-- --

The Programming Language EFL

Stuart I. Feldman

Fortran
Preprocessors
Ratfor

ABSTRACT

EFL is a clean, general purpose computer language intended to encourage portable programming. It has a uniform and readable syntax and good data and control flow structuring. EFL programs can be translated into efficient Fortran code, so the EFL programmer can take advantage of the ubiquity of Fortran, the valuable libraries of software written in that language, and the portability that comes with the use of a standardized language, without suffering from Fortran's many failings as a language. It is especially useful for numeric programs. The EFL language permits the programmer to express complicated ideas in a comprehensible way, while permitting access to the power of the Fortran environment. EFL can be viewed as a descendant of B. W. Kernighan's Ratfor [1]; the name originally stood for 'Extended Fortran Language'. The current version of the EFL compiler is written in portable C.

1. INTRODUCTION

1.1. Purpose

EFL is a clean, general purpose computer language intended to encourage portable programming. It has a uniform and readable syntax and good data and control flow structuring. EFL programs can be translated into efficient Fortran code, so the EFL programmer can take advantage of the ubiquity of Fortran, the valuable libraries of software written in that language, and the portability that comes with the use of a standardized language, without suffering from Fortran's many failings as a language. It is especially useful for numeric programs. Thus, the EFL language permits the programmer to express complicated ideas in a comprehensible way, while permitting access to the power of the Fortran environment.

1.2. History

EFL can be viewed as a descendant of B. W. Kernighan's Ratfor [1]; the name originally stood for 'Extended Fortran Language'. A. D. Hall designed the initial version of the language and wrote a preliminary version of a compiler. I extended and modified the language and wrote a full compiler (in C) for it. The current compiler is much more than a simple preprocessor: it attempts to diagnose all syntax errors, to provide readable Fortran output, and to avoid a number of niggling restrictions. To achieve this goal, a sizable two-pass translator is needed.

1.3. Notation

In examples and syntax specifications, **boldface** type is used to indicate literal words and punctuation, such as **while**. Words in *italic* type indicate an item in a category, such as an *expression.* A construct surrounded by double brackets represents a list of one or more of those items, separated by commas. Thus, the notation *item*

could refer to any of the following:

item
item, *item*
item, *item*, *item*

The reader should have a fair degree of familiarity with some procedural language. There will be occasional references to Ratfor and to Fortran which may be ignored if the reader is unfamiliar with those languages.

2. LEXICAL FORM

2.1. Character Set

The following characters are legal in an EFL program:

<table>
<tr><td>letters</td><td>a b c d e f g h i j k l m</td></tr>
<tr><td></td><td>n o p q r s t u v w x y z</td></tr>
<tr><td>digits</td><td>0 1 2 3 4 5 6 7 8 9</td></tr>
<tr><td>white space</td><td>blank tab</td></tr>
<tr><td>quotes</td><td>´ "</td></tr>
<tr><td>sharp</td><td>#</td></tr>
<tr><td>continuation</td><td>_</td></tr>
<tr><td>braces</td><td>{ }</td></tr>
<tr><td>parentheses</td><td>( )</td></tr>
<tr><td>other</td><td>, ; : . + – * /</td></tr>
<tr><td></td><td>= < > & ~ | $</td></tr>
</table>

Letter case (upper or lower) is ignored except within strings, so '**a**' and '**A**' are treated as the same character. All of the examples below are printed in lower case. An exclamation mark ('**!**') may be used in place of a tilde ('**~**'). Square brackets ('[' and ']') may be used in place of braces ('{' and '}').

2.2. Lines

EFL is a line-oriented language. Except in special cases (discussed below), the end of a line marks the end of a token and the end of a statement. The trailing portion of a line may be used for a comment. There is a mechanism for diverting input from one source file to another, so a single line in the program may be replaced by a number of lines from the other file. Diagnostic messages are labeled with the line number of the file on which they are detected.

2.2.1. White Space

Outside of a character string or comment, any sequence of one or more spaces or tab characters acts as a single space. Such a space terminates a token.

2.2.2. Comments

A comment may appear at the end of any line. It is introduced by a sharp (#) character, and continues to the end of the line. (A sharp inside of a quoted string does not mark a comment.) The sharp and succeeding characters on the line are discarded. A blank line is also a comment. Comments have no effect on execution.

2.2.3. Include Files

It is possible to insert the contents of a file at a point in the source text, by referencing it in a line like

include joe

No statement or comment may follow an **include** on a line. In effect, the **include** line is replaced by the lines in the named file, but diagnostics refer to the line number in the included file. **Include**s may be nested at least ten deep.

2.2.4. Continuation

Lines may be continued explicitly by using the underscore (**_**) character. If the last character of a line (after comments and trailing white space have been stripped) is an underscore, the end of line and the initial blanks on the next line are ignored. Underscores are ignored in other contexts (except inside of quoted strings). Thus

1_000_000_
000

equals 10^9.

There are also rules for continuing lines automatically: the end of line is ignored whenever it is obvious that the statement is not complete. To be specific, a statement is continued if the last token on a line is an operator, comma, left brace, or left parenthesis. (A statement is not continued just because of unbalanced braces or parentheses.) Some compound statements are also continued automatically; these points are noted in the sections on executable statements.

2.2.5. Multiple Statements on a Line

A semicolon terminates the current statement. Thus, it is possible to write more than one statement on a line. A line consisting only of a semicolon, or a semicolon following a semicolon, forms a null statement.

2.3. Tokens

A program is made up of a sequence of tokens. Each token is a sequence of characters. A blank terminates any token other than a quoted string. End of line also terminates a token unless explicit continuation (see above) is signaled by an underscore.

2.3.1. Identifiers

An identifier is a letter or a letter followed by letters or digits. The following is a list of the reserved words that have special meaning in EFL. They will be discussed later.

array	**exit**	**precision**
automatic	**external**	**procedure**
break	**false**	**read**
call	**field**	**readbin**
case	**for**	**real**
character	**function**	**repeat**
common	**go**	**return**
complex	**goto**	**select**
continue	**if**	**short**
debug	**implicit**	**sizeof**
default	**include**	**static**
define	**initial**	**struct**
dimension	**integer**	**subroutine**
do	**internal**	**true**
double	**lengthof**	**until**
doubleprecision	**logical**	**value**
else	**long**	**while**
end	**next**	**write**
equivalence	**option**	**writebin**

The use of these words is discussed below. These words may not be used for any other purpose.

2.3.2. Strings

A character string is a sequence of characters surrounded by quotation marks. If the string is bounded by single-quote marks (´), it may contain double quote marks ("), and vice versa. A quoted string may not be broken across a line boundary.

```
´hello there´
"ain´t misbehavin´"
```

2.3.3. Integer Constants

An integer constant is a sequence of one or more digits.

0
57
123456

2.3.4. Floating Point Constants

A floating point constant contains a dot and/or an exponent field. An *exponent field* is a letter **d** or **e** followed by an optionally signed integer constant. If *I* and *J* are integer constants and *E* is an exponent field, then a floating constant has one of the following forms:

.*I*
*I***.**
*I***.***J*
IE
*I***.***E*
.*IE*
*I***.***JE*

2.3.5. Punctuation

Certain characters are used to group or separate objects in the language. These are

parentheses	( )
braces	{ }
comma	,
semicolon	;
colon	:
end-of-line	

The end-of-line is a token (statement separator) when the line is neither blank nor continued.

