META-3

A Syntax-Directed Compiler Writing Compiler to Generate Efficient Code

by Frederick W. Schneider and Glen D. Johnson, UCLA Computing Facility, Los Angeles

ABSTRACT

The basic compilation method is a top to bottom recursive scan without backtrack based on the compiler written for the IBM 1401 by Val Schorre. Each statement of the language is written in a form closely resembling Backus Normal Form; that is, a sequence of tests to be performed to determine whether or not the sequence of characters in the input string is a valid program in the language described. In addition, output instructions are interspersed with the syntactic elements to generate the desired code. The following features were added to the language to facilitate the direct generation of efficient machine code:

1. *A symbol table*
2. *A push-down operand stack*
3. *Mode flags and a register manipulation generator*
4. *A push-down first-in first-out list*
5. *Direct communication in a simplified manner between the compiler and hand coded routines.*

A complete description of both the META-3 compiler and of the compilation algorithm are given.

META-3

Contrary to popular opinion, syntax-directed compilers can rapidly generate quite efficient machine code for machines without push-down hardware. The method used in our compiler is based on the META II compiler developed by D. V. Schorre for the IBM 1401, but it is modified to facilitate direct generation of sequential code rather than polish-like code.

The META-3 compiler constructs a series of tests and references to external routines from an input language resembling Backus Normal Form, with code defining clauses added. This construction assembles into a compiler for the language defined.

Two types of operations are basic in the meta-language: actions and tests. An action is an unconditional operation such as outputting, setting flags and so forth. There are two major types of test. One is to test internal status such as the type of a variable, the other is to test the input stream for the occurance of an identifier, a specific character string, or a general form of string. Each test returns the value *true* or *false* depending upon whether the tested condition was met or not. The Meta-compiler generates the code to test this value after every test and either proceeds, if true, or, if false, trys to return the value *false* to the caller. Since anything tested for and found is deleted from the input stream, any *false* return other than the first of a sequence of tests will be made to transfer control to a diagnostic routine which prints the top element of the stack, the present status of the input stream, and a complaint about bad syntax. The discussion of the syntax equations for META-3 as written in META-3 will show the usage and definition of the basic syntax elements. For a further discussion of the basic algorithm or references on the subject see Schorre's paper in this volume.

Each syntax equation begins by naming the construct which it isdefining, and ends with a semicolon (written '.'), The definition is a series of tests and actions, which may be grouped by parenthesization. A string in quotes (e.g. 'STRING') is a test which is *true* only if the specified characters appear next in the input stream. '.ID' is a test which is *true* only if an identifier is the next thing in the input stream. An identifier is an alphabetic character followed by a series of alphabetic or numeric characters, and terminated by the first unrecognizable character, usually a blank. The first six characters of the identifier must be unique and are the only portion of the identifier retained. '.ID' causes this identifier to be placed in the push-down operand stack. Alternate definitions are identified by separating them with slashes (/). For simplicity of writing the sequence operator '$' is used to reduce the number of recursive definitions needed, and is read: ' a sequence *(which may be empty)* of...'. The test following the sequence operator is performed until it returns *false*, at which point the sequence is satisfied. 'EMPTY' is a test which always has the value *true*. An identifier indicates a syntactical structure, usually defined by another equation, which is to be tested for. '.STRING' is a test which removes a string from the input stream, assigns it storage, and places its symbolic address (of the form .Z.nnn) in the operand stack.

Outputting is indicated by the '.OUT' or '.CALL' verbs. '.OUT' is followed by a list, in parentheses, of output arguments to be placed in the fields of a symbolic card to be turned over to the assembler. There are three fields; label, operation, and variable. Fields are separated by commas, and cards are terminated by slashes. There are three forms which each argument may take:

a) strings to be inserted litterally
b) '*' indicating the uppermost element of the stack
c) '*n' indicating the n^{th} label stack, each of which has a unique constant value at each usage of each statement.

