Assembler Language as a Higher Level Language: Conditional Assembly and Macro Techniques

SHARE 115

John R. Ehrman
Ehrman@us.ibm.com

IBM Silicon Valley (Santa Teresa) Laboratory
555 Bailey Avenue
San Jose, California 95141

August, 2010

Conditional Assembly and Macro Overview

"In so far as programming languages constrain utterance, they also constrain what a programmer may contemplate productively."
– W. Kahan

Why is the Conditional Assembly Language Necessary?

- Assembler Language has a reputation for difficulty, complexity
  - It's verbose
  - Requires knowledge of underlying architecture

    ...and its instructions
  - Large “Semantic Gap” between conceptual model and its implementation
- Good design:
  - No need to go below your conceptual model to know how it really works
  - Address problems farther from the machine, closer to the problem
- Conditional Assembly and Macros help you...
  - Reduce the “semantic gap”
  - Write programs closer to the application's conceptual model

Why is the Conditional Assembly Language Interesting?

- Adds great power and flexibility to the base (ordinary) language
  - You write little programs that write programs for you!
  - Lets the language and the assembler do more of the work
- Lets you build programs “bottom-up”
  - Repeated code patterns become macros

    ...and, you create reusable code!
  - Enhances program readability, reduces program size, simplifies coding
- You can adapt and change the implementation language to fit the problem
  - Each application encourages design of useful language elements

    Unlike HLLs, where you must make the problem fit the language
  - Addresses problems closer to the application, farther from the machine
  - Lets you raise the level of abstraction in your programs

Part 1: The Conditional Assembly Language

The Two assembler Languages

- The IBM z/Architecture assembler supports two (nearly) independent languages
  - "Ordinary" or "base" assembly language: you program the machine
    - Translated by the assembler into machine language
    - Usually executed on a z/Architecture processor
  - "Conditional" assembly language: you program the assembler
    - Macros are "compiled" at assembly time
    - Conditional assembly statements are **interpreted** and **executed** by the assembler at assembly time
    - Tailors, selects, and creates sequences of statements

Conditional Assembly Language

- Conditional Assembly Language:
  - Analogous to preprocessor support in some languages
    - But the assembler's is much more powerful!
  - General purpose (if a bit primitive): data types and structures; variables; expressions and operators; assignments; conditional and unconditional branches; built-in functions; I/O; subroutines; external functions
- The inner and outer languages manage different classes of symbols:
  - Ordinary (inner) assembly: **ordinary** symbols (internal and external)
  - Conditional (outer) assembly: **variable** and **sequence** symbols
    - Variable symbols: for **evaluation** and **substitution**
    - Sequence symbols: for **selection**
- Fundamental concepts of conditional assembly apply
  - Outside macros (“open code,” the primary input stream)
  - Inside macros (“assembly-time subroutines”)

Evaluation, Substitution, and Selection

- Three key concepts of conditional assembly:

  1. Evaluation

     – Assigns values to **variable symbols**, based on the results of computing complex expressions.

  2. Substitution

     – You write the name of a variable symbol where you want the assembler to substitute the **value** of the variable symbol.

     – Permits tailoring and modification of the “ordinary assembly language” text stream.

  3. Selection

     – Use **sequence symbols** to alter the normal, sequential flow of statement processing.

     – Selects different sets of statements for further processing.

Variable Symbols

- Written as an ordinary symbol prefixed with an ampersand **(&)**
- Examples:

```
&A    &Time    &DATE    &My_Value
```

- Variable symbols starting with **&SYS** are reserved to the assembler
- Three variable symbol **types** are supported:
  - Arithmetic: values represented as signed 32-bit (2's complement) integers
  - Boolean: values are 0, 1
  - Character: strings of 0 to 1024 EBCDIC characters
- Two **scopes** are supported:
  - local: known only within a fixed, bounded context; not shared across scopes (other macros, “open code”)
  - global: shared in all contexts that declare the variable symbol as global
- Most variable symbol values are modifiable (“SET” symbols)

Declaring SET Symbols

- There are six **explicit** declaration statements (3 types × 2 scopes)

	Arithmetic Type	**Boolean Type**	**Character Type**
Local Scope	**LCLA**	**LCLB**	**LCLC**
Global Scope	**GBLA**	**GBLB**	**GBLC**
Initial Values	0	0	null

- Examples of scalar-variable declarations:

```
LCLA  &J,&K          Local arithmetic variables
GBLB  &INIT          Global Boolean variable
LCLC  &Temp_Chars    Local character variable
```

- May be **subscripted**, in a 1-dimensional array (positive subscripts)

```
LCLA  &F(15),&G(1)     No fixed upper limit; (1) suffices
```

- May be **created**, in the form &(**e**) (where **e** is a character expression starting with an alphabetic character)

```
&(B&J&K)   SETA  &(XY&J.Z)-1
```

Declaring SET Symbols ...

- All explicitly declared variable symbols are SETtable
  - Their values can be changed
- Three forms of **implicit** declaration:
  1. by the assembler (for **System Variable Symbols**)
     - names always begin with characters **&SYS**
     - most have local scope
     - you can't change them; HLASM does
  2. by appearing as **symbolic parameters** (dummy arguments) in a macro prototype statement
     - symbolic parameters always have local scope; you can't change them
  3. as local variables, if first appearance is as the name-field symbol of a SETx assignment statement
     - this is the only implicit form whose values may be changed (SET)

Assigning Values to Variable Symbols: SET Statements

- One SET statement for each type of variable symbol:
  SETA, SETB, SETC
- General form is (where x = A, B, or C)

```
&x_varsym   SETx  x_expression   Assigns value of x_expression to &x_varsym
```

- SETA operand uses familiar arithmetic operators and internal (built-in) functions

```
&A_varsym   SETA  arithmetic_expression
```

- SETB operand uses “logical-expression” notation

```
&B_varsym   SETB  Boolean_expression
```

- SETC operand uses strings, specialized forms and internal functions

```
&C_varsym   SETC  character_expression
```

Assigning Values to Variable Symbols: SET Statements ...

- The target variable symbol may be subscripted

```
&A(6)    SETA  9            Sets &A(6) = 9
&A(7)    SETA  2            Sets &A(7) = 2
```

- Values can be assigned to successive elements in one statement

```
&Subscripted_x_VarSym  SETx  x_Expression_List      'x' is A, B, or C

&A(6)    SETA  9,2,5+5      Sets &A(6) = 9, &A(7) = 2, &A(8) = 10
```

  - Leave an existing value unchanged by omitting the expression

```
&A(3)    SETA  6,,3         Sets &A(3) = 6, &A(4) is unchanged, &A(5) = 3
```

- External functions use SETAF, SETCF (more at slide Cond-39)
  - SETAF for arithmetic-valued functions

```
&ARITH  SETAF 'AFUN',2,87          Passes arguments 2, 87 to AFUN
```

  - SETCF for character-valued functions

```
&CHAR   SETCF 'CFUN','AB','CDZ'   Passes arguments 'AB','CDZ' to CFUN
```

Substitution

- In appropriate contexts, a variable symbol is replaced by its **value**
- Example: Suppose the value of &A is 1. Then, substitute &A:

```
 Char&A DC    C'&A'      Before substitution
+Char1  DC    C'1'       After substitution
```

- Note: '+' in listing's "column 0" indicates a generated statement
- This example illustrates why paired ampersands are required if you want a single & in a character constant or self-defining term!
- To avoid ambiguities, mark the end of a variable-symbol substitution with a period:

```
Write:    CONST&A.B DC    C'&A.B'     &A followed by 'B'
Result:  +CONST1B   DC    C'1B'       Value of &A followed by 'B' !!

Not:      CONST&AB  DC    C'&AB'      &A followed by 'B' ??  No: &AB !
       ** ASMA003E Undeclared variable symbol - OPENC/AB
```

  - **OPENC/AB** means "in Open Code, and &AB is an unknown symbol"

Substitution, Evaluation, and Re-Scanning

- Points of substitution identified only by variable symbols
  - HLASM is not a general string- or pattern-matching macro processor
- Statements once scanned for points of substitution are not re-scanned
  (But there's a way around this with the AINSERT statement... more later)

```
&A      SETC    '3+4'
&B      SETA    5*&A    Is the result 5*(3+4) or (5*3)+4 ??
   ** ASMA102E Arithmetic term is not self-defining term; default = 0
(Neither! The characters '3+4' are not a self=defining term!)
```

- Substitutions cannot create points of substitution
  (But there's a way around this with the AINSERT statement... more later)
- Another example (the SETC syntax and the &&s are explained later):

```
&A      SETC    '&&B'       &A has value &&B
&C      SETC    '&A'(2,2)   &C has value &B

&B      SETC    'XXX'       &B has value XXX
Con     DC      C'&C'       Is the result &B or XXX?
    ** ASMA127S Illegal use of Ampersand
```

The operand **'&B'** is not re-scanned; the statement gets a diagnostic

Internal Conditional-Assembly Functions

- Many powerful internal (built-in) functions
- Internal functions are invoked in two formats:
  - logical-expression format takes two forms:

```
     (expression operator expression)      (&A OR &B)   (&J SLL 2)
or
     (operator expression)                 (NOT &C)     (SIGNED '&J')
```

    (logical-expression format also used in SETB and AIF statements)

  - function-invocation format takes a single form:

```
     function(argument[,argument]...)      NOT(&C)   FIND('ABC','&CV')
```

- Eight functions can use either invocation format:
  BYTE, DOUBLE, FIND, INDEX, LOWER, NOT, SIGNED, UPPER
- We'll look at functions using logical-expression format first

Arithmetic-Valued Functions Using Logical-Expression Format

- Arithmetic functions operate on 32-bit integer values
- Masking/logical operations: AND, OR, NOT, XOR

```
((&A1 AND &A2) AND X'FF')
(&A1 OR (&A2 OR &A3))
(&A1 XOR (&A3 XOR 7))
(NOT &A1)+&A2
(7 XOR (7 OR (&A+7)))    Round &A to next multiple of 8
```

- Shifting operations: SLL, SRL, SLA, SRA

```
(&A1 SLL 3)          Shift left 3 bits, unsigned
(&A1 SRL &A2)        Shift right &A2 bits, unsigned
(&A1 SLA 1)          Shift left 1 bit, signed (can overflow!)
(&A1 SRA &A2)        Shift right &A2 bits, signed
```

- Any combination...

```
(3+(NOT &A) SLL &B)/((&C-1 OR 31)*5)
```

Arithmetic-Valued Functions Using Logical-Expression Format

- DCLEN determines length of constant if substituted in DC

```
If &S = && then      DCLEN('&S') = 1      (2 & paired to 1)
If &S = a''b then    DCLEN('&S') = 3      (2 ' paired to 1)
```

- FIND returns offset in 1st argument of 1st matching character from 2nd

```
&First_Match SetA  Index('&BigStrg','&SubStrg')  First string match
&First_Match SetA  Index('&HayStack','&OneLongNeedle')

Find('abcdef','dc') = 3   Find('abcdef','DE') = 0   Find('123456','F4') = 4
```

- INDEX returns offset in 1st argument of 2nd argument

```
&First_Char  SetA Find('&BigStrg','&CharSet')  First char match
&First_Char  SetA Find('&HayStack','&ManySmallNeedles')

Index('abcdef','cd') = 3   Index('abcdef','DE') = 0   Index('abcdef','F4') = 0
Index('ABCDEF','DE') = 4   Index('ABCDEF','Ab') = 0   Index('123456','23') = 2
```

Character-Valued Functions Using Logical-Expression Format

- Character-valued (unary) character operations:
  UPPER, LOWER, DOUBLE, SIGNED, BYTE
- UPPER converts all EBCDIC lower-case letters to upper case

```
(UPPER  '&X')    All letters in &X set to upper case
```

- LOWER converts all EBCDIC upper-case letters to lower case

```
(LOWER  '&Y')    All letters in &Y set to lower case
```

- DOUBLE converts single occurrences of & and ' to pairs

```
(DOUBLE '&Z')    Ampersands/apostrophes in &Z doubled
```

- SIGNED converts arithmetic values to character, with prefixed minus sign if negative

```
(SIGNED  &A)     Converts arithmetic &A to string (signed if negative)
```

- BYTE converts arithmetic values to a single character (just a byte!)

```
(BYTE    X'F9')  Creates a one-byte character value '9'
```

Functions Using Function-Invocation Format

- Representation-conversion functions
  - From arithmetic value to character: A2B, A2C, A2D, A2X
  - From binary-character string: B2A, B2C, ,B2D, B2X
  - From characters (bytes): C2A, C2B, C2D, C2X
  - From decimal-character string: D2A, D2B, D2C, D2X
  - From hexadecimal-character string: X2A, X2B, X2C, X2D
    - All D2* functions accept optional sign; *2D functions always generate a sign
- String validity-testing functions: ISBIN, ISDEC, ISHEX, ISSYM
- String functions: DCLEN, DCVAL, DEQUOTE
- Symbol attribute retrieval functions: SYSATTRA, SYSATTRP

Conditional Assembly Conversion Functions

A-value SetA arithmetic expression

B-string SetC string of binary-digit characters

C-string SetC string of characters treated as 8-bit bytes

D-string SetC string of (optionally) signed decimal-digit characters

X-string SetC string of hexadecimal-digit characters

	Function Value				
Argument	**A-value**	**B-string**	**C-string**	**D-string**	**X-string**
A-value	–	A2B	A2C, BYTE	A2D, SIGNED	A2X
B-string	B2A	–	B2C	B2D	B2X
C-string	C2A	C2B	–	C2D	C2X
D-string	D2A	D2B	D2C	–	D2X
X-string	X2A	X2B	X2C	X2D	–

Converting from Arithmetic to Character Value Types

- A2B converts to a string of 32 binary digits

```
          A2B(1) = '00000000000000000000000000000001'
         A2B(-1) = '11111111111111111111111111111111'
A2B(2147483647) = '01111111111111111111111111111111'
```

- A2C converts to 1 or more bytes; BYTE converts to only 1

```
A2C(-1) = 'ffff'          (X'FFFFFFFF')   (f = byte of all 1-bits)
A2C(-9) = 'fff7'          (X'FFFFFFF7')
A2C(80) = 'nnn&'          (X'00000050')   (n = byte of all 0-bits)
BYTE(60) = '-'            (X'60')         (only a single byte)
```

- A2D converts an arithmetic value to an always-signed decimal string (SIGNED adds a sign only for negative numbers)

```
A2D(0) = '+0'      A2D(1) = '+1'     A2D(-1) = '-1'    A2D(-8) = '-8'
               SIGNED(1) = '1'   SIGNED(-1) = '-1'
```

- A2X converts to a string of 8 hex digits

```
A2X(0) = '00000000'   A2X(1) = '00000001'   A2X(-8) = 'FFFFFFF8'
```

Converting from Character Value Types to Arithmetic Values

- B2A converts from bit-strings to arithmetic:

```
B2A('1001') = 9   B2A('11111111111111111111111111111111') = -1
                  B2A('11111111111111111111111111111000') = -8
```

- C2A converts from bytes to arithmetic:

```
    C2A('-') = 96        (Argument is a single byte, X'60')
    C2A('a') = 129
C2A('nnn2') = 242        (n = byte of all 0-bits)
```

- D2A converts from decimal strings to arithmetic: (+ signs optional)

```
D2A('') = 0   D2A('-001') = -1   D2A('+1') = 1   D2A('1') = 1
```

- X2A converts from hex-strings to arithmetic:

```
X2A('fffffff8') = -8   X2A('63') = 99   X2A('7fffffff') = 2147483647
```

Converting to and from Decimal Characters

- B2D converts binary characters to a signed decimal string

```
B2D('1') = '+1'  B2D('1001') = '+9'
B2D('11111111111111111111111111111111') = '-1'
```

- C2D converts bytes to a signed decimal string

```
C2D('a') = '+129'     C2D('nnn2') = '+242'
```

- X2D converts hex characters to a signed decimal string

```
X2D('') = '+0'            X2D('0') = '+0'
X2D('ffffffff') = '-1'    X2D('63') = '+99'
```

- D2B converts decimal characters to binary characters

```
      D2B('+0999') = '00000000000000000000001111100111'
  D2B('123456789') = '00000111010110111100110100010101'
```

- D2C converts decimal characters to bytes

```
 D2C('-001') = 'ffff'   (=X'FFFFFFFF')
D2C('+0231') = 'nnnX'   (=X'000000E7')
```

- D2X converts decimal characters to hex characters

```
     D2X('-001') = 'FFFFFFFF'          D2X('+0999') = '000003E7'
D2X('123456789') = '075BCD15'    D2X('-2147483647') = 80000001
```

Converting Among Binary, Bytes, and Hex

- B2C converts binary digits to bytes

```
B2C('') = '' (null string)      B2C('10000001') = 'a' (X'81')
B2C('0000') = 'n' (X'00')   B2C('100000001011100') = ' *'    (X'405C')
```

- B2X converts binary digits to hex digits

```
B2X('') = ''   B2X('1000000') = '40'    B2X('001111101') = '07D'
B2X('0000000000001') = '0001'   B2X('1011100010111000101110001011100') = '5C5C5C'
```

- C2B converts bytes to binary digits

```
C2B('a') = '10000001'   C2B('****') = '01011100010111000101110001011100'
 C2B('') = ''           C2B(' *') = '0100000001011100'
```

- C2X converts bytes to hex characters

```
C2X('') = ''   C2X(' ') = '40'   C2X('a') = '81'   C2X('9999') = 'F9F9F9F9'
```

- X2B converts hex digits to binary digits

```
X2B('') = ''   X2B('7D') = '01111101'   X2B('040') = '000001000000'
```