2.3.6. Operators

The EFL operators are written as sequences of one or more non-alphanumeric characters.

```
+   –   ∗   /   ∗∗
<   <=   >   >=   ==   ˜=
&&   ||   &   |
+=   –=      /=   ∗∗=
&&=   ||=      &=   |=
–>   .   $
```

A dot (‘**.**’) is an operator when it qualifies a structure element name, but not when it acts as a decimal point in a numeric constant. There is a special mode (see the Atavisms section) in which some of the operators may be represented by a string consisting of a dot, an identifier, and a dot (*e.g.,* **.lt.**).

2.4. Macros

EFL has a simple macro substitution facility. An identifier may be defined to be equal to a string of tokens; whenever that name appears as a token in the program, the string replaces it. A macro name is given a value in a **define** statement like

```
define count     n += 1
```

Any time the name **count** appears in the program, it is replaced by the statement

n += 1

A **define** statement must appear alone on a line; the form is

define *name* *rest-of-line*

Trailing comments are part of the string.

3. PROGRAM FORM

3.1. Files

A *file* is a sequence of lines. A file is compiled as a single unit. It may contain one or more procedures. Declarations and options that appear outside of a procedure affect the succeeding procedures on that file.

3.2. Procedures

Procedures are the largest grouping of statements in EFL. Each procedure has a name by which it is invoked. (The first procedure invoked during execution, known as the *main* procedure, has the null name.) Procedure calls and argument passing are discussed in Section 8.

3.3. Blocks

Statements may be formed into groups inside of a procedure. To describe the scope of names, it is convenient to introduce the ideas of *block* and of *nesting level.* The beginning of a program file is at nesting level zero. Any options, macro definitions, or variable declarations there are also at level zero. The text immediately following a **procedure** statement is at level 1. After the declarations, a left brace marks the beginning of a new block and increases the nesting level by 1; a right brace drops the level by 1. (Braces inside declarations do not mark blocks.) (See Section 7.2). An **end** statement marks the end of the procedure, level 1, and the return to level 0. A name (variable or macro) that is defined at level *k* is defined throughout that block and in all deeper nested levels in which that name is not redefined or redeclared. Thus, a procedure might look like the following:

```
#   block 0
procedure george
real x
x = 2
 . . .
if(x > 2)
          {          # new block
          integer x  # a different variable
          do x = 1,7
                     write(,x)
          . . .
          }          # end of block
end       # end of procedure, return to block 0
```

3.4. Statements

A statement is terminated by end of line or by a semicolon. Statements are of the following types:

Option
Include
Define

Procedure
End

Declarative
Executable

The **option** statement is described in Section 10. The **include, define,** and **end** statements have been described above; they may not be followed by another statement on a line. Each procedure begins with a **procedure** statements and finishes with an **end** statement; these are discussed in Section 8. Declarations describe types and values of variables and procedures. Executable statements cause specific actions to be taken. A block is an example of an executable statement; it is made up of declarative and executable statements.

3.5. Labels

An executable statement may have a *label* which may be used in a branch statement. A label is an identifier followed by a colon, as in

```
        read(, x)
        if(x < 3) goto error
        . . .
error:  fatal("bad input")
```

4. DATA TYPES AND VARIABLES

EFL supports a small number of basic (scalar) types. The programmer may define objects made up of variables of basic type; other aggregates may then be defined in terms of previously defined aggregates.

4.1. Basic Types

The basic types are

logical
integer
field(*m* **:***n* **)**
real
complex
long real
long complex
character(*n* **)**

A logical quantity may take on the two values true and false. An integer may take on any whole number value in some machine-dependent range. A field quantity is an integer restricted to a particular closed interval ([*m* :*n*]). A ‘real’ quantity is a floating point approximation to a real or rational number. A long real is a more precise approximation to a rational. (Real quantities are represented as single precision floating point numbers; long reals are double precision floating point numbers.) A complex quantity is an approximation to a complex number, and is represented as a pair of reals. A character quantity is a fixed-length string of *n* characters.

4.2. Constants

There is a notation for a constant of each basic type.

A logical may take on the two values

true
false

An integer or field constant is a fixed point constant, optionally preceded by a plus or minus sign, as in

17
–94
+6
0

A long real (‘double precision’) constant is a floating point constant containing an exponent field that begins with the letter **d.** A real (‘single precision’) constant is any other floating point constant. A real or long real constant may be preceded by a plus or minus sign. The following are valid **real** constants:

17.3
−.4
7.9e−6 $(= 7.9\times10^{-6})$
14e9 $(= 1.4\times10^{10})$

The following are valid **long real** constants

7.9d−6 $(= 7.9\times10^{-6})$
5d3

A character constant is a quoted string.

4.3. Variables

A variable is a quantity with a name and a location. At any particular time the variable may also have a value. (A variable is said to be *undefined* before it is initialized or assigned its first value, and after certain indefinite operations are performed.) Each variable has certain attributes:

4.3.1. Storage Class

The association of a name and a location is either transitory or permanent. Transitory association is achieved when arguments are passed to procedures. Other associations are permanent (static). (A future extension of EFL may include dynamically allocated variables.)

4.3.2. Scope of Names

The names of common areas are global, as are procedure names: these names may be used anywhere in the program. All other names are local to the block in which they are declared.

4.3.3. Precision

Floating point variables are either of normal or **long** precision. This attribute may be stated independently of the basic type.

4.4. Arrays

It is possible to declare rectangular arrays (of any dimension) of values of the same type. The index set is always a cross-product of intervals of integers. The lower and upper bounds of the intervals must be constants for arrays that are local or **common.** A formal argument array may have intervals that are of length equal to one of the other formal arguments. An element of an array is denoted by the array name followed by a parenthesized comma-separated list of integer values, each of which must lie within the corresponding interval. (The intervals may include negative numbers.) Entire arrays may be passed as procedure arguments or in input/output lists, or they may be initialized; all other array references must be to individual elements.

4.5. Structures

It is possible to define new types which are made up of elements of other types. The compound object is known as a *structure;* its constituents are called *members* of the structure. The structure may be given a name, which acts as a type name in the remaining statements within the scope of its declaration. The elements of a structure may be of any type (including previously defined structures), or they may be arrays of such objects. Entire structures may be passed to procedures or be used in input/output lists; individual elements of structures may be referenced. The uses of structures will be detailed below. The following structure might represent a symbol table:

```
struct tableentry
        {
        character(8) name
        integer hashvalue
        integer numberofelements
        field(0:1) initialized, used, set
        field(0:10) type
        }
```

5. EXPRESSIONS

Expressions are syntactic forms that yield a value. An expression may have any of the following forms, recursively applied:

primary
(*expression* **)**
unary-operator expression
expression binary-operator expression

In the following table of operators, all operators on a line have equal precedence and have higher precedence than operators on later lines. The meanings of these operators are described in sections 5.3 and 5.4.

```
−>  .
**
*  /     unary +  −  ++  −−
+  −
<  <=  >  >=  ==  ~=
&  &&
|  ||
$
=  +=  −=  *=  /=  **=   &=  |=  &&=  ||=
```

Examples of expressions are

```
a<b && b<c
−(a + sin(x)) / (5+cos(x))**2
```

5.1. Primaries

Primaries are the basic elements of expressions, as follows:

5.1.1. Constants

Constants are described in Section 4.2.

5.1.2. Variables

Scalar variable names are primaries. They may appear on the left or the right side of an assignment. Unqualified names of aggregates (structures or arrays) may only appear as procedure arguments and in input/output lists.

