September 13, 1972

A Manual for the Tmg Compiler-writing Language

M. D. McIlroy
Bell Laboratories
Murray Hill, New Jersey

ABSTRACT

Tmg is a string processing language especially intended for writing translators for computer languages. It deals with string scanning, building of tables and output generation, and provides some integer arithmetic. The experience of many years has been distilled into a new version running on the PDP-11 under the UNIX operating system.

A Manual for the Tmg Compiler-writing Language

M. D. McIlroy
Bell Laboratories
Murray Hill, New Jersey
September 13, 1972

1. INTRODUCTION

1.1 Parsing rules and functions

At the heart of the language are parsing rules (3.1). A rule is a sequence of actions, written simply by naming the actions one after the other. For example, this typical parsing rule

```
smark any(letter) string(alpha) install
```

might be used to recognize an identifier of arbitrary length and install it in a table. The example invokes two scanning functions, "any(.)" and "string(.)", which recognize respectively precisely one, and an arbitrary string of characters from a character class. Sandwiched around the scanning are "smark" and "install", which note the beginning of the string, and enter the completed string into the table.

A parsing statement (2.) is a parsing rule labeled with a name and ended with a semicolon. This parsing statement contains the previous parsing rule:

```
ident: smark any(letter) string(alpha) install;
```

When the meaning is obvious from context, a parsing statement may also be called simply a rule.

As it happens, "smark", "any(.)" and "string(.)" are all functions intrinsic to Tmg, which we call builtins, while "install" must be defined by another rule in terms of other actions. No character classes are built in, so "letter" and "alpha" also must be defined somewhere else in the program.

The function of matching a specifed literal string is so common that it has been given a special notation, the string surrounded by angle brackets, <>. Thus a Fortran DO statement might be recognized by

```
do: <DO> number ident <=> limits;
```

1.2 Success, failure and branches

Execution of a parsing rule may have several effects. (We have already noted that it may make entries in a table.) Every rule must *succeed* or *fail*. A rule succeeds when execution proceeds to the end of the rule without failing. Certain builtins can fail--"any(letter)" fails unless the cursor (see below) points to a letter. In general the failure of any action invoked by a rule causes the rule itself to fail without doing any subsequent actions. However, there are ways to continue conditionally upon failure.

A rule may specify an alternate *branch* in case of failure, as in this rule for a DO limit, which consists either of an identifier or of a number.

```
limit:  ident/lm1;
lm1:    number;
```

The branch, designated by a slash and the name of a rule, is a conditional go-to. If "ident" fails , then the rule for "limit" continues at "lm1" as if "ident" had never been invoked (but see 3.7 for a qualification). The success or failure of "number" then determines the success or failure of "limit".

Conditional branches may also be made on success, indicated by a backslash.

```
limits: limit <,> limit <,>\lms1;
lms1:   limit;
```

This rule continues at "lms1" after a second comma has been found. Because a test has been made, execution of "limits" continues right on when a second comma is not found, to terminate (successfully) at the semicolon.

1.3 The input cursor

A successful action may move a *cursor* along the input--"any(letter)" succeeds and moves the cursor to the next character if it points to a letter. When a rule succeeds the cursor is left where it was left by the last action in the rule. When a rule fails, the cursor is restored to the place where it was when the rule was invoked, regardless of what happened later.

1.4 Translation rules

A successful parsing rule may *deliver* a *translation rule*, often called simply a *translation*. Since the proper order of output may not be the same as the order in which translation rules are delivered, the execution of a translation rule is delayed until explicitly called for (1.5).

A translation rule, like a parsing rule, consists of a sequence of actions, which may be other translation rules or literal strings to place into the output. A translation rule is always enclosed in braces, {}.

This simple program defines the translation of fully parenthesized infix expressions to polish postfix for a stack machine.

```
expr: <(>/exp1 expr operator expr <)> = { 3 1 2 };
exp1: ident = { < LOAD > 1 };
operator:
op0: <+>/op1 = { < ADD> };
op1: <->/op2 = { < SUB> };
op2: <*>/op3 = { < MPY> };
op3: </>     = { < DIV> };
```

The "=" in each rule intrcduces the translation to be delivered by the rule. The numbers in a translation refer to translations delivered by *those actions that delivered translations* to the parsing rule. Those translations are identified by *counting backward from the = sign*. Actions with no translations (e.g. recognizing a literal) are not counted. This awkward convention happens to be very efficient to implement, so we live with it; however some syntactic sugaring will get around counting in many cases (4.2).

