symbols quickly in the table. The hash key of a symbol is calculated by the routine st_hash, which simply converts each of the characters in the symbol to a numeric value and adds them all together. This sum is divided by the table size, which is a prime number, and the resulting remainder is used as the key. If a collision occurs, the quotient of the division is used as an increment to calculate the next key.

ROUTINES

The function st_enter is used to add symbols to the

table. It calls st_hash to calculate the key for the symbol and then looks in the table to see if a symbol already resides in that location. If not, the new symbol is entered there and the key value is returned. If that spot in the symbol table is occupied, the next key is calculated using the increment returned by st_hash. The new key is then used to read the table again, and the process is repeated until an empty spot is found, the table is found to be full, or the symbol itself is found in the table. These last two possibilities both cause the compiler to report a fatal error and stop since, when the symbol table is full, compilation cannot continue, and an attempt to re-enter a symbol is a semantic impossibility.

Symbols are retrieved from the table by the routine st_lookup, which follows a similar procedure. Once the initial key is calculated, the chain that begins in that spot is searched. If the symbol is found, st_lookup returns the key value. If the end of the chain is found, a value of -1 is returned, indicating that the symbol was not found.

Since the global symbols must be saved for all routines within a single source file, it is not possible to simply create another copy of the skeleton symbol table before compiling each routine. Therefore, as symbols are entered into the table, their scope (local or global) is saved in a field in the entry. This allows the local symbols for one procedure to be removed from the table before compiling the next procedure in the file. The routine st_flush performs this removal process, and also resets several pointers and indicators to the values they had prior to compiling the first routine.

Removal of symbols from a hashed table can cause problems if the hash "chains" are broken in the middle because this causes the remainder of each chain to be lost. In AL this situation cannot occur. Since all global identifiers are declared at the beginning of a file, no global identifiers will ever be added to the table after local identifiers. Thus, the removal of local symbols simply truncates chains, so that the beginning portions of the chains which contain the global identifiers will be intact.

4.
The YACC Parser

YACC GRAMMARS

One of the many software tools provided with UNIX is YACC (Yet Another Compiler Compiler), written by S. C. Johnson and described in the UNIX documentation. YACC converts a context-free grammar into a set of tables for a simple automaton which executes an LR(1) parsing algorithm. If the grammar is ambiguous, precedence rules may be specified to resolve the ambiguities. YACC was used in ALc in order to get a working parser quickly, and was translated for use in AL4 because of the intimate interrelationships between the grammar rules and their action statements.

As an example of a context-free grammar accepted by YACC, a portion of the AL grammar is shown in figure 4-1. These grammar rules recognize simple expressions (simple_exp) used in declaring constants. In the AL grammar, precedence rules (not shown here) are specified for these grammar rules so that if an ambiguity arises, the operations are performed in the proper order. The lexical analyzer emits tokens for the operators (the characters between apostrophies) and for numeric constants (T_N_CONST). If the simple expression "5*(2+3)" appeared in a program, the lexical analyzer would emit the following tokens:

```
simple_exp:
rule 1:          simple_exp '+' simple_exp

rule 2:          | simple_exp '-' simple_exp

rule 3:          | simple_exp '*' simple_exp

rule 4:          | simple_exp '/' simple_exp

rule 5:          | '-' simple_exp

rule 6:          | '(' simple_exp ')'

rule 7:          | T_N_CONST
```

Figure 4-1
Grammar Rules for Simple Expressions

```
T_N_CONST
    '*'
    '('
T_N_CONST
    '+'
T_N_CONST
    ')'
```

This would be parsed with the substitutions shown in figure 4-2.

Each grammar rule may be accompanied by one or more action statements. These action statements will be executed when that grammar rule is used to make a reduction. The action statements for a grammar rule may specify that the new nonterminal symbol (the left side of the grammar rule) be given an integer value. This value may be used by the action statements of other grammar rules in which the nonterminal appears. The symbol $$ refers to the present value of a grammar rule, which is the value that the nonterminal symbol will have when it next appears in other grammar rules. The symbol $n refers to the value of the n'th component of a grammar rule. The grammar rules in the example above are shown in figure 4-3 with their action statements enclosed in braces.

The variable "yy_lval" is set by the lexical analyzer to the value of the numeric constant. The sample parse shown above is repeated in figure 4-4 with the current value of each component in the expression specified by placing it in brackets after that component. In this example, the value of the nonterminal symbol "simple_exp" is always the numeric value of the portion of the expression it represents. Other nonterminals in the AL grammar may represent things other than expression values, such as locations in the program or pointers to data stored elsewhere.

AL GRAMMAR DEVELOPMENT

The AL grammar was created over a period of about two weeks. It was built part by part, with a new construct or statement being added only after those already in the grammar were represented correctly. The

```
tokens:     T_N_CONST '*' '(' T_N_CONST '+' T_N_CONST ')'

rule 7:     simple_exp '*' '(' T_N_CONST '+' T_N_CONST ')'

rule 7:     simple_exp '*' '(' simple_exp '+' T_N_CONST ')'

rule 7:     simple_exp '*' '(' simple_exp '+' simple_exp ')'

rule 1:     simple_exp '*' '(' simple_exp ')'

rule 6:     simple_exp '*' simple_exp

rule 3:     simple_exp
```

Figure 4-2
Sample Parse

```
simple_exp:  simple_exp '+' simple_exp          {$$ = $1 + $3}
          |  simple_exp '-' simple_exp          {$$ = $1 - $3}
          |  simple_exp '*' simple_exp          {$$ = $1 * $3}
          |  simple_exp '/' simple_exp          {$$ = $1 / $3}
          |  '-' simple_exp                     {$$ = - $2}
          |  '(' simple_exp ')'                 {$$ = $2}
          |  T_N_CONST                          {$$ = yy_lval}
```

Figure 4-3
Action Statements

```
T_N_CONST [5] '*' '(' T_N_CONST [2] '+' T_N_CONST [3] ')'
simple_exp [5] '*' '(' T_N_CONST [2] '+' T_N_CONST [3] ')'
simple_exp [5] '*' '(' simple_exp [2] '+' T_N_CONST [3] ')'
simple_exp [5] '*' '(' simple_exp [2] '+' simple_exp [3] ')'
simple_exp [5] '*' '(' simple_exp [5] ')'
simple_exp [5] '*' simple_exp [5]
simple_exp [25]
```

Figure 4-4
Parse with Values

original intent was for a language with no separators between statements because separators are purely syntactic and do nothing to improve the representation of the algorithm. As more and more statements were added to the grammar, though, the YACC processor rapidly approached its limit of "lookahead sets". A lookahead set is a collection of tokens that may legally appear next during a parse. They are used to resolve potential conflicts between grammar rules that have identical beginnings by looking farther down the input to find the token that determines the difference. Even though each statement in the language begins with its own keyword, the number of these lookahead sets greatly increased when the grammar construct "statement list" was added, and finally exceeded its limit when the "case" statement was added. The "case" statement was discarded, but the need for some kind of statement terminator became obvious, and the semi-colon was reluctantly ap-

pended to the rule for every statement in the grammar. This reduced the number of lookahead sets by 25%, and allowed YACC to process the rest of the grammar. When the grammar rules were correct, the action statements were added to each grammar rule to produce the appropriate intermediate code.

YACC recognizes action statements written in C or RATFOR. This posed no problem in ALc, which was written in C. Since YACC does not recognize AL statements, the AL parser was produced indirectly by YACC using the following process. The AL grammar was rewritten so that each grammar rule had only one action statement, a comment that identified the rule. YACC processed this grammar and produced the parsing tables and action program for it. Then, the AL action statements were manually inserted into the empty action program using the comments to locate each rule. This approach had the advantage of producing action routines that have better structure than those in ALc, but had the disadvantage of absolutely freezing the grammar. Because the action statements were inserted into the YACC output by hand, any change of the grammar would mean a repetition of that process. For this reason, the grammar for AL cannot be changed without a major overhaul of the parser. This is the reason that so much time was spent improving the grammar for ALc, as that was much easier to experiment with.

The object code for the AL action routines proved to be so large compared to the C version that the source program was

```
/*
 * Sample AL program #1: print prime numbers.
 */
label   test;           /* labels loop that gets next prime to test */

const   max_prime=100;  /* largest value to test */

int     test_prime,     /* number to test for primeness */
        divisor;        /* divisor to try next */

test: for test_prime = 2 to max_prime do        /* check if prime */

        for divisor = 2 to test_prime-1 do      /* try all divisors */
                if test_prime % divisor == 0 then       /* no prime */
                        next test;
                endif;
        endfor;

        /* if it gets here, must be prime */
        print(1,"%d\n",test_prime);
endfor;
end
```

Figure 4-5
Sample AL Source Program

split into three separate modules. The parser calls one of the three depending on which one contains the actions for the grammar rule in use. There are two reasons for the increase in size. First, even though the source programs for AL's actions are actually smaller than those for ALc, the compiled programs are larger due to the additional code needed to check subscripts. Also, the AL version of the parsing routine contains many more array references than the original C version, which utilizes pointer variables heavily.

INTERMEDIATE CODE

Parsers commonly generate some form of intermediate code; often it is three-address code. The output of the AL parser is a form of three-address code called quadruples, or quads, because each one contains four fields. The "r" field specifies where the result of the quad is to be placed, the "op" field indicates what operation is to be performed, and the "a" field and "b" field specify the operands to be manipulated.

AL quads are actually composed of nine integers. One field, called q_lineno, contains the source program line number from which the quad was generated, which is used with the debug option to show the user exactly where in the source program run-time errors occurred. The q_count field is always zero when the quads are generated. When used by the optimizers, it will contain a count of how many transfer quads point to this one. The "r", "a", and "b" fields are each represented by two integers, one indicating the type of the entry, and one for its value. The types are called q_rtype, q_atype and q_btype, and the value fields are q_rvalue, q_avalue and q_bvalue.

The type null (indicated in quads by "--") signifies that there is no argument in a field. Its value is ignored. The type "q_var" indicates the variable whose symbol table pointer is in the value field. "q_nconst" signifies a numeric constant; the value field contains the constant itself. "q_sconst" is used for string constants, and its value field contains a pointer to the string constant table. "q_temp" specifies a temporary location; the value field contains the temp number. "q_atemp" is used for temporaries that contain address, such as those produced by array references, and its value field also contains the temp number. Because they contain addresses, these temps are always accessed using indirect addressing. "q_quadno" indicates that the value field contains the number of another quad in the program. These appear only in the "r" field of transfer quads. Finally, "q_lib" indicates that the field contains the name of a run-time support program; its value is the program number.

Figure 4-5 shows a simple AL program to compute prime numbers, and figure 4-6 lists the quads produced by the parser for this program. Figure 4-7 is a list of all types of quads the parser produces, along with their meanings.

UNIVERSAL NODES

Many nonterminals that result from grammar reductions represent several values. For

```
Quad (stmt): count, r, a, op, b
   0 (4):  0, -- , nconst 0, lcall, lib setup
   1 (11):  0, var test_prime, -- , assgn, nconst 2
   2 (11):  0, quad #4, -- , goto, --
   3 (11):  0, var test_prime, var test_prime, plus, nconst 1
   4 (11):  0, quad #22, var test_prime, ifgt, nconst 100
   5 (13):  0, temp #1, var test_prime, minus, nconst 1
   6 (13):  0, -- , temp #1, temp, --
   7 (13):  0, var divisor, -- , assgn, nconst 2
   8 (13):  0, quad #10, -- , goto, --
   9 (13):  0, var divisor, var divisor, plus, nconst 1
  10 (13):  0, quad #17, var divisor, ifgt, temp #1
  11 (14):  0, temp #2, var test_prime, rem, var divisor
  12 (14):  0, quad #14, temp #2, ifeq, nconst 0
  13 (14):  0, quad #15, -- , goto, --
  14 (15):  0, quad #3, -- , goto, --
  15 (17):  0, quad #9, -- , goto, --
  16 (17):  0, -- , -- , temp, temp #1
  17 (20):  0, -- , nconst 3, call, var print
  18 (20):  0, nconst 1, -- , param, --
  19 (20):  0, sconst #1, -- , param, --
  20 (20):  0, var test_prime, -- , param, --
  21 (21):  0, quad #3, -- , goto, --
  22 (22):  0, -- , -- , return, nconst 0
```

Figure 4-6
Quadruples Produced by the AL Parser

example, conditions have two values, a true exit and a false exit. Universal nodes are used for nonterminals that have several values. The universal nodes are actually three arrays, nd_1, nd_2, and nd_link. The set of the elements in each array with the same subscript comprises one node. In general, numeric values are stored in nd_1 and nd_2. When a linked list of nodes is required, nd_link contains the number of the next node in the list, otherwise it may also contain a numeric value. If a nonterminal requires two or more values, they will be stored in a node and the nonterminal will be assigned the number of that node. In grammar rules where the nonterminal appears, the action statements will extract the desired values from the node.

The nodes are initially stored in a linked list called the free list. The routine nd_get removes a node from the free list and returns its number. The routine nd_kill returns a node to the free list. There are 300 nodes in the AL compiler; if the supply is exhausted by an overly complex program, a fatal error is issued within the compiler, which then stops.

The universal nodes in ALc consisted of only two elements, a data value and a link. This proved adequate for the backpatching lists (see below), but some grammar rules required saving three or four values. This was possible with the smaller

OPERATION	MEANING	REMARKS
assgn	r := b	simple assignment
clr	r := Ø	
plus	r := a+b	
inc	r := r+1	increment
minus	r := a-b	
dec	r := r-1	decrement
mult	r := a*b	low-order 16 bits of product
div	r := a/b	quotient of integer division
and	r := a&b	bitwise logical AND
or	r := a|b	bitwise logical OR
rem	r := a%b	remainder of integer division
uminus	r := -b	unary minus
not	r := ~b	one's complement
goto	transfer to quad r	
ifeq	if a==b goto r	
iflt	if a<b goto r	
ifgt	if a>b goto r	
ifle	if a<=b goto r	
ifge	if a>=b goto r	
ifne	if a!=b goto r	
array	r := addr of a(b)	creates address temp
param		parameter in procedure/function call
call		call external routine
lcall		call library routine
return		return b to calling routine
stop		halt program, print value b
temp		birth and death of perm temps

Figure 4-7
AL Quads

nodes by constructing linked lists, but they were clumsy to traverse, and the subscripting in the action statements became quite complex. AL's universal nodes are not completely filled when they are used in backpatching lists, but this inefficiency is more than offset by the simpler action statements that resulted for the complex grammar rules.

BACKPATCHING

The AL parser is a one-pass process. Because of this, some technique was needed to allow quads that had already been generated to be modified. For example, the "then" clause of an "if" statement must end with an instruction to jump to the first quad after the "else" clause. However, when the "then" clause is being parsed, it is not yet known where that quad will be. This situation is handled by

generating an empty "goto" quad and saving its number. After the "else" clause is parsed and the number of the first quad in the next statement is known, that value is inserted into the empty "goto" quad. This technique is described in Aho and Ullman and is called backpatching.

Often, many quads are generated which should all point to the same target. This condition is handled by creating a linked list of the quad numbers. When the number of the target quad is determined, the list is scanned to find each quad into which the number must be inserted. This operation is performed by the routine nd_bakpatch. New quad numbers are added to an existing linked list with the routine nd_merge, which merges two lists into one, and new lists are created with nd_makelist.

In the AL grammar, the nonterminal symbol for an executable statement has a value of -1 if there are no quads within that statement that must be backpatched. If there are such quads, the nonterminal's value will be the number of the first node in the linked list that contains those quad numbers. All of the quads in a "statement list" are backpatched to point to the statement that follows it. The list of quads to be backpatched in the new "statement list" is simply the list for its last statement.

The parser makes no effort to prevent transfer quads from referring to other transfer quads since the peephole optimizer can remove these occurences much more easily than the parser could prevent them.

ARGUMENTS

When a procedure or function is called in the source program, a calling sequence is generated in the object program that consists of a "call" or "lcall" quad followed by a "param" quad for each actual argument. One field in the call quad contains the number of parameters that will follow it, something that is not known when the quad is generated. This field could have been filled in later by a process similar to backpatching, except that a new routine would have been required to do it. Instead, as arguments are parsed, they are saved in a table and the quads for the calling sequence are generated after it has been completely parsed. This eliminates the need for another backpatching routine and also makes the grammar for procedure and function calls clearer.