5.1.3. Array Elements

An element of an array is denoted by the array name followed by a parenthesized list of subscripts, one integer value for each declared dimension:

```
a(5)
b(6, −3, 4)
```

5.1.4. Structure Members

A structure name followed by a dot followed by the name of a member of that structure constitutes a reference to that element. If that element is itself a structure, the reference may be further qualified.

a.b
x(3).y(4).z(5)

5.1.5. Procedure Invocations

A procedure is invoked by an expression of one of the forms

procedurename **( )**
procedurename **(** *expression* **)**
procedurename **(** *expression-1***,** ...**,** *expression-n* **)**

The *procedurename* is either the name of a variable declared **external** or it is the name of a function known to the EFL compiler (see Section 8.5), or it is the actual name of a procedure, as it appears in a **procedure** statement. If a *procedurename* is declared **external** and is an argument of the current procedure, it is associated with the procedure name passed as actual argument; otherwise it is the actual name of a procedure. Each *expression* in the above is called an *actual argument*. Examples of procedure invocations are

f(x)
work()
g(x, y+3, 'xx')

When one of these procedure invocations is to be performed, each of the actual argument expressions is first evaluated. The types, precisions, and bounds of actual and formal arguments should agree. If an actual argument is a variable name, array element, or structure member, the called procedure is permitted to use the corresponding formal argument as the left side of an assignment or in an input list; otherwise it may only use the value. After the formal and actual arguments are associated, control is passed to the first executable statement of the procedure. When a **return** statement is executed in that procedure, or when control reaches the **end** statement of that procedure, the function value is made available as the value of the procedure invocation. The type of the value is determined by the attributes of the *procedurename* that are declared or implied in the calling procedure, which must agree with the attributes declared for the function in its procedure. In the special case of a generic function, the type of the result is also affected by the type of the argument. See Chapter 8 for details.

5.1.6. Input/Output Expressions

The EFL input/output syntactic forms may be used as integer primaries that have a non-zero value if an error occurs during the input or output. See Section 7.7.

5.1.7. Coercions

An expression of one precision or type may be converted to another by an expression of the form

attributes **(** *expression* **)**

At present, the only *attributes* permitted are precision and basic types. Attributes are separated by white space. An arithmetic value of one type may be coerced to any other arithmetic type; a character expression of one length may be coerced to a character expression of another length; logical expressions may not be coerced to a nonlogical type. As a special case, a quantity of **complex** or **long complex** type may be constructed from two integer or real quantities by passing two expressions (separated by a comma) in the coercion. Examples and equivalent values are

integer(5.3) = 5
long real(5) = 5.0d0
complex(5,3) = $5+3i$

Most conversions are done implicitly, since most binary operators permit operands of different arithmetic types. Explicit coercions are of most use when it is necessary to convert the type of an actual argument to match that of the corresponding formal parameter in a procedure call.

5.1.8. Sizes

There is a notation which yields the amount of memory required to store a datum or an item of specified type:

sizeof (*leftside* **)**
sizeof (*attributes* **)**

In the first case, *leftside* can denote a variable, array, array element, or structure member. The value of **sizeof** is an integer, which gives the size in arbitrary units. If the size is needed in terms of the size of some specific unit, this can be computed by division:

sizeof(x) / sizeof(integer)

yields the size of the variable **x** in integer words.

The distance between consecutive elements of an array may not equal **sizeof** because certain data types require final padding on some machines. The **lengthof** operator gives this larger value, again in arbitrary units. The syntax is

lengthof (*leftside* **)**
lengthof (*attributes* **)**

5.2. Parentheses

An expression surrounded by parentheses is itself an expression. A parenthesized expression must be evaluated before an expression of which it is a part is evaluated.

5.3. Unary Operators

All of the unary operators in EFL are prefix operators. The result of a unary operator has the same type as its operand.

5.3.1. Arithmetic

Unary **+** has no effect. A unary – yields the negative of its operand.

The prefix operator **++** adds one to its operand. The prefix operator **−−** subtracts one from its operand. The value of either expression is the result of the addition or subtraction. For these two operators, the operand must be a scalar, array element, or structure member of arithmetic type. (As a side effect, the operand value is changed.)

5.3.2. Logical

The only logical unary operator is complement (**~**). This operator is defined by the equations

~ true = false
~ false = true

5.4. Binary Operators

Most EFL operators have two operands, separated by the operator. Because the character set must be limited, some of the operators are denoted by strings of two or three special characters. All binary operators except exponentiation are left associative.

5.4.1. Arithmetic

The binary arithmetic operators are

+	addition
−	subtraction
*	multiplication
/	division
**	exponentiation

Exponentiation is right associative: a**b**c = a**(b**c) = $a^{(b^c)}$ The operations have the conventional meanings: 8+2 = 10, 8−2 = 6, 8*2 = 16, 8/2 = 4, 8**2 = 8^2 = 64.

The type of the result of a binary operation *A op B* is determined by the types of its operands:

	Type of B				
Type of A	integer	real	long real	complex	long complex
integer	integer	real	long real	complex	long complex
real	real	real	long real	complex	long complex
long real	long real	long real	long real	long complex	long complex
complex	complex	complex	long complex	complex	long complex
long complex	long complex	long complex	long complex	long complex	long complex

If the type of an operand differs from the type of the result, the calculation is done as if the operand were first coerced to the type of the result. If both operands are integers, the result is of type integer, and is computed exactly. (Quotients are truncated toward zero, so 8/3=2.)

5.4.2. Logical

The two binary logical operations in EFL, **and** and **or,** are defined by the truth tables:

A	B	A and B	A or B
false	false	false	false
false	true	false	true
true	false	false	true
true	true	true	true

Each of these operators comes in two forms. In one form, the order of evaluation is specified. The expression

a && b

is evaluated by first evaluating **a**; if it is false then the expression is false and **b** is not evaluated; otherwise the expression has the value of **b.** The expression

a || b

is evaluated by first evaluating **a;** if it is true then the expression is true and **b** is not evaluated; otherwise the expression has the value of **b.** The other forms of the operators (**&** for **and** and **|** for **or**) do not imply an order of evaluation. With the latter operators, the compiler may speed up the code by evaluating the operands in any order.

5.4.3. Relational Operators

There are six relations between arithmetic quantities. These operators are not associative.

EFL Operator	Meaning	
<	$<$	less than
<=	$\leq$	less than or equal to
==	$=$	equal to
~=	$\neq$	not equal to
>	$>$	greater than
>=	$\geq$	greater than or equal

Since the complex numbers are not ordered, the only relational operators that may take complex operands are **==** and **~=** . The character collating sequence is not defined.

5.4.4. Assignment Operators

All of the assignment operators are right associative. The simple form of assignment is

basic-left-side = *expression*

A *basic-left-side* is a scalar variable name, array element, or structure member of basic type. This statement computes the expression on the right side, and stores that value (possibly after coercing the value to the type of the left side) in the location named by the left side. The value of the assignment expression is the value assigned to the left side after coercion.

There is also an assignment operator corresponding to each binary arithmetic and logical operator. In each case, $a \ op = b$ is equivalent to $a = a \ op \ b$. (The operator and equal sign must not be separated by blanks.) Thus, **n+=2** adds 2 to n. The location of the left side is evaluated only once.