The four operators +-*/ are translated into "ADD", "SUB", "MPY" and "DIV". An expression consisting of a single identifier is rendered as "LOAD" followed by the translation of the identifier, which we assume is unchanged in translation. A little inspection shows that the parsing rules correspond to this simple BNF, in which angle brackets have the same meaning as in Tmg:

```
expr ::= <(> expr operator expr <)> | ident
operator ::= <+> | <-> | <*> | </>
```

and that the expression ((a*x)+(b*y)) translates into

```
LOAD a LOAD x MPY LOAD b LOAD y MPY ADD
```

Here we see an important difference between the Tmg, which is a *program*, and the BNF, which is a *pattern*. In Tmg the branch was placed after <(>, not after the matching <)> as was the alternate in the BNF. The latter place would be wrong, for the rule would fail without ever getting there to test for tne branch.

1.5 Getting output

Mere delivery of a translation rule does not create any output. Indeed a delivered translaticn may never get used, for example if the rule to which it was delivered fails. The builtin function "parse(.)" is provided tc cause the execution cf a parsing rule followed immediately by the execution of its translation (if the rule succeeds). Data is placed on the output file only while translations are being executed. Once executed, the translation is forgotten.

These two rules might be used to parse and translate Fortran card-by-card.

```
                program: comment\program
                         endcard\done
                         parse(statement)\program
                         diag(error)\program;
                done: ;
```

First each card is checked tc see if it is a comment, and if so the rule loops. (It is understood that "comment" delivers no translation lest the process get clogged with delivered but unexecuted translations.) Next the card is checked to see whether it is the end; if it is, the rule terminates by going to the empty rule "done". When "statement" succeeds, its translation is output; "parse(statement)" then succeeds in turn, and the rule loops. When "statement" fails, so does "parse(statement)" and the rule goes on to "diag(error)".

"Diag(.)" is just like "parse(.)", except that it sends output to the diagnostic file. We assume that "error" has been coded to eat up any card and perhaps deliver a copy of the card along with a message. Thus unless there is no card there at all (end of file), the rule loops after giving the diagnostic.

1.6 Tables

As characters are scanned over by "any(.)" and "string(.)", but not by quoted literals, they are gathered into a current string. The current string may be looked up in or entered into a table. Recalling the rule on page 1 for identifiers, we now show how to accomplish "install".

```
     ident: smark any(letter) string(alpha) install = { 1 };
     install: enter(t,i) getnam(t,i) = { 1 };
```

The current string is cleared by "smark" and gathered by "any(.)" and "string(.)". "Enter(t,i)" enters the current string into table t and assigns the index of the entry to variable i. (See Section 6 for how to create a table.) "Getnam(t,i)" delivers the string which is the ith entry of table t. The "= { 1 }" in each rule arranges to deliver to its invoker the one translation that was delivered to it.

1.7 Character classes

A character class is defined by enclosing a set of characters in double angle brackets. Sets may be unioned by juxtaposition as in the next example.

```
letter: <<abcdefghijklmnopqrstuvwxyz>>
        <<ABCDEFGHIJKLMNOPQRSTUVWXYZ>>;
```

An exclamation mark complements a set as in this example that defines the class of all ascii characters as the complement of the empty class.

```
ascii:  !<<>>;
```

2. PROGRAMS

A Tmg program consists of a sequence of statements. Each statement has one of these forms

(i) a comment bracketed by /* */ in the style of PL/I
(ii) a parsing statement
(iii) a labeled translation rule (4.)
(iv) a labeled character class (3.2)
(v) a labeled list of octal constants separated by semicolons (5.1,5.7)

All statements except comments are terminated by semicolons. Spaces, tabs and newline characters delimit tokens but not statements. Execution of the program begins with the first noncomment statement, which must be a parsing statement, and ends when execution of that rule (as extended by go-to's) ends.

A parsing statement is a labeled parsing rule followed by a semicolon or by a parsing statement. In the latter case the execution of the containing statement flows into the contained rule as if the contained label were not there.

Instructions for compiling and executing a program on the PDP-11 are reproduced from the UNIX manual [8] as Section 10 of this manual.

3. PARSING RULES

3.1 General form

A parsing rule is a possibly empty sequence of disjuncts separated by | signs.

A disjunct consists of a nonempty sequence of parsing elements (called simply elements when the context is obvious). An element may be any one of

(i) a literal (10.2)
(ii) a name of a builtin function (8.)
(iii) a name of a parsing statement (2.)
(iv) an output element (a translation) (3.3,4.)

(v) a reference to a parameter of the parsing statement (3.5)

(vi) an arithmetic element (5.1)

(vii) any of the preceding with a success or failure branch (1.2)

(viii) a parsing rule in parentheses ()

The elements specify actions to be performed in order, except as modified by failure, branches or disjunction.

3.2 Literals and character classes

A literal consists of one or more ascii characters enclosed in angle brackets <>. A > sign may appear within a literal only as the first character.

A literal consisting of a single newline character may be designated by a special notation, an unbracketed asterisk *.

