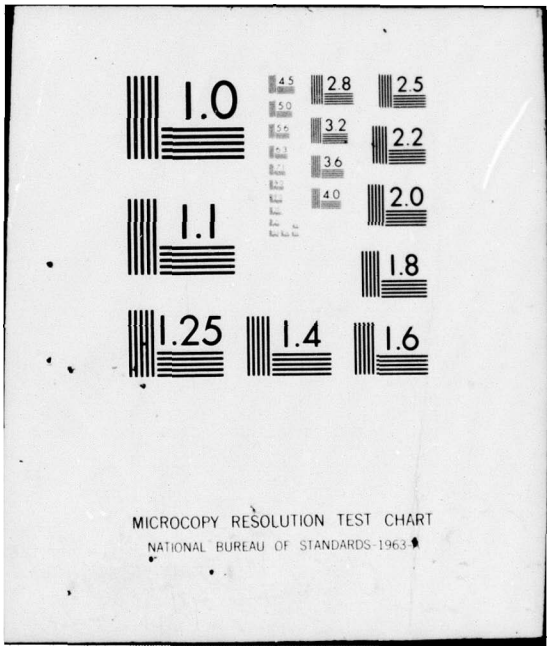
AD-A044 415 CALIFORNIA UNIV SANTA CRUZ INFORMATION SCIENCES F/G 9/2
HIGHLIGHTS OF L011.(U)
AUG 77 D ROSS, W M MCKEEMAN N00014-76-C-0682
UNCLASSIFIED TR-77-8-005 NL

1 OF 1
AD A044415

END
DATE FILMED
10-77
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

HIGHLIGHTS OF LO11

Daniel Ross and W. M. McKeeman

Technical Report No. 77-8-005

INFORMATION SCIENCES
UNIVERSITY OF CALIFORNIA
SANTA CRUZ, CALIFORNIA 95064

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: (14) TR-77-8-005
2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle): (6) HIGHLIGHTS OF LO11.
5. TYPE OF REPORT & PERIOD COVERED: (9) Technical rept.,
6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s): (10) Daniel Ross and W. M. McKeeman
8. CONTRACT OR GRANT NUMBER(s): (15) N00014-76-C-0682

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Board of Studies in Information Sciences
University of California
Santa Cruz, California 95064

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: (12) 21 p.

11. CONTROLLING OFFICE NAME AND ADDRESS
Office of Naval Research
Arlington, Virginia 22217

(11) 3

12. REPORT DATE: Aug[illegible], 1977
13. NUMBER OF PAGES: 19

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)
Office of Naval Research
University of California
553 Evans Hall
Berkeley, California 94720

15. SECURITY CLASS. (of this report): Unclassified
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

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

DDC
RECEIVED
SEP 21 1977
B

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
Evelyn system, LO11, PDP-11, compiler, system implementation language.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
This paper outlines the main features of a low-level language compiler for the PDP-11/20. Some of these features are applicable to the design of system implementation languages, or of high-level languages in general.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

Unclassified 410 350
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Highlights of LO11

Daniel Ross and
W. M. McKeeman

August 3, 1977

Board of Studies in Information Sciences
University of California
Santa Cruz, California 95064

Abstract -- This paper outlines the main features of a low-level language compiler for the PDP-11/20. Some of these features are applicable to the design of system implementation languages, or of high-level languages in general.

This research was supported in part by:
1. National Science Foundation contract GJ-483.
2. Office of Naval Research contract N00014-76-C-0682.
3. The Board of Studies in Information Sciences, UCSC.

Contents

Introduction

The PDP-11/20 is both host and target computer for the LO11 language and compiler. It is operated in a single-user, hands-on environment. LO11 is the system implementation language among a planned family of languages (MID11, HI11) and a single-user operating system, collectively called the Evelyn system. The LO11 compiler and the operating system have been implemented, but the other compilers have not.

Currently, LO11 is implemented as a single-pass, recursive descent compiler. However, the design of the language and the design of some aspects of the operating system reflect our intent to implement the Evelyn compilers using multiple-pass techniques based on an LR(1) parser.

Language facilities

LO11 is a member of the "PL" class of languages, based on the ideas introduced in Wirth's PL360 [6], and including BLISS [7], C [3], PL-11 [5], and LIL [2]. The language facilities for modular construction of programs, and for control of the sequence of execution, are high-level -- comparable to those of algebraic languages. The language facilities for data access are low-level -- approximately 1:1 with the hardware instructions of the target computer, comparable to an assembler.

Included in the modular facilities are the declarations of modules, procedures, and data, and access restriction statements and compile-time definitions. The modular facilities only indicate the existence of data declarations; the details of data declarations are included in the data facilities.