'.CALL(...)' is eqivalent to '.OUT(,'CALL',...)' and generates the op-code CALL with the first argument going in the variable field.

Since the compiler is fully defined by its syntax equations, the following discussion of each equation will complete the description of the META-3 compiler.

.SYNTAX PROGRAM
: Defines the principal syntactic element of this compiler.

PROGRAM =
: Begins the definition of the syntactic element 'PROGRAM'

'.SYNTAX'
: Tests the input stream for the quoted string. If *false* (since this is the first test of this definition) 'PROGRAM' will be *false*.

.ID
: Tests for an identifier in the input stream. If found the first six characters are placed in the operand stack, and the entire identifier is deleted from the input stream. If not found the diagnostic routine is entered.

.OUT(,'ENTRY',*)
: Outputs a symbolic card with the op-code ENTRY and the variable field containing the identifier in the top of the stack. The stack is popped up.

$ ST
: Tests for the syntactic element 'ST' (defined below, and keeps going back for more until they are exausted.

'.END'
: Removes the string '.END' from the input stream, giving a diagnostic if not found.

'.'
: End of statement.

The entire statement discussed so far is:

```
PROGRAM ='.SYNTAX'  .ID .OUT(,'ENTRY',*)
                    $ ST '.END' .,
```

It may be expressed in Backus Normal Form as:

```
<program> ::= .SYNTAX <identifier> <stseq> .END
<stseq> ::= <st> | <st> <stseq> | <empty>
```

The equations for META-3 continue:

```
ST = .ID .OUT( *,'PXA',',4'/, 'CALL','..PUSH')
```

This is the beginning of the defination of a statement and says that a statement starts with an identifier which is output as the label of a PXA ,4 instruction, then followed by a call of ..PUSH.

```
  '=' EX1 '.,' .CALL ('..POPP') .,
```

The identifier must be followed by an equal symbol and an EX1 (see below) and terminated by a At this point a call of routine ..POPP is output. The routines ..PUSH and ..POPP handle the recursion.

```
EX1 = EX2 $( '/' .OUT(,'ZET','..TEST' /,'TRA', *1) EX2)
  .OUT (*1,'NULL') .,
```

An expression one is defined to be an expression two followed by a sequence (*which may be empty*) of slash, at which point output a test for the truth of the previous expression, which, if met, will cause transfer of control to the label contained in '*1' which will be defined later. After the slash must come another expression two. When there no more alternatives in the stream, define the label in '*1' by outputting it on a NULL.

```
EX2 = ( TEST .OUT(, 'NZT' , '..TEST' /, 'TRA',*1)/ACTION
      ) $( TEST .OUT(,'NZT','..TEST'/,'CALL','..DIAG')/
    ACTION) .OUT( *1 ,'NULL') .,
```

An expression two consisists of a number of tests or actions. If the first of these is not met the restof them are skipped. If any of the others is noy met ..DIAG receives control.

```
TEST = .ID .CALL( *)
```

A test is defined to be either an identifier, in which case a call to the identifier is output,

```
    / '.ID' .CALL('..IDNT')
```

or the string '.ID' in which case a call on routine '..IDNT' is generated

```
    / '(' EX1 ')'
```

or, a left parenthesis followed by an EX1 followed by a right parenthesis

```
    / .STRING .CALL('..CMPR(' * ')' )
```

or, a string in quotes whose location is inserted into the stack and then output as an argument to ..CMPR

```
    / 'S.TRING' .CALL ('..STRT')
```

or, the word .STRING which causes a call on ..STRT to be generated

```
    / '.CLA'( '-' DIGIT ALPHABETIC *1 .CALL('..CLAD(=' 
      * ',=H' *1 '*****)' )
```

or, the word .CLA followed by a minus sign and a digit and a letter both of which are placed in the stack as they are found by external routines with the entry points 'DIGIT' and 'ALPHAB' These two characters are output as arguments of a call on routine ..CLAD by placing the letter in the *1 label stack and referencing it in the .CALL statement

```
      / DIGIT ALPHABETIC *1 .CALL('..CLAD(=' * ',=H'
        *1 '*****)' ) )
```

or the .CLA could be followed by just the digit and letter without the minus sign and receive a approximately the same treatment, except that the digit is transmitted to ..CLAD as positive rather than negative.

```
  / '-' ALPHABETIC .CALL('..MINS(=H' * '*****)' )
```