A character class is designated by

(i) a set of characters enclosed in double angle brackets <<>>

(ii) a union of two or more classes of type (i) indicated by juxtaposition

(iii) the complement of a class of type (i) or (ii) indicated by a prefixed !

Type (ii) classes are merely a convenience for splitting a large class up into readable groupings. The characters of a class may be given in any order, with duplications, except that the character >, if included, must come last to avoid confusion with literals.

There is always an ignored class, which is saved upon the invocation and restored upon the return from each parsing rule. The ignored class is initially empty; "ignore(.)" resets it. The function "smark" scans over ignored characters before marking the start of the current string. "Any(.)" and "string(.)" skip ignored characters. Ignored characters are skipped before, but not within, literals.

Here we define the syntax of a Tmg literal, which must contain at least one ascii character. "Ignore(none)" resets the ignored class from the prevailing value, which is space, tab and newline.

```
  literal: smark <<> ignore(none) any(ascii) string(nonket) <>>;
  nonket:  !<<>>>;
  none:    <<>>;
  ascii:   !<<>>;
```

The Tmg character set consists of 127 characters--ascii less NUL.

3.3 Output elements

An output element is an = sign followed by a translation rule, or by the name of a labeled translation rule. In the latter case the element acts as if the designated rule were copied verbatim into the place of the name.

3.4 Disjunctions

An infix | sign separates two or more <u>disjuncts</u>, each of which is a nonempty sequence cf elements. The example of page 3 may be reworked into a go-to-less form using disjunction:

```
expr:     <(> expr cperator expr <)> = { 3 1 2 }
          | ident = { < LOAD > 1 };
operator: <+> = { < ADD> }
          | <-> = { < SUB> }
          | <*> = { < MPY> }
          | </> = { < DIV> };
```

Disjuncts are executed in crder, with the second being tried if and only if the <u>first element</u> of the first disjunct fails and has no branch, and so on. Once past the first element of a disjunct, the rule executes as if the other disjuncts weren't there. For all its BNF-like appearance, a disjunction still represents a program, not a pattern, as may be illustrated by the example of DO limits from page 2.

```
limits: limit <,> limit
        ( <,> limit | () );
```

The following version of this rule is incorrect because the second disjunct is useless--it would only be tried when the first <u>element</u> of the first disjunct failed, rather than when the second comma failed.

```
limits: limit <,> limit <,> limit
        | limit <,> limit;
```

3.5 Parameters

A parsing statement may have one or more parameters. Corresponding arguments are designated in ordinary functional notation. An argument may be

(i) a name of a statement
(ii) a parenthesized parsing rule
(iii) a character class
(iv) a number
(v) a reference to a parameter of the invoking rule
(vi) a literal; corresponding parameter may only be used as an argurent of another element

The forms ii and iii are understood to be shorthand for an unwritten name of a statement containing the given rule or variable.

A parameter is referred to by number, counting 1,2,... _from right to left_ in the argument list, the number being preceded by a dollar sign. Before any parameters are used, they must be made available by means of the builtin "params(.)", whose argument tells how many arguments are expected. "Params(.)" may be used several times to pick off successive arguments from the right end of the argument list; the arguments so obtained are (re)numbered $1,$2,... If the total number of arguments transmitted during execution of a rule is wrong, or if an argument is used in a nonsensical context, chaos usually results.

This example defines a number of constructs of Algol in terms of a "separated list" or "seplist(.)". The $2 argument of seplist defines a list element, the $1 argument defines the separator.

```
seplist: params(2) $2 ( $1 seplist($2, $1) | () );
block:   <begin> seplist(statement, (<;>)) <end>;
actuals: <(> seplist(expr, (<,>)) <)>;
formals: <(> seplist(ident, (<,>)) <)>;
expr:    seplist(term, (<+>|<->));
term:    seplist(factor, (<*>|</>));
factor:  seplist(primary, (<↑>));
```

Parses according to this definition of "seplist" are right associative, and not always appropriate to Algol. The next section tells how to obtain left-associative parses.

The parameters of a rule that is not properly contained in any other rule may be denoted by names instead of numbers, provided they are referred to from within that rule only. The names are declared by beginning the rule with "proc(.)", where the arguments of "proc" are the parameters. "Seplist" can be defined this way:

```
seplist: proc(x,y) x ( y seplist(x,y) | () );
```

The following rule defines "not(.)" to be a parsing element that succeeds only when its argument fails and vice versa. The built-in "fail" does what you expect.

```
not: proc(x) x fail | ();
```

The next example uses "not(.)" to distinguish a < sign from <= and << by looking ahead one character. The double parentheses come from an argument of type (ii), a parsing rule.

```
lessthan: <<> not(( any(<<=<>>) ));
```

"Not((not(.)))" may be used to peek ahead without displacing the input cursor as in this rule for recognizing the end of a statement in BCPL, where a newline (denoted *) can act as a semicolon provided the beginning of the next line is a plausible beginning for a statement.

```
    semicolon: <;> ¦ ( * not(( not(( ident¦keyword )) )) );
```