Included in the sequencing facilities are conditional statements (IF, THEN, ELSE, CASE), procedure calls and loop control (CALL, REPEAT, RETURN), and assignments.

Included in the data facilities are the details of data declarations, the PDP-11 instruction set, PDP-11 addressing modes and register usage, etc.

Modularity of the grammar

The modular, sequencing, and data facilities of LO11 are described by 3 separate subgrammars, whose union forms the complete grammar of LO11. The subgrammars are nearly disjoint; each pair of subgrammars shares only a few non-terminal symbols.

The modular and sequencing subgrammars may be included in the complete grammars of other Evelyn languages, but the data subgrammar is unique to the low-level data access of LO11.

Independence from syntactic sugar

The grammar defines a mapping from LO11 source statements to parse trees. Parse trees are mapped onto abstract syntax trees, which carry structural information to the code generators. Any change in the grammar would result in a change in the parse trees. We define syntactic sugar changes to be those changes in the source language which can be made not to cause changes in the abstract syntax trees.

We consider the LO11 language to be describable by any grammar which differs at most in its syntactic sugar from the grammar presented below. In practical terms, we have freed ourselves from concern over minor stylistic issues, without significantly increasing the burden on the compiler implementor.

Grammar_of_LO11

```
$ Subgrammar of the modular facility

Program = Module + ;
Module =
   Module_name ':'
      'MODULE' ';'
         Module_body_element +
      'END' ';' ;
Module_name = 'Identifier' ;
Module_body_element
   = ACCESS_statement
   | CONSTANT_statement
   | SPECIFY_statement
   | DECLARE_statement
   | Procedure
   ;
ACCESS_statement =
   'ACCESS' Accessed_names 'ONLY' 'FROM' Scopes ';' ;
Accessed_names = Accessed_name ( ',' Accessed_name ) * ;
Accessed_name = 'Identifier' ;
Scopes = Scope ( ',' Scope ) * ;
Scope = Entry_name ( | '(' Path 'THROUGH' Path ')' ) ;
Entry_name = 'Identifier' ;
Path = Direction Distance 'DISTANT' ;
Direction = 'CALLEES' | 'CALLERS' ;
Distance = Constant_expression | 'ARBITRARILY' ;
CONSTANT_statement =
   'CONSTANT' Name_of_constant '=' Constant_expression ';' ;
Name_of_constant = 'Identifier' ;
Constant_expression = Const_numeric_expression
   ( Const_binary_logic_op Const_numeric_expression ) * ;
Const_numeric_expression =
   Const_term ( Const_binary_add_op Const_term ) * ;
Const_term =
   Const_factor ( Const_binary_mult_op Const_factor ) * ;
Const_factor = Const_primary | Const_unary_op Const_factor ;
Const_primary = Value | '(' Constant_expression ')' ;
Const_binary_logic_op = '&' | '|' ;
Const_binary_add_op = '+' | '-' ;
Const_binary_mult_op = '*' | '/' | '//' ;
Const_unary_op = '¬' | '-' ;
SPECIFY_statement = 'SPECIFY' Specification ';' ;
DECLARE_statement = 'DECLARE' Declaration   ';' ;
Procedure =
   Entry_name ':'
      'PROCEDURE' ( | Formal_inputs ) ( | 'RETURNS' Formal_outputs ) ';'
        ( CONSTANT_statement | SPECIFY_statement | DECLARE_statement ) *
        Local_procedure *
        Statement +
      'END' ';' ;
Formal_inputs  = Formal_parameters ;
Formal_outputs = Formal_parameters ;
Formal_parameters = '(' Formal_name ( ',' Formal_name ) * ')' ;
Formal_name = 'Identifier' ;
Local_procedure =
   Entry_name ':'
      'PROCEDURE' ';'
         Statement +
      'END' ';' ;
```

```
$ Subgrammar of the sequencing facility

Statement = IF_statement | Simple_statement ;
IF_statement =
   'IF' Logical_expression
   'THEN' Simple_statement
   ( | 'ELSE' Statement ) ;
Logical_expression = Logical_term ( 'OR' Logical_term ) * ;
Logical_term = Logical_factor ( 'AND' Logical_factor ) * ;
Logical_factor = 'NOT' * ( Test | '(' Logical_expression ')' ) ;
Simple_statement
   = Assignment_statement
   | Group
   | CALL_statement
   | REPEAT_statement
   | RETURN_statement
   | Empty_statement
   ;
Assignment_statement = Assignment ';' ;
Group
   = ( | Label ':' )
      'DO' ';'
         Statement +
      'END' ';'
   | ( | Label ':' )
      'DO' 'CASE' Datum ';'
         ( 'CASE' Constants ':' Statement ) +
      'END' ';'
   ;
Label = 'Identifier' ;
Constants
   = Constant_expression ( ',' Constant_expression ) *
   | 'DEFAULT'
   ;
CALL_statement =
   'CALL'
      Entry_name ( | Actual_inputs )
      ( | 'RETURNS' Actual_outputs ) ';' ;
Actual_inputs  = Actual_parameters ;
Actual_outputs = Actual_parameters ;
Actual_parameters = '(' Actual_parameter ( ',' Actual_parameter ) * ')';
Actual_parameter = | Passed_parameter ;
REPEAT_statement = 'REPEAT' ( | 'Identifier' ) ';' ;
RETURN_statement = 'RETURN' ( | 'FROM' 'Identifier' ) ';' ;
Empty_statement = ';' ;
```

```
$ Subgrammar of the data facility

Specification
   = ( | Data_name ) '[' Size ']'
   |     Data_name   '[' Specification ( ',' Specification ) * ']'
   ;
Declaration
   = ( | Data_name ) '[' Size ']' ( | Initialization )
   |     Data_name   '[' Declaration ( ',' Declaration ) * ']'
   ;
Data_name = 'Identifier' ;
Size = Constant_expression ;
Initialization = '<-' Constant_expression ;
Passed_parameter = Datum ;
Assignment = Expression ;
Expression
   = Prefix Precision Datum
   | Term
   | Expression Infix Precision Term
   ;
Prefix = '-' | '¬' ;
Infix = '<-' | '+' | '-' | '|' | '&¬' | '*' | '/' | '//' | '\' | '\\'
   | '<<' | '<<<' | '>>' | '>>>' ;
Precision = | 'W' | 'B' ;
Term = Datum | '(' Expression ')' ;
Datum
   = Register
   | '@+' Register
   | '-@' Register
   | '@' '(' Register Value ')'
   | '@' Register
   | '@@+' Register
   | '-@@' Register
   | '@@' '(' Register Value ')'
   | Value
   | '@' Value
   | '@@' Value
   | 'NEGATIVE'
   | 'ZERO'
   | 'OVERFLOW'
   | 'CARRY'
   ;
Register = 'R' '.' Value ;
Value = Numeric_constant | Qualified_name ;
Numeric_constant
   = 'Constant'
   | Compile_time_function_name
      '(' Constant_expression ( ',' Constant_expression ) * ')'
   ;
Compile_time_function_name = 'Identifier' ;
Qualified_name = 'Identifier' ( '.' 'Identifier' ) * ;
Test
   = Comparand Relation Signed Precision Comparand
   | Datum '&' Precision Datum Relation Datum
   ;
Comparand = Datum | 'LAST_RESULT' ;
Relation = '=' | '¬=' | '<' | '<=' | '>=' | '>' ;
Signed = | 'U' ;
```