5.5. Dynamic Structures

EFL does not have an address (pointer, reference) type. However, there is a notation for dynamic structures,

leftside –> *structurename*

This expression is a structure with the shape implied by *structurename* but starting at the location of *leftside.* In effect, this overlays the structure template at the specified location. The *leftside* must be a variable, array, array element, or structure member. The type of the *leftside* must be one of the types in the structure declaration. An element of such a structure is denoted in the usual way using the dot operator. Thus,

place(i) –> st.elt

refers to the **elt** member of the **st** structure starting at the i^{th} element of the array **place.**

5.6. Repetition Operator

Inside of a list, an element of the form

integer-constant-expression **$** *constant-expression*

is equivalent to the appearance of the *expression* a number of times equal to the first expression. Thus,

(3, 3$4, 5)

is equivalent to

(3, 4, 4, 4, 5)

5.7. Constant Expressions

If an expression is built up out of operators (other than functions) and constants, the value of the expression is a constant, and may be used anywhere a constant is required.

6. DECLARATIONS

Declarations statement describe the meaning, shape, and size of named objects in the EFL language.

6.1. Syntax

A declaration statement is made up of attributes and variables. Declaration statements are of two form:

attributes variable-list
attributes { declarations }

In the first case, each name in the *variable-list* has the specified attributes. In the second, each name in the declarations also has the specified attributes. A variable name may appear in more than one variable list, so long as the attributes are not contradictory. Each name of a nonargument variable may be accompanied by an initial value specification. The *declarations* inside the braces are one or more declaration statements. Examples of declarations are

```
integer k=2

long real b(7,3)

common(cname)
     {
     integer i
     long real array(5,0:3) x, y
     character(7) ch
     }
```

6.2. Attributes

6.2.1. Basic Types

The following are basic types in declarations

logical
integer
field($m:n$**)**
character(k**)**
real
complex

In the above, the quantities k, m, and n denote integer constant expressions with the properties $k>0$ and $n>m$.

6.2.2. Arrays

The dimensionality may be declared by an **array** attribute

array$(b_1,...,b_n)$

Each of the b_i may either be a single integer expression or a pair of integer expressions separated by a colon. The pair of expressions form a lower and an upper bound; the single expression is an upper bound with an implied lower bound of 1. The number of dimensions is equal to n, the number of bounds. All of the integer expressions must be constants. An exception is permitted only if all of the variables associated with an array declarator are formal arguments of the procedure; in this case, each bound must have the property that $upper-lower+1$ is equal to a formal argument of the procedure. (The compiler has limited ability to simplify expressions, but it will recognize important cases such as **(0:n−1)**. The upper bound for the last dimension (b_n) may be marked by an asterisk ($*$) if the size of the array is not known. The following are legal **array** attributes:

```
array(5)
array(5, 1:5, −3:0)
array(5, ∗)
array(0:m−1, m)
```

6.2.3. Structures

A structure declaration is of the form

struct *structname* **{** *declaration statements* **}**

The *structname* is optional; if it is present, it acts as if it were the name of a type in the rest of its scope. Each name that appears inside the *declarations* is a *member* of the structure, and has a special meaning when used to qualify any variable declared with the structure type. A name may appear as a member of any number of structures, and may also be the name of an ordinary variable, since a structure member name is used only in contexts where the parent type is known. The following are valid structure attributes

```
struct xx
      {
      integer a, b
      real x(5)
      }

struct { xx z(3); character(5) y }
```

The last line defines a structure containing an array of three **xx**′*s* and a character string.

6.2.4. Precision

Variables of floating point (**real** or **complex**) type may be declared to be **long** to ensure they have higher precision than ordinary floating point variables. The default precision is **short**.

6.2.5. Common

Certain objects called *common areas* have external scope, and may be referenced by any procedure that has a declaration for the name using a

common (*commonareaname* **)**

attribute. All of the variables declared with a particular **common** attribute are in the same block; the order in which they are declared is significant. Declarations for the same block in differing procedures must have the variables in the same order and with the same types, precision, and shapes, though not necessarily with the same names.

6.2.6. External

If a name is used as the procedure name in a procedure invocation, it is implicitly declared to have the **external** attribute. If a procedure name is to be passed as an argument, it is necessary to declare it in a statement of the form

external *name*

If a name has the external attribute and it is a formal argument of the procedure, then it is associated with a procedure identifier passed as an actual argument at each call. If the name is not a formal argument, then that name is the actual name of a procedure, as it appears in the corresponding **procedure** statement.

6.3. Variable List

The elements of a variable list in a declaration consist of a name, an optional dimension specification, and an optional initial value specification. The name follows the usual rules. The dimension specification is the same form and meaning as the parenthesized list in an **array** attribute. The initial value specification is an equal sign (=) followed by a constant expression. If the name is an array, the right side of the equal sign may be a parenthesized list of constant expressions, or repeated elements or lists; the total number of elements in the list must not exceed the number of elements of the array, which are filled in column-major order.

6.4. The Initial Statement

An initial value may also be specified for a simple variable, array, array element, or member of a structure using a statement of the form

initial *var* = *val*

The *var* may be a variable name, array element specification, or member of structure. The right side follows the same rules as for an initial value specification in other declaration statements.

7. EXECUTABLE STATEMENTS

Every useful EFL program contains executable statements — otherwise it would not do anything and would not need to be run. Statements are frequently made up of other statements. Blocks are the most obvious case, but many other forms contain statements as constituents.

To increase the legibility of EFL programs, some of the statement forms can be broken without an explicit continuation. A square (□) in the syntax represents a point where the end of a line will be ignored.

7.1. Expression Statements

7.1.1. Subroutine Call

A procedure invocation that returns no value is known as a subroutine call. Such an invocation is a statement. Examples are

```
work(in, out)
run( )
```

Input/output statements (see Section 7.7) resemble procedure invocations but do not yield a value. If an error occurs the program stops.

7.1.2. Assignment Statements

An expression that is a simple assignment (=) or a compound assignment (+= etc.) is a statement:

```
a = b
a = sin(x)/6
x *= y
```

7.2. Blocks

A block is a compound statement that acts as a statement. A block begins with a left brace, optionally followed by declarations, optionally followed by executable statements, followed by a right brace. A block may be used anywhere a statement is permitted. A block is not an expression and does not have a value. An example of a block is

```
{
integer i    # this variable is unknown outside the braces
big = 0
do i = 1,n
   if(big < a(i))
         big = a(i)
}
```

7.3. Test Statements

Test statements permit execution of certain statements conditional on the truth of a predicate.

7.3.1. If Statement

The simplest of the test statements is the **if** statement, of form

if (*logical-expression* **)** □ *statement*

The logical expression is evaluated; if it is true, then the *statement* is executed.

7.3.2. If-Else

A more general statement is of the form

if (*logical-expression* **)** □ *statement-1* □ **else** □ *statement-2*

If the expression is **true** then *statement-1* is executed, otherwise *statement-2* is executed. Either of the consequent statements may itself be an **if-else** so a completely nested test sequence is possible:

```
if(x<y)
   if(a<b)
      k = 1
   else
      k = 2
else
   if(a<b)
      m = 1
   else
      m = 2
```

An **else** applies to the nearest preceding un-**else**d **if**. A more common use is as a sequential test:

```
if(x==1)
   k = 1
else if(x==3 | x==5)
   k = 2
else
   k = 3
```

7.3.3. Select Statement

A multiway test on the value of a quantity is succinctly stated as a **select** statement, which has the general form

select(*expression* **)** □ *block*

Inside the block two special types of labels are recognized. A prefix of the form

case *constant* **:**

marks the statement to which control is passed if the expression in the select has a value equal to one of the case constants. If the expression equals none of these constants, but there is a label **default** inside the select, a branch is taken to that point; otherwise the statement following the right brace is executed. Once execution begins at a **case** or **default** label, it continues until the next **case** or **default** is encountered. The **else-if** example above is better written as

```
select(x)
   {
   case 1:
      k = 1
   case 3,5:
      k = 2
   default:
      k = 3
   }
```

Note that control does not ‘fall through’ to the next case.