- X2C converts hex digits to bytes

```
X2C('') = ''   X2C('81') = 'a'   X2C('405c') = ' *'   X2C('5c5c5c') = '***'
```

Validity-Testing Functions

- ISBIN tests 1-32 characters for binary digits

```
ISBIN('1') = 1   ISBIN('0') = 1   ISBIN('2') = 0
ISBIN('11111111111111111111111111111111') = 1   (32 characters)
ISBIN('000000000000000000000000000000000') = 0  (33 characters)
```

- ISDEC tests 1-10 characters for assignable decimal value

```
ISDEC('1') = 1           ISDEC('-12') = 0   (sign not allowed)
ISDEC('11111111111') = 0  (11 characters)
ISDEC('2147483648') = 0   (value too large)
```

- ISHEX tests 1-8 characters for valid hexadecimal digits

```
ISHEX('1') = 1   ISHEX('a') = 1   ISHEX('G') = 0   ISHEX('ccCCCCcc') = 1
ISHEX('123456789') = 0  (too many characters)
```

- ISSYM tests alphanumeric characters for a valid symbol

```
ISSYM('1') = 0    ISSYM('$') = 1   ISSYM('1A') = 0   ISSYM('Abc') = 1
ISSYM('_@$#1') = 1   ISSYM('ABCDEFGHIJabcd.....ghiABCD') = 0  (64 characters)
```

Character-Valued String Functions

- DCVAL pairs quotes and ampersands, and returns the string generated as if substituted in DC (compare to DCLEN, slide Cond-27)

  If &S = a''b&&c then DCVAL('&S') = a'b&c (single quote and ampersand)

- DEQUOTE removes a single leading and/or trailing quote

  If &S = '' then DEQUOTE('&S') = null string (both quotes removed)
  If &S = '1F1B' then DEQUOTE('&S') = 1F1B (quotes removed at both ends)

- DOUBLE pairs quotes and ampersands (inverse of DCVAL)

  Let &A have value a'b and &B have value a&b; then

  (DOUBLE '&A') = DOUBLE('&A') = a''b (two quotes)
  (DOUBLE '&B') = DOUBLE('&B') = a&&b (two ampersands)

- LOWER converts all letters to lower case

  (LOWER 'aBcDeF') = LOWER('aBcDeF') = 'abcdef'

- UPPER converts all letters to upper case

  (UPPER 'aBcDeF') = UPPER('aBcDeF') = 'ABCDEF'

Arithmetic-Valued String Functions

- DCLEN determines length of string if substituted in DC

```
If &S = ''  then       DCLEN('&S') = 1      (2 ' paired to 1)
If &S = &&  then       DCLEN('&S') = 1      (2 & paired to 1)
If &S = a''b  then     DCLEN('&S') = 3      (2 ' paired to 1)
If &S = a''b&&c then   DCLEN('&S') = 5      (' and & paired)
If &S = &&&'''  then   DCLEN('&S') = 4      (one pairing each)
```

- FIND returns offset in 1st argument of *any* character from 2nd

```
Find('abcdef','dc') = ('abcdef' Find 'dc') = 3  ('c' matches 3rd character)
Find('abcdef','DE') = ('abcdef' Find 'DE') = 0  ('DE' doesn't match 'd' or 'e')
Find('abcdef','Ab') = ('abcdef' Find 'Ab') = 2  ('b' matches 2nd character)
Find('ABCDEF','Ab') = ('ABCDEF' Find 'Ab') = 1  ('A' matches 1st character)
```

- INDEX returns offset in 1st argument of *entire* 2nd argument

```
Index('abcdef','cd') = ('abcdef' Index 'cd') = 3
Index('abcdef','DE') = ('abcdef' Index 'DE') = 0
Index('abcdef','Ab') = ('abcdef' Index 'Ab') = 0
```

Assembler and Program Symbol Attributes

- EQU and DC/DS statements enhanced to support two new attributes:
  - Assembler attributes have fixed “keyword” values known to HLASM; assignable only on EQU statements

```
Symbol2  EQU  value,length,type,program-attribute,assembler-attribute

R1       EQU  1,,,,GR32       Explicitly a 32-bit register
GR1      EQU  1,,,,GR64       Explicitly a 64-bit register
```

    assembler attribute checking enabled by **TYPECHECK(REGISTER)** option

  - Program attribute assignable on DC/DS and EQU statements

```
Symbol1   DC    <type>P(abs-expr)<modifiers>'nominal-value'
Today     DC    CP(C'Date')L10'Thursday'

Symbol2   EQU   value,length,old-type,P(abs-expr)
ThisYear  EQU   2005,,,P(C'Year')
```

    Both assign the value of **abs-expr** as the symbol's program attribute

- Values of these attributes are retrieved with functions:

```
SYSATTRA('symbol')   returns assembler attribute as characters
SYSATTRP('symbol')   returns 4-byte program attribute
```

Evaluating and Assigning Arithmetic Expressions: SETA

- Syntax:

```
&Arithmetic_Var_Sym  SETA  arithmetic_expression
```

- Same evaluation rules as ordinary-assembly expressions
  - Simpler, because no relocatable terms are allowed
  - Richer, because internal functions are allowed
  - Arithmetic overflows always detected! (except anything ÷ 0 = 0!)
- Valid terms include:
  - arithmetic and Boolean variable symbols
  - self-defining terms (binary, character, decimal, hexadecimal)
  - character variable symbols whose value is a self-defining term
  - predefined absolute ordinary symbols (most of the time)
  - arithmetic-valued attribute references (Count, Definition, Integer, Length, Number, Scale)
  - internal function evaluations
- Example:

```
&A  SETA  &D*(2+&K)/&G+ABSSYM-C'3'+L'&PL3*(&Q SLL 5)+D2A('&D')
```

SETA Statements vs. EQU Statements

- Differences between SETA and EQU statements:

SETA Statements	EQU Statements
Active only at conditional assembly time	Active at ordinary assembly time; predefined absolute values usable at conditional assembly time
May assign values to a given *variable* symbol *many* times	A value is assigned to a given *ordinary* symbol *only once*
Expressions yield a 32-bit binary signed absolute value	Expressions may yield absolute, simply relocatable, or complexly relocatable values
No base-language attributes are assigned to variable symbols	Attributes (length, type, assembler, program) may be assigned with an EQU statement

Evaluating and Assigning Boolean Expressions: SETB

- Syntax:

```
&Boolean_Var_Sym  SETB  (Boolean_expression)
```

- Boolean constants: 0 (false), 1 (true)
- Boolean operators:
  - **NOT** (highest priority), **AND, OR, XOR** (lowest)

```
&A     SetB  (&V gt 0 AND &V le 7)        &A=1 if &V between 1 and 7
&B     SetB  ('&C' lt '0' OR '&C' gt '9') &B=1 if &C is a decimal character
&S     SetB  (&B XOR (&G OR &D))
&T     SetB  (&X ge 5 XOR (&Y*2 lt &X OR &D))
```

  - Unary **NOT** also allowed in **AND NOT, OR NOT, XOR NOT**

```
&Z     SetB  (&A AND NOT &B)
```

- Relational operators (for arithmetic and character comparisons):
  - **EQ, NE, GT, GE, LT, LE**

```
&A  SETB  (&N LE 2)
&B  SETB  (&N LE 2 AND '&CVAR' NE '*')
&C  SETB  ((&A GT 10) AND NOT ('&X' GE 'Z') OR &R)
```

Evaluating and Assigning Boolean Expressions: SETB ...

- Cannot compare arithmetic expressions to character expressions
  - Only character-to-character and arithmetic-to-arithmetic comparisons
    - Comparison type determined by the first comparand
  - But you can often substitute one type in the other and then compare
- **Warning!** Character comparisons in relational expressions use the EBCDIC collating sequence, but:
  - Shorter strings **always** compare LT than longer! (Remember: not like CLC!)
  - `'B' > 'A'`, but `'B' < 'AA'`

```
&B  SETB  ('B' GT 'A')      &B is 1 (True)
&B  SETB  ('B' GT 'AA')     &B is 0 (False)
```

  - Shorter strings are *not* blank-padded to the length of the longer string

Evaluating and Assigning Character Expressions: SETC

- Syntax:

```
&Character_Var_Sym  SETC  character_expression
```

- A character constant is a 'quoted string' 0 to 1024 characters long

```
&CVar1   SETC  'AaBbCcDdEeFf'
&CVar2   SETC  'This is the Beginning of the End'
&Decimal SETC  '0123456789'
&Hex     SETC  '0123456789ABCDEF'
&Empty   SETC  ''      Null (zero-length) string
```

- Strings may be preceded by a parenthesized duplication factor

```
&X    SETC  (3)'ST'         &X has value STSTST
&J    SETA  2
&Y    SETC  (2*&J)'*'       &Y has value ****
```

- Strings are quoted; type-attribute and opcode-attribute references, and internal functions are not

```
&TCVar1  SETC  T'&CVar1
```

  - Type/opcode attribute references: no duplication, quoting, or combining

Evaluating and Assigning Character Expressions: SETC ...

- Apostrophes and ampersands in strings must be paired; but...
  - Apostrophes **are** paired internally for assignments and relationals!

```
&QT   SetC  ''''        Value of &QT is a single apostrophe
&Yes  SetB  ('&QT' eq '''')     &Yes is TRUE
```

  - Ampersands **are not** paired internally for assignments and relationals!

```
&Amp  SetC  '&&'        &Amp has value &&
&Yes  SetB  ('&Amp' eq '&&')  &Yes is TRUE
&D    SetC  (2)'A&&B'   &D has value A&&BA&&B
```

  - Use the BYTE function (slide Cond-18) or substring notation (slide Cond-36) to create a single &

- Warning! SETA variables are substituted **without** sign!

```
&A  SETA  -5
    DC    F'&A'  Generates X'00000005'
&C  SETC  '&A'   &C has value 5 (not -5!)
```

  - The SIGNED and A2D functions avoid this problem

```
&C SETC (SIGNED &A) or &C SETC A2D(&A)   &C has value '-5'
```

Character Expressions: String Concatenation

- Concatenate character variables and strings by juxtaposition
- Concatenation operator is the period (.)

```
&C  SETC  'AB'        &C has value AB
&C  SETC  'A'.'B'     &C has value AB

&D  SETC  '&C'.'E'    &D has value ABE
&E  SETC  '&D&D'      &E has value ABEABE
```

- Remember: a period indicates the end of a variable symbol

```
&F  SETC  '&D.&D'     &F has value ABEABE
&D  SETC  '&C.E'      &D has value ABE
```

- Periods are data if not at the end of a variable symbol

```
&G  SETC  '&D..&D'    &G has value ABE.ABE
&B  SETC  'A.B'       &B has value A.B
```

- Individual terms may have duplication factors

```
&J  SETC  (2)'A'.(3)'B'   &J has value AABBB
```

Character Expressions: Substrings

- Substrings specified by **'string'(start_position,span)**

```
&C  SETC  'ABCDE'(1,3)   &C has value ABC
&C  SETC  'ABCDE'(3,3)   &C has value CDE
```

- **span** may be zero (substring is null)

```
&C  SETC  'ABCDE'(2,0)   &C is a null string
```

- **span** may be * (meaning “to end of string”)

```
&C  SETC  'ABCDE'(2,*)   &C has value BCDE
```

- Substrings take precedence over duplication factors

```
&C  SETC  (2)'abc'(2,2)  &C has value bcbc, not bc
```

- Incorrect substring operations may cause warnings or errors

```
&C  SETC  'ABCDE'(6,1)   &C has null value (with a warning)
&C  SETC  'ABCDE'(2,-1)  &C has null value (with a warning)
&C  SETC  'ABCDE'(0,2)   &C has null value (with an error)
&C  SETC  'ABCDE'(5,3)   &C has value E (with a warning)
```

Character Expressions: String Lengths

- Use a Count Attribute Reference (**K'**) to determine the number of characters in a variable symbol's value

```
&N    SETA  K'&C        Sets &N to number of characters in &C

&C    SETC  '12345'     &C has value 12345
&N    SETA  K'&C        &N has value 5

&C    SETC  ''          null string
&N    SETA  K'&C        &N has value 0

&C    SETC  '''&&'''    &C has value '&&'
&N    SETA  K'&C        &N has value 4

&C    SETC  (3)'AB'     &C has value ABABAB
&N    SETA  K'&C        &N has value 6
```

- Arithmetic and Boolean variables converted to strings first

```
&A    SETA  -999        K'&A has value 3    (Remember: no sign!)
```

Conditional Expressions with Mixed Operand Types

- Expressions sometimes simplified with mixed operand types
  - Some limitations on substituted values and converted results
- Let &A, &B, &C be arithmetic, Boolean, character:

Variable Type	SETA Statement	SETB Statement	SETC Statement
Arithmetic	no conversion	zero `&A` becomes 0; nonzero `&A` becomes 1	`'&A'` is decimal representation of magnitude(`&A`)
Boolean	extend `&B` to 32-bit 0 or 1	no conversion	`'&B'` is `'0'` or `'1'`
Character	`&C` must be a self-defining term	`&C` must be a self-defining term; convert to 0 or 1 as above	no conversion

External Conditional-Assembly Functions

- Interfaces to assembly-time environment and resources
- Two types of external, user-written functions
  1. Arithmetic functions: like `&A = AFunc(&V1, &V2, ...)`

```
&A     SetAF  'AFunc',&V1,&V2,...      Arithmetic arguments
&LogN  SetAF  'Log2',&N                Logb(&N)
```

  2. Character functions: like `&C = CFunc('&S1', '&S2', ...)`

```
&C     SetCF  'CFunc','&S1','&S2',...  String arguments
&RevX  SetCF  'Reverse','&X'           Reverse(&X)
```

- Functions may have zero to many arguments
- Standard linkage conventions

Statement Selection

- Lets the assembler select different sequences of statements for further processing
- Key elements are:
  1. Sequence symbols
     - Used to “mark” positions in the statement stream
     - A conditional assembly “label”
  2. Two statements that reference sequence symbols:

     AGO conditional-assembly “unconditional branch”

     AIF conditional-assembly “conditional branch”
  3. One statement that defines a sequence symbol:

     ANOP conditional-assembly “No-Operation”

Sequence Symbols

- Sequence symbol: an ordinary symbol preceded by a period (**.**)

```
.A      .Repeat_Scan      .Loop_Head      .Error12
```

- Used to **mark** a statement
  - **Defined** by appearing in the name field of a statement

```
.A      LR   R0,R9
.B      ANOP ,
```

  - **Referenced** as the target of AIF, AGO statements
- Not assigned any value (absolute, relocatable, or other)
- Purely local scope; no sharing of sequence symbols across scopes
- Cannot be created or substituted (unlike variable symbols)
  - Cannot even be created by substitution in a macro-generated macro (!)
    (AINSERT provides a way around this)
- Never passed as the value of any symbolic parameter

The ANOP Statement

- ANOP: conditional-assembly “No-Operation”
- Serves **only** to hold a sequence-symbol marker before statements that don't have room for it in the name field

```
.NewVal ANOP ,
&ARV    SETA  &ARV+1    Name field required for receiving variable
```

- **No** other effect
  - Conceptually similar to (but **very** different from!)

```
Target  DC   0H         For branch targets in ordinary assembly
```

The AGO Statement

- AGO **unconditionally** alters normal sequential statement processing
  - Assembler breaks normal sequential statement processing
  - Resumes at statement marked with the specified sequence symbol
  - Two forms: Ordinary AGO and Extended AGO
- Ordinary AGO ("Go-To" statement)

```
AGO  sequence_symbol
```

  - Example:

```
AGO  .Target      Next statement processed is marked by .Target
```

- Example of use:

```
          AGO   .BB
  * (1) This statement is ignored
.BB     ANOP
  * (2) This statement is processed
```

The Extended AGO Statement

- Extended AGO (or “Computed Go-To,” “Switch” statement)

```
AGO   (arith_expr)seqsym_1[,seqsym_k]...
```

- Value of arithmetic expression determines which “branch” is taken from sequence-symbol list
  - Value must lie between 1 and number of sequence symbols in “branch” list
- **Warning!** if the value of the arithmetic expression is invalid, no “branch” is taken!
  - This is a god practice to catch bad values of &SW

```
AGO   (&SW).SW1,.SW2,.SW3,.SW4
MNOTE 12,'Invalid value of &&SW = &SW..'   Message is a good practice!
```

The AIF Statement

- AIF **conditionally** alters normal sequential statement processing
- Two forms: Ordinary AIF and Extended AIF
- Ordinary AIF:

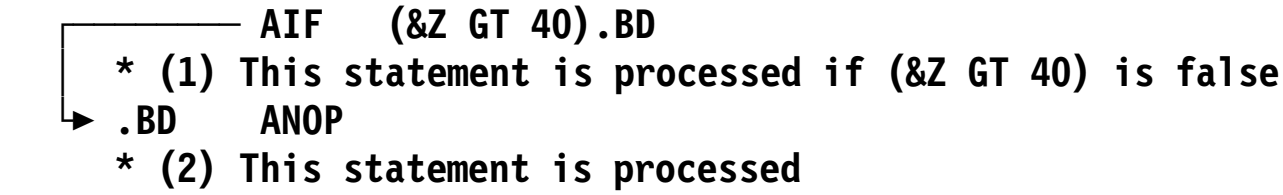
```
AIF   (Boolean_expression)seqsym
```

  - Example:

```
AIF   (&A GT 10).Exit_Loop
```

- If **Boolean_expression** is

  true: continue processing at specified sequence symbol
  false: continue processing with next sequential statement