Program preparation

There is a simple compiler control language by which a user can name the files of LO11 source code which must be compiled (in the order they were named) to create any particular program. The compiler control language is not described in this paper.

Translation of a LO11 program, from source code to start of execution, is accomplished by a compilation step and a loading step. There is no linking step for separately compiled modules. Conceptually, each recompilation of a LO11 program is a compilation of the entire program. External to the LO11 language, there is an optimization mechanism embedded in the LO11 compiler, the Evelyn text editor, and the Evelyn operating system which permits the compiler to perform a partial recompilation of only those source files which have been changed since the last preceding partial recompilation. Our existing single-pass compiler cannot take advantage of the optimization mechanism, but the multiple-pass compiler will be able to.

Semantics, modular facility

Each module (a linguistic entity) is stored in a separate file (an operating system entity).

Access restriction statements are described in a separate section of this report.

Constant expressions are evaluated at compile time. Subsequently during compilation, the name of a constant may be used in any context in which the constant itself could be used.

Data must be specified or declared preceding its first use. Procedures should be declared preceding their first use, although the compiler does permit a few exceptions. The exceptions should be restricted to those procedures which are recursive. Execution begins at the last compiled procedure.

Constants, data specifications and declarations, and procedure declarations are local if they appear within the declaration of some procedure; otherwise they are global. Local procedures, or subroutines, are parameterless and cannot contain any declarations.

Global declared variables are allocated fixed storage. Local declared variables and parameters to procedures are allocated storage on the stack. Specified variables are not allocated storage at compile time, but the specifications are used in the implementation of access algorithms. Global declared variables can be initialized at compile time.