In ALc, this table was a fixed size matrix which could hold up to eight arguments. This limited procedure and function calls to a maximum of eight arguments. It also limited string assignment statements to not more than eight substrings because string assignment is performed by calling a library procedure. This technique worked well until the problem of nested function calls was uncovered by one of the test programs. To process f(g(a)), for instance, parsing of the first call must be suspended so that the inner call can be parsed. To handle this, the table was lengthened, and a stack was created to simulate a recursive procedure. The result of the change was not very efficient: only three levels of nesting were allowed, and errors in the

inner arguments caused errors in the outer lists also because the stacks pointers were scrambled when an error occurred.

AL4 uses a table that holds variable length entries. Since the parser's action routines are selected with a case structure, the recursion was again simulated with a stack. In AL4, however, there is no "eight argument" restriction, and nesting is allowed until the table is full. This allows, for example, a routine with 16 integer arguments followed by a function call which also has 16 integer arguments, or 11 nested functions, each having three integer arguments.

Argument type checking in AL4 is performed in the procedure that generates the calling sequence rather than in the action routine that recognizes argument lists. This eliminates the stack problems that argument errors in ALc caused, so redundant error messages are no longer printed, and with nested lists, errors in the inner lists do not cause errors in the enclosing lists.

LOOP CONTROL STATEMENTS

In order to implement the "next" and "exit" statements, the parser maintains a stack to save information about each loop. Each time a new loop is encountered in the source program, a new frame is pushed on the stack. The frame contains the symbol table pointer of the loop's label (if there is one), and pointers to two empty backpatching lists, one each for "exit" and "next" statements. When "exit" or "next" statements are found in the loop, the numbers of the quads generated are added to the appropriate backpatch list. Loop control statements that refer to outer loops are added to the correct lists by examining previous frames on the stack according to the specified loop count or label. When the end of a loop is found, the list of "next" quads is backpatched to the iteration portion of the loop. The "exit" quads should refer to whatever statement follows the loop, so the "exit" quad list is merged with the list of quads that already branch out of the statement. Then the frame corresponding to that loop is popped off the stack.

5.
Peephole Optimizer

The peephole optimizer reads the quads produced by the parser and performs several transformations on them to obtain better quads from which to generate code. These transformations fall into two broad categories: elimination of useless arithmetic operations and flow modification.

This optimizer was included in AL for two reasons. First, the flexibility with which constants may be declared and used in AL programs leads to many situations where calculations are performed that involve only constants. These calculations should be performed at compile time rather than at execution time. To keep the parser simple though, it does not check the arithmetic quads it produces to see if they have constants in them. The term "useless" refers to operations that can be calculated at compile time. Eliminating useless quads produces a program that will execute faster.

Second, the ability of the peephole optimizer to transform clumsy sequences of quads into straightforward, compact sequences of quads simplifies the design of the parser. The parser often generates quads before it actually knows the context in which they appear. To generate optimum sequences of quads for an "if" statement, for example, the parser would have to generate all of the quads for the expressions and tests, and then back up and rearrange them according to how the statement actually transfers control. Also, the parser cannot determine whether or not to actually compile any "conditional compilation" statements without evaluating the constant expressions itself. The ability to perform this evaluation would have made the grammar more complex. This optimizer eliminates the need for the parser to perform any analysis on the quads after it generates them.

USELESS ARITHMETIC ELIMINATION

In its first pass, the optimizer eliminates all arithmetic operations that can be performed at compile time, such as adding two constants together or multiplying a value by one. These operations can be eliminated in one of three ways. Quads containing redundant operations are simply eliminated. If a calculable operation produces a result that is used later, the result is inserted as a constant into the quad in which it is used. Finally, some quads are simply changed to equivalent quads from which better object code is generated.

Figure 5-1 lists all of the transformations that are per-

ORIGINAL QUAD	ACTION
c + d	substitute c + d
c - d	substitute c - d
c * d	substitute c * d
c / d	substitute c / d
c % d	substitute c % d
c & d	substitute c & d
c ¦ d	substitute c ¦ d
x - x	substitute 0
x % x	substitute 0
x / x	substitute 1
x & x	substitute x
x ¦ x	substitute x
c + x	Rearrange quad reversing a and b fields.
c * x	Rearrange quad reversing a and b fields.
c & x	Rearrange quad reversing a and b fields.
c ¦ x	Rearrange quad reversing a and b fields.
0 - x	change quad to "uminus x"
0 / x	substitute 0
0 % x	substitute 0
uminus c	substitute -c
not c	substitute ~c
x ¦ 0177777	substitute 0177777
x ¦ 0	substitute x
x & 0177777	substitute x
x ¦ 0	substitute 0
x + 1	change quad to "inc x"
x - 1	change quad to "dec x"
x + 0	substitute x
x - 0	substitute x
x * 0	substitute 0
x * 1	substitute x
x * -1	change quad to "uminus x"
x / 1	substitute x
x / -1	change quad to "uminus x"
x % 1	substitute 0
x % -1	substitute 0
x assgn temp	substitute x into quad that produced the temp
x assgn 0	change quad to "clr x"
x assgn x	deleted

Figure 5-1
Useless Arithmetic Transformations

formed. In the examples, "x" indicates a variable, and "c" and "d" indicate any integer constants. If "x" or "c" appears twice, they refer to the same variable or constant each time. An action of "deleted" indicates that the quad containing the operation is redundant and simply eliminated, and the action "substitute n" means that the quad is deleted after the

operation it specifies is calculated and the result shown by "n" is inserted into the quad that uses it. Other changes are described in the table.

Conditional quads that contain one constant and one variable are rearranged if necessary so that the constant is in the b field. This simplifies the logic required in the expression optimizer to analyze these quads.

Conditional quads that contain two constants are evaluated by the optimizer. If the relation is true, the conditional is changed to a "goto" quad since the transfer would always have been taken anyway. If the relation is false, the conditional quad is simply deleted. This process of changing or deleting conditional quads often has a dramatic effect because the flow of control to at least one other quad is altered, perhaps rendering some quads unreachable.

All of the above transformations are performed in pass one of the peephole optimizer. This pass is performed once since none of the transformations produces anything in previous quads that must be processed again.

FLOW MODIFICATION

Flow modification is performed in pass 2 of the optimizer. Pass 2 is repeated as many

```
/*
 * Sample AL program #2: demonstrates peephole optimization
 */
const  type1=1,type2=0, /* select type 1, inhibit type 2 */
       c1=10,c2=20,c3=30;
int    a,b;

/*
 * Conditional compilation
 */
if type1==1 then        /* test whether to compile next statement */
       a=1;
endif;

if type2==1 then        /* test whether to compile next statement */
       a=2;
endif;

/*
 * Constant optimization
 */
a=a+1;                  /* changes to an "inc" */
a=b&b;                  /* changes to a=b */
a=b/(c3/10-c2/10);      /* changes to a=b */
end
```

Figure 5-2
Sample AL Source Program

times as needed to guarantee that everything that can be optimized has been. This is accomplished by setting a flag each time a transformation is made; the pass is simply repeated until the flag is not set.

All transformations described below except the first one are performed only on transfer quads, that is, conditional quads and goto's. A quad is said to be labelled if it is the target of at least one transfer quad. The compiler produces some quads that do not result in the generation of any object code. These are called dummy quads, and there is only one that is currently generated, its purpose is to inform the object code generator of the birth and death of temporary locations whose values must be preserved. Dummy quads are not modified or moved by any of the transformations described below.

Unreachable quads: If the first quad following "goto", "stop", or "return" quad is not labelled, it can never be accessed so the optimizer deletes it. Since the transformation is repeated, all quads up to the next labelled quad will be deleted. This transformation implements the conditional compilation feature.

Useless transfers: If a transfer points to the next quad, it is deleted. This is a useless operation for conditionals as well as goto's.

Changed transfers: If a transfer points to a quad that has been deleted, it is changed to point to the next existing quad. This is not really an optimization, but is required so that the labels generated in the assembly language output will be consistent with each other.

If the transfer is ever changed to point to itself during this process, an infinite loop must have existed in the source program, and the quad is deleted. This would be an appropriate place to print some kind of warning message, though currently no indication is given of the error.

Transfer chains: If a transfer points to a goto quad, it is changed to point to the target of that goto quad rather than the goto itself. This process is repeated on the same quad until its target is no longer a goto. If during this process the quad ever points to itself, it is deleted as described above.

Relational tests: The parser produces two quads for every relational test in the source program, an "if" quad of the correct type which points to the target desired if the relation is true, and a "goto" that points to the desired target if the relation is false. Very often the goto will be unnecessary as the conditional just before it points to the quad just after it.

If a conditional quad is followed by a goto quad, and the conditional points to the quad following the goto, then the relational test in the conditional is complemented, the target of the conditional is changed to be the target of the goto that follows, and the goto itself is deleted.

Duplicate quads: If two sequences of quads are identical and one sequence ends with a

```
Before optimization:
Quad (stmt): count, r, a, op, b
   Ø (4):  Ø, -- , nconst Ø, lcall, lib setup
   1 (1Ø):  Ø, quad #3, nconst 1, ifeq, nconst 1
   2 (1Ø):  Ø, quad #4, -- , goto, --
   3 (11):  Ø, var a, -- , assgn, nconst 1
   4 (13):  Ø, quad #6, nconst Ø, ifeq, nconst 1
   5 (13):  Ø, quad #7, -- , goto, --
   6 (14):  Ø, var a, -- , assgn, nconst 2
   7 (2Ø):  Ø, temp #1, var a, plus, nconst 1
   8 (2Ø):  Ø, var a, -- , assgn, temp #1
   9 (21):  Ø, temp #2, var b, and, var b
  1Ø (21):  Ø, var a, -- , assgn, temp #2
  11 (22):  Ø, temp #3, nconst 3Ø, div, nconst 1Ø
  12 (22):  Ø, temp #4, nconst 2Ø, div, nconst 1Ø
  13 (22):  Ø, temp #5, temp #3, minus, temp #4
  14 (22):  Ø, temp #6, var b, div, temp #5
  15 (22):  Ø, var a, -- , assgn, temp #6
  16 (23):  Ø, -- , -- , return, nconst Ø


After optimization:
Quad (stmt): count, r, a, op, b
   Ø (4):  Ø, -- , nconst Ø, lcall, lib setup

   3 (11):  Ø, var a, -- , assgn, nconst 1

   7 (2Ø):  Ø, var a, -- , inc, var a

  1Ø (21):  Ø, var a, -- , assgn, var b

  15 (22):  Ø, var a, -- , assgn, var b
  16 (23):  Ø, -- , -- , return, nconst Ø
```

Figure 5-3
Sample Quads Before and After Optimization

goto that points to the end of the other sequence, then the first sequence will be removed and the goto will be changed so that it points to the beginning of the second sequence. The program logic will be unchanged, but the program will be smaller because the redundant sequence of quads has been removed. This is illustrated in figure 5-4.

If any of the quads preceding the goto are labelled, the comparison stops with that quad. The labelled quad is not deleted since there is no way to locate all the transfer quads that point to it without performing an exhaustive search.

There is one exceptional case that must be tested: if the quads that did not match happen to be parameters in the calling sequence of a procedure or function, then the goto is changed to point to the first quad after that sequence. This is because the quads for parameters to a

```
Before:
10: var a, -- , assgn, nconst 1     21: var a, -- , assgn, nconst 1
11: var b, -- , assgn, var a        22: var b, -- , assgn, var a
12: (can be any quad)               23: quad #12, -- , goto, --

After:
10: var a, -- , assgn, nconst 1     21: (quad deleted)
11: var b, -- , assgn, var a        22: (quad deleted)
12: (unchanged)                     23: quad #10, -- , goto, --
```

Figure 5-4
Duplicate Quad Elimination

procedure call do not generate executable instructions, so a branch into a sequence of parameters would cause the program to fail. (Parameter quads are fully explained in chapter 7.)

Figure 5-2 shows an AL source program, and figure 5-3 shows the quads produced by the parser and output from the peephole optimizer. The blank lines in the second list of quads indicate places where quads have been removed. This program makes use of the conditional compilation feature. The constants "type1" and "type2" are used to select whether or not different sections of code are to be compiled. The parser produces quads for the entire program, but the peephole optimizer removes the unwanted statements.

6.
Arithmetic Expression Optimizer

The arithmetic expression optimizer evaluates the quads emitted by the peephole optimizer and removes sequences of quads that evaluate redundant expressions. Its usefullness is limited by the fact that no global data flow analysis is done. It was intended primarily to eliminate redundant subscript calculations, and considerable gains can be obtained when optimizing code that contains many array references.

DATA FLOW ANALYSIS

A generalized optimizer works by first analyzing the flow of data through the program, that is, where each datum is born, used, and dies. It uses this global information to detect and eliminate redundant expressions in different parts of the program. A fancy optimizer might also insert infrequently called procedures inline, or move invariant code out of the bodies of loops.

The AL optimizer does not perform any global data flow analysis. Time limitations forced this restriction, which reduced the complexity of the optimizer by about an order of magnitude. Thus, if an expression is redundant because it is equivalent to a previous expression, it will not be eliminated unless it is close enough to the first expression to be analyzed along with it. This critical distance is measured in "basic blocks" whose length depend on the type of quads in the program.

EQUIVALENCE RELATION

The primary instrument in optimization is an equivalence relation (ER), which is implemented as a collection of arrays. This relation is actually a set whose elements are the expressions, constants, and variables that have been discovered so far in the analysis of the program. The set is partitioned into several equivalence classes (EC), and all the entries in any one class are guaranteed to have the same value at the current location in the program. Furthermore, there is exactly one equivalence class for each known value, so two entries that are known to have the same value will always be in the same equivalence class.

BASIC BLOCKS

The optimizer divides the quads in the program into basic blocks. A basic block is a series of quads of which only the first is labelled. This means that the only way to enter a basic block at its beginning. There may be any number of quads

in a basic block that branch out of it, but these do not change the analysis of the block.

Each basic block is optimized independently of all others, that is, any data flow information that was known at the end of one block is always discarded before processing the next block. This is done because the flow analysis depends on the sequential execution of the quads. Each basic block begins with a quad that is the target of a transfer, so it is possible that any block might be executed out of sequential order. Any optimizations done on the block that were based on the assumption that the preceeding block had just executed might be invalid and could cause erroneous results.

OPTIMIZATION

Optimization of a basic block begins after the ER is emptied of any previous contents. Then, the quads in the block are examined in the order in which they appear.

When a quad is found that calls a subprogram, all ER entries that refer to global variables are removed. This is necessary because the called procedure might modify one or more of them. All call-by-reference parameters to the procedure are removed since their values may also be changed by the procedure.

When an "ifeq" quad is found and the two values being compared are in the same EC, then the quad is changed to a "goto" since the two values must be equal. After this change is made, the rest of the basic block is eliminated since the quads in it cannot be reached. This modification is simple because the range of quads to be removed is clearly defined by the extent of the basic block.

More optimizations on conditional quads could be performed here by examining the EC's to which each field belongs to see if they contain constants. If so, the constants could be compared using the relation indicated in the quad. Based upon the actual result of the comparison, the quad could be eliminated or changed to a "goto".

All other non-arithmetic quads are ignored by the optimizer since none can affect any existing data values.

Assignment quads are easy to handle. If the source data and the result location are in the same EC, then they already have the same value, so the quad is deleted. Otherwise, the ER is updated by removing the result location from whatever EC it occupied and placing it in the same class with the source data. The quad is not changed in this case.

Arithmetic quads are handled as follows. First, if the "a" or "b" field of the quad contains a variable or constant that is not in any EC, a new EC is created and the value is added to it. This is done by copying the field from the quad into the array assigned to the new EC. Then, a representation of the operation the quad performs is constructed using its operation code and the EC numbers for the "a" and "b" fields. The ER is searched for this expression. If it is not

found in any EC, the quad must calculate a value that does not already exist. A new EC is created, and the expression and the result field of the quad are both added to it. The quad itself is not changed. If the expression is found in the ER, the current quad is redundant because the result that it computes already exists. The quad is replaced with an assignment quad that copies a datum from the EC in which the expression was found into the result field given in the original quad. This is needed to assure that the result field is given the value the original quad specified. This assignment quad is then analyzed as described above to see if it is redundant.