Parameters are passed by name, in the Algol sense. This example capitalizes on a name parameter to recognize a notorious non-context free language.

```
                f: proc(x) <a> f((<b> x)) <c> ¦ x;
```

The element "parse((f(())))" will recognize $a^nb^nc^n$. The diagram shows the progress of a parse of "aabbcc".

```
|-------------f(())---------------|
|   |------f((<b>()))---------|   |
|   |   |-f((<b>(<b>())))-|   |   |
|   |   |----<b>(<b>())---|   |   |
|   |   |     |--<b>()--| |   |   |
|   |   |     |         | |   |   |
  a   a    b       b        c   c
```

3.6 Bundles

Upon successful completion of a rule, all translations that have been delivered to it are bundled into a single translation, or bundle, to be delivered to its invoker. The elements of the bundle are translation rules. The elements are counted 0,1,2,... from the most to the least recently delivered.

Translations delivered by output elements (designated by =) are no different from translations delivered by other elements. In any case the translation most recently delivered to a rule becomes the 0 element in its bundle. Thus the rules

```
          param:   ident ={ 1 };
          param:   ident;
```

deliver indistinguishable translations (see 4.3 for a qualification). The latter form is more efficient in time and space.

The builtin "bundle" causes a bundling in the rule that invoked it up to the point of its invocation, and delivers that bundle to the same rule. The set of translations delivered up to the invocation of "bundle" is replaced by just one translation--exactly the translation that the rule would have delivered if the final semicolon appeared in place of "bundle". "Bundle" is particularly useful for left-associative parses, as in the following fragment of a translation from infix to postfix with operator precedence. The operators +-*/ are left-associative and ↑ is right-associative.

```
expr:     term
expr1:    addop/done term = { 3 1 2 } bundle\expr1;
term:     factor
term1:    mulop/done factor = { 3 1 2 } bundle\term1;
factor:   primary <↑>/done factor = { 2 1 < EXP> };
primary:  ident = { < LOAD > 1 }
        | <(> expr <)>;
done:     ;
addop:    <+> = { < ADD> }
        | <-> = { < SUB> };
mulop:    <*> = { < MUL> }
        | </> = { < DIV> };
```

The builtin "reduce(n)" performs the same job as "bundle" except that it replaces only the last n delivered translations. There can be no intra- or interbundle references (4.2, 4.4) between elements of a bundle created by "reduce(.)" and translations of earlier elements of the rule.

3.7 Side effects

Some side effects of the execution of a rule are automatically undone upon its completion. All side effects except these persist:

> assignments to variables saved by "push(.)" are undone
> the ignored class is reset
> the cursor is reset on failure

4. TRANSLATION RULES

4.1 General form

The body of a translation rule is a sequence of translation elements enclosed in braces {}. A translation element may be any one of

(i) a name of a labeled translation rule
(ii) a literal (3.2)
(iii) a reference to a parameter of the translation rule (4.3)
(iv) an intrabundle reference optionally accompanied by arguments (4.2)
(v) an interbundle reference optionally accompanied by arguments (4.4)

In general the significance of a translation element is dynamic and depends upon the progress of the parse and upon other translation rules delivered by the parse. If a translation element is the name of a labeled translation rule, which must consist of a body only, the element acts as if the designated rule were copied into its place with the braces stripped.

4.2 Intrabundle references

An intrabundle reference is a number designating another element of the same bundle. An argument list may be supplied to an intrabundle reference (4.3). Intrabundle references are counted backward starting from 0 at the element containing the reference. For example, if all parsing elements in this rule deliver translations

```
r: a = { 1 }  b = { 3 2 1 };
```

the bundle it delivers will have four elements

```
a's translation
{ 1 }
b's translation
{ 3 2 1 }
```

The 1 in the last element refers to the translation of b, 2 to "{ 1 }" and 3 to the translation of a; the other 1 refers to the translation of a. Only the 0 element of a bundle is directly accessible to its invoker; other elements of the bundle are pulled out by intrabundle references in the 0 element (or by interbundle references, 4.4).

Names may be used instead of numbers for intrabundle references within one parsing rule, provided that the parsing elements that have translations are indicated explicitly by suffixing each with a period. A name, or _alias_, to denote the translation delivered by a parsing element may follow the period with no intervening blanks. If no alias is given and the element consists of a name alone, then that name becomes the alias. The preceding rule may be rewritten in these ways, among others:

```
r: a. = { 1 }.t  b. = { a t b };
r: a. = { a }.  b.x = { a 2 x };
```

4.3 Parameters

A translation rule may have parameters, and if it does, their number is declared by a parenthesized integer prefixed to its body. Alternatively parameters may be given names listed in the leading parentheses, valid only in the immediate rule body and contained bodies, but not in bodies copied in place of type (i) elements (4.1). Parameters are referred to by name or by \$1, \$2,... counted from right to left in the associated argument list. An argument corresponding to a parameter is itself a translation rule body, or a reference thereto, and is passed _by name_ (in the sense of Algol), so that intrabundle references and parameter references in an argument are evaluated in the environment of the invoking rule.