There is a clear distinction between input and output parameters to a procedure. Within the body of the procedure, all parameters may be used in exactly the same manner as local variables. The difference is that formal input parameters are initialized to their actual input values during the invocation of the procedure, and the contents of formal output parameters are copied to their actual output variables during the return from the procedure.

Semantics, sequencing facility

Case statements are implemented such that the case values must be small non-negative integers.

Program blocks are procedure bodies or group statements (DO, DO CASE). Repeat and return statements cause execution to go to the beginning or the end of program blocks. The identifier in a repeat or return statement must name the label of a program block which statically encloses the repeat or return statement. In the absence of an identifier, the innermost enclosing program block is repeated or returned from.

Semantics, data facility

Each procedure parameter consists of exactly one 2-byte word.

Specified or declared data is typeless. Only the size (in bytes) is designated. Access to the various fields of structured data is described by using qualified names.

Expression evaluation is performed from left to right, with no hierarchy except that imposed by parenthesization. The leftmost datum in an expression, or in a parenthesized subexpression, acts as the destination operand of every operation executed during expression evaluation. Examples:

LO11	LO11 equivalent	ALGOL equivalent
`@ V1 + 5;`		`V1 := V1 + 5;`
`@ V1 + @ V2 + 5;`	`@ V1 + @ V2; @ V1 + 5;`	`V1 := V1 + V2 + 5;`

Word or byte precision may be designated explicitly for each operation, or default precision may be used. If the leftmost operand is of size = 1 byte, then byte precision is the default. In all other cases, word precision is the default.

The following table shows the LO11 operators, and the set of PDP-11/20 instructions which are used to implement the operators. The exact PDP-11/20 instruction used depends on the value of the operand to the right of the operator, and depends on the precision.

	Operator	PDP-11/20 instructions
prefix	-	NEG
	¬	COM
infix	<-	MOV,CLR
	+	ADD,ADC,INC
	-	SUB,SBC,DEC
	|	BIS
	&¬	BIC
	*	macro: multiply in Extended Arithmetic Element (EAE) peripheral hardware
	/	macro: divide in EAE, retain quotient
	//	macro: divide in EAE, retain remainder
	\	macro: reverse divide in EAE, retain quotient
	\\	macro: reverse divide in EAE, retain remainder
	<<	multiple ALS
	<<<	multiple ROL
	>>	multiple ARS
	>>>	multiple ROR

As an aside, the left operand was chosen to be the destination rather than the right operand, only so that the "-" operator would be intuitive.

The following table shows datum syntax and the usual PDP-11/20 addressing mode. In some context-dependent special cases, the compiler's optimizers will change the addressing mode or the instruction. "@" means "contents of".

Datum	Usual addressing mode
Register	Register
'@+' Register	Autoincrement
'-@' Register	Autodecrement
'@' '(' Register Value ')'	Indexed
'@' Register	Register deferred
'@@+' Register	Autoincrement deferred
'-@@' Register	Autodecrement deferred
'@@' '(' Register Value ')'	Indexed deferred
Value	Immediate
'@' Value	Absolute (for global variables), or Indexed by the stack register (for parameters and local variables)
'@@' Value	Relative deferred (for global variables), or Indexed by the stack register deferred (for parameters and local variables)
'NEGATIVE','ZERO','OVERFLOW','CARRY'	Condition code bits

In a formal sense, a register is identified by its numeric value. But in practice, a named constant is used for the value of the register, so the register can be identified by the functional role for which it is used. Example:

```
DECLARE BUFFER [72];
CONSTANT CURSOR = 3;
R.CURSOR <- BUFFER;
```

If an expression consists of a single term of constant value, that constant value is compiled in line with the surrounding code. This provides a mechanism for implementing some PDP-11/20 instructions (HALT, RESET, WAIT, RTI, BPT) which most easily are described as named constants. It also provides an escape mechanism for generating code not otherwise describable in LO11. For example, the code implementing the LO11 compiler contains (exactly) one GO TO which violates the nesting rules of program blocks.

If an expression consists of a single variable term, that variable is accessed with a PDP-11/20 TST or TSTB instruction.