SAMPLE OPTIMIZATION

Figure 6-1 illustrates an AL program that contains redundant arithmetic expressions and redundant subscript calculations. For ease of reference, each program statement has been labelled by a comment containing its line number. The quads contain a field (labelled "stmt") indicating the number of the line they were generated from.

Figure 6-2 illustrates how the optimizer operates on arithmetic expressions. The quads in the example are those produced by the peephole optimizer. The equivalence relation is shown as it exists after each of the quads is processed using the following notation: The entire relation is enclosed in braces, and each class in the relation is introduced by the class number and a colon. Members of a class are separated by commas, and the classes are separated by semicolons. Variables in an equivalence class are listed by name, temporaries are shown as "Tn", and address temporaries are shown as "An", where in each

```
/*
 * Sample AL program #3: demonstrates redundant expression removal
 */

int    a,b,c,d,e;
array  x,y;

/* line 8 */    a = b+c;
/* line 9 */    d = b;
/* line 10 */   e = c;
/* line 11 */   a = e+d;
/* line 12 */   d = e*a;
/* line 13 */   b = (b+c)*c;

/* line 15 */   x(b+c) = b;
/* line 16 */   b = x(b+c);
/* line 17 */   c = x(b+c);
/* line 18 */   a = x(b+c);
end
```

Figure 6-1
Sample AL Program

```
Quad (stmt): count, r, a, op, b
   Ø (5):  Ø, -- , nconst Ø, lcall, lib setup
       {}

   1 (8):  Ø, var a, var b, plus, var c
       {1:b; 2:c; 3:1+2,a}

   3 (9):  Ø, var d, -- , assgn, var b
       {1:b,d; 2:c; 3:1+2,a}

   4 (10):  Ø, var e, -- , assgn, var c
       {1:b,d; 2:c,e; 3:1+2,a}

   5 (11):  Ø, var a, var e, plus, var d
       {1:b,d; 2:c,e; 3:1+2,a}

   7 (12):  Ø, var d, var e, mult, var a
       {1:b; 2:c,e; 3:1+2,a; 4:2*3,d}

   9 (13):  Ø, temp #4, var b, plus, var c
       {1:b; 2:c,e; 3:1+2,a,T4; 4:2*3,d}

  1Ø (13):  Ø, var b, temp #4, mult, var c
       {2:c,e; 3:a; 4:2*3,d,b}
```

Figure 6-2
Arithmetic Optimization

case n is the temp number. Expressions in a class are shown as an EC number, an operation, and another EC number. Thus the expression "1+2" represents the sum of the values represented by EC #1 and EC #2.

Before processing any quads in the example, the ER is emptied. Quad number 1 causes a new EC to be added for each of the variables b and c, and another EC for the expression b+c. Since the variable a receives the value of b+c, it is also placed into this EC. Note that the expression b+c is shown as 1+2, which are the EC numbers in which b and c currently reside. Representing expressions in terms of their EC numbers increases the number of redundant expressions that can be detected.

After the assignments in quads 3 and 4, the new variables d and e will be equal to b and c respectively, so they are simply added to the appropriate EC's. This reflects the fact that the pairs of variables must be equal after the assignments. The expression in quad 5, e+d, is first changed to 2+1 by substituting the EC numbers of e and d. By convention, this is rearranged to 1+2, because addition is commutative. This convention applies to all operators that are commutative, and increases the chances of finding equivalent expressions. The expression 1+2 is already in the ER, so it must have been computed previously. The calculation need not be done again, so the

quad is changed to an assignment quad that copies any value from EC #3 into the variable a. This assignment quad is then analyzed and found to be redundant because a is already in EC #3, that is, the variable a already had the value that the original quad would have computed. The assignment quad is also redundant, so the analyzer simply deletes quad 5.

Quad 7 adds a new class to the ER containing the expression 2*3 and the variable d. The operation in quad 9, which is translated to 1+2, is redundant. Since the result is a temporary, the temp itself is added to the EC that contains the expression. Future references to this temporary will be translated to something else in that class, in this case the variable a.

The multiplication in quad 10 is actually 2*3, which is redundant, but since the result field contains a variable which is not in the same class as the expression, the quad cannot simply be deleted. Instead, an assignment quad is generated that copies the current value of d (the expression 2*3) into b. The ER is updated by removing b from EC #1 and adding it to EC #4. Since EC #1 is empty, it is completely removed from the ER, and all references to EC #1 in any expressions are also removed. This is why the expression 1+2 in EC #3 is deleted.

Optimization of array references is performed in much the same way as expressions, except that for an address temporary, the ER keeps track of the address of the array element as well as the contents of that element. A redundant assignment involving an array element will be caught in the usual way, and if two different array references refer to the same location, the second subscript calculation will be detected as redundant and will be eliminated. The array address is represented in the ER by the temporary number that contains it, and the contents of the array element are represented by the address temporary with the same number.

The quads in figure 6-3 actually followed the quads in the previous example in the same basic block when the sample program was compiled. Because of this, the ER in this example should contain all of the values it contained at the end of the previous one. They are irrelavant for this example, so the ER is shown as if it began empty in order to simplify the example.

Quad 12 calculates the expression in the subscript, and quad 13 produces the address of the array element. While processing quad 13, EC #4 is added to the ER to hold the variable x. The quad produces an address which is represented by the expression "4 array 3". This address is stored in temporary #7, so EC #5 contains T7 along with the expression. Quad 14 assigns the value of the variable b to the array element, so A7 is put into EC #1 along with b. The EC that contains T7 represents the address of the array element, and the EC that contains A7 represents the value of that element and of b.

In quad 15, the expression b+c is found to be redundant, so T8 is added to EC #3 and the quad is eliminated. The result of quad 16 is the expression "4 array 3", which is redundant, so

```
12 (15):  Ø, temp #6, var b, plus, var c
     {1:b; 2:c; 3:1+2,T6}

13 (15):  Ø, temp #7, var x, array, temp #6
     {1:b; 2:c; 3:1+2,T6; 4:x; 5:4 array 3,T7}

14 (15):  Ø, atemp #7, -- , assgn, var b
     {1:b,A7; 2:c; 3:1+2,T6; 4:x; 5:4 array 3,T7}

15 (16):  Ø, temp #8, var b, plus, var c
     {1:b,A7; 2:c; 3:1+2,T6,T8; 4:x; 5:4 array 3,T7}

16 (16):  Ø, temp #9, var x, array, temp #8
     {1:b,A7; 2:c; 3:1+2,T6,T8; 4:x; 5:4 array 3,T7,T9}

17 (16):  Ø, var b, -- , assgn, atemp #9
     {1:b,A7,A9; 2:c; 3:1+2,T6,T8; 4:x; 5:4 array 3,T7,T9}

18 (17):  Ø, temp #1Ø, var b, plus, var c
     {1:b,A7,A9; 2:c; 3:1+2,T6,T8,T1Ø; 4:x; 5:4 array 3,T7,T9}

19 (17):  Ø, temp #11, var x, array, temp #1Ø
     {1:b,A7,A9; 2:c; 3:1+2,T6,T8,T1Ø; 4:x; 5:4 array 3,T7,T9,T11}

2Ø (17):  Ø, var c, -- , assgn, atemp #11
     {1:b,c; 4:x}

21 (18):  Ø, temp #12, var b, plus, var c
     {1:b,c; 2:1+1,T12; 4:x}

22 (18):  Ø, temp #13, var x, array, temp #12
     {1:b,c; 2:1+1,T12; 3: 4 array 2,T13; 4:x}

23 (18):  Ø, var a, -- , assgn, atemp #13
     {1:b,c; 2:1+1,T12; 3: 4 array 2,T13; 4:x; 5:A13,a}

24 (19):  Ø, -- , -- , return, nconst Ø
```

Figure 6-3
Array Reference Optimization

T9 is added to EC #5 and the quad is removed. T9 is in the same class as T7, so the array element represented by A9 must be the same element as the one represented by A7. Therefore, quad 17 assigns the variable b the value it already has. This is indicated by the fact that b and A9 are in the same EC, so quad 17 is deleted.

Quads 18 and 19 compute the same subscript used before so they are deleted, however, quad 2Ø changes the value of the variable c which was used in the subscript calculation. This causes a great change in the ER, because changing the value of c removes it from its old EC and adds it to EC #1. Since EC #2 no longer exists, all expres-

sions that include EC #2 are deleted. This eliminates the expression 1+2 from EC #3. All the remaining members of EC #3 are the temps that at one time contained the value of the expression that was deleted, so they are also removed. That leaves EC #3 empty, and it is removed. With EC #3 now gone, the expression and all of the temps in EC #5 are deleted. When "T" temps are removed, their corresponding "A" temps are also removed, and the resulting ER is as shown in the figure. The subscript value b+c in quads 21 and 22 must be recalculated so these quads are not removed.

Figure 6-4 shows the actual quads produced by the optimizer for this sample program. By relating the statement number in the quad back to the source program lines, the reader can easily see which statements have been eliminated.

TEMPORARY NUMBERS

One additional function perfomed by the optimizer is the assignment of new numbers to all temporaries that still exist. These new numbers indicate what hardware locations will be used to hold the value of the temp when the program is running. The numbers Ø and -1 refer to registers Ø and 1 respectively. All other temp numbers are offsets from the program's local stack pointer and refer to memory.

The assignment of temp numbers is performed on each basic block after the block has been optimized. Since registers are faster and easier to access than memory locations, the most frequently used temps ought to be assigned to the registers, leaving the least frequently used temps for memory. This is accomplished by determining the frequency of use and range over which each temp in the basic block is used. This list of temps is then sorted in decreasing order by frequency, and the

```
Quad (stmt): count, r, a, op, b
   Ø (5):  Ø, -- , nconst Ø, lcall, lib setup
   1 (8):  Ø, var a, var b, plus, var c
   3 (9):  Ø, var d, -- , assgn, var b
   4 (1Ø):  Ø, var e, -- , assgn, var c
   7 (12):  Ø, var d, var e, mult, var a
  1Ø (13):  Ø, var b, -- , assgn, var d
  12 (15):  Ø, temp #Ø, var b, plus, var c
  13 (15):  Ø, temp #Ø, var x, array, temp #Ø
  14 (15):  Ø, atemp #Ø, -- , assgn, var b
  2Ø (17):  Ø, var c, -- , assgn, atemp #Ø
  21 (18):  Ø, temp #Ø, var b, plus, var c
  22 (18):  Ø, temp #Ø, var x, array, temp #Ø
  23 (18):  Ø, var a, -- , assgn, atemp #Ø
  24 (19):  Ø, -- , -- , return, nconst Ø
```

Figure 6-4
Quads Produced from Arithmetic Optimizer

temps are assigned to registers or memory in that order. Once a temp has been assigned a number, that number must not be assigned to any new temps that appear before the final use of the first temp. This assignment scheme results in smaller object programs.

Assignment of machine locations to temps is usually performed in the object code generator along with all of the other machine-dependent functions so that when the compiler is changed to run on another machine, only the object code generator need be revamped. For the AL code generator to determine the usage and range of the temps, some of the data flow analysis performed in the optimizer would have to be repeated, so by assigning the temp numbers in the optimizer, the code generator is considerably simplified.

SPECIAL TEMPS

The AL "do" statement iterates a loop based upon a counter value. The count is stored in a temporary location, but since it is used repeatedly it must be preserved for the duration of the loop. This is indicated by the appearance of the dummy "temp" quad, which indicates that a previously created temp is actually a "special temporary". These temps must be preserved until "they" are released by another "temp" quad. When temporary numbers are assigned to temps, the optimizer automatically assigns memory locations to all special temps.

Special temps are also used in the "for" statement when the parameters for the loop are not constants. This protects the parameter values from modification within the loop.

7.
Object Code Generator

DEVELOPMENT PROCESS

The code generator for the compiler produces assembly language since the purpose of the project was to investigate compiler problems rather than operating system interface problems. Also, by not having to deal with the UNIX scheme for representing object files, the compiler retains a small degree of portability.

The object code generator underwent significant evolution between ALc and AL4. The original version was one of the most difficult components of ALc to write. Since it was written without prior knowledge of what kind of formatting and editing functions would be required, some of the formatting routines were broken into many smaller components, each performing a single specific function. This made the code generator one of the largest modules in the compiler, as well as being one of the most difficult to understand.

In AL4, the code generator turned out to be a smaller, neater module than its ancestor because of the experience gained. The support routines, such as generating a line of assembly language code from several operands or generating the address of an operand, were consolidated into single modules, and the entire program was modularized according to function. Also, since the temporaries in AL4 are assigned permanent locations in the expression optimizer, that function was not included in the AL4 code generator.

Because of the arithmetic expression optimizer in AL4, any temp may be used several times in a basic block. The optimizer takes care of assigning locations to the temps, but the AL4 code generator cannot destroy the contents of a temp to perform an operation since that same temp might be needed later. This change dictated some modifications to the algorithms used in the original version for several types of quads and, because several special cases were eliminated, the optimizer became even smaller.

One particularly convenient feature of the object code generator is the automatic generation of comments for the assembly language statements. Since variables are stored on a stack, access to them is made using an integer offset rather than the variable name, so it is difficult to relate the assembly language statements to the source program. The comments, however, provide the reader with the variable names that were used in the source program, which simplifies working with

the assembly program.

OBJECT CODE

The architecture of the PDP-11 greatly simplified the object code generation process. Since all variables are on the user's stack, they are referenced with an offset from the pointer to the base of the stack. This reference can be done in a single PDP-11 instruction. Local and global variables reside on the stack in different areas, so two stack pointers are used, one for local values and one for global values.

Because most arithmetic operations can be performed in memory locations as well as in registers, many quads that involve temporary locations do not actually require any temporary space for the calculation. Whenever possible, each operation is performed in the result field of the quad, which eliminates many temps and move instructions.

The "array" quad generates several instructions to calculate an address. In ALc, each step of the calculation that might be affected by subscript errors was checked to prevent them. Normally AL4 generates a slightly shorter version of these same instructions, but by specifying a compile-time option, the error checking instructions can be suppressed to produce smaller, faster object code for tested programs. The decrease in program size can approach 30% to 50% depending on how many array references the program contains.

If the debug option is selected, the code generator examines the line number field of each quad it processes. Whenever a new line number is encountered, an instruction is generated to save its value. If the program should abort, the error routine will have access to the source program line number of the faulty statement.

CALLING/RECEIVING SEQUENCES

Any type of data can be passed as an argument to an AL program. In addition, arguments may be call by reference or call by value. In order to implement this flexible scheme without using excessive amounts of memory for the calling and receiving

```
/*
 * Sample AL program 4: calling/receiving sequences
 */

function abcd(val int a, ref int b, val string c);

int d;

d=abcd(a+1,b,c(b,d));
end;
```

Figure 7-1
Sample AL Program

```
_abcd:
        jsr     pc,setup
        pname
        10.                     /arg stack space
        stack                   /addr of local stack size
        3.
        010400                  /int param
        -2.                     /local stack offset
        110500                  /ref int param
        -4.                     /local stack offset
        030400                  /string param
        -10.                    /local stack offset
```

Figure 7-2
Receiving Sequence for Sample Program

sequences, they are generated in the compressed format described below.

Figure 7-1 is a simple AL function. The object code produced when it is compiled contains the receiving sequence illustrated in figure 7-2. It begins with a "jsr" instruction to the "setup" library program. This is followed by a pointer to the program's name, the amount of stack space needed for arguments, and a pointer to the amount of local stack space that will be required. These values are used by the setup program to create the new stack frame for the routine. Next is an integer indicating how many formal parameters there are, followed by the parameters themselves.

