Published: 04/17/67

Identification

TMGL
Robert R. Fenichel and M. D. McIlroy

Purpose

This section describes the syntax and semantics of TMGL. TMGL is of interest in Multics because the EPL compiler is written in TMGL.

Structure of this Section

The remainder of this section consists of eleven subdivisions as follows:

Those of these pieces which are not motivated by others are made intelligible by others. No single order of reading can be recommended.

For tutorial purposes, the most useful portions of this description are thought to be subsections A, B, C, H, and I. There is no particular order in which these sections should be read, however. Instead, the reader is advised to utilize a chimneying technique, climbing not one wall or another, but rather all at once. Section BN.4.02 A is an index to this section.

A. Components

A TMGL program consists largely of sequences of components. Components are frequently the names of syntactic types for which representatives are being sought in the input stream. Other components alter internal tables, and still others test the contents of these tables. Finally, an important class of components is concerned with the output generation process. In brief, when a component C is matched,

1. It may or may not advance the scanning of the input stream.

2. It may or may not have side effects on the values of compiler parameters or contents of compiler tables.

3. It may or may not add one line to the output table being produced for the larger component C' of which C is a part. In particular, this new line will be either

   a. A pointer to the output table which was produced as the substructure of C was matched, or
   b. A definition of the output to be produced for C'. This definition may include literal information as well as pointers to previous entries in the output table of C'.

4. It may succeed or fail. For example, C may test the input stream for the presence of a certain literal string. If this string is not present, this component will fail. A component which fails does not add anything to any output table, nor does it advance the scanning of the input stream.

A component may be matched because it is a subcomponent of another component, or -- via a mechanism which will be described later -- it may be matched at the top level. For example, the entire compilation process might consist

of matching the component "program". After a component has been successfully matched at the top level, the output table that it has produced is activated. To activate an output table is to generate output according to the specifications of the definition at the end of this table.

The output just mentioned is appended, as soon as it is produced, to the object stream. The other output streams of TMG are the console stream, usually used for infrequent diagnostic messages, and the BCD stream, usually used for a printable listing.

Types of Components

This subsection includes a number of quasi - BNF forms. Do not trust them too much. The real syntax of TMGL is given in subsection E.

1. Literal string components

   A literal string component is written

   $<string of length n>$

   or

   <name of string of length n>

The mechanism for naming character classes is described in subsection E. This component will fail unless the next n characters of the input stream are those of the displayed or named string. After this component has been successfully matched, the input scan is advanced by n characters.

2. Character class components

   A character class component is written

   <name of character class>

The mechanism for naming character classes is described in subsection E. The character-class component will fail if the next character of the input stream is not a member of the named class. After this component has been successfully matched, the input scan is advanced by one character.

3. Starred character class components

A starred character class component is written

<name of character class>*

This component will always succeed. It has the effect of spacing forward in the input (say, n characters) until the character "under the read head" is not a member of the named class.

The n=0 case is perfectly legitimate.

4. OR Components

An OR component is written

<component> .OR. <component>

If the first subcomponent of an OR component succeeds, the second is ignored. If the first fails, the second is matched. If and only if both subcomponents fail, the whole OR component fails.

5. Function call components

A function call component is written

<function name>

or

<function name> (<arguments>)

A component of this sort may or may not advance the input scan. It may or may not prepare output on one of the output streams, or in a tentative output table. Each function used in a TMGL program must be declared; the syntax of declarations is described in subsection E. The available functions are listed in subsection F, but a few examples will be listed here.

MARKS This component always succeeds. It queers the input program so that until further notice, scanned input characters are placed in the symbol buffer.

NAME This component fails only if the symbol buffer is empty. Otherwise, the MARKS mode is turned off and the symbol buffer is copied into the symbol table. An internal name, of the form XXnnnn, is generated for this symbol, and this internal name is appended to the output table of the component whereof the NAME component is a part. This symbol is made the current symbol.

GETNAM This component always succeeds. It causes the external name (that is, the actual letters) of the current symbol to be placed in the output table of the component whereof this component is a part.

INSTAL This component functions exactly like NAME, except that the actual symbol (not the internal name) is placed in the output table of the component whereof this component is a part.

COMPUTE (VARIABLE = EXPRESSION)
This component always succeeds. It has the effect of performing the assignment indicated. Each variable appearing in the argument of COMPUTE must be declared; syntax for declarations and expressions is described in subsection E.

One important class of variables is that of system cells. The available system cells are listed in subsection G, but two important ones will be mentioned here.

J is the character-scanner. One way of skipping two characters is to match the component COMPUTE (J = J+2).

EQUADR is the symbol-table position of the current symbol. To return attention to the symbol with position n, match the component COMPUTE (EQUADR = n).

6. Anomalous Components

There are exactly two anomalous components.

** This component always fails.

// This component fails unless the next character in the input stream is a carriage return. After this component has been successfully matched, the scan has been advanced beyond the carriage return.

7. Definition Components

A definition component is written

$(<string>$)

or

(<integer>) $ (<string>$)

A definition component always succeeds. The component is literally added to the output table of the component whereof this component is a part.

8. Sentence-name components

A sentence-name component is written

<name of sentence>

A character-string is the name of a sentence if the string, immediately followed by two periods, appears at the left-hand edge of some TMGL line. When a character string so appears, it defines the start of a trail whose passage involves the matching of various components.

The trail-following algorithm is discussed in subsection H.