All compile-time values are constant. A quoted constant 'Constant' is a constant recognized by the compiler's scan mechanism. A 'Constant' may be:

- an unsigned decimal integer in the interval 0 <= integer <= 32767
- '#' followed by an unsigned octal integer in the interval
  #0 <= integer <= #177777
- '%' followed by an unsigned binary integer in the interval
  %0 <= integer <= %1111111111111111
- a quoted string of 0, 1, or 2 characters, delimited by "" or by '' (longer quoted strings may be used only for initializing global declared data)

The compiler has the built-in compile-time functions:

SIZE(Qualified_name)	Value = size (in bytes) of specified or declared data.
END_LOC(Qualified_name)	Value = 1st loc. after end of specified or global declared data. All specified data is assumed to start at loc. = 0.
BITS(Constant_expression)	Value = bit count of a mask for the binary representation of the value of the expression.
CONST_VALUE(Constant_expression)	Value = value of the expression. Permits a constant expression to be used as a single numeric constant, without having to assign a name to the expression using a constant statement.

Unmodified relations are determined between 15-bit signed comparands. Relations modified by 'U' are determined between 16-bit unsigned comparands.

The comparand LAST_RESULT utilizes the PDP-11/20 condition codes which were set as a side effect of a previously executed instruction. If LAST_RESULT is used as a comparand, the other comparand must be a (constant) value = 0.

A test described by
   Datum '&' Precision Datum Relation Datum
is a comparison of bits, implemented by the PDP-11/20 BIT or BITB instruction. The 3rd datum described above must be a (constant) value = 0.

Constraints on user programs

In order that the side effects of any stated computation can be understood completely, LO11 clearly grants authority for controlling the target computer.

The PDP-11/20 hardware interrupt mechanism, and the high-level compiled code (CALL, REPEAT, RETURN, IF, THEN, ELSE, CASE) can change the run-time contents of the stack register (R.6) and the program counter (R.7). The user's program cannot change the contents of these registers by explicit assignment statements.

The user's assignment statements can change the run-time contents of the other registers (R.0 to R.5) and the declared variables. The high-level compiled code does not change these, except that formal output parameters are copied into actual output parameters during the return from a procedure.

Partly for these reasons, LO11 does not include:

- hierarchical evaluation of expressions
- automatic loop control statements (FOR, WHILE, UNTIL)
- subscripted variables
- I/O statements
- memory management statements

In most computer environments, user programs request "system" or "monitor" services by executing hardware trap instructions. Any additional parameters for the trap instructions are passed on the stack. LO11 forbids the user to expand or contract the stack by explicit assignment statements, thereby precluding effective use of the trap instructions. In our single-user, hands-on environment, there is no need for hardware protection against the actions of user programs. Since each compilation is (in principle) a total compilation, there is no need to distinguish "system" procedures from "user" procedures. System services are requested by conventional procedure calls.

Access restriction statements

Access restriction statements grant to specific procedures the use of the names of global data, global named constants, or global procedures. This is the protection mechanism which LO11 substitutes for the nesting facilities commonly found in high-level languages. Access to the name of declared data (for example) may be granted to the procedures whose entry names appear in the scope of the access restriction statement.

If a path also appears, access also may be granted to the callers or callees of the named procedure. The designated distance is a count of the number of calls of calls of procedures, leading to or from the named procedure.

Unrestricted access is permitted to global data, global named constants, and global procedures whose names do not appear in access restriction statements.

Access restriction statements have not been implemented in the single-pass LO11 compiler.