Or, a test may be a minus sign followed by a letter. In this case a call on ..MINS with the letter as an argument is generated. This is used with the symbol table discussed below.

```
  / '*' ( DIGIT .CALL ('..MOVE(=' * ')' )
```

Or, an asterisk followed by a digit which is compiled as an argument to routine ..MOVE. This is the routine which moves identifiers between the operand stack and the label stacks

```
  / '-' DIGIT .CALL( '..MOVE(=-' * ')' )
```

Or, the asterisk could be followed by a minus sign and a digit which is given as an argument to move the '*n' stack to the operand stack.

```
  / ALPHABETIC .CALL ('..STAR(=H' * '*****)' ) )
```

Or, the asterisk could have been followed by a letter which is compiled as an argument to ..STAR. Again, this is for the symbol table q. v.

```
  / '.T' ALPHABETIC .CALL('..SETT(=H' * '*****)' )
```

Or, the final thing which a test may be is .T followed by some letter which is used as an argument in the generated call on .SETT. This test references the mode flag.

```
ACTION = OUTPUT
```

An action is defined to be either an output (defined later),

```
  /'.EMPTY .OUT( 'STL' , '..TEST')
```

or, .EMPTY in which case the test cell ..TEST will be set non-z ero to indicate that indeed an empty has been found.

```
  / '$' .OUT(*2, 'NULL') TEST
```

Or, a dollar sign, at which point the label in *1 is output,followed by a test (defined above)

```
    .OUT(,'ZET','..TEST'/,'TRA',*1/,'STL','..TEST')
```

after which test, if it was met it will be repeated, otherwise, ..TEST is set non-zero to indicate *true.*

```
  /'.STO' ALPHABETIC .CALL('..STOR(=H' * '*****)' )
```

Or, .STO followed by a letter which is compiled as an argument to ..STOR.

```
  /'+' ALPHABETIC .CALL('..PLUS(=H' * '*****)' )
```

Or, a plus sign followed by a letter which compiles as a call on ..PLUS with the letter as an argument (set attribute register to indicate this property).

```
  /'.S' ALPHABETIC .CALL('..SETS(=H' * '*****)' ) .,
```

Or, finally, an action may be .S followed by a letter which becomes, at object time, an argument to ..SETS (which sets the mode flag for later testing with .T).

```
OUTPUT=
```

An output is defined to be

```
  '.OUT' '(' OUT2 ')'
```

.OUT followed by an OUT2 in parentheses

```
 /'.CALL' .CALL('..FELD').CALL('..LITG(=0232143437700)')
       .CALL('..FELD')
```

Or, .CALL which generates thesame instructions as .OUT(,'CALL', ...)

```
      '(' $OUT1 ')' .CALL('..PUBG')
```

followed by a parenthesis and a sequence *(which may be empty)* of OUT1's after which a call on ..PUBG is generated

```
  /'.ERITE' '(' DIGIT $OUT1 ')' .CALL('.ERITE(='*')').,
```

Or, finally an output may be .ERITE followed by, in parenthses, a digit and a sequence (*which may be empty*) of OUT1's, in which case the digit is given to .ERITE as an argument.

```
OUT2 = OUT2A $ ('/' .CALL('..PUBG') OUT2A) .CALL
       ('..PUBG') .,
```

An OUT2 is defined to be an OUT2A followed by a sequence (*which may be empty*) of slash (at which point a call to ..PUBG is output) followed by OUT2A's. at the end ..PUBG is called.

```
OUT2A = $ OUT1 $(',' .CALL('..FELD') $OUT1) .,
```