A calling sequence begins with a "jsr" instruction to the target procedure. Figure 7-3 illustrates a calling sequence, but it has a "mov" and an "inc" instruction before the "jsr". These two instructions evaluate the expression "a+1" which is

```
        mov     -2.(r4),r0      /var a, reg temp #0
        inc     r0              /reg temp #0
        jsr     r0,_abcd
        3.+100000
        010400                  /int param (1)
        2.                      /local stack offset
        010200                  /int param (2)
        0.                      /register number
        110500                  /ref int param (3)
        -4.                     /local stack offset
        030454                  /string param (4)
        -10.                    /local stack offset
        -4.                     /local stack offset
        2.                      /local stack offset
```

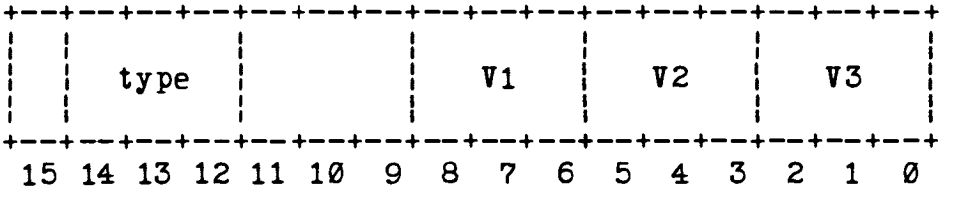
Figure 7-3
Calling Sequence in Sample Program

passed as the first argument. The "jsr" is followed by an argument count, which includes a flag (octal 100000) indicating whether or not a result is expected. The actual arguments follow the count, with the parameter that describes the returned value's location appearing first. In the example, the result location is specified in parameter one using the local stack offset of the variable "d".

Each argument or parameter is represented by a control word and one or more value words. The format of the control word and meaning of the value words that follow it are shown in figure 7-4. Bit 15 indicates whether a formal parameter is call by reference or call by value. If bit 15 is set for an actual argument, then the argument itself is a call by reference parameter of the current procedure. The type field identifies the argument or parameter's type. Different types of arguments may require one, two or three values to pinpoint their locations. The fields V1, V2 and V3 in the control word indicate which of the corresponding values appear in the calling sequence and what each represents. The letter V is used in the descriptions below to represent any of the fields V1, V2, or V3.

If the V field in the control word contains a zero, then no corresponding value word is

PARAMETER CONTROL WORD



BITS	MEANING
15	Argument mode 0: call by value 1: call by reference
14-12	Argument type 001: integer 010: array 011: string
11-9	Reserved for future use
8-6	V1 indicator
5-3	V2 indicator
2-0	V3 indicator

V1 - V3 MEANINGS

000: null, no value word
001: value is numeric constant
010: value is register number
011: value is register number, accessed indirectly
100: value is offset from local stack pointer
101: value is offset from local stack pointer, accessed indirectly
110: value is offset from global stack pointer
111: value is an address

Figure 7-4
Parameters

generated. The first three parameters in the example use only V1, so V2 and V3 in those control words are zero. If V is ØØ1, the argument is the numeric constant that appears for the value. If V is Ø1Ø, the argument is in a register, and the value is the register number. The second parameter specifies the expression in register zero. The code just prior to the "jsr" instruction calculates the expression "a+1" and leaves the result in register zero. If V is Ø11, the address of the argument is in a register, and the register number number is given as the value. For local variables 1ØØ is used, which indicates that the value is an offset from the local stack pointer. Parameter one specifies the position of the local variable "d", which will receive the result. A V of 1Ø1 also means that the value is an offset from the local stack pointer, but this offset points to the address of the argument. The second argument, "b", is itself a call by reference formal. The third parameter indicates indirect addressing by using a 1Ø1 in the V1 field. The corresponding offset is negative because "b" is a formal parameter rather than a local variable. V is 11Ø for global variables; the value is an offset from the global stack pointer. If V is 111, the value is the actual address of the argument. This is most often used to pass string literals as arguments. The value points to the appropriate entry in the string literal table.

The fourth parameter illustrates how a substring is passed as an argument. V1 points to the string itself, and V2 and V3 indicate the limits of the desired substring.

When a procedure call is executed, the setup routine matches the number and type of arguments to the number and type of the formal parameters and aborts the program if any errors are discovered. Each parameter is set up according to its mode; call by value parameters are duplicated in the new routine, and the address of each call by reference parameter is copied into the new routine. The value or address is placed in the area of the new routine's local stack that was reserved for formal parameters. (This is segment #2 of the local stack, as described in chapter 8.) Call by value parameters are accessed just like local variables since that is what they really are. Call by reference parameters are accessed using indirect addressing because the address of the argument is stored in the local stack.

When arrays or strings are passed with call by reference, a pointer to the descriptor to the string or array, which contains length and subscript information, is put into the local stack. However, with call by value, a copy of the entire array or string must be created in the new routine. The setup program performs both these tasks.

If the calling routine expects a result to be returned, the first parameter in the list describes where the result should be stored. Setup handles this by calculating the absolute address in the old program's stack where the result should be put, and saving this address in the new routine's stack. When the called program returns an answer, it is stored indirect

through the saved address.

8.
Run-time Environment

An AL program cannot execute without several standard routines which are referred to as the run-time library. These routines perform many functions such as interfacing with UNIX and maintaining the user's stack, and are all stored in the AL library. The loader selects whichever routines are needed and includes them with the user's program.

THE USER STACK

All data in an AL program is stored on the user's stack. The stack starts at the first free memory location after the user's program and grows toward higher memory addresses. As illustrated in figure 8-1, the user's stack consists of two main segments: the global data area and the local data area. When an AL program is executed, the initialization routine reserves the appropriate amount of memory for the global stack, and stores a pointer to the base of that stack in register 5. The next available memory location is the beginning of the user's local stack, and that address is saved in register 4. These registers are utilized in assembly language instructions which access variables in the stack, so they must not be destroyed during program execution.

STACK FRAMES

Every routine requires some space on the stack, and this space is called a frame. Figure 8-2 shows how frames are organized. When a routine is called, a new frame is allocated by enlarging the stack.

A frame contains four seg-

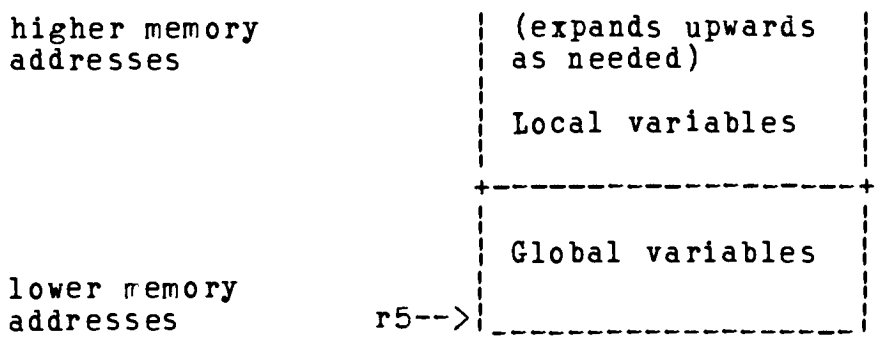


Figure 8-1
Stack Organization

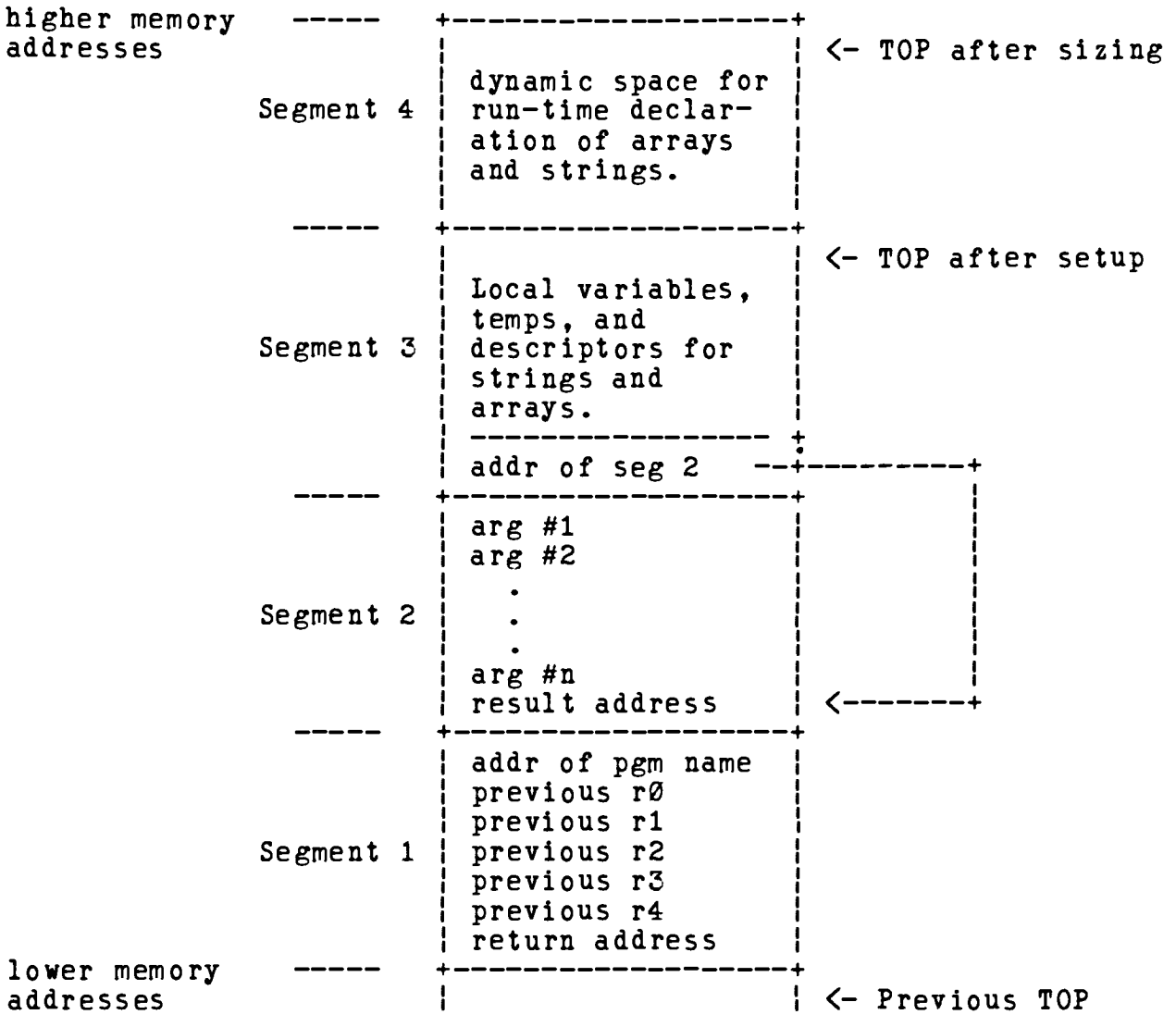


Figure 8-2
Local Frame Layout

ments. Segment one contains information pertaining to the previous environment. This segment is consulted to find the stack pointer of the calling program when arguments are being retrieved. This segment also contains the address of the current routine's name, which is used by the error routine.

Segment two varies in size depending on how many arguments were passed to the routine. Each call-by-reference argument reserves one word in the stack which contains the address of the actual argument in the calling routine. If the actual argument being passed is itself a call-by-reference formal parameter, the indirection is automatically followed by the setup routine so that the value stored points to the actual argument. Call-by-value parameters are stored in the argument segment just as they are stored in the

data segment, described below.

Segment three is the data segment. Register 4 in the user's program always points to the beginning of this segment, which contains the address of the beginning of segment two. This pointer is used to locate the data required to return to the previous routine by automatically skipping over the arguments. The rest of segment three contains local variables. These variables are accessed using offsets from register 4, the local stack pointer. Arguments, which are in segment two, have negative offsets, and local variables have positive offsets so the members of each segment are accessed correctly. An integer is stored by placing its current value in the single word at the appropriate offset. Since arrays and strings are dynamic, the offset for a variable of one of these types points to a descriptor for the data; these descriptors are illustrated in figure 8-3. The "setup" program automatically reserves enough space in segment three for the string and array descriptors. It also fills all descriptors with zeros so that any attempt by the programmer to access an array or string before specifying its dimensions with the "size" statement will result in a subscript error. When a "size" statement for an array or string is executed, the proper amount of memory in segment four is obtained from the operating system, and the address of the beginning of that area is stored in the descriptor. For arrays, the number of dimensions and the subscript limits are also stored. Currently arrays are limited to one dimension. For strings, the maximum length is saved in the descriptor and the current length is set to zero.

Segment four is the dynamic area in which strings and arrays are allocated. The top of the stack increases as new arrays or strings are sized. When another routine is called, a new frame is built starting with the first word after the old stack top, but when the new routine returns the environment is completely restored. The current routine may then continue normally, allocating more strings or arrays as needed.

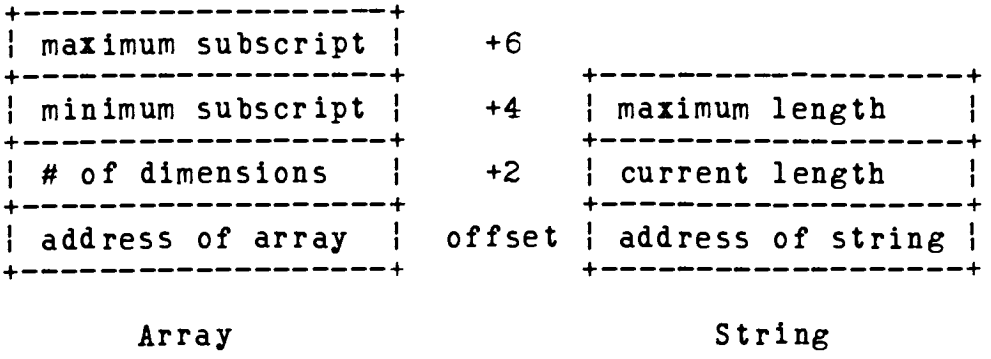


Figure 8-3
Array and String Descriptors

GLOBAL ARRAYS AND STRINGS

Global strings and arrays, like any other strings or arrays, must be sized while the program executes. The global stack space is allocated before the program begins to execute, so the main program's stack is allocated before any global variables can be sized. This leads to a potential problem. If global variables are sized from the main program, they will be assigned space in segment four of the main program's stack frame. This causes no problems because the main program's local stack is never released. If a global array or string is sized from any routine other than the main program, the array or string will be assigned space in segment four of that routine's frame. When that routine returns, its frame is released. A new frame will be allocated when the next routine is called, which might overwrite the space assigned to the global variables. It is for this reason that global variables must not be sized in any routine other than the main program.

RUN-TIME LIBRARY

The run-time library includes the "setup" program for processing procedure calls, the "init" program which handles program initialization, as well as several interface and utility programs. This section describes several of these utilities.

"ctoi" converts a single character to an integer, and supports the inclusion of single character substrings in arithmetic expressions. "itoc" performs the opposite conversion and is used when a numeric value is assigned to a string.

"open", "close", "read", "write", and "delete" interface with the corresponding UNIX file system calls to perform input and output.

"exec" is used in an AL program to execute another AL program. First, a UNIX "fork" operation is done to create two processes. In the child process, "exec" sets up the arguments to the desired program and invokes it. This program may return a one byte code by executing a "return" from the main program. In the parent process, "exec" simply waits for the child to complete and returns the code, if any, to the original program.

"fatal" is called when an AL program aborts for a known reason, such as a subscript error. It prints a brief description of the error, its location in the source program, the contents of the registers, and a call history. It then forces a memory dump by executing an IOT instruction. If the description and location of the error are not enough to allow the programmer to find the problem, the memory dump can be interrogated with the help of the call history to determine values of variables and arguments.

"getmem" obtains memory from the operating system to expand the stack. It is called from "setup", "init", and "size". In order to avoid the overhead involved with repeated operating system calls, once memory is obtained it is never given back to UNIX. Also, each AL program contains a location called "alloc" which can be ini-

tialized with the UNIX debugger after the program is loaded. If this location contains a non-zero value, it is treated as a byte count of the amount of memory to be pre-allocated for the user stack. This avoids calling the system several times to obtain the same amount of memory in small pieces.

The "size" routine supports the size statement and allocates memory in the user's stack for arrays and strings.

"strcat" supports string assignment and concatenation, and "strcmp" supports relational tests between substrings.

9.
Conclusions

AL is a working language. Although it is a comfortable language in which to write programs, it will probably not be used by anyone other than the author since there are so many comfortable languages in existance already. As a production language, it probably has little chance of being widely used.