7.4. Loops

The loop forms provide the best way of repeating a statement or sequence of operations. The simplest (**while**) form is theoretically sufficient, but it is very convenient to have the more general loops available, since each expresses a mode of control that arises frequently in practice.

7.4.1. While Statement

This construct has the form

while (*logical-expression* **)** □ *statement*

The expression is evaluated; if it is true, the statement is executed, and then the test is performed again. If the expression is false, execution proceeds to the next statement.

7.5. For Statement

The **for** statement is a more elaborate looping construct. It has the form

for (*initial-statement* **,** □ *logical-expression* **,** □ *iteration-statement* **)** □ *body-statement*

Except for the behavior of the **next** statement (see Section 7.6.3), this construct is equivalent to

initial-statement
while (*logical-expression* **)**
   {
   body-statement
   iteration-statement
   }

This form is useful for general arithmetic iterations, and for various pointer-type operations. The sum of the integers from 1 to 100 can be computed by the fragment

```
n = 0
for(i = 1, i <= 100, i += 1)
      n += i
```

Alternatively, the computation could be done by the single statement

```
for( { n = 0 ;  i = 1 } ,  i<=100 ,  { n += i ;  ++i } )
     ;
```

Note that the body of the **for** loop is a null statement in this case. An example of following a linked list will be given later.

7.5.1. Repeat Statement

The statement

repeat □ *statement*

executes the statement, then does it again, without any termination test. Obviously, a test inside the *statement* is needed to stop the loop.

7.5.2. Repeat...Until Statement

The **while** loop performs a test before each iteration. The statement

repeat □ *statement* □ **until (** *logical-expression* **)**

executes the *statement,* then evaluates the logical; if the logical is true the loop is complete; otherwise control returns to the *statement.* Thus, the body is always executed at least once. The **until** refers to the nearest preceding **repeat** that has not been paired with an **until.** In practice, this appears to be the least frequently used looping construct.

7.5.3. Do Loops

The simple arithmetic progression is a very common one in numerical applications. EFL has a special loop form for ranging over an ascending arithmetic sequence

do *variable* **=** *expression-1, expression-2, expression-3*
    statement

The variable is first given the value *expression-1.* The statement is executed, then *expression-3* is added to the variable. The loop is repeated until the variable exceeds *expression-2.* If *expression-3* and the preceding comma are omitted, the increment is taken to be 1. The loop above is equivalent to

```
t2 = expression-2
t3 = expression-3
for(variable = expression-1 , variable <= t2 , variable += t3)
    statement
```

(The compiler translates EFL **do** statements into Fortran DO statements, which are in turn usually compiled into excellent code.) The **do** *variable* may not be changed inside of the loop, and *expression-1* must not exceed *expression-2.* The sum of the first hundred positive integers could be computed by

```
n = 0
do i = 1, 100
        n += i
```

7.6. Branch Statements

Most of the need for branch statements in programs can be averted by using the loop and test constructs, but there are programs where they are very useful.

7.6.1. Goto Statement

The most general, and most dangerous, branching statement is the simple unconditional

goto *label*

After executing this statement, the next statement performed is the one following the given label. Inside of a **select** the case labels of that block may be used as labels, as in the following example:

```
select(k)
        {
        case 1:
                error(7)

        case 2:
                k = 2
                goto case 4

        case 3:
                k = 5
                goto case 4

        case 4:
                fixup(k)
                goto default

        default:
                prmsg("ouch")
        }
```

(If two **select** statements are nested, the case labels of the outer **select** are not accessible from the inner one.)

7.6.2. Break Statement

A safer statement is one which transfers control to the statement following the current **select** or loop form. A statement of this sort is almost always needed in a **repeat** loop:

```
repeat
  {
  do a computation
  if ( finished )
       break
  }
```

More general forms permit controlling a branch out of more than one construct.

break 3

transfers control to the statement following the third loop and/or **select** surrounding the statement. It is possible to specify which type of construct (**for**, **while**, **repeat**, **do**, or **select**) is to be counted. The statement

break while

breaks out of the first surrounding **while** statement. Either of the statements

break 3 for
break for 3

will transfer to the statement after the third enclosing **for** loop.

7.6.3. Next Statement

The **next** statement causes the first surrounding loop statement to go on to the next iteration: the next operation performed is the test of a **while,** the *iteration-statement* of a **for,** the body of a **repeat,** the test of a **repeat...until,** or the increment of a **do.** Elaborations similar to those for **break** are available:

next
next 3
next 3 for
next for 3

A **next** statement ignores **select** statements.

7.6.4. Return

The last statement of a procedure is followed by a return of control to the caller. If it is desired to effect such a return from any other point in the procedure, a

return

statement may be executed. Inside a function procedure, the function value is specified as an argument of the statement:

return (*expression* **)**

7.7. Input/Output Statements

EFL has two input statements (**read** and **readbin**), two output statements (**write** and **writebin**), and three control statements (**endfile**, **rewind**, and **backspace**). These forms may be used either as a primary with a **integer** value or as a statement. If an exception occurs when one of these forms is used as a statement, the result is undefined but will probably be treated as a fatal error. If they are used in a context where they return a value, they return zero if no exception occurs. For the input forms, a negative value indicates end-of-file and a positive value an error. The input/output part of EFL very strongly reflects the facilities of Fortran.

7.7.1. Input/Output Units

Each I/O statement refers to a ‘unit’, identified by a small positive integer. Two special units are defined by EFL, the *standard input unit* and the *standard output unit.* These particular units are assumed if no unit is specified in an I/O transmission statement.

The data on the unit are organized into *records.* These records may be read or written in a fixed sequence, and each transmission moves an integral number of records. Transmission proceeds from the first record until the *end of file.*

7.7.2. Binary Input/Output

The **readbin** and **writebin** statements transmit data in a machine-dependent but swift manner. The statements are of the form

writebin(*unit* **,** *binary-output-list* **)**
readbin(*unit* **,** *binary-input-list* **)**

Each statement moves one unformatted record between storage and the device. The *unit* is an integer expression. A *binary-output-list* is an *iolist* (see below) without any format specifiers. A *binary-input-list* is an *iolist* without format specifiers in which each of the expressions is a variable name, array element, or structure member.

7.7.3. Formatted Input/Output

The **read** and **write** statements transmit data in the form of lines of characters. Each statement moves one or more records (lines). Numbers are translated into decimal notation. The exact form of the lines is determined by format specifications, whether provided explicitly in the statement or implicitly. The syntax of the statements is

write(*unit* **,** *formatted-output-list* **)**
read(*unit* **,** *formatted-input-list* **)**

The lists are of the same form as for binary I/O, except that the lists may include format specifications. If the *unit* is omitted, the standard input or output unit is used.

7.7.4. Iolists

An *iolist* specifies a set of values to be written or a set of variables into which values are to be read. An *iolist* is a list of one or more *ioexpressions* of the form

expression
{ *iolist* **}**
do-specification **{** *iolist* **}**

For formatted I/O, an *ioexpression* may also have the forms

ioexpression **:** *format-specifier*
: *format-specifier*

A *do-specification* looks just like a **do** statement, and has a similar effect: the values in the braces are transmitted repeatedly until the **do** execution is complete.