Example_program

```
FOUR_FUNCTION_INFIX_CALCULATOR_OF_OCTAL_INTEGERS:
   MODULE;

   $This module is a complete, self-contained program which emulates the
   $operation of a four-function calculator.  The principal operational
   $characteristics of this program are:
   $1.  Integer I/O is in octal.
   $2.  I/O is performed on a TTY.
   $3.  Expressions are evaluated from left to right with no hierarchy,
   $    using infix operators and ending with "=".

   $This program accepts only the keys:
   $ the digits 0 to 7
   $ + - * / =
   $ C                    meaning cancel this computation

   $Names for PDP-11 hardware constants
   CONSTANT BYTE = 1;
   CONSTANT WORD = 2;

   $TOP_OF_THE_STACK must be the first declaration of the program.
   $During initialization, the stack is made to extend downward in
   $memory from TOP_OF_THE_STACK.
   DECLARE TOP_OF_THE_STACK     [0];

   $Type out 1 character to the TTY, synchronous I/O
TTY_OUT:
   PROCEDURE (CHAR);
      $Unibus addresses of TTY output registers
      CONSTANT TTY_OUT_CSR      = #177564;
      CONSTANT TTY_OUT_DATA     = TTY_OUT_CSR + WORD;
      $Idle until the TTY has finished any previous typing
      IF @ TTY_OUT_CSR >=B 0 THEN REPEAT;
      @ TTY_OUT_DATA <- @ CHAR;
   END;

TYPE_OUT_CR_LF:
   PROCEDURE;
      CONSTANT ASCII_CAR_RET    = #15;
      CONSTANT ASCII_LINE_FEED  = #12;
      CALL TTY_OUT(ASCII_CAR_RET);
      CALL TTY_OUT(ASCII_LINE_FEED);
   END;
```

```
   $Read and echo 1 character typed in from the TTY, synchronous I/O
TTY_IN:
   PROCEDURE RETURNS (CHAR);
      $Unibus addresses of TTY input registers
      CONSTANT TTY_IN_CSR       = #177560;
      CONSTANT TTY_IN_DATA      = TTY_IN_CSR + WORD;
      $Mask for the complement of a 7-bit ASCII character
      CONSTANT ASCII_MASK_COMPLEMENT = #177600;
      $Idle until a character has been typed in
      IF @ TTY_IN_CSR >=B 0 THEN REPEAT;
      @ CHAR <- @ TTY_IN_DATA &¬ ASCII_MASK_COMPLEMENT;
      $Echo the character
      CALL TTY_OUT(@ CHAR);
   END;

TYPE_OUT_OCTAL:
   PROCEDURE (INTEGER);
      $All integers are typed out as the same number of octal digits
      CONSTANT DIGITS_PER_OCTAL_INTEGER          = 6;
      CONSTANT OCTAL_MASK_COMPLEMENT             = #370; $One byte only
      CONSTANT TOP_3_BITS                        = #160000;
      DECLARE COUNT_OF_DIGITS_REMAINING          [WORD];
      DECLARE BUFFER_FOR_DIGITS [DIGITS_PER_OCTAL_INTEGER];
      $Designate the registers used in this procedure
      CONSTANT ASCII_DIGIT                       = 0;
      CONSTANT BUF_CURSOR                        = 1;
      $The digits are generated in the reverse of the order in which
      $they are typed out
      R.BUF_CURSOR <- BUFFER_FOR_DIGITS;
      $Generate the digits
      @ COUNT_OF_DIGITS_REMAINING <- DIGITS_PER_OCTAL_INTEGER;
      DO;
         $Extract an octal digit, but do not convert it to ASCII yet
         @  R.BUF_CURSOR <-B @ INTEGER;
         @+ R.BUF_CURSOR &¬B OCTAL_MASK_COMPLEMENT;
         $Rotate right, discarding any carry bits that are rotated in
         @  INTEGER      >>> 3 &¬ TOP_3_BITS;
         @  COUNT_OF_DIGITS_REMAINING - 1;
         IF LAST_RESULT > 0 THEN REPEAT;
      END;
      $Type out the digits
      @ COUNT_OF_DIGITS_REMAINING <- DIGITS_PER_OCTAL_INTEGER;
      DO;
         $Expand the byte to a full word and convert to an ASCII octal
         $digit
         R.ASCII_DIGIT <-B -@ R.BUF_CURSOR + "0";
         CALL TTY_OUT(R.ASCII_DIGIT);
         @ COUNT_OF_DIGITS_REMAINING - 1;
         IF LAST_RESULT > 0 THEN REPEAT;
      END;
   END;

   $Table of operator characters
   DECLARE OPERATORS [
         [BYTE] <- "+",
         [BYTE] <- "-",
         [BYTE] <- "*",
         [BYTE] <- "/",
      EQ [BYTE] <- "=" ];
```

```
CALCULATOR:
   PROCEDURE;
      CONSTANT TOP_3_BITS                 = #160000;
      $Designate the registers used in this procedure
      CONSTANT CHAR                       = 0;
      CONSTANT OPERAND                    = 1;
      CONSTANT ACCUMULATOR                = 2;
      CONSTANT INDEX_OF_LAST_OPERATOR     = 3;
      CONSTANT INDEX_OF_NEXT_OPERATOR     = 4;
      $Unibus addresses of the registers in the Extended Arithmetic
      $Element (EAE) peripheral hardware device for performing integer
      $multiplication and division
      CONSTANT EAE_DIVIDE                 = #177300;
      CONSTANT EAE_AC                     = EAE_DIVIDE + WORD;