The next rule compiles Honeywell 6000 assembly code for a simple case of Fortran subscripted variables:

```
svar: ident. <(> ident.subscr <)>
        = (1){ <LxL0 > subscr *
               $1  < > ident<,0> * }
```

A machine opcode is to be filled in for the parameter. Assuming that "expr" compiles code to leave a result in the Q register, this rule would handle assignments to such subscripted variables:

```
assign: svar. <=> expr. = { expr svar({<STQ>}) };
```

The next example compiles code for Boolean expressions over a set of unspecified elementary predicates that set a condition code, the state of which determines the outcome of "bt" (branch on true) and "bf" (false) instructions. The Boolean operators are disjunction | and conjunction &. Each translation rule has two parameters, $2 and $1, which are respectively the destinations of branches to be taken upon determining the truth or falsity of the subexpression in question. "Lbl" is a rule that delivers a unique label every time it is invoked.

```
disj:  conj.
       ( <|> lbl. disj.
          = (T,F){ conj({T},{lbl}) lbl<:> disj({T},{F})}
       | () );
conj:  prim.
       ( <&> lbl. conj.
          = (T,F){ prim({lbl},{F}) lbl<:> conj({T},{F})}
       | () );
prim:  pred. = (T,F){ pred< bt >T< bf >F * }
       | <(> disj <)>;
```

Suppose that predicates are denoted by single letters and that "lbl" generates the labels #1, #2, ... Then the parsing element

```
parse({ disj. = { disj({<T>},{<F>}) } })
```

applied to the expression a&(b|c&d) would yield (except for spacing) the output

```
    a bt #1 bf F
#1: b bt T  bf #2
#2: c bt #3 bf F
#3: d bt T  bf F
```

4.4 Interbundle references

An interbundle reference is a translation element of the form m.n, where m and n are both numbers; m must be a legal intrabundle reference. Then the interbundle reference m.n picks out the same translation as would an intrabundle reference n in the 0 element of that bundle.

Interbundle references furnish a trick for getting several translations from one parsing rule to be put together by an invoking translation rule. The following example "doubles" a parenthesized list of identifiers. Members of the first output list are separated by /, of the second by \, and the lists are separated by |. The input "(a,b,c)" yields the output "a/b/c|a\b\c".

```
   double:   <(> dbla. <)> = { dbla.1 <|> dbla.0 };

   dbla:  ident.
          ( <,> dbla. = {ident </> dbla.1}. = {ident <\> dbla.0}
          | = { ident }. = { ident } );
```

The rule "dbla" builds two different output lists that are finally pasted together by "double". The notation "dbla.0" means just the same as "dbla"; it has been used to emphasize the fact that it is expected to evcke only part of the designated bundle.

5. ARITHMETIC

5.1 Variables and arithmetic elements

All arithmetic is performed on 16-bit two's complement integer data. An integer variable is declared and initialized by a labeled unsigned *octal* number, thus

```
          n:     1;
          size:  0;
```

An *arithmetic element* of a parsing rule is an expression enclosed in brackets [ ] that specifies a calculation to be performed as a parsing action. Expressions involve variables, called lvalues as in the language B [7], octal constants, parentheses and, in decreasing order of precedence

unary operators
infix operators
conditional operators
assignment operators

All operators except unary * return an rvalue. Their meanings (but not their precedences) are taken from B.

5.2 Unary operators

Unary operators in a primary expression are evaluated right to left. The unaries are

prefixed to an lvalue

++	increment and return new value
--	decrement and return new value
&	return lvalue

postfixed to an lvalue	
++	increment and return old value
--	decrement and return old value
prefixed to an rvalue	
*	indirection, take rvalue to be lvalue
¬	1's complement
!	not, !x means x?0:1
-	2's complement

5.3 Infix operators

Infix operators associate left-to-right

+	add
-	subtract
*	multiply
/	divide
%	remainder
&	and
!	or
^	exclusive or
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	equal to
!=	not equal to
>>	right shift (logical)
<<	left shift (logical)

The comparison operators return 1 or 0 for true or false.

5.4 Conditional operators

If e1, e2 and e3 are three rvalues, then the conditional expression

e1?e2:e3

has the value of e2 if e1 is nonzero, and otherwise e3. Only one of e2 or e3 is evaluated.

The operator : (regarded as an infix operator between e1?e2 and e3) associates from the right.