7.7.5. Formats

The following are permissible *format-specifiers.* The quantities *w* , *d* , and *k* must be integer constant expressions.

i(*w***)**	integer with *w* digits
f(*w***,***d***)**	floating point number of *w* characters, *d* of them to the right of the decimal point.
e(*w***,***d***)**	floating point number of *w* characters, *d* of them to the right of the decimal point, with the exponent field marked with the letter **e**
l(*w***)**	logical field of width *w* characters, the first of which is **t** or **f** (the rest are blank on output, ignored on input) standing for **true** and **false** respectively
c	character string of width equal to the length of the datum
c(*w***)**	character string of width *w*
s(*k***)**	skip *k* lines
x(*k***)**	skip *k* spaces
" ... "	use the characters inside the string as a Fortran format

If no format is specified for an item in a formatted input/output statement, a default form is chosen.

If an item in a list is an array name, then the entire array is transmitted as a sequence of elements, each with its own format. The elements are transmitted in column-major order, the same order used for array initializations.

7.7.6. Manipulation statements

The three input/output statements

backspace(*unit* **)**
rewind(*unit* **)**
endfile(*unit* **)**

look like ordinary procedure calls, but may be used either as statements or as integer expressions which yield non-zero if an error is detected. **backspace** causes the specified unit to back up, so that the next read will re-read the previous record, and the next write will over-write it. **rewind** moves the device to its beginning, so that the next input statement will read the first record. **endfile** causes the file to be marked so that the record most recently written will be the last record on the file, and any attempt to read past is an error.

8. PROCEDURES

Procedures are the basic unit of an EFL program, and provide the means of segmenting a program into separately compilable and named parts.

8.1. Procedure Statement

Each procedure begins with a statement of one of the forms

procedure
attributes **procedure** *procedurename*
attributes **procedure** *procedurename* **( )**
attributes **procedure** *procedurename* **(** *name* **)**

The first case specifies the main procedure, where execution begins. In the two other cases, the *attributes* may specify precision and type, or they may be omitted entirely. The precision and type of the procedure may be declared in an ordinary declaration statement. If no type is declared, then the procedure is called a *subroutine* and no value may be returned for it. Otherwise, the procedure is a function and a value of the declared type is returned for each call. Each *name* inside the parentheses in the last form above is called a *formal argument* of the procedure.

8.2. End Statement

Each procedure terminates with a statement

end

8.3. Argument Association

When a procedure is invoked, the actual arguments are evaluated. If an actual argument is the name of a variable, an array element, or a structure member, that entity becomes associated with the formal argument, and the procedure may reference the values in the object, and assign to it. Otherwise, the value of the actual is associated with the formal argument, but the procedure may not attempt to change the value of that formal argument.

If the value of one of the arguments is changed in the procedure, it is not permitted that the corresponding actual argument be associated with another formal argument or with a **common** element that is referenced in the procedure.

8.4. Execution and Return Values

After actual and formal arguments have been associated, control passes to the first executable statement of the procedure. Control returns to the invoker either when the **end** statement of the procedure is reached or when a **return** statement is executed. If the procedure is a function (has a declared type), and a **return**(*value*) is executed, the value is coerced to the correct type and precision and returned.

8.5. Known Functions

A number of functions are known to EFL, and need not be declared. The compiler knows the types of these functions. Some of them are *generic;* i.e., they name a family of functions that differ in the types of their arguments and return values. The compiler chooses which element of the set to invoke based upon the attributes of the actual arguments.

8.5.1. Minimum and Maximum Functions

The generic functions are **min** and **max.** The **min** calls return the value of their smallest argument; the **max** calls return the value of their largest argument. These are the only functions that may take different numbers of arguments in different calls. If any of the arguments are **long real** then the result is **long real**. Otherwise, if any of the arguments are **real** then the result is **real;** otherwise all the arguments and the result must be **integer.** Examples are

```
min(5, x, −3.20)
max(i, z)
```

8.5.2. Absolute Value

The **abs** function is a generic function that returns the magnitude of its argument. For integer and real arguments the type of the result is identical to the type of the argument; for complex arguments the type of the result is the real of the same precision.

8.5.3. Elementary Functions

The following generic functions take arguments of **real**, **long real**, or **complex** type and return a result of the same type:

sin	**sine function**
cos	**cosine function**
exp	**exponential function (e^x).**
log	**natural (base e) logarithm**
log10	**common (base 10) logarithm**
sqrt	**square root function ($\sqrt{x}$).**

In addition, the following functions accept only **real** or **long real** arguments:

atan	$atan(x) = \tan^{-1} x$
atan2	$atan2(x, y) = \tan^{-1} \frac{x}{y}$

8.5.4. Other Generic Functions

The **sign** functions takes two arguments of identical type; $\mathbf{sign}(x, y) = sgn(y)\,|x|$. The **mod** function yields the remainder of its first argument when divided by its second. These functions accept integer and real arguments.

9. ATAVISMS

Certain facilities are included in the EFL language to ease the conversion of old Fortran or Ratfor programs to EFL.

9.1. Escape Lines

In order to make use of nonstandard features of the local Fortran compiler, it is occasionally necessary to pass a particular line through to the EFL compiler output. A line that begins with a percent sign (‘**%**’) is copied through to the output, with the percent sign removed but no other change. Inside of a procedure, each escape line is treated as an executable statement. If a sequence of lines constitute a continued Fortran statement, they should be enclosed in braces.

9.2. Call Statement

A subroutine call may be preceded by the keyword **call.**

call joe
call work(17)

9.3. Obsolete Keywords

The following keywords are recognized as synonyms of EFL keywords:

Fortran	EFL
double precision	**long real**
function	**procedure**
subroutine	**procedure** *(untyped)*

9.4. Numeric Labels

Standard statement labels are identifiers. A numeric (positive integer constant) label is also permitted; the colon is optional following a numeric label.

9.5. Implicit Declarations

If a name is used but does not appear in a declaration, the EFL compiler gives a warning and assumes a declaration for it. If it is used in the context of a procedure invocation, it is assumed to be a procedure name; otherwise it is assumed to be a local variable defined at nesting level 1 in the current procedure. The assumed type is determined by the first letter of the name. The association of letters and types may be given in an **implicit** statement, with syntax

implicit (*letter-list* **)** *type*

where a *letter-list* is a list of individual letters or ranges (pair of letters separated by a minus sign). If no **implicit** statement appears, the following rules are assumed:

implicit (a−h, o−z) real
implicit (i−n) integer

9.6. Computed goto

Fortran contains an indexed multi-way branch; this facility may be used in EFL by the computed GOTO:

goto (*label* **),** *expression*

The expression must be of type integer and be positive but be no larger than the number of labels in the list. Control is passed to the statement marked by the label whose position in the list is equal to the expression.

9.7. Go To Statement

In unconditional and computed **goto** statements, it is permissible to separate the **go** and **to** words, as in

go to xyz

9.8. Dot Names

Fortran uses a restricted character set, and represents certain operators by multi-character sequences. There is an option (**dots=on**; see Section 10.2) which forces the compiler to recognize the forms in the second column below:

<	**.lt.**
<=	**.le.**
>	**.gt.**
>=	**.ge.**
==	**.eq.**
~=	**.ne.**
&	**.and.**
|	**.or.**
&&	**.andand.**
||	**.oror.**
~	**.not.**
true	**.true.**
false	**.false.**

In this mode, no structure element may be named **lt, le,** etc. The readable forms in the left column are always recognized.

9.9. Complex Constants

A complex constant may be written as a parenthesized list of real quantities, such as

(1.5, 3.0)

The preferred notation is by a type coercion,

complex(1.5, 3.0)

9.10. Function Values

The preferred way to return a value from a function in EFL is the **return**(*value*) construct. However, the name of the function acts as a variable to which values may be assigned; an ordinary **return** statement returns the last value assigned to that name as the function value.