      CONSTANT EAE_MQ                     = EAE_DIVIDE + 2*WORD;
      CONSTANT EAE_MULTIPLY               = EAE_DIVIDE + 3*WORD;
CANCEL:
      DO;
         R.ACCUMULATOR                 <- 0;
         R.INDEX_OF_NEXT_OPERATOR      <-
            CONST_VALUE(OPERATORS.EQ - OPERATORS);
         CALL TYPE_OUT_CR_LF;
         DO;                                 $Next operand
            R.OPERAND                  <- 0;
            R.INDEX_OF_LAST_OPERATOR <- R.INDEX_OF_NEXT_OPERATOR;
NEXT_CHAR:
            DO;
               CALL TTY_IN RETURNS (R.CHAR);
               IF R.CHAR = "C" THEN REPEAT CANCEL;
SEARCH_FOR_OPERATOR:
               DO;
                  R.INDEX_OF_NEXT_OPERATOR <- OPERATORS;
                  DO;
                     IF @* R.INDEX_OF_NEXT_OPERATOR =B R.CHAR
                     THEN RETURN FROM SEARCH_FOR_OPERATOR;
                     IF R.INDEX_OF_NEXT_OPERATOR <U END_LOC(OPERATORS)
                     THEN REPEAT;
                  END;
                  $If execution passes through here, the character just
                  $typed in is not an operator.
                  $Convert from ASCII to an octal digit
                  R.CHAR - "0";
                  $Verify that the character is an octal digit
                  IF R.CHAR >U 7
                  THEN
                     DO;
                        CALL TTY_OUT("?");
                        $Ignore the character
                        REPEAT NEXT_CHAR;
                     END;
                  $Test for overflow
                  IF R.OPERAND & TOP_3_BITS ¬= 0
                  THEN
                     DO;
                        CALL TTY_OUT("¬");
                        REPEAT CANCEL;
                     END;
                  $Accumulate the octal digit into the operand
                  R.OPERAND << 3 | R.CHAR;
                  REPEAT NEXT_CHAR;
               END;
```

```
            $Determine the relative loc. of the character in the
            $table of operator characters
            R.INDEX_OF_NEXT_OPERATOR - CONST_VALUE(OPERATORS + BYTE);
            DO CASE R.INDEX_OF_LAST_OPERATOR;
         CASE 0:                                                $ +
               R.ACCUMULATOR + R.OPERAND;
               $Ignore possible overflow
         CASE 1:                                                $ -
               R.ACCUMULATOR - R.OPERAND;
               $Ignore possible overflow
         CASE 2:                                                $ *
               DO;
                  R.ACCUMULATOR * R.OPERAND;
                  $Test for overflow in the EAE peripheral hardware
                  IF @ EAE_MQ >= 0 AND @ EAE_AC ¬= 0          OR
                     @ EAE_MQ <  0 AND @ EAE_AC ¬= #177777
                  THEN
                     DO;
                        CALL TTY_OUT("¬");
                        REPEAT CANCEL;
                     END;
               END;
         CASE 3:                                                $ /
               $Test for divide by 0
               IF R.OPERAND = 0
               THEN
                  DO;
                     CALL TTY_OUT("¬");
                     REPEAT CANCEL;
                  END;
               ELSE R.ACCUMULATOR / R.OPERAND;
         CASE 4:                                                $ =
               R.ACCUMULATOR <- R.OPERAND;
            END;
            $Test whether "=" just was typed in
            IF R.INDEX_OF_NEXT_OPERATOR =
               CONST_VALUE(OPERATORS.EQ - OPERATORS)
            THEN
               DO;
                  CALL TYPE_OUT_OCTAL(R.ACCUMULATOR);
                  REPEAT CANCEL;
               END;
         END; $of NEXT_CHAR
         REPEAT;
      END; $of next operand
   END; $of CANCEL
END; $of CALCULATOR
```

```
   $Execution starts here.  This must be the last compiled procedure,
   $and it must not have any parameters or local variables.
MAIN:
   PROCEDURE;
      CONSTANT RESET_INSTRUCTION = #5;
      RESET_INSTRUCTION;
      $Initialize the stack register.  This must be done once at the
      $start of every program, before any procedures can be called.  It
      $violated the normal LO11 rules, so a kluge is necessary.
      #012706;                              $ MOV #Next,R.STACK
      TOP_OF_THE_STACK;
      CALL CALCULATOR;
      $Procedure CALCULATOR never returns.
   END;

   END;
```