5.5 Assignment operators

The operator = assigns the rvalue on its right to the lvalue on its left. An = concatenated with any infix operator O is a "two-address code" assignment operator; x=O y means the same as x = x O y provided the evaluation of x has no side effects.

Assignment operators associate right-to-left.

5.6 Success and failure

If the expression in an arithmetic element is followed by a ? mark, then its rvalue is tested for nonzero (success) or zero (failure), otherwise an arithmetic element always succeeds. This is a simple Fortran-style do-loop:

```
begin: [i = 1]
loop:   . . .
        [++i<=n?]\loop;
```

5.7 Arrays

A static array is allocated by initializing more than one location with an octal constant, thus:

```
A:      1;2;3;4;
```

Subscripting is not directly provided for static arrays, but can be simulated by address computation, as in the expression *(&A+4), which when applied to the array A as initialized above would pick out the rvalue 3. (Addresses of successive words differ by 2, as is usual on the PDP-11.)

5.8 Variables in functions

Although the names of variables have global scope, their values can be pushed down for the duration of a rule, as in SNOBOL. The builtin "push(n,v1,v2,...,vn)" saves the current values of the n variables v1,v2,...,vn, to be automatically restored when the rule terminates. Variables to be pushed right at the start of a rule may be listed after a semicolon within the "proc(.)" declaration (3.5) instead of in a "push(.)".

The following rule recognizes an octal integer and assigns to its argument the numerical equivalent of the integer. The rule pushes its temporary, i, to avoid conflict with other uses of i in the program.

```
integer: proc(n;i) [n=0] inta
int1:    [n = n*10+i] inta\int1;
inta:    char(i) [i<70?] [(i =- 60)>=0?];
```

The builtin "char(i)" moves the cursor and assigns the ascii value of the scanned character to variable i. The rule depends upon knowing that the ascii codes for 0, 1, ... are octal 60, 61, ...

The next rule has the same effect as the builtin "octal(.)" for binary-to-octal conversion, provided the argument is not negative.

```
        octal:    proc(n;m,i) [i = (m=n)%10]
                  ( [i==0?] = {<0>}
                  | . . .
                  | [i==7?] = {<7>} )
                  [m =/ 10?]/done
                  octal(m) = { 1 2 };
        done:     ;
```

The purpose of m is to effect call by value. Were the rule written as below, it would nct work because of a collision between the argument passed by name and the temporary.

```
        octal:    proc(n;i) [i = n%10]
                  ( [i==0?] = {<0>}
                  | . . .
                  | [i==7?] = {<7>} )
                  [i = n/10]/done
                  octal(i) = { 1 2 };
        done:     ;
```

5.9 Character class operations

Each character class is represented by a one-word mask. The mask for each class declared in angle brackets <<>> contains exactly one nonzero bit, different for each class. A zero mask denotes the empty class, so "ignore(0)" and "ignore(<<>>)" behave similarly, except the latter uses up one of the 16 mask bits.

Words made by or-ing may serve as masks for classes made from unions of other classes. For example, given the following declarations, the element [letter = ucase | lcase] makes "letter" become the class of all letters in either case:

```
          ucase:    <<ABCDEFGHIJKLMNOPQRSTUVWXYZ>>;
          lcase:    <<abcdefghijklmnopqrstuvwxyz>>;
          letter:   0;
```

6. TABLES

A table is a dynamically allocated array, identified by an nonzero integer designator. The builtin "table(t)" allocates a new table and assigns its designator to the integer variable t. The builtin "discard(t)" destroys the designated table.

Bytes of a table are indexed. In arithmetic expressions the indexing notation, t[i], refers to the word occupying bytes i and i+1 of table t (i may be odd).

A table may be used as a symbol table that holds strings and one value word of arbitrary information associated with each. The builtins

```
                         find(t,i)
                         enter(t,i)
```

look up the current string in table t. If the string isn't already there, "find(t,i)" fails, while "enter(t,i)" adds the string and sets its value word to zero, unless the string is empty. When they succeed, both assign the index of the value word to variable i. No arithmetic assignments should be made to any words of a symbol table other than value words.

The builtin "getnam(t,i)" delivers the string of a symbol table entry for a given index. Here is a variant of the rule "install", given on page 4:

```
        install: enter(t,i) getnam(t,i);
```

This version delivers a trivial alias--"X" followed by the index:

```
        install: enter(t,i) octal(i) = { <X> 1 };
```

The next version delivers an alias that counts the temporal order of entries. The first entry has alias X1, the second X2, and so on. (In reading this example, remember that "=" does an assignment, not a comparison.)

```
        install: enter(t,i)
                 ([temp=t[i]?] | [temp=t[i]=++count])
                 octal(temp) = { <X> 1 };
        count:   0;
        temp:    0;
```

Notice that the argument of "octal(.)" is a simple variable. "Octal(t[i])" is not a legal function call.