9.11. Equivalence

A statement of the form

equivalence $v_1, v_2, \ldots, v_n$

declares that each of the v_i starts at the same memory location. Each of the v_i may be a variable name, array element name, or structure member.

9.12. Minimum and Maximum Functions

There are a number of non-generic functions in this category, which differ in the required types of the arguments and the type of the return value. They may also have variable numbers of arguments, but all the arguments must have the same type.

Function	Argument Type	Result Type
amin0	**integer**	**real**
amin1	**real**	**real**
min0	**integer**	**integer**
min1	**real**	**integer**
dmin1	**long real**	**long real**
amax0	**integer**	**real**
amax1	**real**	**real**
max0	**integer**	**integer**
max1	**real**	**integer**
dmax1	**long real**	**long real**

10. COMPILER OPTIONS

A number of options can be used to control the output and to tailor it for various compilers and systems. The defaults chosen are conservative, but it is sometimes necessary to change the output to match peculiarities of the target environment.

Options are set with statements of the form

option *opt*

where each *opt* is of one of the forms

optionname
optionname **=** *optionvalue*

The *optionvalue* is either a constant (numeric or string) or a name associated with that option. The two names **yes** and **no** apply to a number of options.

10.1. Default Options

Each option has a default setting. It is possible to change the whole set of defaults to those appropriate for a particular environment by using the **system** option. At present, the only valid values are **system=unix** and **system=gcos.**

10.2. Input Language Options

The **dots** option determines whether the compiler recognizes **.lt.** and similar forms. The default setting is **no.**

10.3. Input/Output Error Handling

The **ioerror** option can be given three values: **none** means that none of the I/O statements may be used in expressions, since there is no way to detect errors. The implementation of the **ibm** form uses ERR= and END= clauses. The implementation of the **fortran77** form uses IOSTAT= clauses.

10.4. Continuation Conventions

By default, continued Fortran statements are indicated by a character in column 6 (Standard Fortran). The option **continue=column1** puts an ampersand (**&**) in the first column of the continued lines instead.

10.5. Default Formats

If no format is specified for a datum in an iolist for a **read** or **write** statement, a default is provided. The default formats can be changed by setting certain options

Option	Type
iformat	integer
rformat	real
dformat	long real
zformat	complex
zdformat	long complex
lformat	logical

The associated value must be a Fortran format, such as

option rformat=f22.6

10.6. Alignments and Sizes

In order to implement **character** variables, structures, and the **sizeof** and **lengthof** operators, it is necessary to know how much space various Fortran data types require, and what boundary alignment properties they demand. The relevant options are

Fortran Type	Size Option	Alignment Option
integer	**isize**	**ialign**
real	**rsize**	**ralign**
long real	**dsize**	**dalign**
complex	**zsize**	**zalign**
logical	**lsize**	**lalign**

The sizes are given in terms of an arbitrary unit; the alignment is given in the same units. The option **charperint** gives the number of characters per **integer** variable.

10.7. Default Input/Output Units

The options **ftnin** and **ftnout** are the numbers of the standard input and output units. The default values are **ftnin=5** and **ftnout=6.**

10.8. Miscellaneous Output Control Options

Each Fortran procedure generated by the compiler will be preceded by the value of the **procheader** option.

No Hollerith strings will be passed as subroutine arguments if **hollincall=no** is specified.

The Fortran statement numbers normally start at 1 and increase by 1. It is possible to change the increment value by using the **deltastno** option.

11. EXAMPLES

In order to show the flavor or programming in EFL, we present a few examples. They are short, but show some of the convenience of the language.

11.1. File Copying

The following short program copies the standard input to the standard output, provided that the input is a formatted file containing lines no longer than a hundred characters.

```
procedure   # main program
character(100) line

while( read( , line) == 0 )
        write( , line)
end
```

Since **read** returns zero until the end of file (or a read error), this program keeps reading and writing until the input is exhausted.

11.2. Matrix Multiplication

The following procedure multiplies the $m \times n$ matrix a by the $n \times p$ matrix b to give the $m \times p$ matrix c. The calculation obeys the formula $c_{ij} = \sum a_{ik} b_{kj}$.

```
procedure matmul(a,b,c, m,n,p)
integer i, j, k, m, n, p
long real a(m,n), b(n,p), c(m,p)
do i = 1,m
do j = 1,p
                {
                c(i,j) = 0
                do k = 1,n
                                c(i,j) += a(i,k) * b(k,j)
                }
end
```

11.3. Searching a Linked List

Assume we have a list of pairs of numbers (x,y). The list is stored as a linked list sorted in ascending order of x values. The following procedure searches this list for a particular value of x and returns the corresponding y value.

```
define LAST             0
define NOTFOUND         −1

integer procedure val(list, first, x)

#   list is an array of structures.
#   Each structure contains a thread index value, an x, and a y value.
struct
                {
                integer nextindex
                integer x, y
                } list(*)
integer first, p, arg

for(p = first , p~=LAST && list(p).x<=x , p = list(p).nextindex)
                if(list(p).x == x)
                                return( list(p).y )

return(NOTFOUND)
end
```

The search is a single **for** loop that begins with the head of the list and examines items until either the list is exhausted (p==LAST) or until it is known that the specified value is not on the list (list(p).x > x). The two tests in the conjunction must be performed in the specified order to avoid using an invalid subscript in the **list(p)** reference. Therefore, the **&&** operator is used. The next element in the chain is found by the iteration statement **p=list(p).nextindex**.

11.4. Walking a Tree

As an example of a more complicated problem, let us imagine we have an expression tree stored in a common area, and that we want to print out an infix form of the tree. Each node is either a leaf (containing a numeric value) or it is a binary operator, pointing to a left and a right descendant. In a recursive language, such a tree walk would be implement by the following simple pseudocode:

```
if this node is a leaf
        print its value
otherwise
        print a left parenthesis
        print the left node
        print the operator
        print the right node
        print a right parenthesis
```

In a nonrecursive language like EFL, it is necessary to maintain an explicit stack to keep track of the current state of the computation. The following procedure calls a procedure **outch** to print a single character and a procedure **outval** to print a value.

```
procedure walk(first)   # print out an expression tree

integer first           # index of root node
integer currentnode
integer stackdepth
common(nodes) struct
                {
                character(1) op
                integer leftp, rightp
                real val
                } tree(100)  # array of structures

struct
                {
                integer nextstate
                integer nodep
                } stackframe(100)

define NODE             tree(currentnode)
define STACK            stackframe(stackdepth)

#   nextstate values
define DOWN             1
define LEFT             2
define RIGHT            3

#   initialize stack with root node
stackdepth = 1
STACK.nextstate = DOWN
STACK.nodep = first
```

```
while( stackdepth > 0 )
    {
    currentnode = STACK.nodep
    select(STACK.nextstate)
        {
        case DOWN:
            if(NODE.op == " ")   # a leaf
                {
                outval( NODE.val )
                stackdepth -= 1
                }
            else    {   # a binary operator node
                outch( "(" )
                STACK.nextstate = LEFT
                stackdepth += 1
                STACK.nextstate = DOWN
                STACK.nodep = NODE.leftp
                }

        case LEFT:
            outch( NODE.op )
            STACK.nextstate = RIGHT
            stackdepth += 1
            STACK.nextstate = DOWN
            STACK.nodep = NODE.rightp

        case RIGHT:
            outch( ")" )
            stackdepth -= 1
        }
    }
end
```

12. PORTABILITY

One of the major goals of the EFL language is to make it easy to write portable programs. The output of the EFL compiler is intended to be acceptable to any Standard Fortran compiler (unless the **fortran77** option is specified).