The Extended AIF Statement

- Extended AIF (Multi-condition branch, Case statement)

```
AIF  (bool_expr_1)seqsym_1[,(bool_expr_n)seqsym_n]...
```

- Equivalent to a sequence of ordinary AIF statements

```
AIF  (bool_expr_1)seqsym_1
. . .
AIF  (bool_expr_n)seqsym_n
```

- Boolean expressions are evaluated in turn until first **true** one is found
  - Remaining Boolean expressions are not evaluated
- Example:

```
         AIF   (&A GT 10).SS1,(&BOOL2).SS2,('&C' EQ '*').SS3

&OpPosn  SetA  Find('+-*/','&String')   Search for operator character
         AGo   (&OpPosn).Plus,.Minus,.Mult,.Div   Branch accordingly
.*       Do something if none is found!
```

Logical Operators in SETA, SETB, and AIF

- "Logical" operators may appear in SETA, SETB, and AIF statements:
  - AND, OR, XOR, NOT
- Interpretation in SETA and SETB is well defined (see slide Cond-38)
  - SETA: treated as 32-bit masking operators
  - SETB: treated as Boolean connectives
- In AIF statements, possibly ambiguous interpretation:

```
AIF   (1 AND 2).Skip
```

  - **Arithmetic** evaluation of **(1 AND 2)** yields 0 (bit-wise AND)
  - **Boolean** evaluation of **(1 AND 2)** yields 1 (both operands TRUE)
- Rule: AIF statements use ***Boolean*** interpretation
  - Consistent with previous language definitions

```
AIF   (1 AND 2).Skip  will go to .Skip!
```

Displaying Symbol Values and Messages: The MNOTE

- Useful for diagnostics, tracing, information, error messages
  - See macro debugging discussion (slide Mac-92)
- Syntax:

```
MNOTE  severity,'message text'
```

- **severity** may be
  - any arithmetic expression with value between 0 and 255
    - value of **severity** is used to determine assembly completion code
  - an asterisk; the message is treated as a comment
  - omitted
    - if the following comma is present, severity = 1
    - if the following comma is also omitted, treat as a comment
- Displayable quotes and ampersands must be paired
- Examples:

```
.Msg_1B MNOTE  8,'Missing Required Operand'                    (severity 8)
.X14    MNOTE   ,'Conditional Assembly has reached .X14'       (severity 1)
.Trace4 MNOTE  *,'Value of &&A = &A., value of &&C = ''&C.''' (no severity)
        MNOTE    'Hello World (How Original!)'                 (no severity)
```

Example: Generate a Byte String with Values 1-N

- Sample 0: write everything by hand

```
N       EQU   5                Predefined absolute symbol
        DC    AL1(1,2,3,4,N)   Define the constants
```

  - Defect: if the value of N changes, must rewrite the DC statement

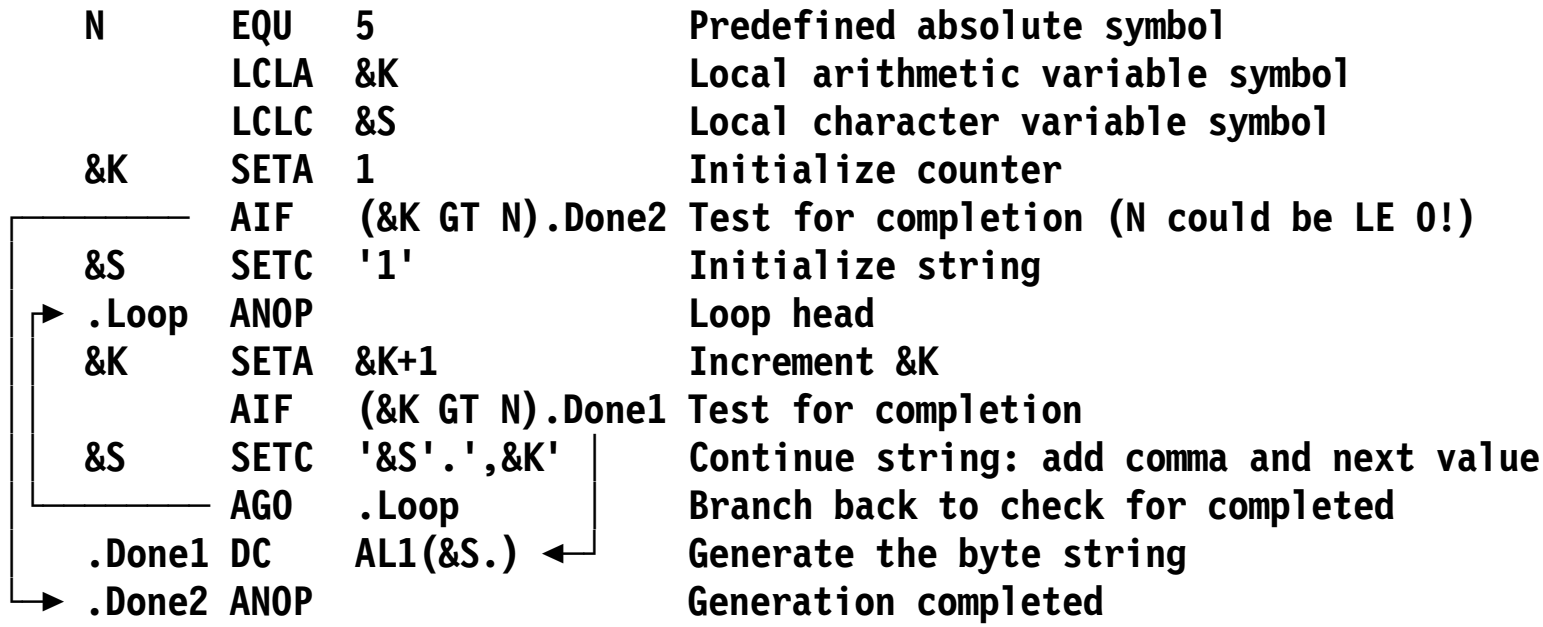
- Sample 1: generate separate statements using arithmetic variables
  - Pseudocode: **DO** for K = 1 to N [**GEN**(DC AL1(K))]

```
   N       EQU   5                 Predefined absolute symbol
           LCLA  &J                Local arithmetic variable symbol, initially 0
.Test      AIF   (&J GE N).Done    Test for completion (N could be LE 0!)
&J         SETA  &J+1              Increment &J
           DC    AL1(&J)           Generate a byte constant
           AGO   .Test             Go to check for completion
.Done      ANOP                    Generation completed
```

- Try it!

Example: Generate a Byte String with Values 1-N ...

- Sample 2: generate a string with the values (like '1,2,3,4,5')
  - Pseudocode:
    Set S='1'; **DO** for K = 2 to N [S = S || ',K'); **GEN**(DC AL1(S)]



```
N       EQU   5                 Predefined absolute symbol
        LCLA  &K                Local arithmetic variable symbol
        LCLC  &S                Local character variable symbol
&K      SETA  1                 Initialize counter
        AIF   (&K GT N).Done2   Test for completion (N could be LE 0!)
&S      SETC  '1'               Initialize string
.Loop   ANOP                    Loop head
&K      SETA  &K+1              Increment &K
        AIF   (&K GT N).Done1   Test for completion
&S      SETC  '&S.',&K'         Continue string: add comma and next value
        AGO   .Loop             Branch back to check for completed
.Done1  DC    AL1(&S.)          Generate the byte string
.Done2  ANOP                    Generation completed
```

- Try it with **'N EQU 30', 'N EQU 90', 'N EQU 300'**

Example: System-Dependent I/O Statements

- Suppose a system-interface module declares I/O control blocks for MVS, CMS, and VSE:

```
&OpSys   SETC  'MVS'              Set desired operating system
         — — —
         AIF   ('&OpSys' NE 'MVS').T1   Skip if not MVS
Input    DCB   DDNAME=SYSIN,...etc...   Generate MVS DCB
         — — —
         AGO   .T4
.T1      AIF   ('&OpSys' NE 'CMS').T2   Skip if not CMS
Input    FSCB  ,LRECL=80,...etc...      Generate CMS FSCB
         — — —
         AGO   .T4
.T2      AIF   ('&OpSys' NE 'VSE').T3   Skip if not VSE
Input    DTFCD LRECL=80,...etc...       Generate VSE DTF
         — — —
         AGO   .T4
.T3      MNOTE 8,'Unknown &&OpSys value ''&OpSys''.'
.T4      ANOP
```

- Setting of &OpSys selects statements for running on **one** system
  - Then, assemble the module with a system-specific macro library

Conditional Assembly Language Eccentricities

- Some items described above...
  1. Character string comparisons: shorter string is **always less** (slide Cond-31)
  2. Different pairing rules for ampersands and apostrophes (slide Cond-34)
  3. SETC of an arithmetic value uses its magnitude (slide Cond-34)
  4. Character functions may not be recognized in SetA expressions (slide Cond-18)
  5. Computed AGO may fall through (slide Cond-44)
  6. Logical operators in SETx and AIF statements (slide Cond-47)
- Normal, every-day language considerations:
  - Arithmetic overflows in arithmetic expressions
  - Incorrect string handling (bad substrings, exceeding 1024 characters)
- Remember, it's not a very high-level language!
  - But you can use it to **create** one!

Part 2: Basic Macro Concepts

What is a Macro Facility?

- A mechanism for extending a language
  - Introduce new statements into the language
  - Define how the new statements translate into the “base language”
    - Which may include existing macros!
  - Allow mixing old and new statements
- In Assembler Language, “new” statements are called **macro instructions** or **macro calls**
- Easy to create your own application-specific languages
  - Not only extend base language, but you can even hide it entirely!
  - Create higher-level language appropriate to application needs
  - Can be made highly portable, efficient

Benefits of Macro Facilities

- Code re-use: write once, use many times and places
- Reliability and modularity: write and debug “localized logic” once
- Reduced coding effort: minimize focus on uninteresting details
- Simplification: hide complexities, isolate impact of changes
- Easier application debugging: fewer bugs and better quality
- Standardize coding conventions painlessly
- Encapsulated, insulated interfaces to other functions
- Increased flexibility, portability, and adaptability of programs

The Macro Concept: Fundamental Mechanisms

- Macro processors rely on two basic mechanisms:

  1. **Macro recognition**: identify a character string as a macro “call”

  2. **Macro expansion**: generate a character stream to replace the “call”

     – Some macro processors allow the generated string to be re-scanned

- Macro processors typically do three things:

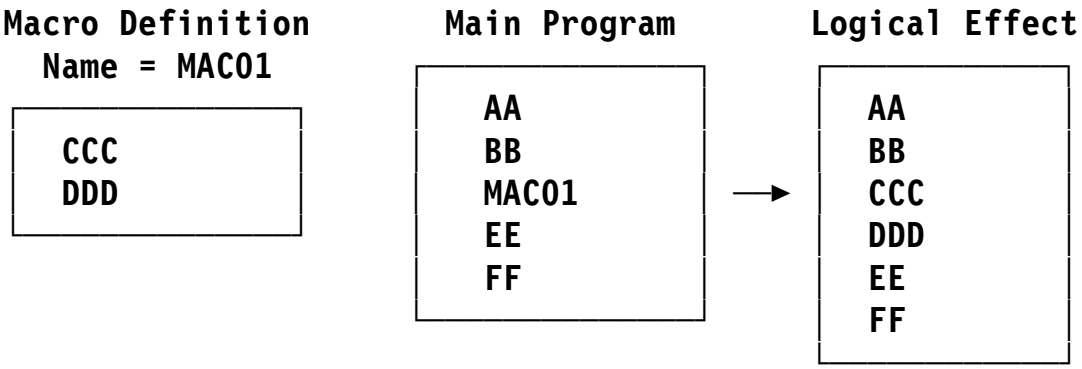
  1. **Text insertion**: injection of one stream of source program text into another stream

  2. **Text modification**: tailoring (“parameterization”) of the inserted text

  3. **Text selection**: choosing alternative text streams for insertion

Basic Macro Concepts: Text Insertion

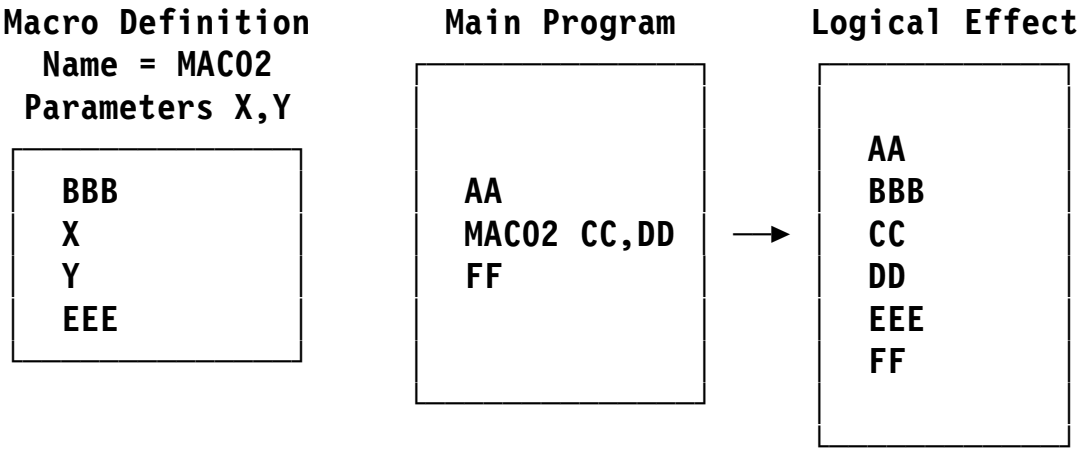
- Insertion of one stream of source program text into another stream



- The processor recognizes **MAC01** as a macro name
- The text of the macro definition replaces the **MAC01** “macro call” in the Main Program
- When the macro ends, processing resumes at the next statement (EE)

Basic Macro Concepts: Text Parameterization

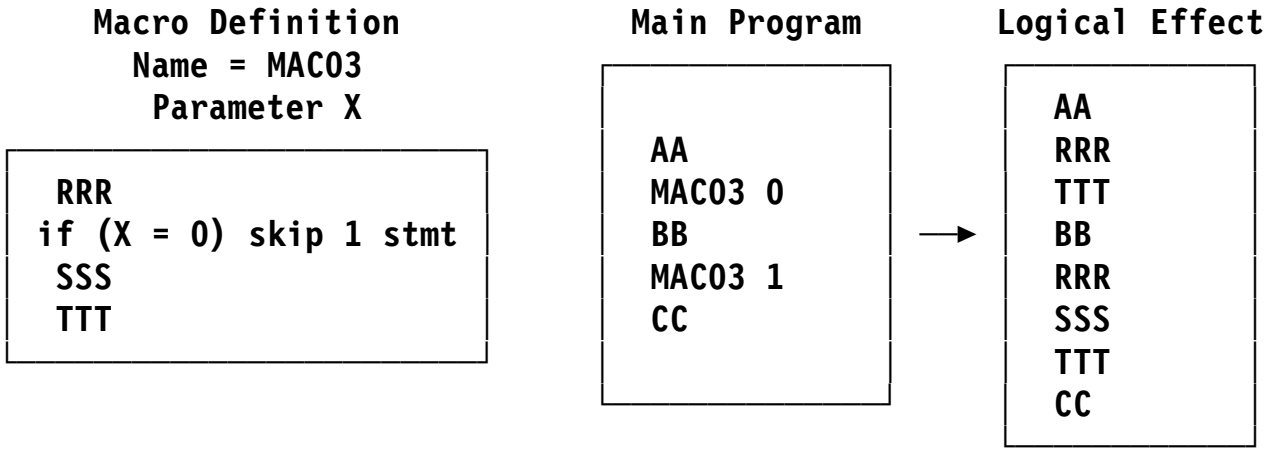
- Parameterization: tailoring of the inserted text



- Processor recognizes **MAC02** as a macro name, with arguments **CC,DD**
  - Arguments **CC,DD** are **associated** with parameters **X,Y** by **position**
    - As in all high-level languages
- The text generated from the macro definition is modified during insertion
  - The macro definition itself is unchanged

Basic Macro Concepts: Text Selection

- Selection: choosing alternative text streams for insertion



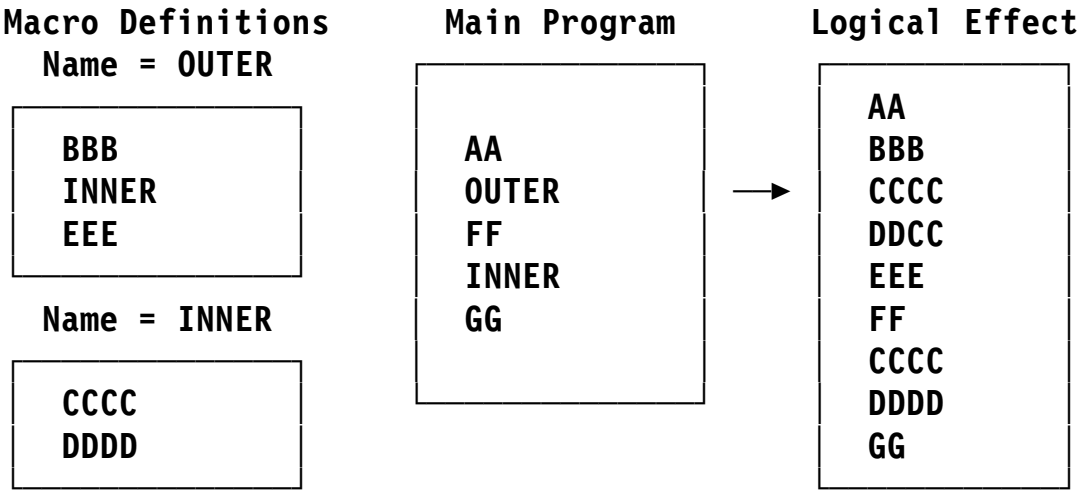
- Processor recognizes **MAC03** as a macro name with argument 0 or not 0
- Conditional actions in the macro definition allow selection of different insertion streams