Symbol tables are kept tree-sorted. Tables are stored on disk and pertinent pages are brought into addressable memory as needed. Erratic accessing patterns through large tables can thus be costly in time. If no information is to be stored with them, the strings may not have to be tabled at all. The first version of "install" on this page can be simulated, except in the handling of null strings, by the builtin "scopy", which delivers the current string.

```
                         install: scopy;
```

7. REDUCTIONS ANALYSIS

A "pure" Tmg program, which uses no builtins except perhaps the basic lexical functions "smark", "any(.)", "string(.)" and "scopy" and no arithmetic, is a "top down" parsing and translation mechanism with limited backup capability. However the builtins "bundle" and "reduce(.)" are bottom-up actions characteristic of reductions analysis. A few other builtins have been added to facilitate reductions analysis.

S. C. Johnson and A. V. Aho have automated the construction of reductions analysis parsers for certain deterministic grammars, and the transliteration of these parsers into Tmg programs. Their methods promise to make Tmg translators considerably more perspicuous and less tedious to write, since they start from a BNF pattern for translation instead of from a parsing program. Most notably, they are able to handle ambiguous grammars, which are especially useful for describing special-case optimization. With Tmg available underneath it is possible to mix top-down and bottom-up to get the best of both.

The new builtins for simple LR(k) parsing are "stack", "unstack()", "accept" and "gotab" (8.2). "Bundle" and "reduce(n)" should not be intermixed in the same rule with "stack", "unstack(.)" and "accept". The use of these actions in real translators will be described by Johnson and Aho.

8. BUILTIN FUNCTIONS

8.1 General catalog

This catalog tells for each builtin what kind of arguments it requires, if any, and whether it may:

C	move the cursor
T	deliver a translation
F	fail

Conventional meanings

c	character class or name thereof
i	name of variable
n	number or name of variable
r	parenthesized parsing rule or name of rule
t	name of table designator

Name	CTF	Function
any(c)	CF	scan current character; succeed if in class c and add character to current string (see pages 1,6)
append(l)		append literal l to the current string
bundle	T	deliver (and make otherwise unavailable) to this rule the translation, if any, that this rule would deliver to its invoker if this rule terminated here (9)
char(i)	CF	assign the ascii equivalent of the next input character to variable i; fail if no more characters (15)
decimal(n)	T	deliver n as a decimal string, with a - sign if required

diag(r)	CF	execute rule r and execute the translation it delivers; append result to the diagnostic file; fail if r fails (4)
emit		execute and forget last translation delivered to this rule
discard(t)		discard table t
enter(t,i)	F	look up the current string in table t; enter if not there; assign its index to variable i; fail if current string is empty (17)
fail	F	fail unconditionally
find(t,i)	F	look up the current string in table t; assign its index to variable i; fail if not there (17)
getnam(t,i)	T	deliver the string of entry i in table t (17)
goto(r)		same as succ\r, but saves space and time
ignore(c)		the ignored class becomes c (6)
octal(n)	T	deliver n as an octal string
params(n)		make n parameters available to this rule (8)
parse(r)	CF	execute rule r and execute the translation it delivers; append result to output file; fail if r fails (3)
proc(l1;l2)		a declaration, not a true builtin; l1 and l2 are lists of names; performs "params(n1) push(n2,l2)" where n1 and n2 are lengths of l1 and l2 (8,15)
push(n,list)		list has form i1,i2,...,in, where i1,...,in are variable names; save the n values; restore them when this rule ends (15)
reduce(n)	T	bundle the last n translations delivered to this rule (10)
scopy	T	deliver the current string (17)
size(i)		assign the number of characters in current string to variable i
smark	C	move to next nonignored character; clear the current string (1,6)
string(c)	C	scan up to next character not in class c; add the scanned characters to the current string (1,6)
stop		stop the program and dump it
succ		succeed; a no-op
table(t)		make a new table; assign its designator to t (16)

8.2 Special builtins for shift-reduce parsing

Name	CTF	Function
accept	T	unstack remaining labels stacked during this rule and bundle (18)
gotab(list)		list has form s1,l1,s2,l2,...,0,ln; if top stacked label is s1 go to l1, if s2 go to l2, ... else go to ln
stack		place label of this element on stack (18)
unstack(n)	T	remove last n labels stacked during this rule and bundle all translations delivered since the label so uncovered was stacked (18)

9. SYNTAX

9.1 Conventions

In the following syntactic specification terminal symbols are underlined; nonterminals have names one or more letters long; all symbols are separated by spaces. Each rule gives the name of a nonterminal followed by the metasymbol ::=, then displays the productions for that nonterminal separated by ¦ signs.

Brackets [] surround parts of a rule that may be repeated. The right bracket is followed by a subscript denoting the minimum number of repetitions, and a superscript for the maximum. A missing superscript permits unbounded repetition.