The compiler is fairly modular, so additions or modifications could be made to it with few problems. Therefore, AL could be used as a tool for research into various language problems, such as error recovery or optimization. Also, existing modules could be modified or replaced with programs that use different algorithms to investigate the behavior of these algorithms in a working language. This chapter contains a discussion of what might be changed in AL, or what might be added to it.

NEW FEATURES

One feature that was eliminated from the design of AL was an assembly language optimizer. The original intent of this optimizer was to remove redundant or useless instructions that were generated because of hardware restrictions. For example, multiplication requires two registers, an even and an odd numbered one. If a multiplication must be performed on an intermediate value of an expression, then the temp that holds that result ought to be assigned to a register in which multiplication can be done to eliminate extra move instructions. The arithmetic expression optimizer removed some of the redundant code related to special temps, but in cases such as the example given above, further optimization could improve the object code. To implement the assembly language optimizer, the object code generator would be changed so that its output was symbolic rather than actual text, and the optimizer would read it, reassign temps as needed, and generate assembly language statements.

The arithmetic expression optimizer performs no global data flow analysis. This is an obvious area for improvement. Many more transformations could be included, such as removing invariant code from loops, analyzing the reachability of code based upon global data flow information, expansion of infrequently called procedures inline, and so forth. Some of the work usually associated with constant folding is already performed by the peephole optimizer, but more could be done in this area using the global data flow information.

One of AL's interesting

features is the run-time declaration of arrays. This eliminates the problem of trying to estimate maximum limits on the size of arrays that change in size from time to time. These arrays involve more complex subscript calculations, though, since it is not known at compile time what the subscript limits will be. A useful addition to the language would be static arrays or strings, perhaps declared by sizing them with constants, whose size and subscript limits were known at compile time. This would allow the compiler to generate better object code for accesses to these arrays and give the programmer a choice between flexibility and speed.

The dynamic arrays are supported by the parser with the "array" quad. In this implementation, a subscript calculation is performed by specifying the array name and the subscript's value in the quad. This method is therefore limited to one-dimensional arrays. Multi-dimensional arrays, a standard feature of nearly all programming languages, could be implemented in AL only by drastically changing the way in which array references are handled. One way to do this might be to have the parser generate code for the subscript checking, and have the array quad simply add an offset to the base address for the array. This approach might also provide side benefits: the code for subscript checking and calculation would then be subject to arithmetic optimization just like any other expression.

SUGGESTED CHANGES

One possible change would be to modify the organization of the stack. Currently the user's stack grows upwards from the highest address in the instruction part of the program, while the system's hardware stack grows downward from the top of memory. When "jsr" instructions are executed, the registers used are pushed on to the system stack, so they must later be copied onto the user's stack. The calling/receiving sequences in some of the run-time routines are somewhat clumsy because of these two stacks; combining them would simplify the code and speed up the programs.

The two-stack approach was initially used to simplify the routines that accessed the stack. It was thought that the user stack would be easier to access if it were not constantly growing and shrinking due to subroutine calls and saving temporary values. The approach turned out to have just the opposite effect though, since registers saved on the system stack had to be copied onto the user's stack, and run-time routines that called each other had no common means of communication. While much of the compiler was written twice, the run-time library was written only once. Rewriting it would produce routines that might be 5% to 10% faster and 10% to 15% smaller.

The interface between YACC and AL is a very strict one. Since YACC does not recognize AL statements, the grammar was processed without any action statements. Once the parsing tables were produced, the action routines were hand-coded. This has the unfortunate effect of freezing the grammar since almost any changes to it would re-

quire the action routines to be almost completely rewritten. If further experimentation were to be done with the grammer, a better interface with YACC would have to be worked out.

One advantage of the current YACC interface is that several grammar rules can share the same action statements, which is not possible in normal YACC action routines. This reduced the size of the action routines and simplified the logic in them considerably over the ALc version. A new YACC interface should preserve this capability.

The YACC parser has a crude capability to recover from syntax errors. The rule "error" may be included in the grammar as alternative reductions in places where errors may occur, and these rules may have action statements to go along with them. If a syntax error is discovered, the parser tries to find the most recent appearance of an error rule and resumes parsing at that point. If no error rule was encountered the parser will abort on a syntax error. This can be rather disturbing in AL programs, since errors in the declaration statements will cause the compiler to simply quit rather than checking the rest of the program as well. An alternative that might be investigated for error recovery would be to assign probabilities to each of the rules in the grammar. When an error is discovered in the source program, the "cost" of modifying the tokens could be weighed against the likelihood of different potential rules, and the most probable result would be selected. If this substitution caused further errors, the algorithm would backtrack and select the next most likely possibility.

10.
APPENDICIES

APPENDIX A - COMPILER LISTINGS

This appendix consists of the block diagrams that follow as well as the compiler listings which appear under a separate cover entitled THE AL COMPILER.

The block diagrams illustrate the relationships between modules in the compiler, and are intended to be used as an aid in understanding how the components of the compiler operate and how they relate to and communicate with each other.

APPENDIX B - LANGUAGE MANUAL

The contents of this appendix appear as a separate part of the report titled THE AL LANGUAGE REFERENCE MANUAL.

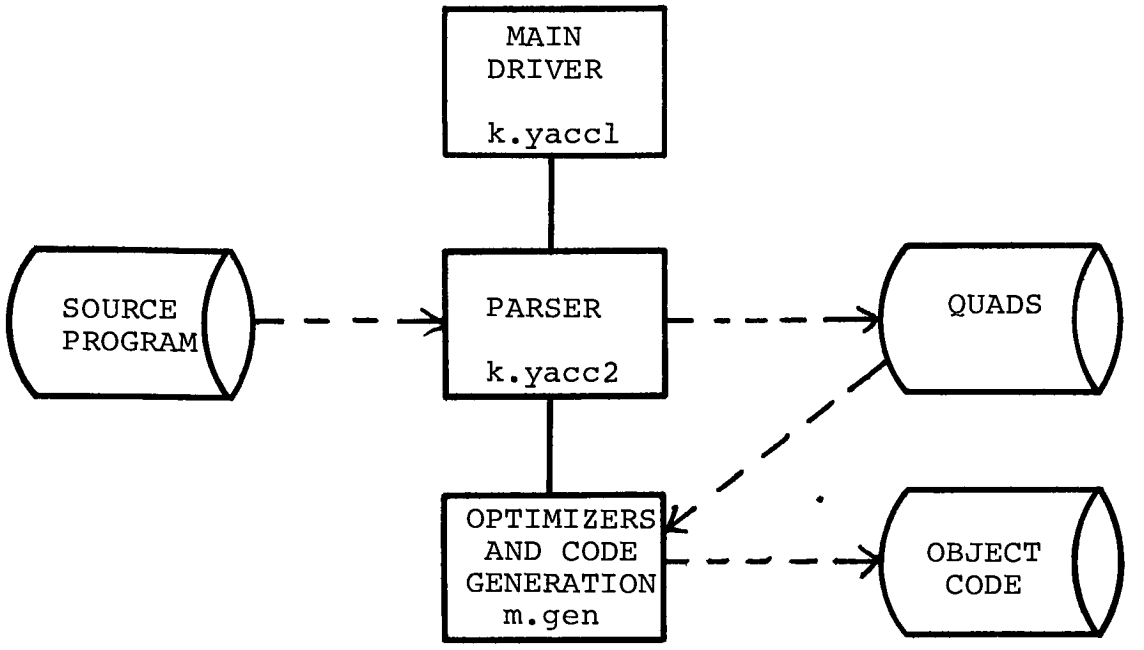


Figure A-1
Compiler Overview

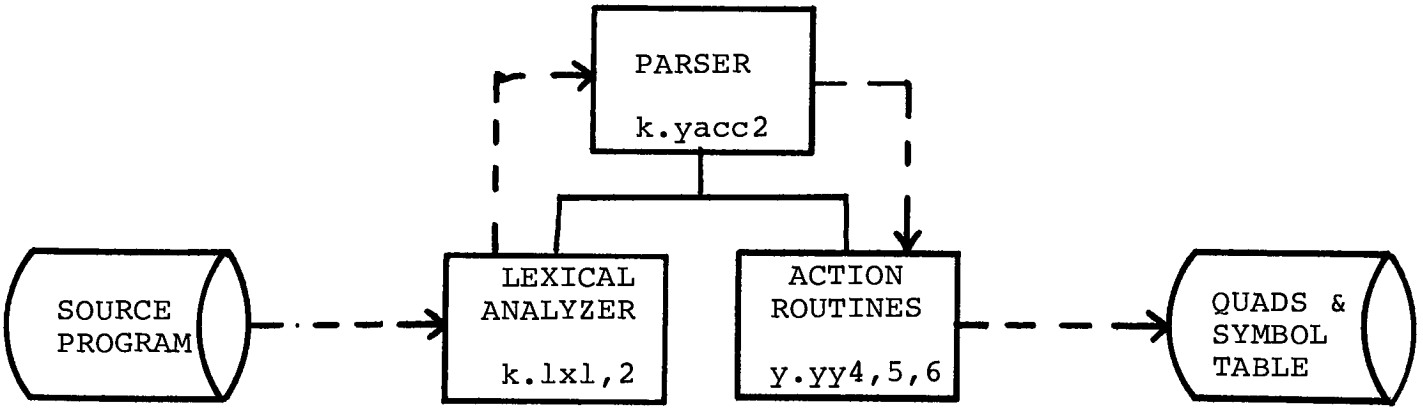


Figure A-2
Parser

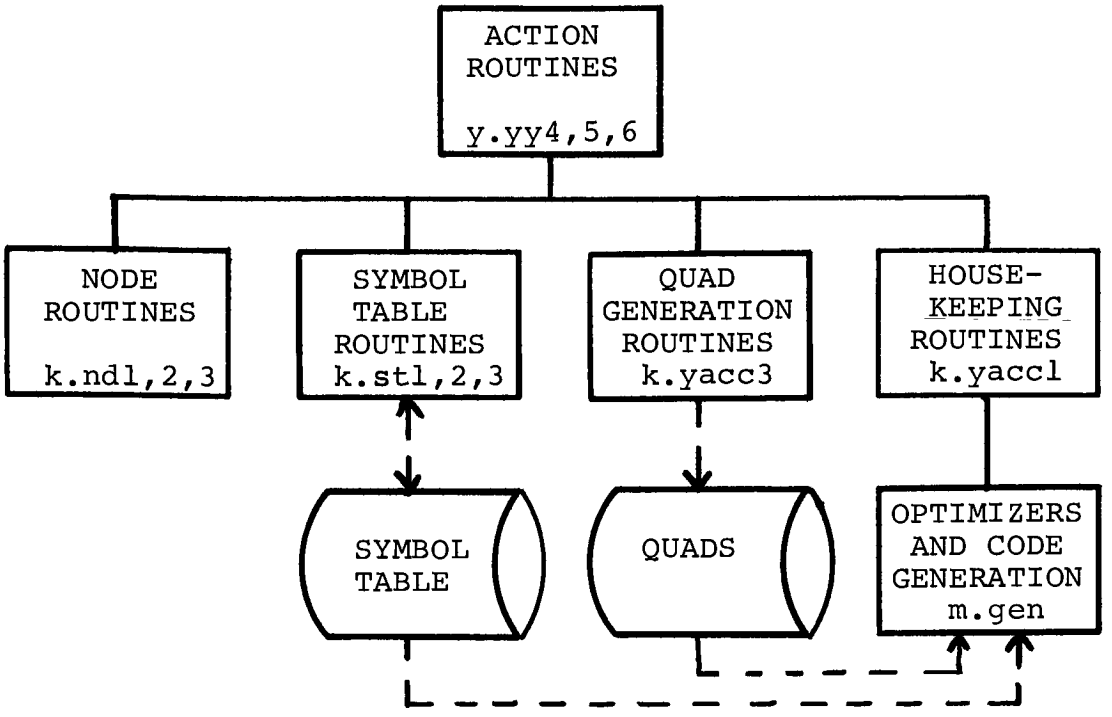


Figure A-3
Quad Generation

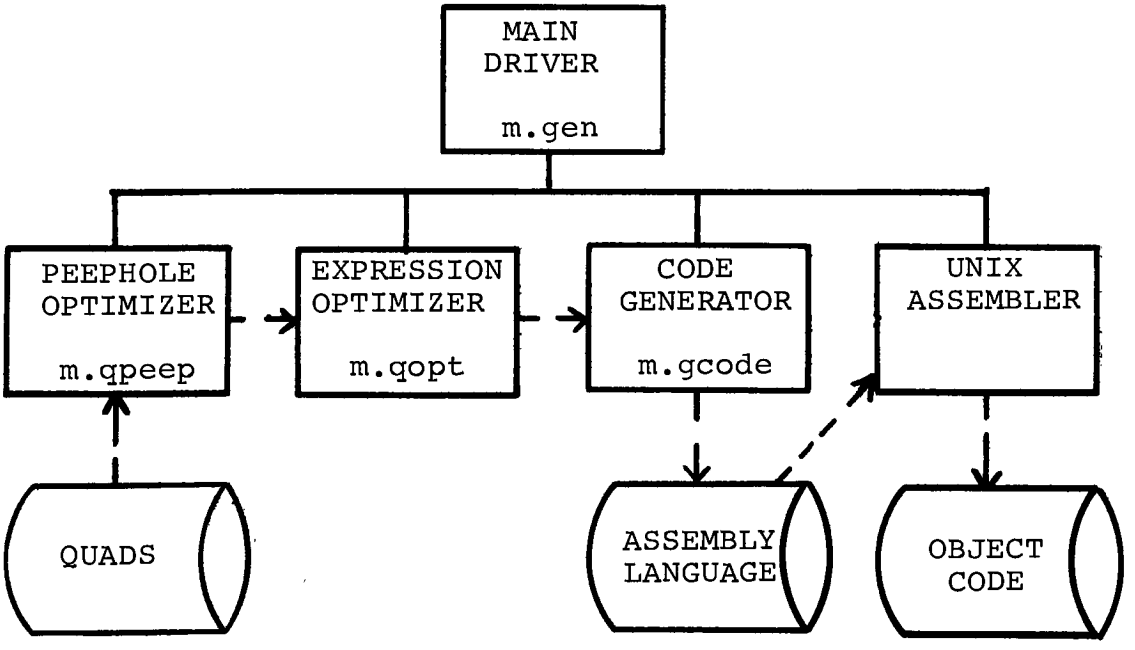


Figure A-4
Object Code Generation

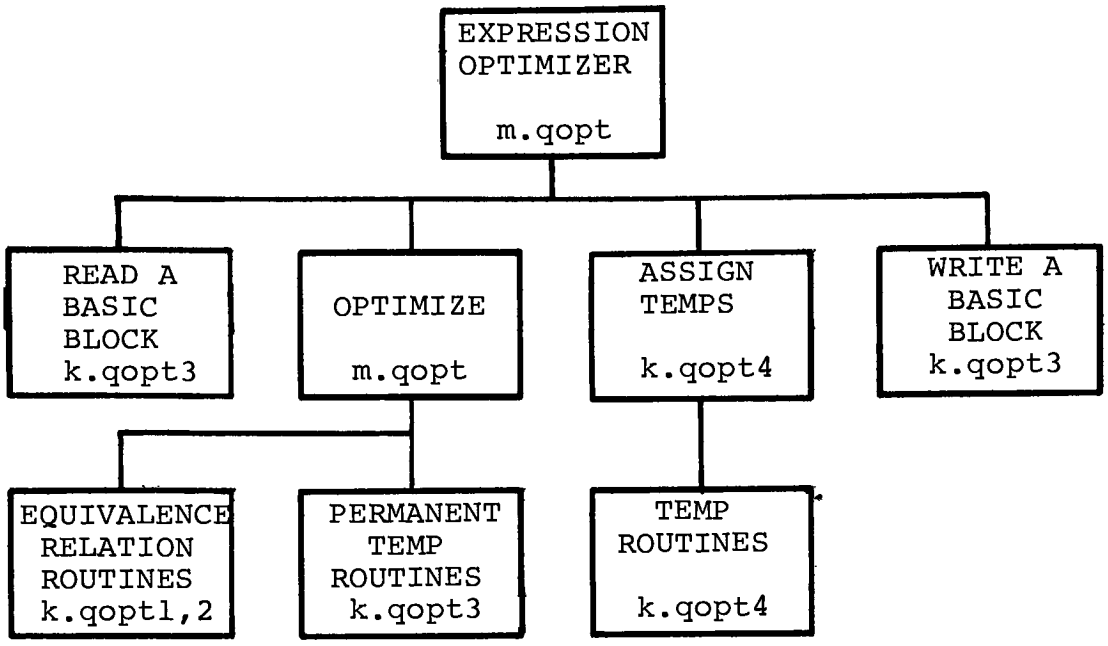


Figure A-5
Expression Optimizer

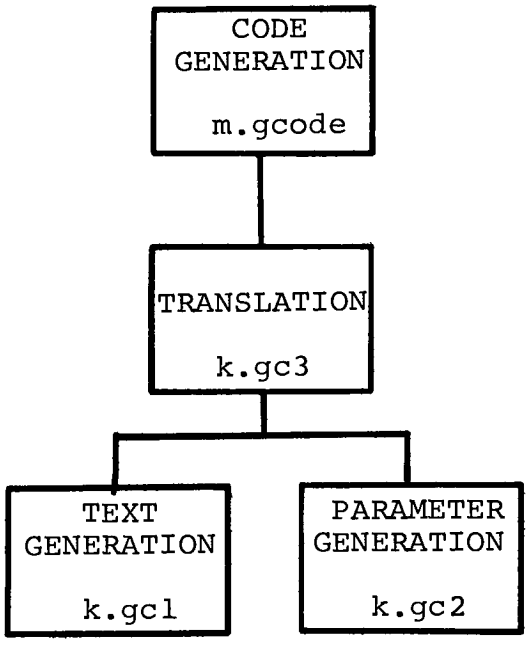


Figure A-6
Code Generation

T H E A L L A N G U A G E
R E F E R E N C E M A N U A L

Submitted by
Kenneth A. Reek
February 12, 1979

C O N T E N T S

PREFACE

AL is an acronym for "Algorithmic Language". As the name suggests, the AL language was originally designed to express clean, well structured algorithms. Programs written in AL illustrate their underlying algorithms exceptionally well. It is also easier to translate algorithms written in pseudo-code into working programs with AL than with most other languages.