Basic Macro Concepts: Call Nesting

- Generated text may include calls on other ("inner") macros
  - New statements can be defined in terms of previously-defined extensions
- Generation of statements by the outer (enclosing) macro is interrupted to generate statements from the inner
- Multiple levels of call nesting OK (including recursion)
- Technical Detail: Inner macro calls recognized during expansion of the outer macro, ***not*** during definition and encoding of the outer macro
  - Lets you pass arguments of outer macros to inner macros that depend on arguments to, and decisions in, outer macros
  - Provides better independence and encapsulation
  - Allows passing parameters through multiple levels
  - Can change definition of inner macros without having to re-define the outer

Macro Call Nesting: Example

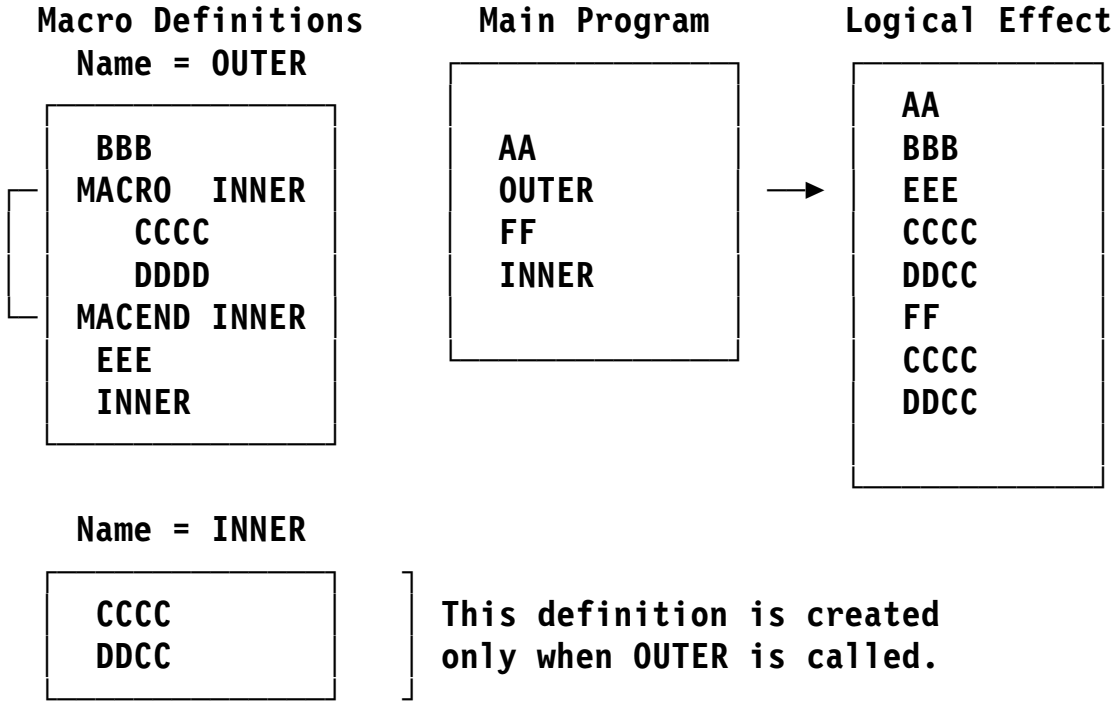
- Two macro definitions: **OUTER** contains a call on **INNER**



- Expansion of **OUTER** is suspended until expansion of **INNER** completes

Macro Definition Nesting: Example

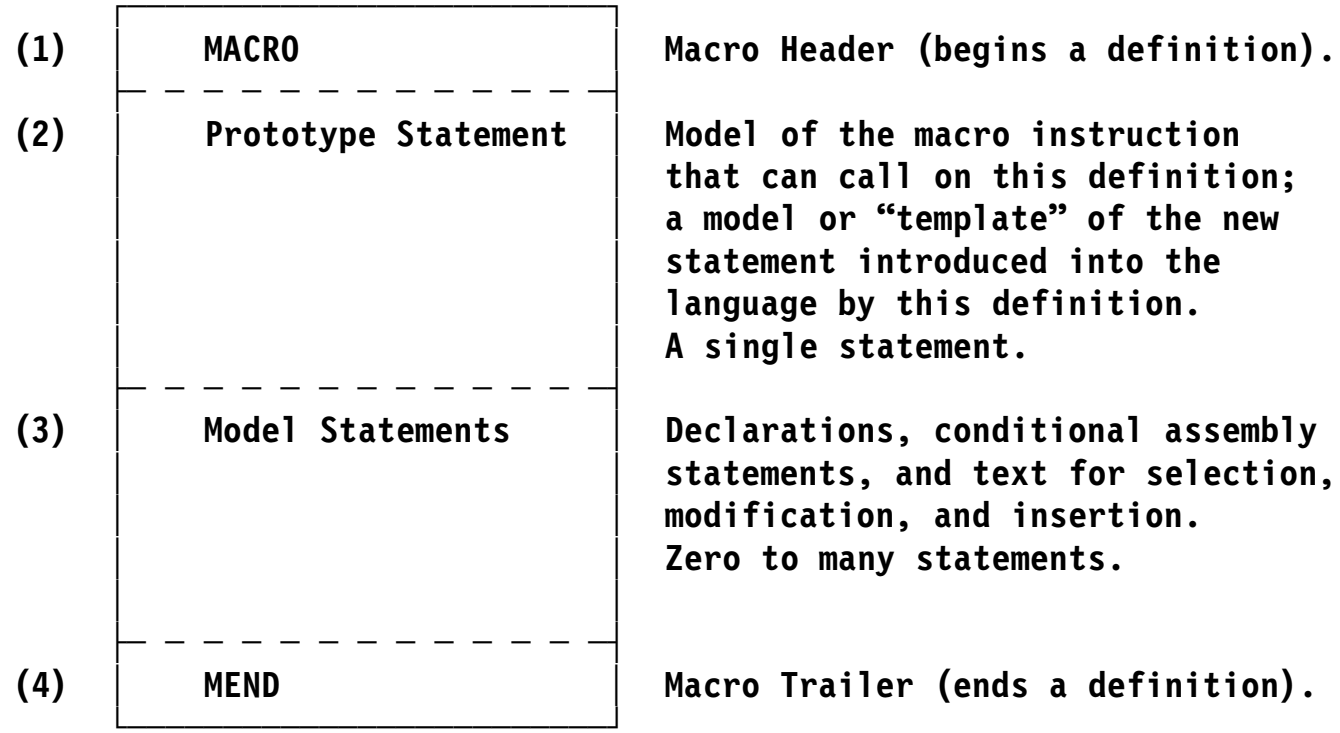
- Macro definitions may contain macro definitions



- Expanding **OUTER** causes **INNER** to be defined
  - **INNER** can then be called anywhere

The Assembler Language Macro Definition

- A macro definition has four parts:



The Assembler Language Macro Definition ...

1. **MACRO** and **MEND** statements delimit start and end of the definition
2. Declares a macro name on a *prototype statement*
   - Prototype statement also declares parameter variable symbols
3. Model statements (“macro body”) provide logic and text

- Definitions may be found
  - “in-line” (a “source macro definition”)
  - in a library (COPY can bring definitions “in-line”)
  - or both
- Macro call recognition rules affected by where the definition is found

Macro-Instruction Recognition Rules

1. If the operation mnemonic is already known as a macro name, use its definition
2. If an operation mnemonic does not match any operation mnemonic already known to the assembler (i.e., it is “possibly undefined”):
   a. Search the library for a macro definition of that name
   b. If found, encode and then use that macro definition
   c. If there is no library member with that name, the mnemonic is flagged as “undefined.”

- Macros may be redefined *during* the assembly!
  - New macro definitions supersede previous mnemonic definitions
- Name recognition activates interpretation of the macro definition
  - Also called “macro expansion” or “macro generation”

Macro-Instruction Recognition: Details

- A macro “call” could use a special CALL syntax, such as

```
         MCALL   macroname(arg1,arg2,etc...)
or       MCALL   macroname,arg1,arg2,etc...
```

- Advantages to having syntax match base language's:
  - Format of prototype dictated by desire not to introduce arbitrary forms of statement recognition for new statements
  - No special characters, statements, or rules to “trigger” recognition
  - No need to distinguish language extensions from the base language
  - Allows overriding of most existing mnemonics; language extension can be natural (and invisible)
- No need for “**MCALL**”; just make “**macroname**” the operation code

Macro-Definition Encoding (“Editing”)

- Assembler compiles a macro definition into an efficient internal format
  - Macro name is identified and saved; all parameters are identified
  - COPY statements processed immediately
  - Model and conditional assembly statements converted to “internal text” for faster interpretation
  - All points of substitution are marked
    - In name, operation, and operand fields
    - But not in remarks fields or comment statements
  - Some errors in model statements are diagnosed
    - Others may not be detected until macro expansion
  - “Dictionary” (variable-symbol table) space is defined
  - Parameter names discarded, replaced by dictionary indexes
- Avoids the need for repeated searches and scans on subsequent uses
- Re-interpretation is more flexible, but much slower
  - AINSERT statement provides some re-interpretation possibilities

Nested Macro Definitions

- Nested macro definitions are supported
  - Nested Macro/MEnd pairs are counted
- Question: should outer macro variables parameterize nested macro definitions?

```
        Macro ,        Start of MAJOR's definition
&L      MAJOR &X
        LCLA  &A       Local variable
        - - -
          Macro ,        Start of MINOR's definition
&N        MINOR &Y
          LCLA  &A       Local variable
          - - -
&A        SetA  2*&A*&Y  Evaluate expression (Problem: MAJOR's or MINOR's &A?)
          - - -
          MEnd  ,        End of MINOR's definition
        - - -
        MNote *,&&A = &A'   Display value of &A
        MEnd  ,        End of MAJOR's definition
```

- Solution: no parameterization of inner macro text (see AINSERT)
  - Statements are “shielded” from substitutions (no nested-scope problems)

Macro Expansion and MEXIT

- Macro **expansion** or **generation** is initiated by **recognition** of a macro instruction
- Assembler suspends current activity, begins to “execute” or “interpret” the encoded definition
  - Parameter values assigned from associated arguments
  - Conditional assembly statements interpreted, variable symbols assigned values
  - Model statements substituted and generated
- Generated statements *immediately* scanned for inner macro calls
  - Recognition of an inner call suspends current expansion, starts a new one
- Expansion terminates when `MEND` is reached, or `MEXIT` is interpreted
  - Some error conditions may also cause termination
  - `MEXIT` is equivalent to “`AGO to MEND`” (but quicker)
- Resume previous activity (calling-macro expansion, open code)

Macro Comments and Readability Aids

- Assembler Language supports two types of comment statement:
  1. Ordinary comments ("*****" in first column position)
     - Can be generated from macros like all other model statements
  2. Macro comments ("**.***" in first two column positions)
     - Not model statements; never generated

```
         MACRO
&N       SAMPLE1   &A
.*   This is macro SAMPLE1. It has a name-field parameter &N,
.*   and an operand-field positional parameter &A.
*    This comment is a model statement, and might be generated
```

- Two instructions help format your macro-definition listings:
  - **ASPACE** provides blank lines in listing of macros
  - **AEJECT** causes start of a new listing page for macros

Example 1: Define General Register Equates

- Generate EQUates for general register names (GR0, ..., GR15)

```
         MACRO                          (Macro Header Statement)
         GREGS                          (Macro Prototype Statement)
GR0      EQU   0                        (First Model Statement)

.*       - - -   etc.                   Similarly for GR1 — GR14

GR15     EQU   15                       (Last Model Statement)
         MEND                           (Macro Trailer Statement)
```

- A more interesting variation with a conditional-assembly loop:

```
            MACRO
            GREGS
            LCLA  &N              Define a counter variable, initially 0
.X          ANOP  ,               Next statement can't be labeled
.* 2 points of substitution in EQU statement
GR&N        EQU   &N
&N          SETA  &N+1            Increment &N by 1
            AIF   (&N LE 15).X    Repeat for all registers 1-15
            MEND
```

Macro Parameters and Arguments

- Distinguish *parameters* and *arguments*:
- Parameters are
  - declared on macro definition prototype statements
  - always local character variable symbols
  - assigned values by association with the arguments of macro calls
- Arguments are
  - supplied on a macro instruction (macro call)
  - almost any character string (typically, symbols)
  - providers of values to associated parameters

Macro-Definition Parameters

- **Parameters** are declared on the prototype statement
  - as operands, and as the name-field symbol
- All macro parameters are *read-only* local variable symbols
  - Name may not match any other variable symbol in this scope
- Parameters usually declared in exactly the same order as the corresponding actual arguments will be supplied on the macro call
  - Exception: keyword-operand parameters are declared by writing an equal sign after the parameter name
  - Can provide default keyword-parameter value on prototype statement
- Example of parameters: one name-field, two positional, one keyword

```
         MACRO
&Name    MYMAC3  &Param1,&Param2,&KeyParm=YES
         . . .
         MEND
```

Macro-Instruction Arguments

- **Arguments** are:
  - Operands (and name field entry) of a **macro instruction**
  - Arbitrary strings (with some syntax limitations)
    - — Most often, just ordinary symbols
    - — Internal quotes and ampersands in quoted strings must be paired
- Separated by commas, terminated by an unquoted blank
  - Like ordinary Assembler-Language statement operands
  - Comma and blank must otherwise be quoted
- Omitted (null) arguments are recognized, and are valid
- Examples:

```
MYMAC1    A,,'String'               2nd argument null (omitted)
MYMAC1    A,'','String'             2nd argument not null!
MYMAC1    Z,RR,'Testing, Testing'   3rd with quoted comma and blank
MYMAC1    A,B,'Do''s, && Don''ts'   3rd argument with everything...
```

Macro Parameter-Argument Association

- Three ways to associate (caller's) arguments with (definition's) parameters:

  1. by position, referenced by declared parameter name (most common way)
  2. by position, by argument number (using &SYSLIST variable symbol)
  3. by keyword: always referenced by name, arbitrary order
     - Argument *values* supplied by writing **keyname=value**

- Example 1: (Assume prototype statement as on slide/foil Mac-73)

```
&Name    MYMAC3  &Param1,&Param2,&KeyParm=YES    Prototype

Lab1     MYMAC3  X,Y,KeyParm=NO      Call: 2 positional, 1 keyword argument

*  Parameter values: &Name    = Lab1
*                    &KeyParm = NO
*                    &Param1  = X
*                    &Param2  = Y
```

Macro Parameter-Argument Association ...

- Example 2:

```
Lab2   MYMAC3   A                    Call: 1 positional argument

*  Parameter values: &Name    = Lab2
*                    &KeyParm = YES
*                    &Param1  = A
*                    &Param2  = (null string)
```

- Example 3:

```
       MYMAC3   H,KeyParm=MAYBE,J   Call: 2 positional, 1 keyword argument

*  Parameter values: &Name    = (null string)
*                    &KeyParm = MAYBE
*                    &Param1  = H
*                    &Param2  = J
```

- Common practice: put positional items first, keywords last

Constructed Keyword Arguments Do Not Work

- Keyword arguments *cannot* be created by substitution
- Suppose a macro prototype statement is

```
&X      TestMac  &K=KeyVal,&P1          Keyword and Positional Parameters
```

- If you construct an “apparent” keyword argument and call the macro:

```
&C      SetC     'K=What'               Create an apparent keyword
        TestMac  &C,Maybe               Call with “keyword” 'K=What'?
```

- This looks like a keyword and a positional argument:

```
        TestMac  K=What,Maybe           Call with ″keyword″?
```

- In fact, the argument is *positional*, with value **'K=What'** !
- Macro calls are not re-scanned after substitutions!
  - The loss of generality is traded for gains in efficiency

Example 2: Generate a Byte Sequence (BYTESEQ1)

- Rewrite previous example (slide Cond-49) as a macro
- BYTESEQ1 generates a separate DC statement for each byte value

```
         MACRO
&L       BYTESEQ1 &N              Prototype stmt: 2 positional parameters &L, &N
.*  BYTESEQ1 — generate a sequence of byte values, one per statement.
.*  No checking or validation is done.
         LclA  &K
         AIF   ('&L' EQ '').Loop  Don't define the label if absent
&L       DS    0AL1               Define the label
.Loop    ANOP
&K       SetA  &K+1               Increment &K
         AIF   (&K GT &N).Done    Check for termination condition
         DC    AL1(&K)
         AGO   .Loop              Continue
.Done    MEND
```

- Examples

```
BSeq1    BYTESEQ1  5
         BYTESEQ1  1
```

Macro Parameter Usage in Model Statements

- Values supplied by arguments in the macro instruction (“call”) are substituted in parameter symbols as character strings
- Parameters may be substituted in name, operation, and operand fields of model statements
  - Substitution points ignored in remarks fields and comment statements
    - — Can sometimes play tricks with operand fields containing blanks
    - — AINSERT lets you generate fully substituted statements
- Some limitations on which opcodes may be substituted in conditional assembly statements
  - Can't substitute **ACTR, AGO, AIF, ANOP, AREAD, COPY, GBLx, ICTL, LCLx, MACRO, MEND, MEXIT, REPRO, SETx, SETxF**
  - The assembler must understand basic macro structures at the time it encodes the macro!
- Implementation trade-off: generation speed vs. generality

Macro Argument Attributes and Structures

- Several mechanisms “ask questions” about macro arguments
- Simplest forms are *attribute references*
  - Determine attributes of the actual arguments
  - Most common questions: “What is it?” and “How big is it?”
- Some provide data about possible base-language properties of symbols: Type (**T'**), Length (**L'**), Scale (**S'**), Integer (**I'**), Defined (**D'**), and “OpCode” (**O'**) attributes
- Count (**K'**) and Number (**N'**) attribute references
  - Neither references any base language attribute
  - **K'** determines the count of characters in an argument
  - **N'** determines the number and nesting of argument list structures
    - Helps extract sublists or sublist elements
    - Decompose argument structures, especially parenthesized lists

Macro Argument Attributes: Type

- Type attribute reference (**T'**) answers
  - "What is it?"
  - "What meaning might it have in the ordinary assembly (base) language?"
    - The answer can be "None" or "I can't tell"!
  - Value of **T'** is a single character
- Assume the following statements in a program:

```
A       DC    A(*)
B       DC    F'10'
C       DC    E'2.71828'
D       MVC   A,B
```

- And, assume the following prototype statement for MACTA:

```
        MACTA &P1,&P2,...,etc.
```

  - Just a numbered list of positional parameters...