Sooner or later, the trail will come to an end. If and only if (but see functions exit, pl, null) a definition component has been reached yet, the sentence-name component has succeeded.

As the trail is followed, the components matched may be adding to the output table of the sentence-name component. When the trail comes to an end, this output table is trimmed back to where it was just after the last definition component was matched.

If the sentence-name component C succeeds, it has some output table T. A pointer to T is placed in the output

table T' of the component C' whereof C is a part. To say that C is a part of C', we now understand, means simply that C is encountered along the trail out of C'.

B. Grand Plan

Every TMGL-written compiler follows the following scheme:

1. By means of a mechanism described in subsection E, a super sentence name is found.

2. The trail from the super sentence name is followed. As soon as any immediate subcomponent of the super sentence name succeeds and produces an output table, that output table is activated. (Warning: see subsection J, items 4 and 8).

3. When the trail comes to an end, the compilation is complete.

Thus, for example, consider the problem of programming a compiler to "translate" English text by reversing the spelling of each word. This compiler would translate "You never outgrow your need for Multics" into "Ouy reven worgtuo ruoy deen rof scitlum."

A complete TMGL program to do this is displayed in subsection I. In brief, the structure of this compiler is as follows:

1. "Word" is a sentence-name component which swallows a word from the input stream and whose output table, when activated, spews out the reverse of the input word.

2. "Punctuation" is a sentence-name component which swallows any stream of spaces, carriage returns, and punctuation, and whose output table, when activated, spews out the string which has been swallowed.

3. The trail starting from the super sentence name loops through "word" and "punctuation" components until it can't find input of either kind; then it comes to an end.

C. Output Tables and Definitions

Activation actually is a process which, like any evaluation

scheme, requires two parameters. These are an expression (in this case, the output table) and a list of bindings for free variables. As it happens, most output tables do not contain any free variables.

At the time an output table is activated, its most recent entry must be a definition component which has placed itself in the table. This fact is readily deducible from the discussion of sentence-name components above. Activation is primarily governed by the form of definition components.

A definition component of the form

=$(<string>$)

can never contain any free variables. Consequently, additional arguments are errors when this component is activated.

A definition component of the form

=(<integer>) $ (<string>$)

may contain up to n free variables, where n is the value of the integer. If bindings are not supplied at activation time, they are generated in the XXnnnn series.

Most of the business of activating a definition component consists of going down the string from left to right, activating the elements found there. There are nine different sorts of elements:

1. If two consecutive slashes are found, or if a new-line character is found, then a new-line character is added to the object stream.

2. If a single slash is found, or if three or more consecutive blanks are found, a tab character is added to the object stream.

3. If $F1 (<identifier>) is found, then the value of the identified variable is incremented by one.

4. If $F2 (<identifier>) is found, then the value of the identified variable is decremented by one.

5. If $F3 (<identifier>) is found, then a decimal representation of the value of the identified variable is added to the object stream.

6. If $Qn is found, where n is some integer, then the nth activation binding is activated. The $Qn's, in other words, are the free variables mentioned in page 10.

7. If $Pn is found, where n is some integer, then the nth previous entry in this output table is activated.

8. If SPn.m is found, where n and m are integers, then the nth previous entry (that is, the (n+1)st topmost entry), in this output table is examined. This entry should be a pointer to another output table (as opposed, for example, to a literal string). The result of activating $Pn.m is the activation of the mth topmost entry in the table T, when T is the (n+1)st topmost in the current table.

Either $Pn or $Pn.m may be followed by arguments, listed between parentheses and separated by commas. If this is done, the output-table entry referred to by the $Pn or $Pn.m must be a pointer to an output table whose top entry is a definition of the form (<integer>) $ (stuff$). The arguments are then the activation bindings of this definition. The syntax for arguments is specified in subsection E.

9. If $ <character> is found, where the character is not Q, P, F, or), the character is appended to the object stream.

10. If the activation process comes upon text which is not of one of the styles (1) - (9), that text is literally appended to the object stream.

D. Character Set

The following ASCII characters are handled unambiguously by TMG:

greater than	less than
blank	new line
period	comma

colon	semicolon
brackets	parentheses
question mark	circumflex
double quote	underline
plus	minus
star	slash
dollar	vertical
equals	ampersand
decimal digits	

The 52 ASCII letters are mapped into a 26-character set of indeterminate case.

Each ASCII HT character is mapped into a blank.

Each other ASCII character is mapped into a percent sign.

E. TMGL Syntax

0. A TMGL Program consists of a header division, a declaration division, a definition division, and an end card.

<tmgl program>::=<header division>
<declaration division>
<sentence division>
<definition division>
<end card>

1. The header division generally consists of two cards. Neither of these contains any information actually used by the TMGL compiler, but both are necessary.

<header division>::=SYNTAX<blank>FOR<blank> <id> <NL>

<.SYNTAX.> <NL>

<id>::=<letter>|<id> <letter>|<id> <digit>|<id> <minus sign>

Optionally, the header division just described may be preceded by

$TMG<blank>N$S$URCE

This suppresses the source listing of the TMGL program; it is sometimes useful.

Also optionally, arbitrary FAP cards may come between the two required cards. These FAP cards will be copied to the FAP file of driving tables being produced from this TMGL program.