Characteristics of the compiler

The single-pass recursive descent compiler maintains a string file, containing one copy of each identifier and each quoted string that has been scanned. In a typical large compilation (of the compiler itself, or of the operating system), the string table occupies 50000 bytes. This necessitates storing the string table on a disk file. There are 4 buffers in main memory, each holding one block of the string table. Also in main memory is a 5-byte reference to each entry in the table.

The symbol table is entirely in main memory. It is accessed serially, in order to avoid the extra storage that would be required if any additional structure were imposed on the symbol table.

As is typical of recursive descent compilers, the code for parsing, emitting, and optimizing is intermingled, with no clean separation of function.

The speed of compilation is about 2400 lines per minute. Most of the time is spent performing disk I/O for the string table.

It has become customary to compare the size and speed of code produced by low-level compilers, with code for the same program produced by some mythical programmer writing in macro assembler language. Since such comparisons can be only wild guesses at best, it is more meaningful to indicate what further improvements could be made by hand optimization of the code generated by an actual compilation. LO11 compiled code is about 12% larger and about 20% slower than the same program would be after careful hand optimization of the code. Almost all the difference is due to the single-pass operation of the compiler, which forces compilation of JMP instructions for most forward branching, where the shorter and faster BR instructions would suffice.

Experiences

LO11 has been used to implement the LO11 compiler itself, the Evelyn operating system, and several other large programs. The only other language alternative for implementing systems would have been macro assembler language, under the constraints of our hardware configuration.

LO11 has been very satisfactory for creating PDP-11/20 system programs. However, we have a Vector-General, Inc. graphics display connected to our PDP-11/20. The VG display has its own special-purpose CPU, which shares the PDP-11/20 Unibus. LO11 never was intended to describe the instruction set of the VG display CPU. It is much more difficult to write VG programs in LO11 than it would be in macro assembler language. Writing programs in LO11 for the combination of the PDP-11/20 and VG display is feasible only because very few VG instructions are compiled; most VG instructions are created dynamically by the PDP-11/20 at run time.

The LO11 compiler was much more difficult to create than we had anticipated. All during its development we were limited by the size of the computer's memory (28K words - the maximum possible for a PDP-11/20) and sometimes we were limited by the size of disk storage (1/2M bytes, later expanded to 1M bytes).

References

1. McKeeman, W. M., and Ross, Daniel; LO11 - A member of the Evelyn family of programming languages for the PDP-11; Proceedings of the Hawaii International Conference on System Sciences (Jan. 1975).
2. Plauger, P. J.; A little implementation language; ACM SIGPLAN Notices, vol. 11 number 4 (April 1976).
3. Ritchie, D. M.; C reference manual; unpublished memorandum, Bell Telephone Laboratories (1973).
4. Ross, Daniel; The Evelyn system deconfusion book, document 22.1; Board of Studies in Information Sciences, Univ. of Calif., Santa Cruz, Calif. 95064 (continuously updated, in machine-readable form).
5. Russell, Robert D.; Experience in the design, implementation, and use of PL-11, a programming language for the PDP-11; ACM SIGPLAN Notices, vol. 11 number 4 (April 1976).
6. Wirth, Niklaus; PL360 - A programming language for the 360 computers; Journal of the ACM, vol. 15 number 1 (1968).
7. Wulf, William, Johnsson, Richard K., Weinstock, Charles B., Hobbs, Steven O., and Geschke, Charles M.; The Design of an Optimizing Compiler; American Elsevier Publishing Company, Inc.; New York, N. Y. (1975).

OFFICIAL DISTRIBUTION LIST

Contract N00014-76-C-0682

Defense Documentation Center
Cameron Station
Alexandria, VA 22314
12 copies

Office of Naval Research
Information Systems Program
Code 437
Arlington, VA 22217
2 copies

Office of Naval Research
Code 102IP
Arlington, VA 22217
6 copies

Office of Naval Research
Code 200
Arlington, VA 22217
1 copy

Office of Naval Research
Code 455
Arlington, VA 22217
1 copy

Office of Naval Research
Code 458
Arlington, VA 22217
1 copy

Office of Naval Research
Branch Office, Boston
495 Summer Street
Boston, MA 02210
1 copy

Office of Naval Research
Branch Office, Chicago
536 South Clark Street
Chicago, IL 60605
1 copy

Office of Naval Research
Branch Office, Pasadena
1030 East Green Street
Pasadena, CA 91106
1 copy

New York Area Office
715 Broadway - 5th Floor
New York, NY 10003
1 copy

Naval Research Laboratory
Technical Information Division
Code 2627
Washington, DC 20375
6 copies

Dr. A. L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps (CodeRD
Washington, D. C. 20380
1 copy

Naval Electronics Laboratory Center
Advanced Software Technology Division
Code 5200
San Diego, CA 92152
1 copy

Mr. E. H. Gleissner
Naval Ship Research & Development Cente
Computation and Mathematics Department
Bethesda, MD 20084
1 copy

Captain Grace M. Hopper
NAICOM/MIS Planning Branch (OP-916D)
Office of Chief of Naval Operations
Washington, D. C. 20350
1 copy

Mr. Kin B. Thompson
Technical Director
Information Systems Division (OP-911G)
Office of Chief of Naval Operations
Washington, D. C. 20350
1 copy