Macro Argument Attributes: Type ...

- Then a call to MACTA like

```
Z       MACTA A,B,C,D,C'A',,'?',Z        Call MACTA with various arguments
```

- would provide these type attributes:

```
T'&P1 = 'A'    aligned, implied-length address
T'&P2 = 'F'    aligned, implied-length fullword binary
T'&P3 = 'E'    aligned, implied-length short floating-point
T'&P4 = 'I'    machine instruction statement
T'&P5 = 'N'    self-defining term
T'&P6 = 'O'    omitted (null)
T'&P7 = 'U'    unknown, undefined, or unassigned
T'&P8 = 'M'    macro instruction statement
```

Macro Argument Attributes: Count

- Count attribute reference (**K'**) answers:
  - "How many characters in a SETC variable symbol's value (or in its character representation, if not SETC)?"
- Suppose macro MAC8 has many positional and keyword parameters:

```
MAC8  &P1,&P2,&P3,...,&K1=,&K2=,&K3=,...
```

- This macro instruction would give these count attributes:

```
MAC8  A,BCD,'EFGH',,K1=5,K3==F'25'

 K'&P1 = 1 corresponding to  A
 K'&P2 = 3                   ABC
 K'&P3 = 6                   'DEFG'
 K'&P4 = 0                   (omitted; explicitly null)
 K'&P5 = 0                   (implicitly null; no argument)
 K'&K1 = 1                   5
 K'&K2 = 0                   (null default value)
 K'&K3 = 6                   =F'25'
```

Macro Argument Attributes: Lists and Number Attribute

- Number attribute reference (**N'**) answers "How many items in a list or sublist?"
- **List**: a parenthesized sequence of items separated by commas

  Examples: **(A)** **(B,C)** **(D,E,,F)**

- List items may themselves be lists, to any nesting

  Examples: **((A))** **(A,(B,C))** **(A,(B,C,(D,E,,F),G),H)**

- Subscripts on parameters refer to argument list (and sublist) items
  - Each added subscript references one nesting level deeper
  - Provides powerful list-parsing capabilities
- **N'** also determines maximum subscript used for a subscripted variable symbol

Macro Argument List Structure Examples

- Assume the same macro prototype as in slide Mac-83:

```
MAC8  &P1,&P2,&P3,&P4,...          Prototype

MAC8  (A),A,(B,C),(B,(C,(D,E)))    Sample macro call
```

- Then, the lists, sublists, and number attributes are:

```
&P1        = (A)            N'&P1        = 1   1-item list: A
&P1(1)     = A              N'&P1(1)     = 1   (A is not a list)
&P2        = A              N'&P2        = 1   (A is not a list)
&P3        = (B,C)          N'&P3        = 2   2-item list: B and C
&P3(1)     = B              N'&P3(1)     = 1   (B is not a list)
&P4        = (B,(C,(D,E)))  N'&P4        = 2   2-item list: B and (C,(D,E))
&P4(2)     = (C,(D,E))      N'&P4(2)     = 2   2-item list: C and (D,E)
&P4(2,2)   = (D,E)          N'&P4(2,2)   = 2   2-item list: D and E
&P4(2,2,1) = D              N'&P4(2,2,1) = 1   (D is not a list)
&P4(2,2,2) = E              N'&P4(2,2,2) = 1   (E is not a list)
```

Macro Argument Lists and &SYSLIST

- &SYSLIST(k): a synonym for the k-th positional parameter
  - Whether or not a named positional parameter was declared
  - Additional subscripts for deeper nesting levels
- N'&SYSLIST = number of **all** positional arguments
- Assume a macro prototype MACNP (with or without parameters)
- Then these four arguments have Number attributes as shown:

```
          MACNP  A,(A),(C,(D,E,F)),(YES,NO)

N'&SYSLIST        = 4                                    MACNP has 4 arguments
N'&SYSLIST(1)     = 1    &SYSLIST(1)     = A             (A is not a list)
N'&SYSLIST(2)     = 1    &SYSLIST(2)     = (A)           a list with 1 item
N'&SYSLIST(3)     = 2    &SYSLIST(3)     = (C,(D,E,F)) a list with 2 items
N'&SYSLIST(3,2)   = 3    &SYSLIST(3,2)   = (D,E,F)       a list with 3 items
N'&SYSLIST(3,2,1) = 1    &SYSLIST(3,2,1) = D             (D is not a list)
N'&SYSLIST(4)     = 2    &SYSLIST(4)     = (YES,NO)      a list with 2 items
```

- &SYSLIST(0) refers to the macro call's name field entry

Macro Argument Lists and Sublists: Details

- HLASM can treat macro argument lists in two ways:
  as lists or as strings

- Old assemblers pass these two types of argument differently:

```
        MYMAC   (A,B,C,D)       Macro call with a list argument

&Char   SetC    '(A,B,C,D)'     Create argument for MYMAC call
        MYMAC   &Char           Macro call with a string argument
```

- COMPAT(SYSLIST) option enforces “old rules”
  - Inner-macro arguments treated as having no list structure
  - Prototype statements must parse the argument one character at a time
- NOCOMPAT(SYSLIST) relaxes restrictions on inner macros

Global Variable Symbols

- Macro calls have a serious defect:
  - Can't *assign* (i.e. return) values to arguments
    - unlike most high level languages
  - “One-way” communication with a macro: arguments in, statements out
  - No “functions” (i.e. macros returning a value)
- Values to be shared among macros (and/or with open code) must use global variable symbols
  - Scope: available to all declarers
  - Can use the same name as a local variable in a scope that does not declare the name as global
- One macro can create (multiple) values for others to use

Variable Symbol Scope Rules: Summary

- Global Variable Symbols
  - Available to all declarers of those variables on GBLx statements (macros and open code)
  - **Must** be declared explicitly
  - **A**rithmetic, **B**oolean, and **C**haracter types; may be subscripted
  - Values persist through an entire assembly
    - Values kept in a single, shared, common dictionary
  - Values are shared by name
  - All declarations must be consistent (type, scalar vs. dimensioned)

Variable Symbol Scope Rules: Summary ...

- Local Variable Symbols
  - Explicitly and implicitly declared local variables
  - Symbolic parameters
    - Values are “read-only”
  - Local copies of system variable symbols
    - Value is constant throughout a macro expansion (except &SYSM_SEV, &SYSM_HSEV)
    - Values kept in a local, transient dictionary
      - Created on macro entry, discarded on macro exit
    - Recursion implies a separate dictionary for each entry
  - Open code has its own local dictionary

Macro Debugging Techniques

- Complex macros can be hard to debug
  - Written in an poorly structured language
- Some useful debugging facilities are available:
  1. MNOTE statement
     - Can be inserted liberally to trace control flows and display values
  2. MHELP statement
     - Built-in assembler trace and display facility
     - Many levels of control; output can be quite verbose!
  3. ACTR statement
     - Limits number of conditional branches within a macro
     - Very useful if you suspect excess looping
  4. LIBMAC Option
     - Library macros appear to be defined in-line
  5. PRINT MCALL statement, PCONTROL(MCALL) option
     - Displays inner-macro calls

Macro Debugging: The MNOTE Statement

- MNOTE allows the most detailed controls over debugging output (see also slide Cond-48)
- **You** specify exactly what to display, and where

```
        MNote *,'At Skip19: &&VG = &VG., &&TEXT = ''&TEXT'''
```

- **You** can control which ones are active (with global variable symbols)

```
        GblB  &DEBUG(20)
        — — —
        AIF   (NOT &DEBUG(7)).Skip19
        MNote *,'At Skip19: &&VG = &VG., &&TEXT = ''&TEXT'''
.Skip19 ANop
```

- **You** can use &SYSPARM values to set debug switches
- **You** can “disable” MNOTEs with conditional-assembly comments

```
.*      MNote *,'At Skip19: &&VG = &VG., &&TEXT = ''&TEXT'''
```

Macro Debugging: The MHELP Statement

- MHELP controls display of conditional-assembly flow tracing and variable "dumping"
  - Use with care; output is potentially large
- MHELP operand value is sum of 8 bit values:

1	Trace macro calls (name, nesting depth, &SYSNDX value)
2	Trace macro branches (AGO, AIF)
4	AIF dump (dump scalar SET symbols before AIFs)
8	Macro exit dump (dump scalar SET symbols on exit)
16	Macro entry dump (dump parameter values on entry)
32	Global suppression (suppress GBL symbols in AIF, exit dumps)
64	Hex dump (SETC and parameters dumped in hex and EBCDIC)
128	MHELP suppression (turn off all active MHELP options)

  - Best to set operand with a GBLA symbol (can save/restore its value), or from &SYSPARM value
- Can also limit total number of macro calls (see Language Reference)

Macro Debugging: The ACTR Statement

- ACTR helps control excessive looping
  - Specifies the maximum number of conditional-assembly branches in a macro or open code

```
        ACTR  200         Limit of 200 successful branches
```

  - Scope is local (to open code, and to each macro)
    - Can set different values for each; default is 4096
  - Count decremented by 1 for each successful branch
  - When count goes negative, macro's invocation is terminated
- Executing erroneous conditional assembly statements halves the ACTR value!

```
.*      Following statement has syntax errors
&J      SETJ  &J+?        If executed, would cause ACTR = ACTR / 2
```

Macro Debugging: The LIBMAC Option

- The LIBMAC option causes library macros to be defined “in-line”
  - Specify as invocation option, or on a *PROCESS statement

    ***PROCESS LIBMAC**

- Errors in library macros can be hard to find:
  - HLASM can only indicate “There's an error in macro XYZ”
  - Specific location (and cause) are hard to determine
- LIBMAC option causes library macros to be treated as “source”
  - Can use **ACONTROL [NO]LIBMAC** statements to limit LIBMAC range
- Errors can be indicated for specific macro statements
- Errors can be found without
  - modifying any source
  - copying macros into the program

Macro Debugging: The PRINT MCALL Statement

- PRINT [NO]MCALL controls display of inner macro calls

```
PRINT MCALL          Turns ON  inner-macro call display
PRINT NOMCALL        Turns OFF inner-macro call display
```

  - Normally, you see only the outermost call and generated code from it and all nested calls
    - Difficult to tell which macro may have received invalid arguments
  - With MCALL, HLASM displays each macro call before processing it
    - Some limitations on length of displayed information

- PCONTROL([NO]MCALL) option
  - Forces PRINT MCALL on [or off] for the assembly
  - Specifiable at invocation time, or on a *PROCESS statement:

```
*PROCESS PCONTROL(MCALL)
```

Part 3: Macro Techniques

Macros as a Higher Level Language

- Can be created to perform very simple to very complex tasks
  - Housekeeping (register saving, calls, define symbols, map structures)
  - Define your own application-specific language increments and features
- Macros can provide much of the “goodness” of HLLs
  - Abstract data types, private data types
  - Information hiding, encapsulation
  - Avoiding side-effects
  - Polymorphism
  - Enhanced portability
- Macros can be built incrementally to suit ***your*** application needs
  - Can develop “application-specific languages” and increments
    - Easier to learn, because it relates to your application
  - Code re-use promotes faster learning, fewer errors
- Avoid struggling with the latest “universal language” fad
  - Add new capabilities to existing applications without converting

Examples of Macro Techniques

- Sample-problem “case studies” illustrate some techniques
  1. Define EQUated names for registers
  2. Generate a sequence of byte values
  3. “MVC2” macro takes implied length from second operand
  4. Generate lists of named integer constants
  5. Create length-prefixed message text strings and free-form comments
  6. Recursion (indirect addressing, factorials, Fibonacci numbers)
  7. Basic and advanced bit-handling techniques
  8. Defining assembler and user-specified data types and operations
  9. Utilizing Program Attributes for strong and private typing
  10. “Front-ending” or “wrapping” a library macro

Case Study 1: EQUated Symbols for Registers

- Write a GREGS macro to define “symbol equates” for GPRs
- Basic form: simply generate the 16 EQU statements
- Variation 1: ensure that “symbol equates” are generated only once
- Variation 2: generate equates once for up to four register types
  - **G**eneral Purpose, **F**loating Point, **C**ontrol, **A**ccess

Define General Register Equates (Simply)

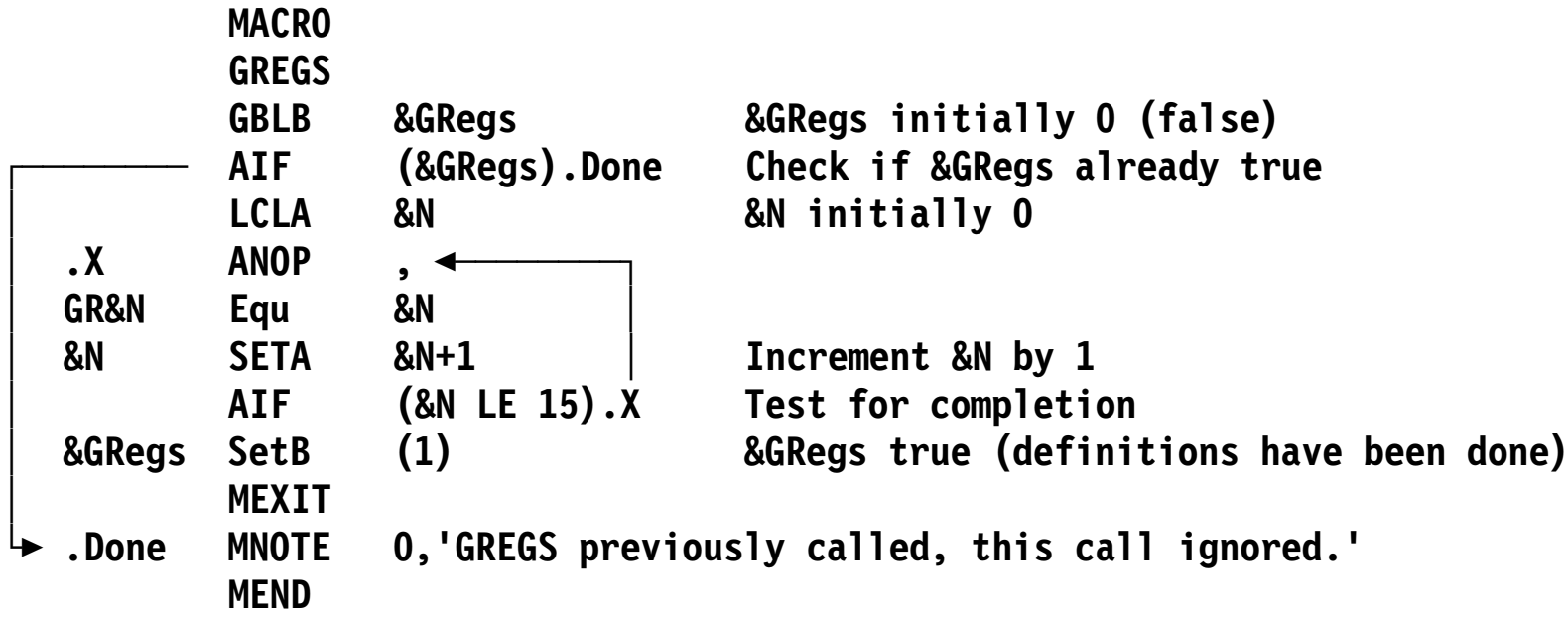
- Define “symbol equates” for GPRs with this macro (see slide Mac-71)

```
         MACRO
         GREGS
GR0      Equ   0
GR1      Equ   1
.*       - - -   etc.
GR15     Equ   15
         MEND
```

- Problem: what if two code segments are combined?
  - If each calls GREGS, could have duplicate definitions
  - How can we preserve modularity, and define symbols only once?
- Answer: use a global variable symbol **&GRegs**
  - Value is available across all macro calls

Define General Register Equates (Safely)

- Initialize **&GRegs** to “false”; set to “true” when EQUs are generated



```
         MACRO
         GREGS
         GBLB    &GRegs           &GRegs initially 0 (false)
         AIF     (&GRegs).Done    Check if &GRegs already true
         LCLA    &N               &N initially 0
.X       ANOP    ,
GR&N     Equ     &N
&N       SETA    &N+1             Increment &N by 1
         AIF     (&N LE 15).X     Test for completion
&GRegs   SetB    (1)              &GRegs true (definitions have been done)
         MEXIT
.Done    MNOTE   0,'GREGS previously called, this call ignored.'
         MEND
```

- If **&GRegs** is **true**, no statements are generated

```
         GREGS
         GREGS  This,Call,Is,Ignored
```