2. The declaration division consists of declarations and interspersed comments. Declarations are used to identify the super sentence name, to assign names to system functions and system cells which will be used, to identify and initialize variables, to name fixed strings, to name fixed character classes, to declare that certain identifiers will name flags which may be set on entries in the symbol table, to set logical tab stops in the object stream, and to identify vectors which will be used within the compiler.

<declaration division>::=<dc>|<declaration division> <dc>

<dc>::=<declaration>|<comment>

<comment>::=*<string not including<NL>> <NL>

<declaration>::=<super sentence declaration>|

<system function declaration>|

<system cell declaration>|

<fixed string declaration>|

<character class declaration>|

<flag declaration>|

<tab declaration>|

<vector declaration>|

2.1 The super sentence declaration determines the overall flow of the compilation, as described in subsection of B. Each compiler needs exactly one super sentence declaration.

<super sentence declaration>::=<sentence name>=
PROGRM <NL>

<sentence name>::=<id>

2.2 Each system function used in a compiler must be declared. At the time of declaring the function, the user must assign to it an arbitrary identifier for use in his compiler. The cases of functions with and without an explicit argument are distinguished.

<system function declaration>::=<function name>=
<fd> <NL>

<function name>::=<id>

<fd>::=*<name of no-argument system function>|

**<name of one-argument system function>

<name of system function>::=See subsection F.

2.3 System cells may be given arbitrary names and then used as variables in a compiler. Each system cell used as a variable must be declared.

<system cell declaration>::=<variable name>=
<system cell name> <NL>

<variable name>::=<id>

<system cell name>::=See subsection G.

2.4 Variables other than system cells must be declared if they are to be used. At the time of declaration,

an initial value must be assigned.

<variable declaration>::=<variable name>=<signed integer><NL>

<signed integer>::=<integer>|-<integer>|+<integer>

<integer>::=<decimal integer>|<octal integer>

<decimal integer>::=<digit>|<decimal integer> <digit>

<octal integer>::=<octal digit>B|<octal digit><octal integer>

2.5 It may be convenient to allow an identifier to denote a fixed character-string throughout a compiler. The syntax of this declaration is complicated by the special role of dollar signs in TMGL.

Dollar signs are used as string delimiters in TMGL, so they may not appear within strings as other characters may. However, the TMGL syntax does not admit zero-length fixed strings. A dollar sign may, therefore, appear as the first character of a TMGL fixed string. In that position, it is not taken to be the final delimiter.

<fixed string declaration>::=<string name>=$<TMG string>$<NL>

<fixed name>::=<id>

<TMG string>::=$<tmgs>|<tmgs>

<tmgs>::=<string of any characters but $ and <NL>>

2.6 A name may be given to any subset of the TMGL character set which does not contain <NL>.

<character class declaration>::=<character class name>=

CHARCL($<TMG string>$)

<character class name>::=<id>

2.7 Certain system functions described in subsection F allow named flags to be individually set and cleared on individual entries in the symbol table. Each flag to be used must be declared.

<flag declaration>::=.FLAGS.<blank><flaglist> <NL>

<flaglist>::=<flagname>|<flaglist> <fbreak> <flagname>

<flagname>::=<id>

<fbreak>::=<comma>|<blank>|<fbreak> <blank>

2.8 The setting of logical tab stops is ignored by the TMGL compiler, but a tab stop declaration must appear in the compiler.

<tab declaration>::=.TABS.<blank> <tab list> <NL>

<tab list>::=<tab stop>|<tab list> <fbreak> <tab stop>

<tab stop>::= <decimal integer>

2.9 Vectors may be useful within the compiler; each must be declared with a subscript bound

<vector declaration>::=.LIST.<blank> <vector list> <NL>

<vector list>::=<vdesc>|<vector list> <fbreak> <vdesc>

<vdesc>::= <vector name> (<integer>)

<vector name>::=<id>

3. The sentence division consists of sentences and interspersed comments. Each sentence begins with its unique sentence name and ends with the first subject after it. Sentence B may begin within sentence A, in which case A and B will end with the same subject.

It is tempting to say that each sentence is a complete description of the syntax of the structure associated with the sentence-name. Actually, things are less simple, but the reader is referred to subsection H for a more careful discussion.

```
<sentence division>::=<sc>|<sentence division> <sc>

<sc>::=<sentence>|<comment>|<pseudo sentence>

<pseudo sentence>::=see subsection F, under "valbra"

<sentence>::=<sentence beginning> <subject> <NL>|
             <sentence beginning> <sentence middle>
                                  <NL> <sentence>

<sentence beginning>::=<sentence label>

<sentence label>::=<sentence name>..

<sentence middle>::=<element list>|
                    <break> <sentence middle>|
                    <sentence middle> <break>

<break>::=<blank>|<break> <break>|<NL> <break>

<element list>::=<element>|<element> <break> <element list>

<subject>::==<definition>|=<definition>/<continuation>

<element>::=<non-definition component>|
            <non-definition component>/<detour>

<non-definition component>::=<literal string component>|
                             <character class component>|
                             <starred character class
                              component>|
                             <OR component>|
                             <function call component>|
                             <anomalous component>|
                             <sentence name component>
```

<literal string component>::=$<TMG string>$|<string name>

<character class component>::=<character class name>

<starred character class component>::=<character class name>*

<OR component>::=<non-definition component> <blanks>
.OR.<break> <non-definition component>

<blanks>::=<blank>|<blank> <blanks>

<function call component>::=<fc with>|<fc without>

<fc without>::=<function name>

<fc with>::=<function name> (<argument>)

<argument>::= format varies from function to function.
See subsection F.