An OUT2A is defined to be a sequence (*which may be empty*) of OUT1's followed by a sequence(*which may be empty*) of comma (output a call on ..FELD) followed by sequence (*which may be empty*) of OUT1's.

```
OUT1 = '*' (DIGIT .CALL('..GENR(='*')' )
```
An OUT1 is defined to be either an asterisk followed by a digit, in which case ..GENR is called with the digit as an argument,

```
     / .EMPTY .CALL('..COPG' ))
```
or else, for an asterisk alone, a call on .. ..COPG is output.

```
   / .STRING .CALL('..LITG(' * ')' ) .,
```
Or, finally, an OUT1 may be a string whose location is compiled into a call on ..LITG.

.END Signals the end of compilation.

Direct Communication Between Hand Coded Routines and the Meta-compiler

While compiling the meta-language description of a compiler, any identifier is assumed to be the name of a meta-linguistic variable, and, as such, has a call to it generated. Upon return the cell ..TEST is tested for the *true* or *false* result of the test performed. The IBMAP assembler assumes that any undefined symbol will be defined as an entry point to some other deck at load time.

This rather rash assumption on the assemblers part allows operations to be added at will with the understanding that if the added routine is actually only an action the compiler still treats it as a test, and tests cell ..TEST on return from the routine, and had better find it non-zero (or *true*) at that point if stray error messages are to be avoided and compilation is to continue.

The Push-down Operand Stack

The meta-linguistic element * is to be treated as a push-down stack. Whenever an identifier is successfully discovered it is placed on the top of this stack. It may be removed (and the stack popped up) either by having* in an output imperative, by the FIFO, or by entering subroutine REMOVE which may be called either from a syntax equation or from a hand coded routine.

This stack is extended by allowing copies to be freely made from the * stack to any one of the four local safe cells (*1, *2, *3, *4) and also allowing back-copying (*-1, *-2, *-3, *-4).

All other operands such as strings, digits, etc. are entered as the topmost element of the stack as they are discovered in the input stream.

The FIFO

An interesting technique implemented in META-3 is the combined push-down and first-in first-out list. The operations FEE, FI, FO, and FUM are used to address it, the elements being inserted by FI and removed by FO. FEE is used to push the list down and insert a level mark, while FUM generates a call statement on the variable in the top of the operand stack, with all the elements in the top of the FIFO as arguments.

The basic structure of the list is that of a number of superimposed FIFO lists(or queues).

FI removes the uppermost element of the operand stack and inserts it into the FIFO list as the last element.

FO removes the first element of the FIFO list and inserts it as the uppermost element of the operand stack, however if the present queue or FIFO area is empty FO will return*false* and pop the stack to the underlying FIFO.

FEE starts a new list, marking the top of the previous one. That is it pushes down the previos queue, and starts a new one on top of it.

FUM generates a call to the uppermost element of the operand stack and gives as arguments all the elements of the uppermost FIFO list(if any).

The following example of the use of this list will give an idea of its use. The first column represents the contents of the operand stack, the second the operation in the compiler, and the third the contents of the FIFO.

Operand stack	Operation	FIFO
A B Γ Δ E	FI	empty initially
A B Γ Δ	FI	E
A B Γ	FI	ΔE
A B	FEE	ΓΔE
A B	FI	ΓΔE|
A	FI	ΓΔE|B
empty	FO	ΓΔE|AB
B	FO	ΓΔE|A
B A	FO	ΓΔE|
B A	*(returns false)* FUM	ΓΔE
B	*(outputs* CALL A(Γ,Δ,E))	empty

Intricate rearrangements and rescanning are possible using this list since anything not wanted now can be Fled and when needed restored in the identical order using a FO.