Defining Register Equates Safely: Pseudo-Code

- Allow declaration of multiple register types on one call:
  Example: REGS $type_1$[,$type_2$]... as in REGS G,F
- Pseudo-code:

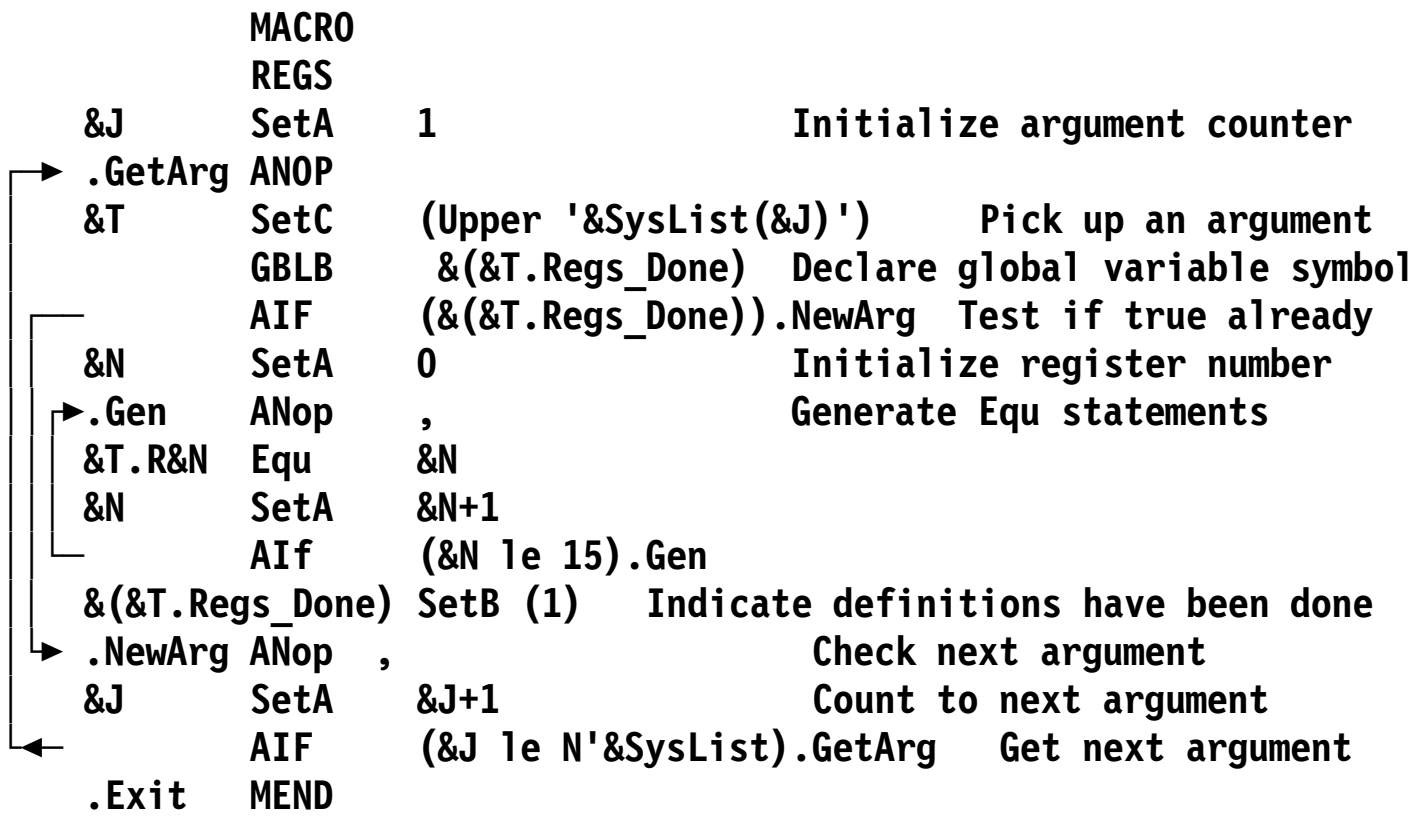
```
IF (number of arguments is zero) EXIT
FOR each argument:
   Verify valid register type &T (values A, C, F, or G):
      IF invalid, ERROR EXIT with message
   IF (that type already done) Give message and ITERATE
   Generate equates
   Set appropriate 'Type_Done' flag and ITERATE
```

- 'Type_Done' flags are global Boolean variable symbols
  - Use created variable symbols **&(&T.Regs_Done)**
- If **&(&T.Regs_Done)** is **true**, no statements are generated

```
REGS   G,F,A,G     G registers are not defined twice!
```

Define All Register Equates Safely

- Improved REGS macro, to generate EQUates for four register types
  - Basic form; error checking not shown



```
         MACRO
         REGS
   &J    SetA   1                 Initialize argument counter
.GetArg ANOP
   &T    SetC   (Upper '&SysList(&J)')     Pick up an argument
         GBLB    &(&T.Regs_Done)  Declare global variable symbol
         AIF    (&(&T.Regs_Done)).NewArg  Test if true already
   &N    SetA   0                 Initialize register number
.Gen     ANop   ,                 Generate Equ statements
   &T.R&N Equ    &N
   &N    SetA   &N+1
         AIf    (&N le 15).Gen
   &(&T.Regs_Done) SetB (1)   Indicate definitions have been done
.NewArg ANop  ,                    Check next argument
   &J    SetA   &J+1               Count to next argument
         AIF    (&J le N'&SysList).GetArg   Get next argument
.Exit    MEND
```

Case Study 2: Generate Sequence of Byte Values

- Generate a sequence of bytes containing values 1,2,...,N
  - An alternative to literals
- Basic form: simple loop generating one byte at a time
- Variation: check input, generate a single DC with all values

Generating a Byte Sequence: BYTESEQ1 Macro

- BYTESEQ1 generates a separate DC statement for each value (compare slides Cond-49 and Mac-78)

```
         MACRO
&L       BYTESEQ1 &N
.*  BYTESEQ1 — generate a sequence of byte values, one per statement.
.*  No checking or validation is done.
         LclA  &K
         AIF   ('&L' EQ '').Loop  Don't define the label if absent
&L       DS    0AL1               Define the label
.Loop    ANOP
&K       SetA  &K+1               Increment &K
         AIF   (&K GT &N).Done    Check for termination condition
         DC    AL1(&K)
         AGO   .Loop              Continue
.Done    MEND

 BSeq1a  BYTESEQ1  3
+BSeq1a  DC    AL1(1)
+        DC    AL1(2)
+        DC    AL1(3)
```

Generating a Byte Sequence: Pseudo-Code

- BYTESEQ2: generate a single DC statement, creating a string of bytes with binary values from 1 to N
  - N has been previously defined as an absolute symbol

```
IF (N not self-defining) ERROR EXIT with message

IF (N > 255) ERROR EXIT with too-big message

IF (N ≤ 0) EXIT with notification

Set local string variable S = '1'
DO for K = 2 to N
   S = S || ',K'          (append comma and next value)
GEN (label  DC  AL1(S) )
```

- Compare to slide Cond-50

Generating a Byte Sequence (BYTESEQ2)

```
        MACRO
&L      BYTESEQ2 &N             Generates a single DC statement
&K      SetA     1              Initialize generated value counter
&S      SetC     '1'            Initialize output string
        AIF      (T'&N EQ 'N').Num   Validate type of argument
        MNOTE    8,'BYTESEQ2 — &&N=&N not self-defining.'
        MEXIT
.Num    AIF      (&N LE 255).NotBig  Check size of argument
        MNOTE    8,'BYTESEQ2 — &&N=&N is too large.'
        MEXIT
.NotBig AIF      (&N GT 0).OK        Check for small argument
        MNOTE    *,'BYTESEQ2 — &&N=&N too small, no data generated.'
        MEXIT
.OK     AIF      (&K GE &N).DoDC     If done, generate DC statement
&K      SetA     &K+1           Increment &K
&S      SetC     '&S.','&K'     Add comma and new value of &K to &S
        AGO      .OK            Continue
.DoDC   ANOP
&L      DC       AL1(&S)
        MEND
```

Case Study 3: MVC2 Macro

- Want to do an MVC, but with the *source* operand's length:

```
         MVC2  Buffer,=C'Message Text'    Want to move only 12 characters...
         - - -
Buffer   DS    CL133                      even though buffer is longer
```

  - MVC would move 133 bytes!

- MVC2 macro uses ORG statements, forces literal definitions

```
         Macro
&Lab     MVC2  &Target,&Source
&Lab     CLC   0(0,0),&Source     X'D500 0000',S(&Source)
         Org   *-6                Back up to first byte of instruction
         LA    0,&Target.(0)      X'4100',S(&Target),S(&Source)
         Org   *-4                Back up to first byte of instruction
         DC    AL1(X'D2',L'&Source-1)  First 2 bytes of instruction
         Org   *+4                Step to next instruction
         MEnd
```

- The CLC instruction puts literal source operands into the assembler's symbol table, so it's available for the L' reference

Case Study 4: Generate Named Integer Constants

- Intent: generate a list of “intuitively” named halfword or fullword integer constants
  - An alternative to using literals
- For example:
  - Fullword constant “1” is named **F1**
  - Halfword constant “−1” is named **HM1**

Generate a List of Named Integer Constants

- Syntax: INTCONS n_1[,n_2]...[,Type=F]
  - Default constant type is F
- Examples:

```
 C1b     INTCONS  0,-1                        Type F: names F0, FM1
+C1b     DC     0F'0'       Define the label
+F0      DC     F'0'
+FM1     DC     F'-1'

 C1c     INTCONS  99,-99,Type=H               Type H: names H99, HM99
+C1c     DC     0H'0'       Define the label
+H99     DC     H'99'
+HM99    DC     H'-99'
```

Generate a List of Named Integer Constants ...

- INTCONS Macro definition (with validity checking omitted)

```
        MACRO
&Lab    INTCONS &Type=F         Default type is F
        AIF    ('&Lab' eq '').ArgsOK  Skip if no label
&Lab    DC     O&Type.'O'       Define the label
.ArgsOK ANOP                    Argument-checking loop
&J      SetA   &J+1             Increment argument counter
        AIF    (&J GT N'&SysList).End  Exit if all done
&Name   SetC   '&Type.&SysList(&J)'  Assume non-negative arg
        AIF    ('&SysList(&J)'(1,1) ne '-').NotNeg  Check arg sign
&Name   SetC   '&Type.M'.'&SysList(&J)'(2,*)  Negative argument, drop -
.NotNeg ANOP
&Name   DC     &Type.'&SysList(&J)'
        AGO    .ArgsOK          Repeat for further arguments
.End    MEND
```

- Exercise: generalize to support + signs on operands

Case Study 5: Using the AREAD Statement

1. Case Study 5a: Generate strings of message text
    - Prefix string with “effective length” byte (length-1)
    - Basic form: count characters
    - Variation 1: create an extra symbol, use its length attribute
    - Variation 2: use the AREAD statement and conditional-assembly functions to support “readable” input
2. Case Study 5b: Block comments
    - Write free-form text comments (without * in column 1)

Case Study 5a: Create Length-Prefixed Message Texts

- Problem: want messages prefixed with “effective length” field

- How such prefixed strings might be used:

```
 HWMsg   PFMSG 'Hello World'       Define a sample message text
+HWMsg   DC    AL1(10),C'Hello World'   Length–prefixed message text
         – – –
         – – –
         LA    2,HWMsg             Prepare to move message to buffer
         IC    1,0(,2)             Effective length of message text
         EX    1,MsgMove           Move message to output buffer
         – – –
MsgMove  MVC   Buffer(*–*),1(2)    Executed to move message texts
```

- Nobody should *ever* have to count characters!
  - Let the assembler do it for you!

Create Length-Prefixed Messages (1)

- PFMSG1: length-prefixed message texts

```
        MACRO
&Lab    PFMSG1 &Txt
.*    PFMSG1 — requires that the text of the message, &Txt,
.*    contain no embedded apostrophes (quotes) or ampersands.
        LclA   &Len              Effective Length
&Len    SetA   K'&Txt-3          (# text chars)-3 (2 quotes, eff. length)
&Lab    DC     AL1(&Len),C&Txt
        MEND
```

- Limited to messages with no quotes or ampersands

```
 M1a    PFMSG1 'This is a test of message text 1.'
+M1a    DC    AL1(32),C'This is a test of message text 1.'

 M1b    PFMSG1 'Hello'
+M1b    DC    AL1(4),C'Hello'
```

Create General Length-Prefixed Messages (2)

- PFMSG2: Allow all characters in text (but: may require pairing)

```
         MACRO
&Lab     PFMSG2 &Txt
.*     PFMSG2 — the text of the message, &Txt, may contain embedded
.*     apostrophes (quotes) or ampersands, so long as they are paired.
&T       SetC   'TXT&SYSNDX.M'   Create TXTnnnM symbol to name the text
&Lab     DC     AL1(L'&T.-1)     Effective length
&T       DC     C&Txt
         MEND

 M2a       PFMSG2 'Test of ''This'' && ''That''.'
+M2a       DC     AL1(L'TXT0001M-1)      Effective length
+TXT0001M DC     C'Test of ''This'' && ''That''.'

 M2b       PFMSG2 'Hello, World'
+M2b       DC     AL1(L'TXT0002M-1)      Effective length
+TXT0002M DC     C'Hello, World'
```

- Quotes/ampersands in message are harder to write, read, translate
- Extra (uninteresting) labels are generated

Readable Length-Prefixed Messages (3): Pseudo-Code

- User writes “plain text” messages (single line, ≤ 72 characters)
- PFMSG3: AREAD statement within the macro “reads” the next source record in the input stream (following the macro call, usually) into a character variable symbol
- Allow all characters in message text without pairing, by using AREAD
- Pseudo-code:

```
IF (any positional arguments) ERROR EXIT with message

[1] AREAD a message from the following source record

[2] Trim off sequence field (73–80) and trailing blanks, note length

[3] Create paired quotes and ampersands (for nominal value in DC)

[4] GEN (label  DC  AL1(Text_Length–1),C'MessageText')
```

Create Readable Length-Prefixed Messages

```
        MACRO
&Lab    PFMSG3  &Null           Comments OK after comma
.*    PFMSG3 — the text of the message may contain any characters.
.*    The message is on a single record following the call to PFMSG3.
        LclA   &L               Local arithmetic variables
        LclC   &T,&C,&M         Local character variables
        AIF    ('&Null' eq '').OK     Null argument OK
        AIF    (N'&SYSLIST EQ 0).OK   No arguments allowed
        MNote  4,'PFMSG3 — no operands should be provided.'
        MEXIT                  Terminate macro processing
.OK     ANOP
.*  Read the record following the PFMSG3 call into &M
&M      ARead  ,               Read the message text [1]
&M      SetC   '&M'(1,72)      Trim off sequence field [2...]
&L      SetA   72              Point to end of initial text string
.*  Trim off trailing blanks from message text
.Trim   AIF    ('&M'(&L,1) NE ' ').C  Check last character
&L      SetA   &L-1            Deduct blanks from length
        AGO    .Trim           Repeat trimming loop
.*      - - -  (continued)
```

Create Readable Length-Prefixed Messages ...

```
.*      - - -  (continuation)
.C      ANOP
&T      SetC   DOUBLE('&M'(1,&L))  Pair-up quotes, ampersands [3]
&L      SetA   &L-1              Set to effective length
&Lab    DC     AL1(&L),C'&T' [4]
        MEnd
```

- Messages are written as they are expected to appear!
- Easier to read and translate to other national languages

```
 M4a     PFMSG3  ,     Test with mixed apostrophes/ampersands
-Test of 'This' & 'That'.
+M4a     DC     AL1(27),C'Test of ''This'' && ''That''.'

 M4c     PFMSG3
-This is the text of a long message & says nothin' very much.
+M4c     DC     AL1(63),C'This is the text of a long message && saysX
+                nothin'' very much.'
```

- '+' prefix in listing for generated statements, '-' for AREAD records

Case Study 5b: Block Comments

- Sometimes want to write “free-form” comments in a program:

```
This is some text
  for a block of
    free-form comments.
```

- Must tell HLASM where the block of comments begins and ends:

```
         COMMENT
This is some text
  for a block of
    free-form comments.
         TNEMMOC         (or 'ENDCOMMENT' or whatever you prefer...)
```

- Restriction: block-end statement (**TNEMMOC**) can't appear in the text

Block Comments Macro

- COMMENT macro initiates block comments:

```
         Macro
&L       Comment &Arg
         LclC    &C
         AIf     ('&L' eq '' and '&Arg' eq '').Read
         MNote   *,'Comment macro: Label and/or argument ignored.'
.Read    ANop
&C       ARead   ,
&C       SetC    Upper('&C')                      Force upper case
&A       SetA    Index('&C'(1,72),' TNEMMOC ')    Note blanks!
         AIf     (&A eq 0).Read
         MEnd
```

- Lets you include your documentation text with the source code!

Case Study 6: Macro Recursion

- Macro recursion illustrated with:

  1. Integer factorial values: `Fac(N) = N × Fac(N−1)`

  2. Integer Fibonacci numbers: `Fib(N) = Fib(N−1) + Fib(N−2)`

  3. “Indirect addressing” via “Load Indirect”

Generate Factorial Values Recursively

- Factorial: defined by Fac(N) = N × Fac(N−1), Fac(0) = Fac(1) = 1

```
        Macro
&Lab    FACTORAL  &N
        GBLA      &Ret              For returning values of inner calls
&L      SetA      &N                Convert from external form
        AIF       (&L GE 2).Calc    Calculate via recursion if N > 1
&Ret    SetA      1                 Fac(0) = Fac(1) = 1
        AGO       .Test             Return to caller
.Calc   ANOP
&K      SetA      &L-1
        FACTORAL  &K                Recursive call
&Ret    SetA      &Ret*&L
.Test   AIF       (&SysNest GT 1).Exit   Check nesting level
.*      MNote     0,'Factorial(&L.) = &Ret.'  Intermediate result
&Lab    DC        F'&Ret'
.Exit   MEnd
```

- Error checking omitted...