<anomalous component>::=**|//

<sentence name component>::=<sentence name>

<continuation>::=<sentence name>

<detour>::=<relative detour>|<sentence name>|(<sentence
fragment>)

<sentence fragment>::=<sentence middle>|<subject>|
(<sentence middle> <subject>

<relative detour>::=*<addop> <integer>

<addop>::= +|<blanks>-

<definition::=<definition name>|
(<integer>)$(<definition text>$)|
$(<definition text>$)

<definition text>::=<definition element>|
<definition element> <definition text>

<definition element>::=<NL definition>|<tab definition>|
<F definition>|<P definition>|
<Q definition>|<quoted character def>|
<free definition>

<NL definition>::=<NL>|//

<tab definition>::=<three or more consecutive blanks>|/

<F definition>::=$F <integer>(<variable name>)

<P definition>::=<P def>|<P def>(<P arglist>)

<P def>::=$P <integer>|$P <integer>.<integer>

<P arglist>::=<definition name>|
<definition name>, <P arglist>

<Q definition>::=$Q <integer>

<quoted character def>::=$<character not Q, P, F, or)>

<free definition>::=<text not containing other definition elements>

4. Definition names are associated with their definitions in the definitions division.

<definitions division>::=.DEFINITIONS.<NL>|
<definitions division> <defc>

<defc>::=<definition line>|<comment>

<definition line>::=<definition name>=<definition> <NL>

<definition name>::=<id>

5. An end card must terminate each TMGL program.

<end card>::=END<NL>

F. System Functions in TMGL

Method of description

When used as components, most functions always succeed. Functions which sometimes (or always) fail are labeled "F" below.

When some functions succeed, they add to the output tables of the components whereof they are parts. Other functions never make any table entries. Functions which do make such entries are labeled "T" below.

Some functions never advance J, the input scan pointer. Functions which sometimes (or always) advance J are labeled "J" below.

1. Mode-setting functions

blanks Causes blanks to be significant in input.

noblks Causes blanks to be ignored in input. This is the default case.

---

cardof Causes new-line characters in input to be treated as blanks.

cardon Causes new-line characters in input to be significant. This is the default case.

---

yescom Causes strings of the form of PL/I comments (/*string*/) to be ignored whenever noblks mode is in effect.

nocom Forces the programmer to handle PL/I comments above the adapter level. This is the default case.

---

lists Causes each compilation to produce a source listing in the BCD stream. Also causes collection of cross-references. This is the default case.

nolsts Suppresses production of source listing and collection of cross-references.

---

refoff Suppresses collection of cross references.

refon Causes collection of cross-references.

2. Text-scanning functions

char(<character class name>) JF
Same as unadorned <character class name>

string (<character class name>) J
Same as <character class name>*

delete J Same as "blank*", where "blank" is the character class whose sole member is the blank character.

eolmrk JF Same as "//".

glot JT Runs wild if used in cardof mode. Otherwise, enters in output table the string which starts at J and is terminated by (does not include) the next new-line character.

mark Saves current scan position (J) in a secret place.

reset J Returns scan to position it had at last use of mark.

3. Symbol table routines

marks — If in noblks mode, executes delete. Then, causes blanks/noblks mode to be set to blanks. Until further notice, all scanned characters go into a symbol buffer, which is initially cleared.

instal TF — Fails if symbol buffer is empty. Otherwise, the symbol buffer is combined with system cell contxt to determine a location (equadr) in symbol table. System cell intval is set to contain the value of the decimal integer, if any, at the head of the symbol buffer. This symbol-table entry, if it was not set before, is set to show the contents of the symbol buffer as the external name of the symbol. This external name is entered into the output table. An internal name, of the form XXnnnn, is associated with the symbol-table slot.

name TF — Same as instal, but the created name, not the external name, is placed in the output table.

find F — Similar to instal, but no output-table or symbol-table entry is made. A typical use for find is in compilers for block-structured languages. Several concentric contexts may give rise to a number of possible symbol-table slots for a variable of given external name. Before deciding that an appearance of the variable is new, the compiler must try to find the variable with contxt set to each surrounding value.

enter (<literal string component>) T
Same as name, but the string is used instead of the contents of the symbol buffer.

alloc T — Similar to "name", but a null string is used instead of the contents of the symbol buffer. Useful for temporary storage; each use produces a new slot.

getnam T — Puts external name of current symbol (identified by equadr) in output table.

namest T — Same as getnam.

getcrn T Puts internal name of current symbol in output table.

getval Sets intval to correspond to current symbol (see instal).

---

noflgs F Fails if any flag is set on the current symbol.

chkflg(<flagname>) F
Fails if the named flag is not set on the current symbol.

setflg(<flagname>)
Sets the named flag on the current symbol.

clrflg(<flagname>)
Clears the named flag from the current symbol.

---

dict Causes printing of cross-reference table in the BCD stream.

---

subsav Stores external name of current symbol in a secret place.

subst Replaces internal name of current symbol by the name last saved by subsav. These two routines are used, for example, in the TMGL compiler itself. The declaration

```
name1 = *name2
```

causes name2 to become the internal name of name1, among other things. This is accomplished by subsav/subst.