This manual, like other user's manuals, is not intended to be a teaching document. It explains the details and features of AL, but it assumes that the reader has had previous exposure to computer languages.

English descriptions of language structures are supplemented by a BNF-like description of the relevant grammar rules. In these descriptions, '::=' is the replacement operator, the vertical bar '|' specifies a choice between alternatives, and angle brackets < and > enclose non-terminals. Curly braces { and } enclose constructs that may be repeated zero or more times. Where language elements conflict with the BNF metalanguage, the context of the grammar rules involved is sufficient to resolve any ambiguity.

NOTATION AND VOCABULARY

The vocabulary of AL consists of all meaningful inputs to the compiler. These inputs are constructed from letters, digits, and special symbols according to certain grammar rules. Letters and digits can be combined to form words, which can be either reserved words or identifiers. A list of special symbols and reserved words is given below.

```
!           ==          enddo       open
!=          >           endfor      print
"           >=          endif       procedure
%           [           endloop     read
&           ]           endwhile    ref
&&          |           exec        repeat
'           ||          exit        return
(                       external    size
)           argc        for         stop
*           argv        forever     string
+           array       function    then
,           by          global      times
-           close       if          to
/           const       int         until
:           do          label       val
;           else        len         while
<           elseif      loop        write
<=          end         next
```

Reserved words may not be used by the programmer as identifiers.

Words in a program are separated from each other by special symbols (subject to statement syntax) or by format characters or comments. Space, tab and newline are format characters (so-called because they are used to format the text of the program). Anywhere a single format character is legal, any number of such characters may also be used.

COMMENTS

Comments may be inserted anywhere in a program (except within lexical structures, such as words or literals) by preceeding them with '/*' and following them with '*/'. Comments may span several lines, if desired, or be placed on the same line as or within program statements. The entire comment is considered to be equivalent to a single format character.

FILE INCLUSION

The characters '{' and '}' have a special meaning to the compiler. The text between them is taken as a UNIX pathname, and the entire construct is replaced by the contents of the named file. For example, if {globals} appeared in the program, it would be replaced by the contents of the file 'globals'. The named file may not contain any inclusions itself, that is, nested

file inclusions are not allowed.

IDENTIFIERS (<id>)

Identifiers are names denoting constants, variables, procedures, functions, or program labels. They must begin with an upper or lower case letter, which may be followed by any combination of characters from the following set:

a-z A-Z 0-9 ' _ (underscore) . #

Identifiers may be any length, however only the first twelve characters are used within the compiler to determine uniqueness. Identifiers denoting procedure or function names must be unique in the first seven characters, due to restrictions imposed by the loader. All of the following are valid identifiers.

a	x1	prime_number	pass#3
B	B'	B''	aBc.39

LITERALS (<int literal>, <string literal>)

A sequence of one or more digits (not part of an identifier) is an integer literal. If a literal begins with a digit from '1' to '9', its value in the program is the decimal value of the number written. If a literal begins with the digit '0', its value is the octal value of the number written. (Though digits '8' and '9' may appear in an octal literal, denoting the octal values 10 and 11, it is not a recommended practice.) Decimal and octal literals may be preceeded by an optional minus sign, '-', to denote a negative value. Size limits for integer literals are the same as those for integer variables.

A single character enclosed in apostrophes ''' denotes a character literal. This is another form of integer literal, whose value is the ASCII value of the character. Any non-control character (by ASCII definition) may appear between the apostrophes.

The following examples illustrate integer literals and their values.

LITERAL	VALUE
535	535
0100	64
' '	32
'A'	65
'''	39
40000	error, value too large
2,000	error, comma not allowed
''	error, no enclosed character

A string literal is any sequence of characters surrounded by quotation marks '"'. To include a quotation mark within a string literal, two adjacent quotation marks are used. Tab and newline

format characters within a string literal are ignored and not included in the literal. This means that long string literals may be written on several lines, formatted into tables, etc. To include the format characters tab and newline in string literals, the following lexical conventions are used:

\t	is translated into a tab,
\n	is translated into a newline,
\Ø	is translated into a null,
\?	is translated into ?, where ? is any character other than 'n', 't', or 'Ø'.

String literals may be up to 5ØØ characters long. If a string literal is longer than 5ØØ characters, a warning message is printed and all characters after the leftmost 5ØØ are discarded.

DATA TYPES, CONSTANTS, AND VARIABLES

Each variable in a program is associated with exactly one type of data, as declared in the beginning of the program. The AL compiler supports three data types, integers, arrays, and strings.

<int var> ::= <int id>
<array var> ::= <array id>
<string var> ::= <string id>

INTEGERS

An integer is stored in one 16-bit word, hence it values may be anywhere in the range

-32768 to +32767.

<num var> ::= <int var>

ARRAYS

An array is a one-dimensional collection of integers. An array element is specified by following an array name with an array subscript, a numeric expression (see chapter 3) enclosed in parentheses. The upper and lower subscript limts for all arrays is specified at execution time with the 'size' statement (see chapter 6). Array elements may be used in a program anywhere integer variables may be used.

<array element> ::= <array var> (<num exp>)
<num var> ::= <array element>

STRINGS

A string variable is a one-dimensional collection of characters. The first character in a string variable occupies position one, the second occupies position two, and so on. Adjacent sequences of one or more characters in a string are called substrings. A substring is specified by using a string subscript to denote the positions in the string where the substring begins and ends. A string subscript is written in one of two ways. A subscript containing two numeric expressions, separated by a comma and enclosed in parentheses, denotes the substring that begins with the character pointed to by the first expression and ends with the character pointed to by the second expression. A single numeric expression in parentheses denotes the substring that begins with the character pointed to by the expression and ends with the last character in the string. A string variable with no subscript refers to the entire contents of the variable.

Substrings from which characters are taken (e.g. those in a 'write' function or on the right hand side of an assignment statement) may not include character positions beyond the actual

length of the string. Substrings into which characters are put (e.g. those in a 'read' function or on the left hand side of an assignment statement) may not begin more than one position beyond the actual length of the string, or extend beyond the maximum length of the string variable. These rules are illustrated with examples in the section on the assignment statement in chapter 6.

A substring enclosed in parentheses may be used as a numeric expression whose value is the ASCII value of the first character in the substring.

<substr> ::= <string id> ¦ <string id> (<num exp>)
¦ <string id> (<num exp> , <num exp>)

OTHER DATA TYPES

Integer variables can be used instead of boolean variables, since the logical (bit) operators apply to integers. Integer variables can store single characters, which are represented as the number associated with the ASCII character. Real numbers are not supported.

CONSTANTS

A constant is an identifier whose value is assigned at compile time. Like variables, every constant corresponds to one of the three data types, depending on the value(s) it represents. Unlike variables, the value of a constant cannot be changed. In particular, it is a syntax error for any constant to appear on the left side of an assignment statement.

Integer constants are replaced in the object program by the value they represent. This means that to use an integer constant instead of a literal requires no sacrifice in program efficiency. Integer constants are recommended for all parameter-type quantities, since such things are often changed.

Array constants are collections of integer constants, and are subscripted exactly like array variables. The lowest subscript of an array constant is always one, and the highest subscript depends upon how many integers are specified in the array.

<array const element> ::= <array const> (<num exp>)

String constants are collections of characters, and may be subscripted exactly like string variables.

EXPRESSIONS AND OPERATORS

NUMERIC EXPRESSIONS

Numeric quantities (numeric expression, literal, constant, integer variable, array element, function call, array constant element, or a substring enclosed in parentheses) can be combined using numeric operators to form numeric expressions. The value of a numeric expression is the result of the operator applied to the values of the components in the expression. There are two kinds of numeric operators, binary and unary. Binary operators connect two numeric components to form an expression, and unary operators preceed a single component to form an expression. A list of all numeric operators is given below.

Prec	Type	Operator	Meaning
1		( )	Grouping
2	unary	~	bitwise NOT (1's complement)
2	unary	-	negation
3	binary	*	multiplication
3	binary	/	division
3	binary	%	remainder of a division
3	binary	&	bitwise AND
4	binary	+	addition
4	binary	-	subtraction
4	binary	¦	bitwise OR

(This table continues with relational operators in chapter 8.)

The order in which numeric expressions are evaluated depends on the precedence of their operators and the use of parentheses. An expression enclosed in parentheses is evaluated before any expressions not so enclosed. Parentheses may be nested, and redundant parentheses are acceptable.

Within a given level of parentheses, the expressions whose operators have the highest precedence are evaluated first. Expressions with operators of the same precedence are evaluated from left to right as they appear in the statement. For example, the expression

```
a % 3 & Ø777 - b / ~ x ¦ 1
```

is evaluated as if it were parenthesized as follows:

```
((((a % 3) & Ø777) - (b / (~ x))) ¦ 1)
```

The value of a substring in a numeric expression is the ASCII value of the first character in the substring. Remember that the substring must be enclosed in parentheses.

```
<num exp> ::= <num exp> <binary op> <num exp>
            | <unary op> <num exp>
            | <int literal> | <int constant>
            | <array const element>
            | <num var> | <function call>
            | ( <substring> )
<unary op> ::= ~ | -
<binary op> ::= * | / | % | & | + | | | -
```

STRING EXPRESSIONS

String quantities (literal, constant, or string variable) can be combined on the right side of an assignment statement to form string expressions. The operator used with strings is '+', denoting concatenation. The result of a concatenation is a string whose length is the sum of the lengths of the components, and whose value is the values of the components, left to right, in the order in which the components appeared. When '+' is used to perform concatenation, it has no precedence because no other operators may appear in the statement with it.

```
<str exp> ::= <substr> { + <substr> }
```

PROGRAM STRUCTURE, DECLARATIONS, AND SCOPE

Every program consists of a heading and a block. The heading contains global declarations, procedures and functions, and external declarations. The block contains local declarations and executable statements.

```
<program> ::= <program heading> <block>
<program heading> ::= <global part> <proc part>
<global part> ::= <null>
        | global <const declares> <var declares> end ;
<block> ::= <label declares> <const declares>
        <var declares> <statement list> end
```

LABEL DECLARATION

The label declaration section contains zero or more 'label' statements, consisting of the reserved word 'label' followed by one or more identifiers separated by commas. All of the identifiers in label statements will be classified as label identifiers, which are used with looping statements (see chapter 8).

Example:

```
    label loop1,pass2,rmtemps;
```

```
<label declares> ::= { < label stmt> }
<label stmt> ::= label <id> { , <id> } ;
<label id> ::= <id>
```

CONSTANT DECLARATION

The constant declaration section contains zero or more 'const' statements, consisting of the reserved word 'const' followed by one or more constant descriptions. A constant description is an identifier followed by an equals sign and a literal or simple expression. Each identifier in a const statement is given the type and value of the assigned literal or simple expression. A simple expression defines an integer constant, a string literal defines a string constant, and a list of simple expressions enclosed in square brackets defines an array constant.

A simple expression is any numeric expression that includes only integer literals and previously defined integer constants, so that its value can be computed at compile time. This allows constants to be defined in terms of other, previously defined, constants, which simplifies the definition of related parameters.

Example:

```
    const int_const=1, array_const=[1,2,3], string_const="hello";
    const x1=1, x2=x1+1, x3=x2+1, x4=((x1-x3)&(x3-x1))|x2;
```

```
<const declares> ::= { <const stmt> }
<const stmt> ::= const <const descr> { , <const descr> } ;
<const descr> ::= <id> = <const value>
```

```
<const value> ::= <simple exp> | <string literal> | <array literal>
<array literal> ::= [ <simple exp> { , <simple exp> } ]
<int const id> ::= <id>
<array const id> ::= <id>
<string const id> ::= <id>
<simple exp> ::= <simple exp> <binary op> <simple exp>
        | <unary op> <simple exp>
        | <int literal> | <int const id>
```

VARIABLE DECLARATION

The variable declarations section consists of zero or more 'var' statements, each consisting of one of the three reserved words, 'int', 'array', or 'string', followed by one or more identifiers separated by commas. All of the identifiers appearing in a var statement are classified as the type denoted by the reserved word used in the statement.

Example:

```
    int a,b,c; array k,l,m; string x,y,z;
```

```
<var declares> ::= { <var stmt> }
<var stmt> ::= <type> <id> { , <id> } ;
<type> ::= int | array | string
<int id> ::= <id>
<array id> ::= <id>
<string id> ::= <id>
```

GLOBAL VARIABLES

Global variables may be accessed by any procedure or function in the program. For this reason, they are declared once in the beginning of the program heading. It is the programmer's responsibility to insure that global declarations in external procedures and functions match the declarations given in the programs with which they will be loaded.

Global integers and the descriptors for global arrays and strings are pre-allocated on the user's stack. Care should be taken in using 'size' statements on global arrays and strings; unpredictable results will occur if they are sized anywhere except from within the main program.

LOCAL VARIABLES

Memory space for local variables is allocated each time the procedure or function is called, and released when it terminates. Consequently, the current values of local variables are lost when the procedure or function returns, and are not available during the next call.

PROCEDURES AND FUNCTIONS

The procedure declaration and function declaration define program segments that are associated with an identifier, the procedure name or function name. These segments, or routines, may be invoked from elsewhere in the program merely by referencing the name of the desired routine. Values may be given to a routine for computation in two ways, through global variables and through arguments.

<proc part> ::= { <proc declaration> }
<proc declaration> ::= <external routine> | <internal routine>

ARGUMENTS

Arguments are passed to a procedure or function by enclosing the desired values in parentheses following the routine name in the invocation. The declaration of the routine includes a similar list, containing variable names. Within the routine for that invocation, the variable names in the declaration list, called formal parameters or simply formals, take on the values of the arguments given in the invocation, called actual arguments or simply actuals.