Generate Fibonacci Numbers: Pseudo-Code

- Defined by Fib(0) = Fib(1) = 1, Fib(n) = Fib(n−1) + Fib(n−2)
  - Sequence is 1, 1, 2, 3, 5, 8, 13, 21, ...
- Use a global arithmetic variable `&Ret` for returned values
- Pseudo-code:

```
IF (argument N < 0) ERROR EXIT with message

IF (N < 2) Set &Ret = 1 and EXIT

CALL myself recursively with argument N–1
Save evaluation in local temporary &Temp

CALL myself recursively with argument N–2
Set &Ret = &Ret + &Temp, and EXIT
```

Generate Fibonacci Numbers Recursively

```
        Macro
&Lab    FIBONACI  &N
        GBLA      &Ret              For returning values of inner calls
        MNote     0,'Evaluating FIBONACI(&N.), Level &SysNest.'
        AIF       (&N LT 0).Error   Negative values not allowed
        AIF       (&N GE 2).Calc    If &N > 1, use recursion
&Ret    SETA      1                 Return Fib(0) or Fib(1)
        AGO       .Test             Return to caller
.Calc   ANOP                        Do computation
&K      SetA      &N-1              First value 'K' = N-1
&L      SetA      &N-2              Second value 'L' = N-2
        FIBONACI  &K                Evaluate Fib(K) = Fib(N-1) (Recursive call)
&Temp   SetA      &Ret              Hold computed value
        FIBONACI  &L                Evaluate Fib(L) = Fib(N-2) (Recursive call)
&Ret    SetA      &Ret+&Temp        Evaluate Fib(N) = Fib(K) + Fib(L)
.Test   AIF       (&SysNest GT 1).Cont   Check nesting level
        MNote     0,'Fibonacci(&N.) = &Ret..'  Display result
&Lab    DC        F'&Ret'
.Cont   MExit                       Return to caller
.Error  MNote     11,'Invalid Fibonacci argument &N..'
        MEnd
```

Indirect Addressing via Recursion

- “Load Indirect” macro for multi-level “pointer following”
- Syntax: each prefixed asterisk adds one level of indirection

```
         LI    3,0(4)              Load from 0(4)
         LI    3,*0(,4)            Load from what 0(,4) points to
         LI    3,**0(,7)           Two levels of indirection
         LI    3,***X              Three levels of indirection
```

- LI macro calls itself for each level of indirection

```
         Macro
&Lab     LI    &Reg,&X             Load &Reg with indirection
         Aif   ('&X'(1,1) eq '*').Ind  Branch if indirect
.*       Generate top-level (direct) reference
&Lab     L     &Reg,&X
         MExit                     Exit from bottom level of recursion
.Ind     ANop
&XI      SetC  '&X'(2,*)           Remove leading asterisk
.*       Generate indirect reference
         LI    &Reg,&XI            Call myself recursively
         L     &Reg,0(,&Reg)
         MEnd
```

Indirect Addressing via Recursion ...

- Examples of code generated by calls to LI macro:

```
          LI    3,0(4)          Load from 0(4)
+         L     3,0(4)

          LI    3,*0(,4)        Load from what 0(,4) points to
+         L     3,0(,4)
+         L     3,0(,3)

          LI    3,**0(,7)       Two levels of indirection
+         L     3,0(,7)
+         L     3,0(,3)
+         L     3,0(,3)

          LI    3,***X          Three levels of indirection
+         L     3,X
+         L     3,0(,3)
+         L     3,0(,3)
+         L     3,0(,3)
```

Case Study 7: Bit-Handling Operations

- We solve a basic problem: addressing individual bits by name
  - Investigate safe bit-manipulation techniques
- Create a “mini-language” for bit-manipulation operations
- Basic forms: macros to
  - Allocate storage to named bits
  - Set bits on and off, and invert their values
  - Test bit values and branch if on or off
- Enhanced forms: macros to
  - Ensure bit names were properly declared
  - Bit names can't be referenced “accidentally”
  - Generate highly optimized code for bit manipulation and testing

Basic Bit Declaration and Manipulation Techniques

- Frequently need to set, test, manipulate “bit flags”:

```
Flag1    DS    X                Define 1st byte of bit flags
BitA     Equ   X'01'            Define a bit flag

Flag2    DS    X                Define 2nd byte of bit flags
BitB     Equ   X'10'            Define a bit flag
```

- Serious defect: *no correlation between bit name and byte name!*

```
         OI    Flag1,BitB       Set Bit B ON  ??
         NI    Flag2,255-BitA   Set Bit A OFF ??
```

- A simpler technique: define a length attribute
  - Then use just one name for all references
  - Advantage: less chance to confuse or misuse bit names and byte names!

Simple Bit-Declaring Macro: Design Considerations

- Several ways to generate bit-name definitions
  1. Allocate storage byte and bit name together:

```
Flag1  DC   X'0',X'80'        Byte with bit value = length attribute
```

  2. Allocate *unnamed* storage byte first, define bits following:

```
       DC   X'0'              Unnamed byte
Bit_A  Equ  *-1,X'80'         Bit_A defined as bit 0
```

  3. Define bits first, allocate *unnamed* storage byte following:

```
Bit_B  DS   0XL(X'40')        Bit_B defined as bit 1
       DC   X'0'              Unnamed byte
```

- Length Attribute used for named bits and *unnamed* bytes

```
TM   Bit_Name,L'Bit_Name Refer to byte and bit using bit name

TM    Flag1,L'Flag1      Test setting of Flag1 bit
NI    BitA,255-L'BitA    Set BitA OFF (uses name 'BitA' only)
OI    BitB,L'BitB        Set BitB ON  (uses name 'BitB' only)
```

Simple Bit-Declaring Macro: Pseudo-Code

- Generates a bit-name EQUate for each argument, allocates storage
- Syntax: `SBitDef  bitname[,bitname]...`
- Examples:

```
SBitDef  b1,b2,b3,b4,b5,b6,b7,b8    Eight bits in one byte

SBitDef  c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v   Many bits+bytes
```

- Pseudo-code:

```
Set Lengths to bit-position weights (128,64,32,16,8,4,2,1)

DO for M = 1 to Number_of_Arguments
   IF (Mod(M,8)=1) GEN (  DC  B'0' )     (Generate unnamed byte)
   GEN (Arg(M)  EQU  *-1,Lengths(Mod(M-1,8)+1) )  (Define bit name)
```

Simple Bit-Declaring Macro: SBITDEF

```
        Macro ,                         Error checking omitted
        SBitDef ,                       No declared parameters
&L(1)   SetA  128,64,32,16,8,4,2,1      Define bit position values
&NN     SetA  N'&SysList                Number of bit names provided
&M      SetA  1                         Name counter
.NB     Aif   (&M gt &NN).Done          Check if names exhausted
&C      SetA  1                         Start new byte at leftmost bit
        DC    B'0'                      Allocate a bit-flag byte
.NewN   ANop  ,                         Get a new bit name
&B      SetC  '&SysList(&M)'            Get M-th name from argument list
&B      Equ   *-1,&L(&C)                Define bit via length attribute
&M      SetA  &M+1                      Step to next name
        Aif   (&M gt &NN).Done          Exit if names exhausted
&C      SetA  &C+1                      Count bits in a byte
        Aif   (&C le 8).NewN            Get new name if byte not full
        Ago   .NB                       Byte is filled, start a new byte
.Done   MEnd

        SBitDef  b1,b2                  Define bits b1, b2
+       DC    B'0'                      Allocate a bit-flag byte
+b1     Equ   *-1,128                   Define bit via length attribute
+b2     Equ   *-1,64                    Define bit via length attribute
```

Simple Bit-Manipulation Macros: Pseudo-Code

- Operations on “named” bits
- Setting bits on: one OI instruction per named bit

```
IF (Label ≠ null) GEN (Label  DC  0H'0')

DO for M = 1 to Number_of_Arguments
   GEN ( OI  Arg(M),L'Arg(M) )          to set bits on
```

- Length Attribute reference specifies the bit
  - As illustrated in the simple bit-defining macro
- Similar macros for setting bits off, or inverting bits

```
IF (Label ≠ null) GEN (Label  DC  0H'0')
   GEN ( NI  Arg(M),255-L'Arg(M) )      to set bits off

   GEN ( XI  Arg(M),L'Arg(M) )          to invert bits
```

- Warning: these simple macros are very trusting!
  - Any name can be used (see slide Tech-139)

Simple Bit-Handling Macros: Setting Bits ON

- Macro `SBitOn` to set one or more bits ON
  - Generates one OI instruction per bitname
- Syntax: `SBitOn bitname[,bitname]...`

```
        Macro ,                         Error Checking omitted
&Lab    SBitOn
&NN     SetA  N'&SysList                Number of Names
&M      SetA  1
        Aif   ('&Lab' eq '').Next       Skip if no name field
&Lab    DC    OH'0'                     Define label
.Next   ANop  ,                         Get a bit name
&B      SetC  '&SysList(&M)'            Extract name (&M-th positional argument)
.Go     OI    &B,L'&B                   Set bit on
&M      SetA  &M+1                      Step to next bit name
        Aif   (&M le &NN).Next          Go get another name
        MEnd
```

Simple Bit-Handling Macros: Setting Bits ON ...

- Examples:

```
 AA1    SBitOn  b1,b3,b8,c1,c2
+AA1    DC    OH'0'                  Define label
+       OI    b1,L'b1                Set bit on
+       OI    b3,L'b3                Set bit on
+       OI    b8,L'b8                Set bit on
+       OI    c1,L'c1                Set bit on
+       OI    c2,L'c2                Set bit on

        SBitOn  b1,b8
+       OI    b1,L'b1                Set bit on
+       OI    b8,L'b8                Set bit on
```

- Observe: one **OI** instruction per bit!
  - We will consider optimizations later

Simple Bit-Handling Macros: Set OFF and Invert Bits

- Macros SBitOff and SBitInv are defined like SBitOn:
  - SBitOff uses **NI** to set bits off

```
        Macro
&Lab    SBitOff
.*      - - -  etc., as for SBitOn
.Go     NI    &B,255-L'&B           Set bit off
.*      - - -  etc.
        MEnd
```

  - SBitInv uses **XI** to invert bits

```
        Macro
&Lab    SBitInv
.*      - - -  etc., as for SBitOn
.Go     XI    &B,L'&B               Invert bit
.*      - - -  etc.
        MEnd
```

Simple Bit-Handling Macros: Set OFF and Invert Bits ...

- Examples:

```
 bb1    SBitOff  b1,b3,b8,c1,c2
+bb1    DC    OH'0'                  Define label
+       NI    b1,255-L'b1            Set bit off
+       NI    b3,255-L'b3            Set bit off
+       NI    b8,255-L'b8            Set bit off
+       NI    c1,255-L'c1            Set bit off
+       NI    c2,255-L'c2            Set bit off

 cc1    SBitInv  b1,b3,b8,c1,c2
+cc1    DC    OH'0'                  Define label
+       XI    b1,L'b1                Invert bit
+       XI    b3,L'b3                Invert bit
+       XI    b8,L'b8                Invert bit
+       XI    c1,L'c1                Invert bit
+       XI    c2,L'c2                Invert bit
```

- One **NI** or **XI** instruction per bit

Simple Bit-Handling Macros: Branch on Bit Values

- Simple bit-testing macros: branch to target if bitname is on/off
- Syntax: `SBBitxxx bitname,target` where xxx = ON or OFF

```
        Macro
 &Lab   SBBitOn &B,&T              Bitname and branch label
 &Lab   TM    &B,L'&B              Test specified bit
        BO    &T                   Branch if ON
        MEnd

        Macro
 &Lab   SBBitOff &B,&T             Bitname and branch label
 &Lab   TM    &B,L'&B              Test specified bit
        BNO   &T                   Branch if OFF
        MEnd

 *      Examples
 dd1    SBBitOn  b1,aa1
+dd1    TM    b1,L'b1              Test specified bit
+       BO    aa1                  Branch if ON
        SBBitOff b2,bb1
+       TM    b2,L'b2              Test specified bit
+       BNO   bb1                  Branch if OFF
```

Bit-Handling Macros: Enhancements

- The previous macros work, and will be enhanced in two ways:

  1. Ensure that “bit names” do name bits! The simple macros don't:

```
X       DC     F'23'          Define a constant
Flag    Equ    X'08'          Define a flag bit (?) 'somewhere'
        SBitOn Flag,X         Set bits ON 'somewhere' ???
```

  2. Handle bits within one byte with one instruction (code optimization!)

- More enhancements are possible (but not illustrated here):
  - Pack all bits (storage optimization), but may not gain much
  - “Hide” declared bit names so they don't appear as ordinary symbols!
  - Provide a “run-time symbol table” for debugging
    - ADATA instruction can put info into SYSADATA file
    - Create separate CSECT with names, locations, bit values

Bit-Handling “Micro-Compiler”

- Goal: a “Micro-compiler” for bit operations
  - Micro: Limit scope of actions to specific data types and operations
  - Compiler: Syntax/semantic scans, code generation
    - Each macro checks syntax of definitions and uses
    - Build symbol tables using created global variable symbols
- Bit Language: same as for the simple bit-handling macros:
  - Data type: named bits
  - Operations: define; set on/off, invert; test-and-branch
- Can incrementally add to and improve each language element
  - As these enhancements illustrate

General Bit-Handling Macros: Data Structures

- Declaring a `bitname` requires three “global” items:
  1. A **Byte_Number** to count bytes in which bits are declared
  2. A **BitCount** for the next unallocated bit in the current byte
  3. An *associatively addressed* Symbol Table
     - Each declared bit name creates a global arithmetic variable
     - Its name &(BitDef_bitname_ByteNo) is constructed from
       - a prefix **BitDef_** (whatever you like, to avoid global-name collisions)
       - the declared `bitname` (the “associative” feature)
       - a suffix **_ByteNo** (whatever you like, to avoid global-name collisions)
     - Its value is the **Byte_Number** in which this bit was allocated
- Remember: the storage bytes themselves are unnamed!

General Bit-Declaring Macro: Design

- Bits may be “packed”; sublisted names are kept in one byte
- Example: `BitDef a,(b,c),d` keeps b and c together
- High-level pseudo-code:

```
DO for all arguments
IF argument is not a sublist
   THEN assign the named bit to a byte (start another byte if needed)
   ELSE IF sublist has more than 8 items, ERROR STOP, can't assign
        ELSE if not enough room in current byte, start another
        Assign sublist bit names to a byte
```

General Bit-Declaring Macro: Pseudo-Code

```
Set Lengths = 128,64,32,16,8,4,2,1  (Bit values, indexed by Bit_Count)
DO for M = 1 to Number_of_Arguments
   Set B = Arg_List(M)
   IF (Substr(B,1,1) ≠ '(') PERFORM SetBit(B)  (not a sublist)
      ELSE (Handle sublist)

      IF (N_SubList_Items > 8) ERROR Sublist too long
      IF (BitCount+N_Sublist_Items > 8) PERFORM NewByte
      DO for CS = 1 to N_Sublist_Items (Handle sublist)
         PERFORM SetBit(Arg_List(M,CS))

SetBit(B): (Save bit name and Byte_Number in which the bit resides:)
   IF (Mod(BitCount,8) = 0) PERFORM NewByte
   Declare created global variable &(BitDef_&B._Byte_Number)
   Set created variable (Symbol Table entry) to Byte_Number
   GEN (B  EQU  *-1,Lengths(BitCount) )
   Set BitCount = BitCount+1 (Step to next bit in this byte)

NewByte: GEN( DC  B'0' ); Increment Byte_Number; BitCount = 1
```

- Created symbol contains bit name; its value is the byte number

General Bit-Handling Macros: Bit Declaration

```
        Macro ,                        Some error checks omitted
        BitDef
        GblA  &BitDef_ByteNo           Used to count defined bytes
&L(1)   SetA  128,64,32,16,8,4,2,1     Define bit position values
&NN     SetA  N'&SysList               Number of bit names provided
&M      SetA  1                        Name counter
.NB     Aif   (&M gt &NN).Done         Check if names exhausted
&C      SetA  1                        Start new byte at leftmost bit
        DC    B'0'                     Define a bit-flag byte
&BitDef_ByteNo SetA &BitDef_ByteNo+1  Increment byte number
.NewN   ANop  ,                        Get a new bit name
&B      SetC  '&SysList(&M)'           Get M-th name from argument list
        Aif   ('&B'(1,1) ne '(').NoL   Branch if not a sublist
&NS     SetA  N'&SysList(&M)           Number of sublist elements
&CS     SetA  1                        Initialize count of sublist items
        Aif   (&C+&NS le 9).SubT       Skip if room left in current byte
&C      SetA  1                        Start a new byte
        DC    B'0'                     Define a bit-flag byte
&BitDef_ByteNo SetA &BitDef_ByteNo+1  Increment byte number
.*      - - -     (continued)
```