12.1. Primitives

Certain EFL operations cannot be implemented in portable Fortran, so a few machine-dependent procedures must be provided in each environment.

12.1.1. Character String Copying

The subroutine **ef1asc** is called to copy one character string to another. If the target string is shorter than the source, the final characters are not copied. If the target string is longer, its end is padded with blanks. The calling sequence is

```
subroutine ef1asc(a, la, b, lb)
integer a(*), la, b(*), lb
```

and it must copy the first **lb** characters from **b** to the first **la** characters of **a.**

12.1.2. Character String Comparisons

The function **ef1cmc** is invoked to determine the order of two character strings. The declaration is

```
integer function ef1cmc(a, la, b, lb)
integer a(∗), la, b(∗), lb
```

The function returns a negative value if the string **a** of length **la** precedes the string **b** of length **lb.** It returns zero if the strings are equal, and a positive value otherwise. If the strings are of differing length, the comparison is carried out as if the end of the shorter string were padded with blanks.

13. ACKNOWLEDGMENTS

A. D. Hall originated the EFL language and wrote the first compiler for it; he also gave inestimable aid when I took up the project. B. W. Kernighan and W. S. Brown made a number of useful suggestions about the language and about this report. N. L. Schryer has acted as willing, cheerful, and severe first user and helpful critic of each new version and facility. J. L. Blue, L. C. Kaufman, and D. D. Warner made very useful contributions by making serious use of the compiler, and noting and tolerating its misbehaviors.

14. REFERENCE

1. B. W. Kernighan, ‘‘Ratfor — A Preprocessor for a Rational Fortran’’, Bell Laboratories Computing Science Technical Report #55

APPENDIX A. Relation Between EFL and Ratfor

There are a number of differences between Ratfor and EFL, since EFL is a defined language while Ratfor is the union of the special control structures and the language accepted by the underlying Fortran compiler. Ratfor running over Standard Fortran is almost a subset of EFL. Most of the features described in the Atavisms section are present to ease the conversion of Ratfor programs to EFL.

There are a few incompatibilities: The syntax of the **for** statement is slightly different in the two languages: the three clauses are separated by semicolons in Ratfor, but by commas in EFL. (The initial and iteration statements may be compound statements in EFL because of this change). The input/output syntax is quite different in the two languages, and there is no FORMAT statement in EFL. There are no ASSIGN or assigned GOTO statements in EFL.

The major linguistic additions are character data, factored declaration syntax, block structure, assignment and sequential test operators, generic functions, and data structures. EFL permits more general forms for expressions, and provides a more uniform syntax. (One need not worry about the Fortran/Ratfor restrictions on subscript or DO expression forms, for example.)

APPENDIX B. COMPILER

B.1. Current Version

The current version of the EFL compiler is a two-pass translator written in portable C. It implements all of the features of the language described above except for **long complex** numbers. Versions of this compiler run under the and UNIX† operating systems.

B.2. Diagnostics

The EFL compiler diagnoses all syntax errors. It gives the line and file name (if known) on which the error was detected. Warnings are given for variables that are used but not explicitly declared.

B.3. Quality of Fortran Produced

The Fortran produced by EFL is quite clean and readable. To the extent possible, the variable names that appear in the EFL program are used in the Fortran code. The bodies of loops and test constructs are indented. Statement numbers are consecutive. Few unneeded GOTO and CONTINUE statements are used. It is considered a compiler bug if incorrect Fortran is produced (except for escaped lines). The following is the Fortran procedure produced by the EFL compiler for the matrix multiplication example (Section 11.2):

```
      subroutine  matmul(a,  b,  c,  m,  n,  p)
      integer  m,  n,  p
      double  precision  a(m,  n),  b(n,  p),  c(m,  p)
      integer  i,  j,  k
      do   3  i  =  1,  m
         do   2  j  =  1,  p
            c(i,  j)  =  0
            do   1  k  =  1,  n
               c(i,  j)  =  c(i,  j)+a(i,  k)*b(k,  j)
1              continue
2           continue
3        continue
      end
```

The following is the procedure for the tree walk (Section 11.4):

† UNIX is a trademark of Bell Laboratories.

```
      subroutine walk(first)
      integer first
      common /nodes/ tree
      integer tree(4, 100)
      real tree1(4, 100)
      integer staame(2, 100), stapth, curode
      integer const1(1)
      equivalence (tree(1,1), tree1(1,1))
      data const1(1)/4h      /
c print out an expression tree
c index of root node
c array of structures
c    nextstate values
c    initialize stack with root node
      stapth = 1
      staame(1, stapth) = 1
      staame(2, stapth) = first
    1 if (stapth .le. 0) goto 9
         curode = staame(2, stapth)
         goto 7
    2       if (tree(1, curode) .ne. const1(1)) goto 3
               call outval(tree1(4, curode))
c a leaf
               stapth = stapth-1
               goto 4
    3          call outch(1h()
c a binary operator node
               staame(1, stapth) = 2
               stapth = stapth+1
               staame(1, stapth) = 1
               staame(2, stapth) = tree(2, curode)
    4       goto 8
    5       call outch(tree(1, curode))
            staame(1, stapth) = 3
            stapth = stapth+1
            staame(1, stapth) = 1
            staame(2, stapth) = tree(3, curode)
            goto 8
    6       call outch(1h))
            stapth = stapth-1
            goto 8
    7       if (staame(1, stapth) .eq. 3) goto 6
            if (staame(1, stapth) .eq. 2) goto 5
            if (staame(1, stapth) .eq. 1) goto 2
    8    continue
         goto 1
    9 continue
      end
```

APPENDIX C. CONSTRAINTS ON THE DESIGN OF THE EFL LANGUAGE

Although Fortran can be used to simulate any finite computation, there are realistic limits on the generality of a language that can be translated into Fortran. The design of EFL was constrained by the implementation strategy. Certain of the restrictions are petty (six character external names), but others are sweeping (lack of pointer variables). The following paragraphs describe the major limitations imposed by Fortran.

C.1. External Names

External names (procedure and COMMON block names) must be no longer than six characters in Fortran. Further, an external name is global to the entire program. Therefore, EFL can support block structure within a procedure, but it can have only one level of external name if the EFL procedures are to be compilable separately, as are Fortran procedures.

C.2. Procedure Interface

The Fortran standards, in effect, permit arguments to be passed between Fortran procedures either by reference or by copy-in/copy-out. This indeterminacy of specification shows through into EFL. A program that depends on the method of argument transmission is illegal in either language.

There are no procedure-valued variables in Fortran: a procedure name may only be passed as an argument or be invoked; it cannot be stored. Fortran (and EFL) would be noticeably simpler if a procedure variable mechanism were available.

C.3. Pointers

The most grievous problem with Fortran is its lack of a pointer-like data type. The implementation of the compiler would have been far easier if certain hard cases could have been handled by pointers. Further, the language could have been simplified considerably if pointers were accessible in Fortran. (There are several ways of simulating pointers by using subscripts, but they founder on the problems of external variables and initialization.)

C.4. Recursion

Fortran procedures are not recursive, so it was not practical to permit EFL procedures to be recursive. (Recursive procedures with arguments can be simulated only with great pain.)

C.5. Storage Allocation

The definition of Fortran does not specify the lifetime of variables. It would be possible but cumbersome to implement stack or heap storage disciplines by using COMMON blocks.