4. Parse control functions

alex Does nothing

exit Somewhat similar in effect to =$($). That is, this component ends the trail which led to it. However, this component is different from =$($) in that it makes no entries in any output table. Even its predecessors' output (since the last subject along the trail) is discarded.

nogo F Same as "**", but nogo stops even deader, ignoring any following detour.

null Acts like "=$($)", even to the point of treating its detour like a continuation. See subsection J, item #3.

parsdo(<parsdo argument>) J
Causes the trail starting at the argument to be followed. Each time a component along this trail produces an output-table entry, thet entry is activated. The trail must end by failing. Running off the right end of the argument leads to disorder. The parsdo function is used by the TMG interpreter to process the super sentence. Rather than store up an enormous output structure, a sentence may utilize parsdo to get rid of output as soon as it is determined. See subsection J, items 4 and 8.

<parsdo argument>::=<sentence name>|

<element> <break> <element list>

not(<sentence middle>) F
Succeeds only if its argument, if it were here in place of this component, would fail. Not may sneak J ahead to test its argument against the input, but the input will later be backed up. For example, matching a string constant with "$$$ marks letter* instal $$$" may, if the second "$" is not found, cause a spurious entry in the symbol table. Better is

$$$ marks letter* not (not($$$)) instal

arbno(<sentence middle>) JT
Same as component x, with

x..=$($)/x1

x1..<sentence middle>=$($p2$p1$)

valbra(<varg>) F
Serves as a subscripted transfer. Valbra (<vs name>) is the same as

valbra (<vs interior>)

where this is the interior associated with the name in a <v pseudo>.

Valbra (n n1/e1 stuff) is the same as

if (n .ne. n1)/e1 valbra (n stuff)

except that subsection J, item 1 does not apply.

valbra (n **) fails.

<pseudo sentence>::=<v pseudo>|<e pseudo>|<l pseudo>

<varg>::=<vs name>|<vs interior>

<e pseudo>::=see below, under "encode"

<l pseudo>::=see below, under "local"

<v pseudo>::=<vs name>..<vs interior> <NL>

<vs name>::=<id>

<vs interior>::=<primary> <break>**|

<primary> <break> <ve list> <break>**

<ve list>::=<primary>/<detour>|

<primary>/<detour> <break> <ve list>

---

savtre Copies last output-table entry into system-cell tree. See subsection J, item 3.

getree T Enters contents of tree into output table. These functions are of some value in handling assignment statements. Code for the left-hand side is easy to prepare when that side is first seen. But there is

no sense in emitting that code until the right-hand side is translated. The left-hand code may be saved and then emitted with savtre/getree.

if (<bexpr>) F Fails if and only if the boolean expression is false.

See subsection J, item 5.

<bexpr>::=<bprimary>|<bprimary>.AND. <bterm>

<bprimary>::=<relation>|(<bexpr>)|.NOT. <bprimary>

<relation>::=<aexpr>.<relop>.<aexpr>

<relop>::= NE|E|L|LE|G|GE|EQ|LT|GT

<aexpr>::=See below, under compute.

5. Miscellaneous Functions

encode (<earg>) JF
Sets a multiple-position switch to show which (if any) of a set of enumerated fixed strings occurs in the input stream.

encode (<es name>) is the same as encode (<es interior>) where this interior is associated with this name in an <e pseudo>.

encode (n s/n1 stuff) is the same as x,

where

x.. s/x1 compute (n=n1) exit

x1..encode (n stuff)

encode (n **) fails

<earg>::=<es name>|<es interior>

<e psuedo>::=<es name>..<es interior> <NL>

<es name>::=<id>

<es interior>::=<primary> <break> **|

<primary> <break> <ee list> <break>**

<ee list>::=$<TMG string>$/<primary>|

$<TMG string>$/<primary> <break> <ee list>

type (<literal string component>)
Causes the last input line and the given string to be appended to the console stream.

---

cvtd (<variable name>) T
Adds to the output table the decimal representation of the value of the given primary. The variable may not be a system cell.

cvto (<variable name>) T
Adds to the output table the octal representation of the value of the given primary. The variable may not be a system cell.

---

compute (<assignment list>) J
Makes assignments to variables and to descriptor-list positions. See subsection J, item 5. The striking thing about the "compute" function is that the effects of "compute" are never withdrawn. This fact deserves careful discussion.

Suppose the sentence-component S is being matched, and suppose that the trail from S runs through non-definition components A, B, and C. If A and B both succeed and C fails, S should fail and the side-effects of matching A and B should be rescinded. For example, matching A might have caused an advance of the scan pointer, J] Before continuing at S's detour, the system must restore the value of J which was in effect when S was entered. Other, nameless, system variables may also be restored.

The effects of matching compute components are not generally rescinded.

Variables mentioned in "compute" arguments may be of any of five types. Ordinary declared variables and system cells are the simplest of these.

Vector elements may be set by simply naming the vector within parentheses. The subscript is implicit, and initially one. To change or to use the value of the subscript of a vector, manipulate the vector name.

Connected to each entry in the symbol table is a description list of indefinite length. To refer to the nth entry in the descriptor list of the current symbol, use the notation "DESC(n)".

<assignment list>::=<assignment>|

<assignment>, <assignment list>

<assignment>::=<lhs>=<aexpr>

<lhs>::=<variable name>|<vector name>|(<vector name>)|

DESC (<aexpr>)

<aexpr>::=<addop> <term>|<term> <addop> <aexpr>|<term>

<term>::=<primary>|<primary> <mop> <term>

<primary>::=<integer>|<lhs>|(<aexpr>)

Note that DESC, if it is used, must be declared as if it were a built-in, argument-taking function of the same name. For example, see the word-counting program in subsection I.

local (<larg>)

Hides the values of named variables (other than system cells). If the trail now passing through local comes to an end before coming to a subject, most components will, as mentioned above with compute, have their effects rescinded.