As is evident in the discussion the principle use of this list in the present version is processing procedure declarations and references, since, for compatibility with the rest of the world it is desirable to have the arguments appear in the object code in the same order as in the source program. Variables in the declaration can be cycled through the FIFO in order to pass a number of tests by doing alternate FOs and FIs and rolling up without changing the length of the area (this action reminds me of a tracked vehicule), while determining the parameters necessary for storage allocation.

The Symbol Table

This routine stores and examines symbols given to it. Each symbol may have any of 26 arbitrary attributes, represented as A through Z. These properties are given to the routine by or'ing a property into the attribute register. For example +R adds the property R to those properties already in the attribute register. The meta-language primitive CLEAR sets all the properties to *false*. The meta-language primitive SET nondestructivly places the element at the top of the * stack into the symbol table along with the contents of the attribute register. The OR verb gives the * stack identifier the properties represented by the contents of the attribute register in addition to its other properties.

The symbol table may be tested with the - property. This returns *true* if and only if the input string is an identifier with the given property. For example -P would test the input string for the property P. mnemonically this could test it for being a procedure name in a statement such as:

```
X =RANDOM
```

where RANDOM is a previously declared procedure. The * stack may be similarly tested by saying for example: *P.

The symbol table is extended to cover block structured languages by marking it and skipping back to the last mark. The marking is done by the verb BEGIN; the popping by END. These may be nested until the symbol table overflows. In addition, Though there is no immediate use, for determining whether or not a variable is local to a block the verb LOCAL returns *true* if the last identifier tested was found in the symbol table before a mark was found.

The Register Manipulation Generator

The register handling routine generates register load, store, and exchange instructions and keeps track of the object time registers. The machine for which we are compiling is assumed to have six registers; an A register for addition, a Q register useful for division, an I register and a L register, both used for logical operations, an R register used for double precision work, and the N register which is a negative A register. It is assumed that that two registers cannot both contain information at the same time.

The safeguarding of the contents is caused by the imperative .STOx where x is a register name. This causes insertion of a dummy register in the * stack, and the maintenance of a pointer to this register in the * stack.

The loading of a register from * is accomplished by .CLAnx where n is the depth of the *stack that the storage reference is to be taken from and x is a register name. The previous register contents, if any, are preserved by the generation of a store instruction. Register exchanges are performed if necessary. The loading imperative is extended by allowing n to be preceeded by -. In this case the register exchange is performed only if it is a pure exchange; that is, the requested operand is already in the registers.

EXAMPLE:

```
.SYNTAX EXPR

EXPR = EXPR0 FREEAC .,

EXPR0 = EXPR1 $ ('+' EXPR1 (.CLA-1A/.CLA2A)
                .OUT('ADD',*) .STOA) .,

EXPR1 = PRIMARY $ ('*' PRIMARY(.CLA-1Q/.CLA2Q)
                .OUT('MPY',*) .STOQ) .,

PRIMARY = '(' EXPR0 ')' / .ID .,    .END
```

Gives for either (A+B) *C or C*(A+B) the following code

```
CLA A
ADD B
XCA
MPY C
STQ .T+000
```

And for the expression (A+(I+J)) gives:

```
CLA I
ADD J
ADD A
STO .T+000
```

The verb FREEAC causes the contents of the registers to be unconditionally emptied.

The verb TPUSH marks a stack used to retain the number of temporaries used in any block. At the end of the block the verb TPOP will generate the instruction:

```
.T   BSS  n
```

where n is the number of temporaries used.

A more complex example of the use of many of the features of META-3 is the listing of the syntax equations for CODOL in the appendix. CODOL is a minimal compiler, designed more to have an assembly listing of less than ten pages than to be useful for computation, and has the severe drawback that in our haste to prepare it provision for constants was completely overlooked, but could be inserted by allowing REALNUMBER as a PRIMARY. This routine is a hand coded one designed for the ALGCL 60 compiler now under development using the successor to META-3, META-4.

Acknowledgments

We thank the UCLA Computing Facility for the generous use of their IBM 7094, and E. M. Manderfield, without whose flogging this paper would never have been written.