Arguments may be passed in either of two modes, call by value, or call by reference. A call by value argument is passed by creating a duplicate variable in the routine, and copying the argument's value into this duplicate. All uses of the formal in the routine use this duplicate, leaving the actual argument untouched. In this manner, the routine can use the formal freely and without regard to the sanctity of its corresponding actual, since the two are distinct. Actual arguments in call by value can be anything of the correct type. Integer formals must correspond to numeric expressions, which includes integer variables, array elements, expressions, function calls and the like. Array formals must correspond to array variables or array constants. String formals must correspond to string variables, any type of substring, string constants, or string literals.

A call by reference argument, on the other hand, merely informs the called routine where the actual is stored. References to the formal access the actual itself, so if the formal is changed, the actual is changed also. This allows values to be returned from the routine, but also introduces some restrictions: actual arguments must be integer variables, array elements, array variables, or string variables. Expressions, constants, literals, or substrings may not be used, since values cannot be returned in these.

The operation and environment of each routine should dictate which mode of argument passing is used. If neither mode appears to be superior in a given instance, then use call by value for integer variables because of its flexibility, and call by reference for arrays and strings because it requires no copying and therefore is faster.

<mode> ::= val | ref
<type> ::= int | array | string
<proc id> ::= <id>
<func id> ::= <id>

EXTERNAL PROCEDURE DECLARATIONS

An external procedure is a routine that is invoked from a program but not defined within that program. In order for the compiler to check the accuracy of actual argument lists in calls to external procedures, they must be declared. The declaration begins with the reserved word 'external', followed by one of the reserved words 'procedure' or 'function'. This is followed by the name of the routine and an abbreviated form of its formal parameter list, which declares the mode and type of each formal parameter, but not its name. Mode is either 'val', for call by value, or 'ref', for call by reference, and type is either 'int', 'array' or 'string'. If there is more than one parameter, the mode and type for each are separated by commas.

Since all procedures and functions must be declared before they are used, routines that call each other would be impossible to arrange correctly. For this reason, procedures and functions that have been declared external may be included later in the program as internal routines. The number, type and mode of parameters in both declarations must match.

Example:
```
external function sqrt(val int);
external procedure swap(ref int, ref int);
```

<external routine> ::= <external procedure> | <external function>
<external procedure> ::= external procedure <id> (<ext list>) ;
<external function> ::= external function <id> (<ext list>) ;
<ext list> ::= <null> | <mode> <type> { , <mode> <type> }

INTERNAL PROCEDURES AND FUNCTIONS

An internal procedure or function is one that is both declared and defined in the program, that is, its executable statements are included. The definition begins with one of the reserved words 'procedure' or 'function', followed by the routine name, followed by the formal parameter list. The formal list contains one entry for each formal, separated by commas. Each entry specifies the mode, type, and name of a formal parameter. The declaration is followed by the body of the routine.

Example: (of declaration only)
```
procedure swap(ref int x1, ref int x2);
function countchar(val string char, ref string target);
```

<internal routine> ::= <internal procedure> | <internal function>
<internal procedure> ::= procedure <id> (<int list>) ; <block> ;
<internal function> ::= function <id> (<int list>) ; <block> ;

```
<int list> ::= <null> | <mode> <type> <id> { , <mode> <type> <id> }
<int id> ::= <id>
<array id> ::= <id>
<string id> ::= <id>
```

BLOCKS AND STATEMENTS

<var declares> <statement list> end

All of the executable statements in a program, whether in the main program or in procedures or functions, fall within the scope of exactly one set of local declarations. This range is called a block. Each block begins with label declarations, then constant and variable declarations, followed by a list of executable statements. When the block is executed, statements are executed in order from the top, until a looping or 'if' statement is encountered. Looping statements (see chapter 8) cause repeated execution of groups of statements, and 'if' statements (see chapter 7) cause selection between groups of statements.

<block> ::= <label declares> <const declares>

ASSIGNMENT STATEMENT

The primary statement for getting things done is the assignment statement. The value represented by the right side of the equals sign is copied into the variable on the left side. There are three variations to this statement.

Numeric assignment is performed when a numeric variable appears on the left and a numeric expression on the right. The expression is evaluated, and its value is copied into the variable on the left.

Example:

```
increment=(sum/size)%(size-1)+1;
class=min(type1,type2);
```

<assgn stmt> ::= <num variable> = <num exp> ;

Numeric to string conversion is performed when a substring appears on the left, and a numeric expression on the right. The expression is evaluated and its low-order byte is converted into a string of length one. This string is then inserted into the specified substring according to the rules outlined under "String assignment", below.

Example:

```
char_var=int_var;
digit(3,3)=(number/100)%10+'0';
```

<assgn stmt> ::= <substr> = <num exp> ;

String assignment is performed when a substring is on the left side of the equals sign, and a string expression is on the right. The characters from the components of the right side are copied in order, one by one, into the substring on the left side until all the characters have been copied or the substring becomes full.

The length of the string variable on the left side is calculated

according to the following rules. If the substring was completely filled, the string's new length is either the position of the last character in the substring, or the string's old length, whichever is greater, and any characters that previously existed beyond the substring are not affected in any way. If the substring was not completely filled, then the string's new length is the position of the last character in the substring, and any characters that previously existed beyond that position are discarded.

The following examples illustrate these rules. Assume the variables x and y are both string variables with a maximum length of 10. The value of x is 'abcde', length 5, and the value of y is '1234567890', length 10. (Both strings are assumed to have these values prior to each example.)

STATEMENT	RESULT
y=x	y=='abcde', len==5
y=x(1)	(same as above)
y=x(2,4)	y=='bcd', len==3
y=x(4,7)	subscript error, x(6,7) undefined
y(3)=x	y=='12abcde', len==7
y(4,6)=x	y=='123abc7890', len==10
y(4,6)=x(1,2)	y=='123ab', len==5
y(4,6)=""	y=='123', len==3
y(1,15)=x	subscript error, y(11,15) beyond max length
x(5)=y	x=='abcd123456', len==10
x(6)=y	x=='abcde12345', len==10
x(7)=y	subscript error, x(6,6) undefined

<assgn stmt> ::= <substr> = <string exp> ;

RETURN STATEMENT

This statement causes the current routine to terminate after passing a numeric value back to the calling routine. The calling routine resumes with the statement following the call statement, or with the evaluation of the expression containing the function call. If no value is given in the return statement, a value of zero is returned. A return from the main program causes the low order byte of the return value to be given to the parent process after the program is terminated.

Example:

```
    return;
    return(count+1);
```

<return stmt> ::= return <return val> ;
<return val> ::= <null> | (<num exp>)

STOP STATEMENT

This statement causes the program to terminate after a message has been printed. The statement consists of the reserved word 'stop', optionally followed by a numeric expression in parentheses. If the expression is included, its value is printed as part of the stopping message, otherwise a zero is printed. The low order byte of the stop value is given to the parent process after the program is terminated.

Example:

```
stop;
stop(error_code);
```

```
<stop stmt> ::= stop <stop val> ;
<stop val> ::= <null> | ( <num exp> )
```

SIZE STATEMENT

This statement is used to declare the upper and lower limits on subscripts for arrays and maximum length for strings, and to allocate stack space for both. Strings and arrays must be sized before they can be used, otherwise, subscript errors will result. Array names in a size statement are followed by numeric expressions for the lower and upper subscript bounds, separated by a colon and enclosed in square brackets. String names are followed by a numeric expression for the maximum length enclosed in square brackets. Once an array or string is sized, its size cannot be changed. When a procedure or function terminates, the memory used by its local variables is released, so arrays and strings must be sized each time the procedure or function is called.

Example:

```
size string_var[25],array_var[lowest-5:2*highest];
```

```
<size stmt> ::= size <size list> ;
<size list> ::= <size element> { , <size element> }
<size element> ::= <array id> [ <num exp> : <num exp> ]
        | <string id> [ <num exp> ]
```

CALL STATEMENT

This statement is used to invoke procedures or functions. It consists of the name of the desired procedure or function, followed by the actual argument list. When the call statement is encountered, the named procedure or function is executed using the values of the actual arguments in the list for its formal parameters. If the called routine returns a value, it is ignored.

Example:

```
sort(tag_array, data_array);
```

```
<call stmt> ::= <proc id> ( <arg list> ) ;
```

```
        | <func id> ( <arg list> ) ;
<arg list> ::= <null> | <arg> { , <arg> }
<arg> ::= <num exp> | <array id> | <substr>
```

NULL STATEMENT

The null statement is useful in places where no action is desired but a statement is required by the grammar. The null statement is written as a semi-colon.

Example:

```
    ;

<null stmt> ::= ;
```

CONDITIONS AND THE IF STATEMENT

CONDITIONS

Conditions are formed by comparing two values or by combining the results of such tests. Conditions result in a value of true or false. Numeric expressions are compared by value, and strings are compared character by character until a difference is found. If one of the strings being compared is a proper prefix of the other one, the shorter string is considered to be less than the longer one.

Comparisons are made using one of the relational operators <, <=, ==, >=, >, !=. These relations may then be combined using the logical operators || (or) and && (and). The precedence of all of these operators is given below. Since the precedence of these operators is lower than the precedence of the arithmetic operators, expressions in a condition are always evaluated before the cndition is evaluatedo. It is an error to use parentheses to force a condition to be evaluated before an expression within the condition, since the condition has no numeric value.

Relational Operators

Prec	Type	Operator	True if left side is _____ right side
5	binary	<	less than
5	binary	<=	less than or equal to
5	binary	>	greater than
5	binary	>=	greater than or equal to
5	binary	==	equal to
5	binary	!=	not equal to

Logical Operators

Prec	Type	Operator	Meaning	True if _____, else false
6	unary	!	not	condition is false
7	binary	&&	and	both true
8	binary	||	or	either true

(Examples are shown with the 'if' statement, below.)

```
<condition> ::= <relation> | ( <condition> ) | ! <condition>
        | <condition> <binary logical oper> <condition>
<binary logical oper> ::= && | ||
<relation> ::= <num exp> <relational oper> <num exp>
        | <substr> <relational oper> <substr>
<relational oper> ::= < | <= | > | >= | == | !=
```

IF STATEMENT

This statement selects one of several groups of statements by testing a condition associated with each group, and executes the group. The conditions are tested in the order in which they appear; if a condition is true, the statements controlled by it are executed, otherwise, the next condition is tested. If none of the conditions are true, then the statements in the 'else' clause (if there is one) are executed. Note that an 'else' clause is required if any 'elseif' clauses are used.

After the selected group of statements is executed, control passes to the statement following the 'endif'; no other conditions in the 'if' are evaluated.

Example:

```
    if index>max_index then
         stop;
    endif;

    if x>y then
         largest=x;
    else
         largest=y;
    endif;

    if code==1 then
         type1();
    elseif code==2 then
         type2();
    elseif code==3 then
         type3();
    else
         error();
    endif;
```

```
<if stmt> ::= if <condition> then <stmt list> endif ;
         | if <condition> then <stmt list> <elseif part>
         else <stmt list> endif ;
<elseif part> ::= { elseif <condition> then <stmt list> }
```

LOOPING STATEMENTS

These statements cause groups of statements to be executed repeatedly depending on some condition or for some number of iterations. The 'next' and 'exit' statements may be used within any of the looping statements to modify or control the looping mechanism.

LABELS

Any looping statement may be tagged with a label. A label is an identifier that was declared a label in the declarations section. It appears just before the loop it is to tag, separated from it with a colon. Labels are optional, and are referenced only in 'exit' and 'next' statements.

(Examples appear with the 'exit' statement, below.)

<label> ::= <null> | <label id> :

EXIT AND NEXT STATEMENTS

These statements are meaningful only inside of some type of loop. The exit statement causes control to be given to the first statement after the loop. The next statement causes the loop to begin its next iteration. Both statements have three forms. The words 'exit' and 'next' by themselves refer to the innermost loop containing them.

If the keyword is followed by a simple expression, all of the loops enclosing the statement are numbered, starting with 1 for the inntermost loop and increasing by 1 for each shallower loop. The value of the expression then determines which loop the statement applies to. For example, 'exit 1' is the same as 'exit', and 'exit 3' exits the 3rd loop containing it, counting from the inside out. An expression with a value less than one or greater than the number of enclosing loops is an error.

The last form has a label identifier following the keyword. Statements in this form refer to the loop that was tagged with the given label. It is an error if no loops enclosing the statement are tagged with the specified label.

Example: (All cases of 'exit' apply to 'next' also.)

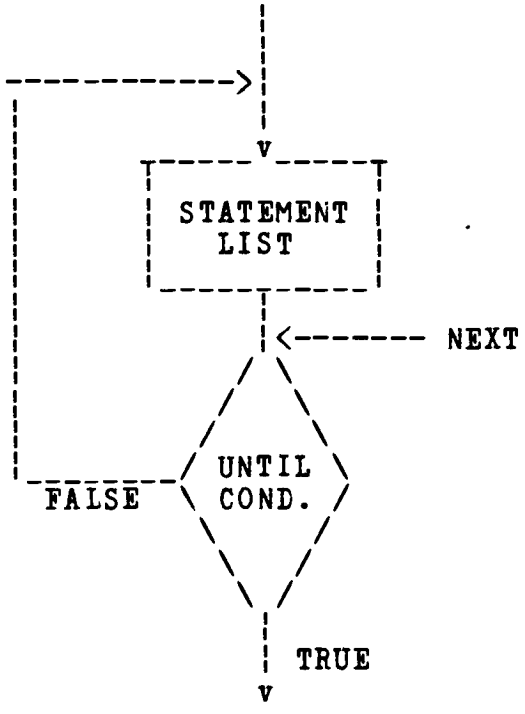
```
    exit;
    exit 3;
    exit pass2;
```

<exit stmt> ::= exit <loop designator> ;
<next stmt> ::= next <loop designator> ;
<loop designator> ::= <null> | <simple exp> | <label id>

REPEAT STATEMENT

This loop repeatedly executes a group of statements until a condition is satisfied. Since the condition is evaluated after each iteration, the statements in the loop are always executed at least once.

The 'next' statement causes the 'until' condition to be evaluated, skipping the rest of the statements in the loop.



Example:

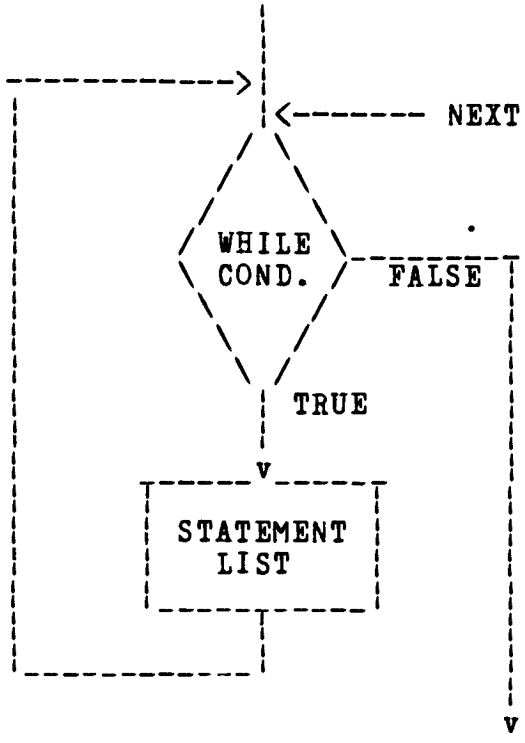
```
repeat
    column=column+1;
until input(column,column)!=" ";
```

<repeat stmt> ::= <label> repeat <stmt list> until <condition> ;

WHILE STATEMENT

The while statement executes a group of statements repeatedly as long as a condition is true. Since the condition is evaluated before each iteration, the statements in the loop are not executed at all if it is initially false.

The 'next' statement causes the 'while' condition to be evaluated, skipping the rest of the statements in the loop.