The effect of rescinding local is to restore the hidden values to the variables.

local (<ls name>) is the same as local (<ls interior>) where this interior is associated with this name in an <l pseudo>

<larg>::=<ls name>|<ls interior>

<l pseudo>::=<ls name>..<ls interior> <NL>

<ls interior>::=<variable name> <blanks> **|

<variable name> <blanks> <ls interior>

G. System Cells in TMG

tree — Used by getree/savtre (subsection F)

contxt — Combined with external name to find a symbol-table slot. See instal, name, find, and enter in subsection F.

J — Scan pointer, counts one per character, goes up to next multiple of 80 at new-line character.

equadr — Specifies which slot of symbol table is that of current symbol. Can be set by the following functions:

compute	find	enter
instal	name	alloc

intval — If equadr set by anything besides compute, or if getval matched, contains value of decimal integer (if any) at beginning of current symbol.

H. Trail-following in TMG

When a sentence-name component is matched, a trail is followed through various parts of the compiler. The course of this trail determines whether or not the sentence-name component will succeed, and, if it will succeed, what output table will be prepared for it.

This subsection describes TMG's trail-following algorithm.

When sentence-name component A is to be matched, TMG looks for a sentence beginning "A..". Exactly one such sentence should be found. The trail from "A.." generally proceeds from left to right, ignoring card boundaries and ignoring sentence labels (e.g., "B..") of embedded sentences.

As subsection E makes plain, there are then two sorts of objects which may be encountered in this left-to-right motion: elements and subjects.

Subsection E also notes that there are two sorts of elements, those with detours and those without.

An element without a detour is just a non-definition component. This component is matched. If the match succeeds, the blanks/noblks mode is restored to what it was before the match, and the trail continues to the right. If the match fails, the trail comes to an end (almost - see Item 1 in subsection I).

Each element E of the other sort consists of a non-definition component and a detour. The component is matched, and if the match is successful, the trail continues to the right.

If the component fails, the detour is examined. There are three distinguishable cases.

The case of a relative detour is simple in concept. The detour "*+1", for example, generally allows the trail to continue at the next element right-ward. Thus, an optional comma could be matched by the element "$,$/*+1". Before using relative detours, check with Item 2 of subsection I.

When the detour is a sentence name S, the trail continues from the sentence label S.

When the detour consists of some parenthesized sentence fragment, the trail moves through this fragment (and probably beyond it) as if it had stood in the place of the original element E.

When the trail reaches a subject, the definition is matched. If there is no continuation portion, the trail comes to an end. If a continuation "C" is present, the trail continues from "C..".

I. Sample TMGL Programs

This subsection consists entirely of three highly-annotated TMGL programs. These programs are directed to (and written by) one who starts from ignorance of both TMGL and compiler structure. Consequently, the examples have nothing to do with compilers; TMGL is hairy enough by itself.