References

1. Schorre, Val, 'META II A Syntax-oriented Compiler Writing Language', 1964 ACM Natl. Conf.

2. Schorre, Val, 'A Syntax-directed Smalgol for the 1401' 1963 ACM Natl. Conf.,Denver, Colo.

3. Irons, E. T., 'The Structure and Use of the Syntax-directed Compiler', Annual Revue in Automatic Programming, The Macmillan Co. New York.

4. Schmidt, L., 'Implementation of a Symbol Manipulator for Heuristic Translation', 1963 ACM Natl. Conf.

5. Bastian, Lewis, 'A Phrase-Structure Language Translator', AFCRL-Rept-62-549, Aug. 1962.

Appendix A

Instructions generated by register exchanges

Source	Destination A	N	Q	R	L	I	Storage
A	—	CHS	XCA	LDQ =0	XCA XCL	XCA XCL PAI	STO
N	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
Q	XCA	XCA CHS	—	XCA LDQ =0	XCL	XCL PAI	STQ
R	—	CHS	XCA	—	XCA XCL	XCA XCL PAI	DST
L	XCL XCA	XCL XCA CHS	XCL	XCL XCA LDQ =0	—	PAI	SLW
I	PIA XCL XCA	PIA XCL XCA CHS	PIA XCL	PIA XCL XCA CHS LDQ =0	PIA	—	STI
Storage	CLA	CLS	LDQ	DLD	CAL	LDI	—

Note: The transfers between I or L and any of A,Q,R or N are for completeness only, there no convenient instructions for this.
Note: The imperative .STON is illegaal, there being no convenient instruction for this.

Apendix B

Subroutines used and their Functions

Name	Usage	Statement	Function
..POPP	=	ST	Saves location of caller for recursion
..PUSH	,	ST	Recursive return
..TEST	any test	EX1, EX2	Non-zero if *true*, zero if *false*.
..DIAG	any test	TEST	Prints stack, input stream and nasty message about bad syntax.
..IDNT	.ID	TEST	Test for an identifier in the input stream, places first six characters in the stack if found.
..COMP	'XYZ.....	TEST	Tests for a string in the input stream returns *true* on match.
..STRT	.STRING	TEST	Tests for any string in the input stream, outputs it as: USE .STRN. .Z.nnn BCI m, the string USE PREVIOUS and places .Z.nnn in the stack.
..CLAD	.CLAnx or .CLA-nx	TEST	Entry point to register manipulator for register loads and exchanges.
..MINS	-x	TEST	Used to test input string and compare it with the symbol table.
..MOVE	*x or *-x	TEST	Moves single elements from the stack to the label stacks and vica-versa.
..STAR	*Z	TEST	Used to compare the stack with the symbol table.
..SETT	.Tx	TEST	Used to test the mode flag.
..STOR	.STOx	ACTION	Tells the register manipulator to hang onto the contents of x.
..PLUS	+x	ACTION	Sets the symbol table.
..SETS	.Sx	ACTION	Sets the mode flag to X.
..FELD	,	OUTPUT	Begins a new field on output.
..LITG	'.....'	OUTPUT	Moves a fixed string into the output stream.
..PUBG	.OUT or .CALL	OUTPUT	Ends a card image.
.RITE	.ERITEn	OUTPUT	Writes an error message.
..GENR	*n	OUT1	Moves the label from the *n label stack to the output stream.
..COPG	*	OUT1	Moves the stack to the output stream and pops it up.
DIGIT	n	TEST	Moves one character of the specified type into the stack. DIGIT and ALPHAB may return *false*, CHARAC never.
ALPHAB	x	TEST	
CHARAC	x or n	ACTION	
CLEAR		ACTION	Resets attribute register.
TPUSH		ACTION	Marks the beginning of a block of temporaries.
TPOP		ACTION	Ends a block of temporaries and allocates storage to them.
USE		ACTION	Begins block of separate code.
USEPOP		ACTION	Ends block of searate code and returns to previous block.
OR		ACTION	Defines a symbol in the stack with
SET		ACTION	the properties in the attribute register, and puts it in the symbol table.
REMOVE		ACTION	Deletes the top of the stack.
FEE		ACTION	
FI		ACTION	
FO		TEST	Reference the FIFOlist (see text).
FUM		ACTION	
REALNU		TEST	Trys to get a double-precision floating-point number from the input stream.