These primitive nonterminal symbols are used

name	a string of letters and digits beginning with a letter
number	a nonempty string of octal digits
char	any ascii character except NUL

In general one or more blanks (ascii SP, HT or NL) must appear between successive constituents of a production; however they may be dropped when no ambiguities are so introduced.

9.2 The grammar

program ::= [statement $]_0$

statement ::= comment ¦ [label $]_1$ tail ;

label ::= name :

tail ::= [proc $]_0^1$ prule [label prule $]_0$
¦ trule ¦ charcl ¦ number [; number $]_0$

proc ::= proc([names $]_0^1$ [; [names $]_0^1$ $]_0^1$)

prule ::= [disj [| disj $]_0$ $]_0^1$

disj[1] ::= [pelem [, [name $]_0^1$ $]_0^1$ $]_1$

pelem ::= pprime [[\ ¦ / $]_1^1$ pname $]_0^1$ ¦ (prule)

pprime[2] ::= pname [, $]_0^1$ [(parg [, parg $]_0$) $]_0^1$
¦ [expr [? $]_0^1$] ¦ literal
¦ = name ¦ = trule

pname ::= name ¦ $ number

parg ::= pname ¦ number ¦ (prule) ¦ literal ¦ charcl

```
expr         ::=  lv assign expr ¦ rv

rv           ::=  [ rv ? rv : ]_0 primary [ infix primary ]_0

primary      ::=  lv [ incdec ]_0^1 ¦ incdec lv

                  ¦ & lv ¦ ( expr ) ¦ unary primary ¦ number

lv           ::=  pname ¦ * primary ¦ ( lv ) ¦ lv [ expr ]

unary        ::=  - ¦ ! ¦ ~

assign[2]    ::=  = [ infix ]_0^1

incdec       ::=  ++ ¦ --

infix        ::=  + ¦ - ¦ * ¦ / ¦ % ¦ & ¦ ^ ¦ << ¦ >>

                  ¦ == ¦ != ¦ < ¦ > ¦ <= ¦ >=

trule        ::=  [ ( [ number ¦ names ]_1^1 ) ]_0^1 tbody

tbody        ::=  { [ telem ]_0 }

telem        ::=  name ¦ literal ¦ $ number

                  ¦ bundleref [ ( targ [ , targ ]_0 ) ]_0^1

bundleref    ::=  [ name ¦ number ]_1^1 [ , number ]_0^1

targ         ::=  name ¦ tbody

names        ::=  name [ , name ]_0

literal[3]   ::=  < [ char ]_1 > ¦ *

charcl[4]    ::=  [ ! ]_0^1 [ << [ char ]_0 >> ]_1

comment[5]   ::=  /* [ char ]_0 */
```

[1] if pelem begins with pname , it can not be followed by ,

[2] no spaces are permitted:
within an assignment operator
just before (in pprime

[3] no char after first may be >, blanks count as chars

[4] no char before last may be >, blanks count as chars

[5] [char $]_0$ must not contain */

10. SOURCES

Tmg has a long history stemming from McClure's work on the CDC 1604, and subsequent development on the IBM 7090, GE 635-645, and DEC PDP-7 by the author, R. Morris and M. E. Barton [1,2]. Some of the present design derives from insights from language theory for which I am indebted to A. V. Aho and J. D. Ullman [3,4]. I have freely borrowed code and appropriated language ideas from R. Morris, L. L. Cherry, S. C. Johnson, K. L. Thompson and D. M. Ritchie.

Enough like its predecessors to deserve the same name, this implementation of Tmg and its unpublished predecessor on the PDP-7 introduced a new parsing discipline that has made possible the use of reductions analysis, the avoidance of backup within a rule (thereby augmenting the class of languages "naturally" parsable by Tmg) and rules with parameters. Shallower, but nonetheless useful innovations are tables and the handling of the current string, uniform treatment of diagnostic and translated output, the form of translation bodies, success branches, disjunctions and other syntactic conventions. Internally, improved handling of character classes and the elimination of many levels of subroutine call both in parsing and translation have improved the speed of Tmg; dynamically allocated tables have extended its capacity so that it may fit comfortably in a minicomputer.

[1] R. M. McClure, TMG--A syntax-directed compiler, Proc. ACM 20th Natl. Conf. (1965) 262-274

[2] R. R. Fenichel and M. D. McIlroy, Reference Manual for TMGL, Multics System Programmer's Manual, Project MAC, MIT (1967) Section BN 4.02

[3] A. V. Aho, P. J. Denning and J. D. Ullman, Weak and Mixed Strategy Precedence Parsing, JACM (19) 225-243

[4] A. Birman and J. D. Ullman, Parsing Algorithms with Backtrack, Conf. Record 11th Annual Symposium on Switching and Automata Theory, IEEE (1970) 153-174