The three examples should be studied in the order in which they appear.

```
syntax for wordflipper
.syntax.
*
*     identify super sentence name
*
flipper=progrm
*
*     declare functions used
*
refoff=*refoff
blanks=*blanks
marks=*marks
install=*instal
type=**type
*
*     name useful character classes
*
letter=charcl($abcdefghijklmnopqrstuvwxyz*$)
puncts=charcl($ .,"():;?[ ]$)
garbage=charcl($$%<>^_+-/|=&0123456789$)
*
*       mandatory ".tabs." declaration
*
.tabs. 1
*       here is the super sentence.  it looks for "***" to
*  stop on.  If it doesnt get "***", it peels a word or string of
*  punctuation marks off the input.  Since this is the super
*  sentence, not just an ordinary one, the output table
*  prepared by each word or punctuation match is immediately
*  activated.
*
flipper.. refoff
flipper3.. $***$/flipper1 lastline   **
*
*        If the "***" had been found, we would have moved on to
*  "**" and to a halt.  But getting to flipper1 means no "***" yet.
*
flipper1.. word/flipper2
*
*  whether or not we have peeled off a word, we drop from flipper1
*  to flipper2,
*
```

```
flipper2.. punctuation/flipper4  **/flipper3
*
*  and if we find any punctuation, back we go to look for "***".
*
*     we use flipper4 to plow through illegal characters.
*
flipper4.. garbage*  type($illegal character in above line$)  **/flipper3


*       To handle a word, we split off the first letter and
*  put this out after the flipped version of the "word" which is left.
*  If there is nothing left, the flipped version of the original
*  word is just the first (only) letter.
*
word.. blanks marks letter install word/(=p1) =$($p1$p2$)
*
*       Notice that in the detour to the "word" component, we use
*  the abbreviation "p1" which we must define below.  The punctuation
*  copier picks up either a string of graphics or a newline character
*  each time it succeeds.  Notice that install will fail if "puncts*"
*  picks up nothing.
*
punctuation.. blanks marks puncts* install/punct2=p1
punct2.. //
lastline.. =$(//$)
*
*       here is that definition we needed.
*
.definitions.
p1=$($p1$)
end
```

```
Syntax for English-to-Pig-Latin
.syntax.
*
*       Pig Latin is probably the nearest approximation to cant
*   (e.g., Cockney rhyming slang) that the United States has ever
*   produced.  Several dialects exist, and this compiler may be
*   described as follows:
*
*       1.  Single-letter words are unchanged, as is punctuation.
*
*       2.  Each word beginning with a vowel has "YAY" appended.  Words
*   beginning with "y" immediately followed by a consonant ("yclept," for
*   example) are considered to fall into this category.
*
*       3.  Each word beginning with a consonant string is altered by
*   rotating that consonant string to the end and then appending "AY".
*   If a "q" is moved, as in "quid" or "squid," its "u" moves with it.
*   The letter "y" is not part of the initial consonant string in "scythe,"
*   although it is in "yaws."
*
*       For the purists, we here acknowledge that Pig Latin is really an
*   aurally constructed language.  Our rules would be rejected by any
*   nine-year-old, who would know that the Pig Latin for "once"
*   is "unsway," not "onceyay"; "hour" should become "ouryay," and so on.
*
*       There is another peculiarity to written Pig Latin, or rather to
*   the written-English-to-written-Pig-Latin translator.  The transformation
*   is expansive, possibly by a factor of two.  Accordingly, this compiler
*   maintains an output column counter, and a new-line character is
*   emitted whenever this output column counter gets large.  New-line
*   characters in the input file are ignored unless they precede paragraph
*   indentation.
*
*                   These declarations should not be mysterious
*
pigger=progrm
delete=*delete
refoff=*refoff
blanks=*blanks
marks=*marks
type=**type
install=*instal
vowel=charcl($aeiou$)
consonant=charcl($bcdfghjklmnpqrstvwxyz*$)
letter=charcl($abcdefghijklmnopqrstuvwxyz-$)
consonant-not-qy=charcl($bcdfghjklmnprstvwxz$)
puncts=charcl($ .,"():;?[ ]$)
garbage=charcl($$%<>^_+/|=&0123456789$)
.tabs. 1
*
```

```
*          A paragraph begins with five spaces
*
paragraph=$     $
*
*          declare variables for column-counting
*
startword=0
column-count=0
*
*         we need access to the system scan pointer, j
*
j=*j
*
*      If column-count gets here, we generate a new-line
*
right-margin=60
*
*              here is that super-sentence
*
pigger.. refoff
pigger-1.. $***$/pigger-2  lastline  **
pigger-2.. test-width  word/pigger-3
pigger-3..  punctuation/pigger-5  **/pigger-1
pigger-5.. pigger-4  **/pigger-1
pigger-4..  garbage*  type($mess above$)  =$( $)
*
*       "test-width" is a modest little routine which always succeeds
*  and which emits new-line characters as needed.
*
test-width.. if(column-count .ge. right-margin)/(=$($))
          blanks  delete  compute(column-count=0)
lastline..  =$(//$)
*
*       The "word" routine is not trivial.  First, it sorts out  the
*  easy case of one-letter words.
*
word.. blanks  one-letter-word .OR. big-word =p1
one-letter-word..  marks  letter  install
       not(letter)
       =$($f1(column-count)$p1$)
*
*      Now comes the monster.  It will be put down first, then explained
*
big-word..  compute(startword=j)  marks
            vowel/big-word-1  **/big-word-4
big-word-1.. $y$/big-word-2   **/big-word-3
big-word-2.. consonant-not-qy*  $qu$/*+1  install
       marks  letter*  install
       compute(column-count=column-count+j -startword+2)
       porker = $($p1(p3,p2)$)
big-word-3.. consonant/(marks  compute(column-count=column-count -1))
big-word-4..  letter*  install
          compute(column-count=column-count+j -startword+3)
          porker = $($p1(y,p2)$)
porker.. = porcify
*
```

```
*       Here we go.  You should know that "porcify" is a definition which
*   takes a split word (say, "squid," split into "squ" and "id") and
*   produces Pig Latin.  So the point of "bigword" is mainly just to split
*   words into pieces for "porcify."
*       We start by noting the input scan pointer, J.  We will use J
*   to compute the quantity of output which we have produced; in the
*   case of words starting with consonants, for example, the output
*   produced is as long as the input (ending jminus starting j)
*   plus two for the added "ay."
*       Now we use "mark" and we start collecting letters.
*   Suppose the first letter is a vowel.  In this case, the detour of
*   the "vowel" component is irrelevant, and we proceed to "big-word-4."
*   The "letter*" component collects the rest of the word and
*   the "compute" ups the column-count to show the word and the
*   coming "yay".  Now we go to "porcify with an imaginary word,
*   split into a leading "y" and a trailing string which is the
*   word we really found.
*       Suppose the first letter of the word is not a vowel ("big-word-1"),
*   but rather it is "y".  Then we skip the detour of the "y" component, and
*   we continue with big-word-3.  If a consonant follows the "y", then we
*   drop immediately to the next line, which is exactly where we were a
*   minute ago when the word began with a vowel.  If a non-consonant follows
*   the "y," however, we slyly reinitialize the letter-collector with a new
*   "marks" component.  Only then do we drift down to the next line, where
*   the code which thinks it is converting "am" into "amyay" is actually
*   converting "yam."
*       Finally, suppose the word begins with some letter not one of
*   [a,e,i,o,u,y].  We collect its initial string, collect the remainder,
*   and porcify the word split into this pair.
*       The punctuation-handler is rather an anticlimax.
*
punctuation..  blanks  compute(startword=j)
    marks  puncts*  install/(
         // paragraph/(=$( $))  compute(column-count=5)  =$(///$))
    compute(column-count=column-count+j -startword)=p1
*
*       Finally, the definitions
*
.definitions.
porcify=(2)$($1$2$q1ay$)
p1=$($p1$)
p2=$($p2$)
p3=$($p3$)
y=$(y$)
ay=$(ay$)
end
```

```
syntax for word-counter
.syntax.
*       This compiler accepts an English text.  From this, it
*  produces a table showing which words occurred in the text, and
*  how frequently each occurred.
*
*            here are the same old declarations
*
marks=*marks
letter=charcl($abcdefghijklmnopqrstuvwxyz-$)
garbage=charcl($$%.,"()<>:;?[ ]^_+/|=&01234567890$)
.tabs. 1
cardof=*cardof
refoff=*refoff
count=progrm
install=*instal
new-word=*noflgs
setflag=**setflg
equadr=*equadr
desc=**desc
parsdo=**parsdo
checkflag=**chkflg
getnam=*getnam
*
*       We will be doing a fair amount of computing in this compiler.
*  "Total-words" will grow by one for every word in the input text;
*  "distinct-words" will only grow when new words are found.  Each
*  distinct word will be labeled with a pointer to the previous such
*  word ("chain") and an occurrence-count ("occurrences").
*
total-words=0
distinct-words=0
chain=1
occurrences=2
*
*        scratch cells
*
n=0
previous-word=0
*
*       The "chain" pointers will be used when we run back through all the
*  words, printing them and their occurrence-counts.  A flag ("first-word")
*  will be set on the first word in, so that we will know where to stop
*  our run.  Also, each word will be given an "exists" flag so that the
*  first appearance of the word may be easily distinguished.
*
.flags. first-word,exists
*
*       Now we just pull in words until one of them is "***".  When that
*  happens, we print a summary.
```

```
*
count.. cardof refoff
count-1.. $***$/count-2  header  output  **
count-2.. count-3  word/count-1  count-3  **/count-1
count-3.. garbage*  = $($)
*
*       When a new word comes in, we (1) set a pointer to the previous word,
*  and (2) start an occurrence count.  Old words we just count.
*
word.. marks  letter*  install
       (total-words=total-words+1)
       new-word/((desc(occurrences)=desc(occurrences)+1)  =$($))
       setflag(exists)
       if(total-words .g. 1)/(setflag(first-word))
       (distinct-words=distinct-words+1)
       (desc(chain)=previous-word, previous-word=equadr)
       (desc(occurrences)=1)
    =$($)
*
*       So much for the words.  When they are all read in, we start producing
*  our summary:
*
header.. =$($f3(total-words) words altogether, $)/h1
h1.. =$($p1$f3(distinct-words) distinct words$.//$)
*
*       "Output" does not accumulate a giant, multi-line output structure.
*  Instead, "output" produces its lines as part of the matching process.
*
output..  (equadr=previous-word)  if(total-words .e. 0)/out1  =$($)
out1..  parsdo(output-line  **)
       checkflag(first-word)/((equadr=desc(chain))  **/out1)  =$($)
output-line..  getnam (n=desc(occurrences))  =$($p1/$f3(n)//$)
*
.definitions.
end
```