Appendix C

7094 META-COMPILER COMPILED BY ITSELF.

```
.SYNTAX  PROGRAM
OUT1 =
     '*'  ( DIGIT  .CALL(  '..GENR(=' *  ')'  )
          /  .EMPTY  .CALL(  '..COPG'  )  )
/    .STRING  .CALL(  '..LITG(' *  ')'  )
.,
OUT2A =
     $  OUT1   $  (  ','  .CALL(  '..FELD'  )  $ OUT1   )
.,
OUT2 =
     OUT2A  $ (  '/'  .CALL(  '..PUBG'  )  OUT2A )  .CALL( '..PUBG'  )
.,
OUTPUT =
     '.OUT'  '('  OUT2  ')'
/    '.CALL'  .CALL(  '..FELD'  )   .CALL(  '..LITG(=0232143437700)'  )
               .CALL(  '..FELD'  )
          '('  $ OUT1  ')'   .CALL(  '..PUBG'  )
/    '.ERITE'   '('  DIGIT  $ OUT1  ')'  .CALL(  '.ERITE(=' *  ')'  )
.,
ACTION =
     OUTPUT
/    '.EMPTY'   .OUT( , 'STL' , '..TEST' )
/    '$'  .OUT(  *1 , 'NULL'  )     TEST
                    .OUT( , 'ZET' , '..TEST'  / , 'TRA' , *1  /
                                              , 'STL' , '..TEST'   )
/    '.STO'  ALPHABETIC  .CALL(  '..STOR(=H'  *  '*****)'  )
/    '+'  ALPHABETIC   .CALL(  '..PLUS(=H'  *  '*****)'  )
/    '.S'  ALPHABETIC  .CALL(  '..SETS(=H'  *  '*****)'  )
.,
TEST =
     .ID   .CALL(  *  )
/    '.ID'   .CALL(  '..IDNT'  )
/    '('  EX1  ')'
/    .STRING  .CALL(  '..CMPR(' *  ')'   )
/    '.STRING'  .CALL(  '..STRT'  )
/    '.CLA'    (  '-'  DIGIT  ALPHABETIC  *1
                    .CALL(  '..CLAD(=-'  *  ',=H'  *1  '*****)'  )
               /  DIGIT  ALPHABETIC  *1  .CALL(  '..CLAD(='  *  ',=H'
                    *1  '*****)'  )    )
/    '-'  ALPHABETIC   .CALL(  '..MINS(=H'  *  '*****)'  )
/    '*'  ( DIGIT  .CALL(  '..MOVE(=' *  ')'  )
          / '-' DIGIT  .CALL(  '..MOVE(=-'  *  ')'  )
          / ALPHABETIC   .CALL(  '..STAR(=H'  *  '*****)'  )   )
/    '.T'  ALPHABETIC  .CALL(  '..SETT(=H'  *  '*****)'  )
.,
EX2 =
     (  TEST   .OUT( , 'NZT' , '..TEST'  / , 'TRA' , *1  )  /  ACTION )
$ ( TEST  .OUT( , 'NZT' , '..TEST'  / , 'CALL' , '..DIAG'  )
          / ACTION  )   .OUT( *1 , 'NULL'  )
.,
EX1 =
     EX2 $ (  '/'  .OUT( , 'ZET' , '..TEST'  / , 'TRA' , *1  )
               EX2  )    .OUT( *1 , 'NULL'  )
.,
ST =
     .ID   .OUT( * , 'PXA' , ',4'  / , 'CALL' , '..PUSH'  )
         '='   EX1   '.,'    .CALL(  '..POPP'  )
.,
PROGRAM =
     '.SYNTAX'  .ID   .OUT( , 'ENTRY' , *  )