Example:

```
    while tag(pointer)<max do
        pointer=pointer+1;
    endwhile;
```

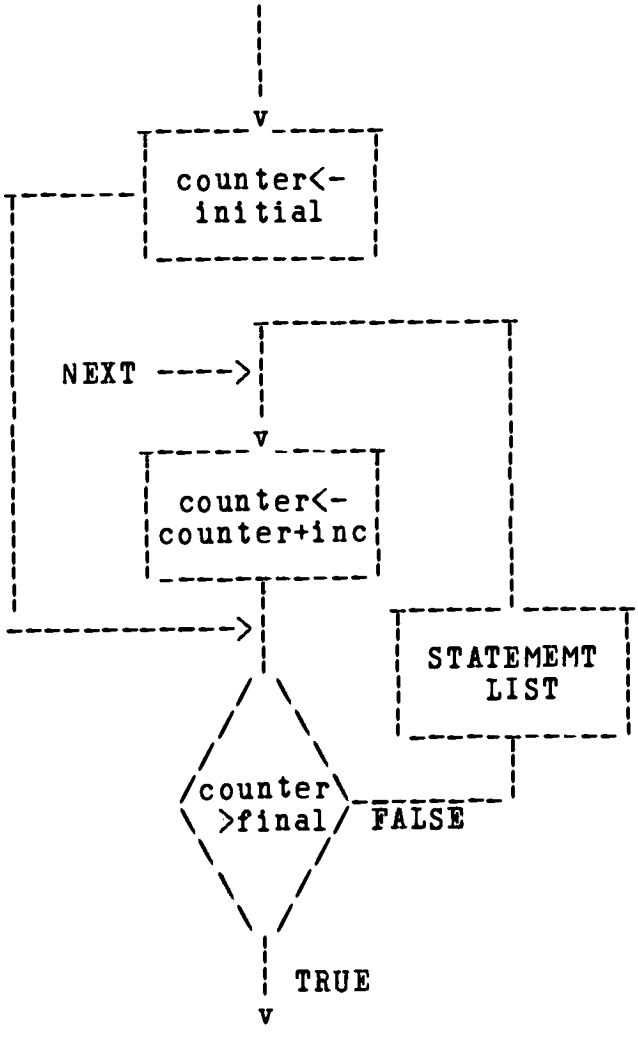
<while stmt> ::= <label> while <condition> do <stmt list> endwhile ;

FOR STATEMENT

This statement executes a group of statements under the control of a counter variable. The counter may be an integer variable or an array element. The initial and final values of the counter are specified with numeric expressions, and the increment may also be specified.

The loop executes as follows. First, all expressions in the statement are evaluated and their values are saved. The counter is given the value of the initial expression. If this value is not greater than the final expression, the statements in the loop are executed. The counter is then incremented by the specified value or, if no value is specified, by one, and compared again to the final value. The loop completes when the counter exceeds the final expression.

The counter may be modified by the programmer within the loop, and when the loop exits, the counter's value will be the value that terminated the loop.



Example:

```
    for pointer=2 to top by 2 do
        array(pointer)=tag(pointer)+tag(pointer-1);
    endfor;

    for char=1 to 80 do
        count((card(char)))=count((card(char)))+1;
    endfor;
```

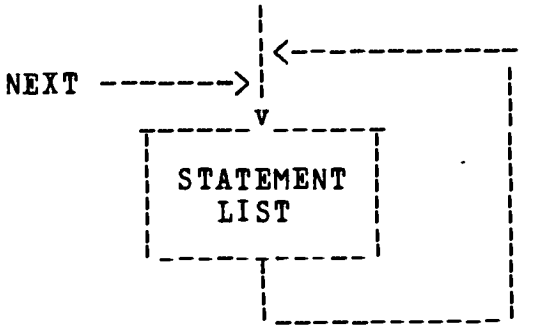
```
<for stmt> ::= <label> for <num var> = <initial exp> to
                    <final exp> <by clause> do <stmt list> endfor ;
<by clause> ::= <null> | by <increment exp>
<initial exp> ::= <num exp>
<final exp> ::= <num exp>
<increment exp> ::= <num exp>
```

DO STATEMENT

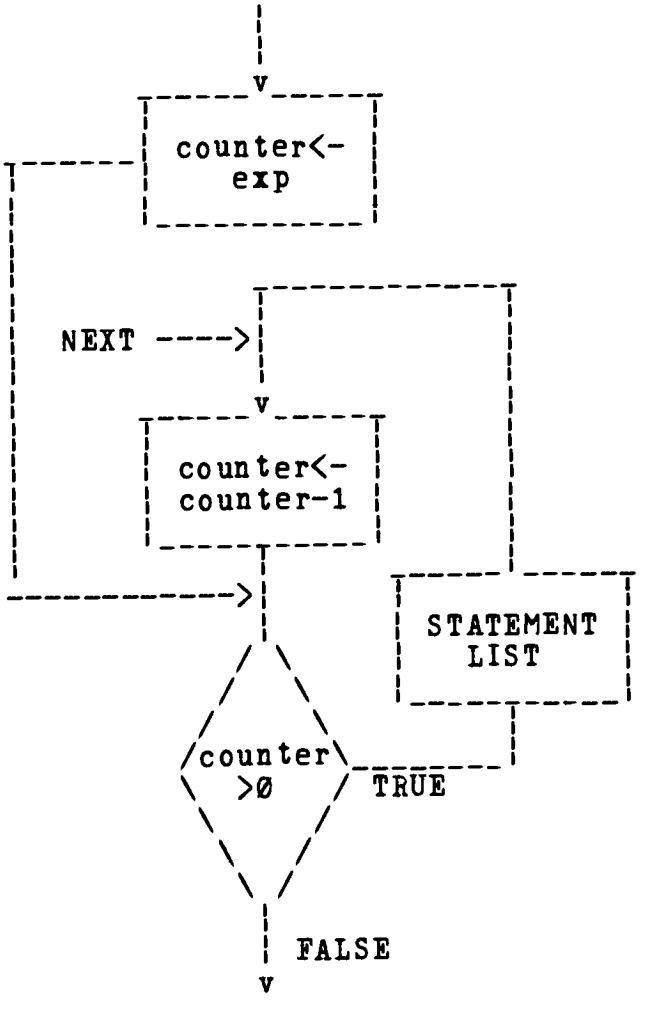
This statement is used to execute a group of statements infinitely ('do forever'), or under the control of a counter ('do times'). The counter in the do statement is not available to the statements within the loop, which allows the loop to run faster than a 'for' loop.

The 'next' statement causes control to pass to the first statement in a 'do forever' loop, and to the iteration part of a 'do times' loop.

do forever



do <num exp> times



Example:

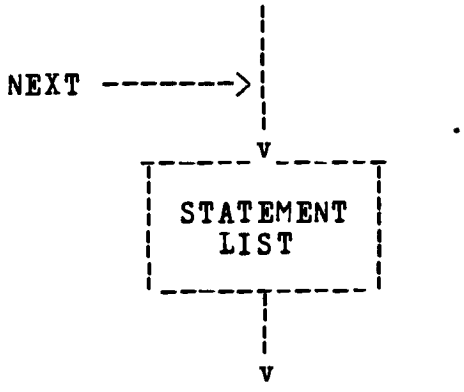
```
    do 20 times
        x=randomize(x);
    enddo;

    do forever
        read(5,-1,value);
        if value<0 then
            exit;
        endif;
        sum=sum+value;
    enddo;
```

<do stmt> ::= <label> do <count clause> <stmt list> enddo ;
<count clause> ::= <num exp> times | forever

LOOP STATEMENT

This statement is used to define the boundaries of a loop without imposing any looping mechanism upon it. Program control passes through a loop statement as if only the enclosed statements were there. Looping is accomplished by means of the 'exit' and 'next' statements. A loop statement is used when the iteration conditions are too complex to be expressed by any of the other looping statements.

The 'next' statement causes control to pass to the first statement in the loop.



Example:

```
phase1: loop
    read(1,-1,switch);
    if switch&1==Ø then
        exit phase1;
    endif;
    (. . .)
endloop;
```

<loop stmt> ::= <label> loop <stmt list> endloop ;

BUILT-IN PROCEDURES AND FUNCTIONS

There are several procedures and functions in the AL library to perform interface chores with the operating system and handle input/output. Since they are all pre-declared as special procedures and functions in the compiler, syntax errors will result if user routines with identical names are compiled.

function argc()

This function returns the number of arguments that were given when the main program was invoked by the user. If it is called from a procedure or function, it still counts the invocation arguments, not the arguments passed to that procedure or function.

function argv(val int arg#, ref string arg)

Argv takes two arguments, a numeric expression (arg#) and a string variable (arg). Arg# may have any value between zero and argc()-1, and selects one of the invocation arguments. The selected argument is returned in arg, and it's actual length is returned as the function value. The string variable passed for arg will automatically be sized to the actual length of the argument; if it was sized previously, the former contents are lost. Furthermore, the same variable may be used over and over in several argv calls. Similar to argc above, argv always refers to the invocation arguments.

function close(val int file#)

This function performs a close system call, using the value of file# as the file descriptor. UNIX error codes are returned negated as the function value, with zero indicating a successful close.

function create(val string filename, val int mode)

This function performs a create system call using the specified filename and mode. Any pathname may be specified in filename, provided that the user has access to all of the intermediate directories. If the file did not already exist, it is created with the specified mode, otherwise the file is truncated to zero length and its mode is unchanged. UNIX errors are returned negated, with zero indicating a successful create. Note that a created file is not opened by this function.

function delete(filename)

This function performs a delete system call using the specified filename. If the file exists and the user has the appropriate

access permissions, it will be deleted. UNIX errors are returned negated, with zero indicating a successful delete.

function exec(val string filename, val string args)

This function causes another program to be executed, using the contents of args for arguments. Arguments in the string should appear in the proper order separated by one or more spaces. By convention, the first argument to a program is usually its name. Exec initiates the program, waits for it to finish executing, obtains the low-order byte of the last stop or return value, and returns that byte to the calling program as the function value. If the specified filename cannot be executed, a fatal error is reported and the UNIX error code is returned to the calling program.

fatal

This routine is not callable by the user. It gains control when a fatal error has been detected and prints information that is helpful in debugging. All numbers printed out are in octal.

The first line of the printout shows the address from which fatal was called. This address is never within the user's own program, but if listings of the runtime support library are available, it will be useful.

If the debug option was selected at compile time, or if the error is an array subscript error, a message is printed that shows the line number in the source program where the error occured (remember that it is octal!).

A descriptive message is then printed that tells what the error was, followed by the current contents of registers 0-5.

A calling history is then printed, giving the names and local stack pointers (register 4) of each routine on the stack back to the main program. The global stack pointer, which never changes, is in register 5.

An IOT instruction is then executed to force a memory dump, which the user can autopsy with one of the UNIX debuggers. The program then terminates, sending back to its parent process the UNIX status code for an IOT trap.

function len(ref string str)

This function takes a string variable argument and returns a count of the number of characters currently in it, that is, its

current length.

function open(val string filename, val int mode)

This function opens the specified filename. Mode can be Ø for reading, 1 for writing, or 2 for reading and writing. The file number is returned as the function value. UNIX error codes are returned negated, so any returned value >=Ø indicates a successful open.

procedure print(?)

This procedure produces formatted output. It has no fixed argument structure since the arguments may differ from call to call.

The first argument may be a string variable (ref) or a numeric expression (val). If the string variable is specified, print will perform the specified conversions and place the results in the variable. If a numeric expression is given, print will use the value as a file descriptor and write the conversions to that file.

The second argument is a call by value string, which specifies the format for the conversions. The format is copied character by character to the desired output location until a field specifier is encountered, which is replaced by the value of a subsequent argument.

There are four field codes, each specifying a different conversion.

CODE	CONVERSION
b	include blank spaces
d	decimal integer
o	octal integer
s	string

The field codes are written in either of two ways, %x or %nx, where x is one of the codes above, and n is an integer literal. If n is specified, it determines the width of the field, and the converted value will either be padded with blanks or truncated to fit. If n is not specified, the field width will vary, and will be exactly as large as necessary to hold the converted value.

The remaining arguments correspond one to one with the field codes in the format, and must match in type: call by value strings for %s, call by value integers for %d or %o. %b does not correspond to any arguments, it simply creates a blank field of the specified width.

If the number of arguments does not match the format string, or

if an argument is the wrong type, a fatal error occurs.

function read(val int file#, val int address, ?)

This function performs input. The first argument is used as a file descriptor, and the second value specifies the address within the file where reading is to begin. If the address specified is negative, reading is done sequentially, and begins from wherever it left off before.

Additional arguments, which must all be call by reference, specify variables into which data is read. An integer variable will cause two bytes to be read, and an array variable will cause two bytes to be read for each of its elements. If the actual number of bytes read does not match the expected number, a fatal error is reported.

When a string variable is read, the function tries to read enough bytes to completely fill the variable. If not enough bytes are available, though, as is often the case when reading from a terminal, no error is reported. The length of the string will correctly reflect the number of bytes that were actually read into it.

If an error occurs during a read, the UNIX error code is returned (positive), otherwise, a zero is returned. An end of file condition will return a value of -1.

function write(val int file#, val int address, ?)

The write function operates exactly like the read function above, except for string variables. In a write, only the current length of the string is written, and if the number of bytes actually written does not match the expected number, a fatal error is reported.

OPERATIONS

Until the AL compiler becomes an integral portion of the UNIX system, it will reside in one of the author's directories, "/usr/accts/kar/al.d". The instructions that follow for using it are based upon this location. It is suggested that links be created in the user's directory to point to the compiler and the run-time routines so that the long pathnames need not be repeatedly typed.

INVOKING THE COMPILER

The compiler is invoked with the following shell command:

/usr/accts/kar/al.d/al [-sqxfdn] [s_1 ... s_n]

The pathname may be replaced by the name of an appropriate link in the user's directory for ease of operation.

S_1 through s_n are the filenames of the AL source programs to be compiled; any number of them may appear. If no filenames are specified, then whatever appears in the standard input is compiled. If two or more filenames appear in the same command, then the files are compiled one by one in the order in which they appear. As each file is being compiled, the names of the procedures and functions contained in it are typed to identify which routine caused any error messages that might be typed. Also, if two or more files were listed in the command line, the compiler types the name of each file just before it begins compiling it.

The s option causes a symbol summary to be printed for each file. This will include a table of all local symbols for each procedure or function in the file, and a table of all global symbols at the end of the file.

The q option causes the compiler to print the quads (intermediate code) that it produced just prior to the generation of assembly language statements. If the n option is also specified, three sets of quads are printed, the raw parser output, the output of the peephole optimizer, and the output of the expression optimizer.

If the x option is specified, the compiler performs a syntax check of the source programs, and doesn't produce any object code. This option is useful since the compiler executes faster when it doesn't have to generate code.

The f option causes generation of fast code for array references. Normally, complete subscript checking is done at run time for array references to guarantee that out-of-bounds subscripts will be detected. The fast option suppresses these checks, which results in object code for array references that is about 50% shorter. This option should be specified only when compiling production versions of fully tested and debugged programs.

When the d option is specified, any fatal error that occurs in the program will be identified with the source program line number of the offending statement, simplifying the debugging process considerably.

The n option causes each module of the compiler to announce itself when it is invoked. The parser announces itself by typing the name of the routine being compiled, and the following letters are used for the other modules.

LETTER	MODULE
P	Peephole optimizer, pass 1
2	Peephole optimizer, pass 2 (may appear several times)
O	Expression optimizer
G	Assembly code generation
A	Assembly of output file
M	Moving object file to final name

OBJECT CODE

Object code is produced for a routine only if the x option was not specified, and it compiles without errors. In that event, the compiler outputs an assembly language program that is equivalent to the source program, and invokes the UNIX assembler to produce an object file suitable for use by the loader. The names of these files are *.as and *.ld, where * represents the name of the routine itself. For example, if AL compiled a procedure called "sqrt", it would write the assembly program in the file "sqrt.as" and assemble that into the file "sqrt.ld". Each main program is written to files whose names begin with the source program's filename. If the main program in a file consists of only the null statement, the compiler will ignore it, and not produce object code for it.

LOADING OBJECT PROGRAMS

The UNIX loader is used to produce executable programs from the .ld files of the desired AL routines. The following command is used to produce an executable program from a group of .ld files:

```
ld /usr/accts/kar/al.d/init *.ld /usr/accts/kar/al.d/libk
```

It is suggested that the user create links in his own directory to the init and libk modules to avoid having to type the long pathnames for each load operation.

In the command, "*.ld" should be replaced by the names of the .ld files of the routines to be included in the program. Exactly one of these must be a main program.

If there were no loading errors (undefined symbols resulting from omission of one or more routines called for), the executable program will be placed in the file "a.out", which may be executed with the shell command:

a.out

If desired, the program may be moved to whatever other filename the user desires.