J. TMGL Dirties

1. Detour-less elements

Suppose a detour-less element fails, while matching a sentence component C. One might expect this event to entail the immediate failure of C, with the scan backed up to where it was when C was encountered. This will happen in some cases.

In fact, the scan is backed up along the trail back to C only until C or some element having a detour is reached. In the latter case, the trail continues out that detour.

For example, the sentence

```
C.. ALEX/(=$($)) **
```

will always succeed, even though it should, according to subsection H, always fail. To get around this sort of thing, try

```
C.. ALEX/(=$($)) **/(**)
```

This anomaly is not observed along trails being followed by "PARSDO".

2. Relative detours

The effect of relative detours is highly implementation-dependent. As the detour for any component except a <fc with>, "*+1" is reliably "continue to the the right". To get this effect with a <fc with> component, use "*+2". Don't use other relative detours.

3. P1, Savtre

These functions are supposed to treat the most recent output-table entry. They will do this successfully only if that entry was made by the last component matched along the trail. Intervening components, even though they made no entries in any output table, will queer it.

4. Parsdo, blanks, noblks

The blanks and noblks functions have no effect when encountered along a parsdo-argument trail.

5. Compute, if

The word "compute" may be omitted from components using the compute function. Also, compute and if must never be declared.

6. Line length

An input line to TMGL should not be longer than 80 characters. If it is, the Diagnostic "preceding card unrecognizable" may occur with reference to a blank line which does not, of course, exist.

7. Functions and character classes undeclared

If a TMGL program makes use of a function without declaring it, the "card not blank after component list" Diagnostic may appear.

8. Parsdo

Along trails followed by parsdo (for example, within the super sentence), no function which requires an argument will be properly interpreted. Starred character-class components will also be mistreated.

K. REFERENCES

1. McClure, R. M. "TMG - A Syntax Directed Compiler," in Proceedings of the 20th National Conference. New York: Association for Computing Machinery, 1965. Pp. 262 - 274.

2. Fasciano, V. A. "TMG - A Syntax Directed Compiler." BTL Paper Pd - 550029.