               $ (  ','  .ID   .OUT( , 'ENTRY' , *  )   )
                 $ ST   '.END'  .,
.END
```

```
          CODOL COMMON DEMONSTRATION ORIENTED LANGUAGE
.SYNTAX PROGRAM

PROGRAM=.OUT('......','SAVE') TPUSH SEGMENT .OUT(,'RETURN','......')
        $(.ID *1 .OUT(*1,'SAVEN') SEGMENT .OUT('RETURN',*1))
                 TPOP .,

SEGMENT= DECLARATION'.,' $(DECLARATIONS '.,')  ST $(',' ST ) .,

DECLARATION = 'REAL' .OUT(,'USE','.STOR.') CLEAR +R .ID SET
                       .OUT( *,'PZE') $(',' .ID SET .OUT(*,'PZE'))
                      .OUT(,'USE','PREVIOUS')
        /'FORMAT' .ID CLEAR +S SET *1 .STRING .OUT(*1,'EQU',*)  .,

ST = '*' .ID  .OUT( * ,'TRA','*+1') ST
   / 'GO' 'TO' .ID  .OUT ( , 'TRA' , * )
   / 'CALL' .ID FEE ( '(' EXPR FREEAC FI $( ',' EXPR FREEAC FI)  ')'
                    / .EMPTY )   FUM
   / 'SET' FEE .ID FI $( ',' .ID FI ) '=' EXPR .CLA1A
                    FO .OUT( ,'STO',*) $(FO .OUT(,'STO',* ) )
   / 'IF' EXPR .CLA1A ( 'PLUS'.OUT( ,'TMI',*1) /'MINUS' .OUT(,'TPL',*1)
                    / 'ZERO'.OUT( ,'TNZ',*1) /'NON''ZERO' .OUT(,'TZE'
                                             ,*1))   ST  .OUT(*1,'NULL')
   / 'ALTER' .ID 'TO' .ID .OUT(,'AXT',*',4'/,'SXA',*',4')
   / 'PRINT' .ID *S .CALL('.FWRD.(.UN06.,'*')')
                  $(',' EXPR .CLA1A .OUT(,'TSX','.FCNV.,4'))
   / 'READ' .ID *S .CALL('.FRDD.(.UN05.,'*')')
                    $( ',' .ID .OUT(,'TSX','.FCNV.,4'/,'STO',*)  )   .,

EXPR = '-' NEXPR / ('+'/.EMPTY) EXPR1  .,

EXPR1 = EXPR2 $( '+' EXPR2(.CLA-1A / .CLA2A) .OUT(,'FAD' , *) .STOA
                /'-' EXPR2(.CLA-1N .OUT(,'FAD',*)/.CLA2A .OUT(,'FSB',*))
                         .STOA) .,

EXPR2 = EXPR3 $( '*' EXPR3(.CLA-1Q/.CLA2Q) .OUT(,'FMP',*) .STOA
               /'//' EXPR3 .CLA2A .OUT(,'FDP',*/,'XCA'/,'FAD','=164B8')
                         .STOA
               /'/' EXPR3 .CLA2A .OUT(,'FDP',*) .STOQ)   .,

EXPR3 = PRIMARY $('**' PRIMARY FREEAC *1 .CALL ('.FXP2.('*','*1')')
                         .STOA  ) .,

PRIMARY = .ID / '(' EXPR ')' .,

NEXPR = EXPR2('+' EXPR1 (.CLA-1A .OUT(,'FSB',*)/.CLA2N .OUT(,'FAD',*))
                .STOA   /'-' NEXPR (.CLA-1A/.CLA2A).OUT(,'FAD' ,*) .STOA
                        / .EMPTY .CLA1N .STOA) .,

    .END
```