General Bit-Handling Macros: Bit Declaration ...

```
.*      – – –     (continuation)    Name is in a sublist
.SubT   ANop  ,                     Generate sublist equates
&B      SetC  '&SysList(&M,&CS)'    Extract sublist element
        GblA  &(BitDef_&B._ByteNo)  Created var sym with ByteNo for this bit
&B      Equ   *–1,&L(&C)            Define bit via length attribute
&(BitDef_&B._ByteNo) SetA &BitDef_ByteNo  Byte no. for this bit
&CS     SetA  &CS+1                 Step to next sublist item
        Aif   (&CS gt &NS).NewA     Skip if end of sublist
&C      SetA  &C+1                  Count bits in a byte
        Ago   .SubT                 And go do more list elements
.NoL    ANop  ,                     Not a sublist
        GblA  &(BitDef_&B._ByteNo)  Declare byte number for this bit
&B      Equ   *–1,&L(&C)            Define bit via length attribute
&(BitDef_&B._ByteNo) SetA &BitDef_ByteNo  Byte no. for this bit
.NewA   ANop  ,                     Ready for next argument
&M      SetA  &M+1                  Step to next name
        Aif   (&M gt &NN).Done      Exit if names exhausted
&C      SetA  &C+1                  Count bits in a byte
        Aif   (&C le 8).NewN        Get new name if not done
        Ago   .NB                   Bit filled, start a new byte
.Done   MEnd
```

Examples of Bit Declaration

- Example: Define ten bit names (with macro-generated code)

```
 a4      BitDef   d1,d2,d3,(d4,d5,d6,d7,d8,d9),d10    d4 starts new byte
+         DC     B'0'                      Define a bit-flag byte
+d1       Equ    *-1,128                   Define bit via length attribute
+d2       Equ    *-1,64                    Define bit via length attribute
+d3       Equ    *-1,32                    Define bit via length attribute
+         DC     B'0'                      Define a bit-flag byte
+d4       Equ    *-1,128                   Define bit via length attribute
+d5       Equ    *-1,64                    Define bit via length attribute
+d6       Equ    *-1,32                    Define bit via length attribute
+d7       Equ    *-1,16                    Define bit via length attribute
+d8       Equ    *-1,8                     Define bit via length attribute
+d9       Equ    *-1,4                     Define bit via length attribute
+d10      Equ    *-1,2                     Define bit via length attribute
```

- Bits named d4-d9 are allocated in a single byte
  - Causes some bits to remain unused in the first byte

General Bit-Setting Macro: Data Structures

Two “phases” used to generate bit-operation instructions:

1. Check that bit names are declared (“strong typing”), and collect information about bits to be set:
   a. Number of distinct Byte_Numbers (which bytes “own” the bit names?)
   b. For each byte, the number of instances of bit names in that byte
   c. An associatively addressed variable-symbol “name table”
      - Name prefix is **BitDef_Nm_** (whatever, to avoid global-name collisions)
      - Suffix is a “double subscript,” **&ByteNumber._&InstanceNumber**
      - Value of the symbol is the bit name
2. Use the information to generate optimal instructions
   - Names and number of name instances needed to build operands

General Bit-Setting Macro: Design

- Optimize generated code for setting bits on
- Syntax: `BitOn bitname[,bitname]...`
  Example: `BitOn a,b,c,d`
- High-level pseudo-code:

```
DO for all arguments (Pass 1)

   Verify that the argument bit name was declared (check global symbol)
   IF not declared, STOP with error message for undeclared bit name

   Save argument bit names and their associated byte numbers

DO for all saved distinct byte numbers (Pass 2)

   GEN Instructions to handle argument bits belonging to each byte
```

- Pass 1 captures bit names & byte numbers, pass 2 generates code

General Bit-Setting Macro: Pseudo-Code

```
Save macro-call label
Set NBN (Number of known Byte Numbers) = 0
DO for M = 1 to Number_of_Arguments [phase 1]
   Set B = Arg(M)
   Declare created global variable &(BitDef_&B._Byte_Number)
   IF (Its value is zero) ERROR EXIT 'Undeclared Bitname &B'  <-- Key
   DO for K = 1 to NBN (Check byte number from the global variable)
      IF (This Byte Number is known) Increment its count
      ELSE Increment NBN (this Byte Number is new: set its count = 1)
   Save B in bitname list for this Byte Number

(End Arg scan: have all byte numbers and their associated bit names)

DO for M = 1 to number of distinct Byte Numbers [phase 2]
   Set Operand = 'First_Bitname,L''First_Bitname'  (local character string)
   DO for K = 2 to Number of bitnames in this Byte
      Operand = Operand || '+L''Bitname(K)'
   GEN (label  OI  Operand  ); set label = ''
```

- Easy generalization to BitOff (NI) and BitInv (XI) macros

General Bit-Setting Macros: Set Bits ON

- `BitOn` optimizes generated instructions (most error checks omitted)

```
         Macro
&Lab     BitOn
&L       SetC  '&Lab'                    Save label
&NBN     SetA  0                         No. of distinct Byte Nos.
&M       SetA  0                         Name counter
&NN      SetA  N'&SysList                Number of names provided
.NmLp    Aif   (&M ge &NN).Pass2         Check if all names scanned
&M       SetA  &M+1                      Step to next name
&B       SetC  '&SysList(&M)'            Pick off a name
         Aif   ('&B' eq '').Null         Check for null item
         GblA  &(BitDef_&B._ByteNo)      Declare GBLA for Byte No.
         Aif   (&(BitDef_&B._ByteNo) eq 0).UnDef  Exit if undefined
&K       SetA  0                         Loop through known Byte Nos
.BNLp    Aif   (&K ge &NBN).NewBN        Not in list, a new Byte No
&K       SetA  &K+1                      Search next known Byte No
         Aif   (&BN(&K) ne &(BitDef_&B._ByteNo)).BNLp  Check match
.*       ---   continued
```

General Bit-Setting Macros: Set Bits ON ...

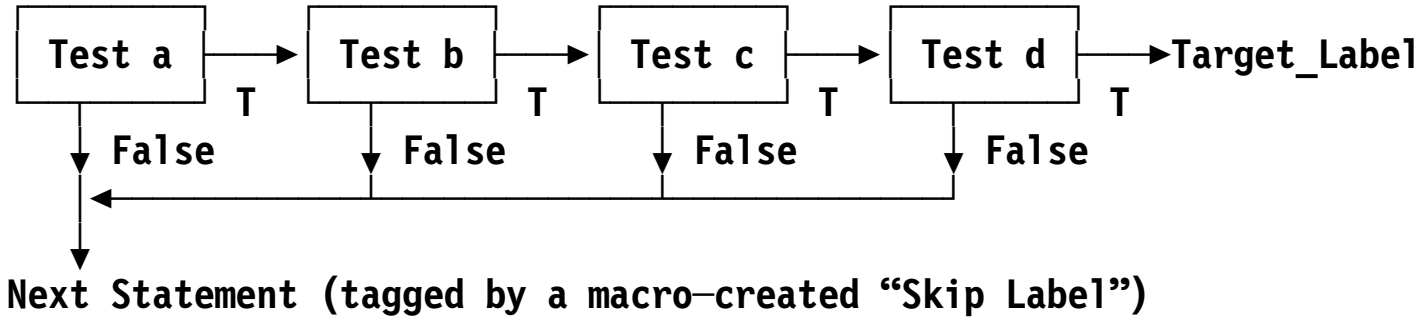
```
.*      - - -     (continuation)
&J       SetA  1                        Check if name already specified
.CkDup   Aif   (&J gt &IBN(&K)).NmOK    Branch if name is unique
         Aif   ('&B' eq '&(BitDef_Nm_&BN(&K)._&J)').DupNm  Duplicated
&J       SetA  &J+1                     Search next name in this byte
         Ago   .CkDup                   Check further for duplicates
.DupNm   MNote 8,'BitOn: Name ''&B'' duplicated in operand list'
         MExit
.NmOK    ANop  ,                        No match, enter name in list
&IBN(&K) SetA  &IBN(&K)+1  Matching BN, bump count of bits in this byte
         LclC  &(BitDef_Nm_&BN(&K)._&IBN(&K))    Slot for bit name
&(BitDef_Nm_&BN(&K)._&IBN(&K)) SetC '&B'  Save K'th Bit Name, this byte
         Ago   .NMLp                    Go get next name
.NewBN   ANop  ,                        New Byte No
&NBN     SetA  &NBN+1                   Increment Byte No count
&BN(&NBN) SetA &(BitDef_&B._ByteNo)     Save new Byte No
&IBN(&NBN) SetA 1                       Set count of this Byte No to 1
         LclC  &(BitDef_Nm_&BN(&NBN)._1) Slot for first bit name
&(BitDef_Nm_&BN(&NBN)._1) SetC '&B'     Save 1st Bit Name, this byte
         Ago   .NMLp                    Go get next name
.*       - - -   continued
```

General Bit-Setting Macros: Set Bits ON ...

```
.*       - - -     (continuation)
.Pass2   ANop  ,                          Pass 2: scan Byte No list
&M       SetA  0                          Byte No counter
.BLp     Aif   (&M ge &NBN).Done          Check if all Byte Nos done
&M       SetA  &M+1                       Increment outer-loop counter
&X       SetA  &BN(&M)                    Get M-th Byte No
&K       SetA  1                          Set up inner loop
&Op      SetC  '&(BitDef_Nm_&X._&K).,L''&(BitDef_Nm_&X._&K)' 1st operand
.OpLp    Aif   (&K ge &IBN(&M)).GenOI     Operand loop, check for done
&K       SetA  &K+1                       Step to next bit in this byte
&Op      SetC  '&Op.+L''&(BitDef_Nm_&X._&K)'  Add "+L'bitname" to operand
         Ago   .OpLp                      Loop (inner) for next operand
.GenOI   ANop  ,                          Generate instruction for Byte No
&L       OI    &Op                        Turn bits ON
&L       SetC  ''                         Nullify label string
         Ago   .BLp                       Loop (outer) for next Byte No
.UnDef   MNote 8,'BitOn: Name ''&B'' not defined by BitDef'
         MExit
.Null    MNote 8,'BitOn: Null argument at position &M.'
.Done    MEnd
```

General “Branch if Bits On” Macro: Design

- Function: branch to `target` if all named bits are on
- Syntax: `BBitOn (bitlist),target`
  Example: `BBitOn (a,b,c,d),Label`
- Optimize generated code using data created by `BitDef`
- If more than one byte is involved, need “skip-if-false” branches



- Need only one test instruction for multiple bits in a byte!

General “Branch if Bits On” Macro: Pseudo-Code

```
Save macro-call label; Set NBN (Number of known Byte Numbers) = 0
DO for M = 1 to Number_of_1st-Arg_Items [phase 1]
   Set B = Arg(M)
   Declare created global variable &(BitDef_&B._Byte_Number)
   IF (Its value is zero) ERROR EXIT, undeclared bitname
   DO for K = 1 to NBN (Check byte number from the global variable)
      IF (This Byte Number is known) Increment its count
      ELSE Increment NBN (this Byte Number is new: set its count = 1)
   Save B in bit name list for this Byte Number

(End Arg scan: have all byte numbers and their associated bit names)

Create Skip_Label (using &SYSNDX)
DO for M = 1 to NBN [phase 2]
   Set Operand = 'First_Bitname,L''First_Bitname'  (first operand)
   DO for K = 2 to Number of bitnames in this Byte
      Operand = Operand || '+L''Bitname(K)'
   IF (M < NBN) GEN (label TM Operand ; BNO Skip_Label); set label = ''
   ELSE         GEN (label TM Operand ; BO  Target_label)
IF (NBN > 1) GEN (Skip_Label DS 0H)
```

General Bit-Handling Macros: Branch if Bits On

- `BBitOn` macro optimizes generated instructions (most error checks omitted)
- Two “passes” over bit name list:
  1. Scan, check, and save names, determine byte numbers (as in `BitOn`)
  2. Generate optimized tests and branches;
     if multiple bytes, generate “skip” tests/branches and label

```
         Macro
&Lab     BBitOn &NL,&T                  Bit Name List, Branch Target
         Aif   (N'&SysList ne 2 or '&NL' eq '' or '&T' eq '').BadArg
&L       SetC  '&Lab'                   Save label
&NBN     SetA  0                        No. of distinct Byte Nos.
&M       SetA  0                        Name counter
&NN      SetA  N'&NL                    Number of names provided
.NmLp    Aif   (&M ge &NN).Pass2        Check if all names scanned
.*       - - -    (continued)
```

General Bit-Handling Macros: Branch if Bits On ...

```
.*       – – –    (continuation)
&M       SetA  &M+1                      Step to next name
&B       SetC  '&NL(&M)'                 Pick off a name
         GblA  &(BitDef_&B._ByteNo)      Declare GBLA with Byte No.
         Aif   (&(BitDef_&B._ByteNo) eq 0).UnDef  Exit if undefined
&K       SetA  0                         Loop through known Byte Nos
.BNLp    Aif   (&K ge &NBN).NewBN        Not in list, a new Byte No
&K       SetA  &K+1                      Search next known Byte No
         Aif   (&BN(&K) ne &(BitDef_&B._ByteNo)).BNLp  Check match
&J       SetA  1                         Check if name already specified
.CkDup   Aif   (&J gt &IBN(&K)).NmOK     Branch if name is unique
         Aif   ('&B' eq '&(BitDef_Nm_&BN(&K)._&J)').DupNm  Duplicated
&J       SetA  &J+1                      Search next name in this byte
         Ago   .CkDup                    Check further for duplicates
.DupNm   MNote 8,'BBitOn: Name ''&B'' duplicated in operand list'
         MExit
.NmOK    ANop  ,                         No match, enter name in list
&IBN(&K) SetA  &IBN(&K)+1                Have matching BN, count up by 1
         LclC  &(BitDef_Nm_&BN(&K)._&IBN(&K))    Slot for bit name
&(BitDef_Nm_&BN(&K)._&IBN(&K)) SetC '&B'  Save K'th Bit Name, this byte
         Ago   .NMLp                     Go get next name
.*       – – –    (continued)
```

General Bit-Handling Macros: Branch if Bits On ...

```
.*       – – –    (continuation)
.NewBN   ANop  ,                          New Byte No
&NBN     SetA  &NBN+1                     Increment Byte No count
&BN(&NBN) SetA &(BitDef_&B._ByteNo)       Save new Byte No
&IBN(&NBN) SetA 1                         Set count of this Byte No to 1
         LclC  &(BitDef_Nm_&BN(&NBN)._1) Slot for first bit name
&(BitDef_Nm_&BN(&NBN)._1) SetC '&B'       Save 1st Bit Name, this byte
         Ago   .NMLp                      Go get next name
.Pass2   ANop  ,                          Pass 2: scan Byte No list
&M       SetA  0                          Byte No counter
&Skip    SetC  'Off&SysNdx'               False–branch target
.BLp     Aif   (&M ge &NBN).Done          Check if all Byte Nos done
&M       SetA  &M+1                       Increment outer–loop counter
&X       SetA  &BN(&M)                    Get M–th Byte No
&K       SetA  1                          Set up inner loop
&Op      SetC  '&(BitDef_Nm_&X._&K).,L''&(BitDef_Nm_&X._&K)' Operand
.OpLp    Aif   (&K ge &IBN(&M)).GenBr     Operand loop, check for done
&K       SetA  &K+1                       Step to next bit in this byte
&Op      SetC  '&Op.+L''&(BitDef_Nm_&X._&K)'  Add next bit to operand
         Ago   .OpLp                      Loop (inner) for next operand
.*       – – –    (continued)
```

General Bit-Handling Macros: Branch if Bits On ...

```
.*       — — —    (continuation)
.GenBr   ANop  ,                      Generate instruction for Byte No
         Aif   (&M eq &NBN).Last      Check for last test
&L       TM    &Op                   Test if bits are ON
         BNO   &Skip                 Skip if not all ON
&L       SetC  ''                    Nullify label string
         Ago   .BLp                  Loop (outer) for next Byte No
.Last    ANop  ,                      Generate last test and branch
&L       TM    &Op                   Test if bits are ON
         BO    &T                    Branch if all ON
         Aif   (&NBN eq 1).Done      No skip target if just 1 byte
&Skip    DC    OH'0'                 Skip target
         MExit
.UnDef   MNote 8,'BBitOn: Name ''&B'' not defined by BitDef'
         MExit
.BadArg  MNote 8,'BBitOn: Improperly specified argument list'
.Done    MEnd
```

Case Study 8: Utilizing Assembler Data Types

- Overview of data typing
- Using base-language type attributes
  - Case Study 8a: use operand type attribute to generate correct literal types
- Shortcomings of assembler-assigned type attributes
  - Case Study 8b: create macros to check conformance of instructions and operand types
  - Extension: instruction vs. operand vs. register consistency checking
- User-assigned (and assembler-maintained) data types
  - Case Study 8c: declare user data types and “operators” on them

Using and Defining Assembler Data Types

- We're familiar with type sensitivity in higher-level languages:
  - Instructions generated from a statement depend on data types:

```
A = B + C ;        '=' and '+' are polymorphic operators
```

  - **A, B, C** might be integer, float, complex, Boolean, string, ...
- Most named assembler objects have a type attribute
  - Usually assigned by the assembler
  - Can exploit type attribute references for type-sensitive code sequences and for operand validity checking
- Extensions to the assembler's “base language” types are possible:
  - Assign our own type attributes (avoiding conflicts with Assembler's)
  - Utilize created variable symbols to retain type information

Base-Language Type Sensitivity: Simple Polymorphism

- Intent: `INCR` macro increments **var** by a constant **amt** (or 1)

  Syntax: `INCR var[,amt]` (default amt=1)
- Usage examples:

```
Day     DS    H             Type H: Day of the week
Rate    DS    F             Type F: Rate of speed
MyPay   DS    PL6           Type P: My salary
Dist    DS    D             Type D: A distance
Wt      DS    E             Type E: A weight
WXY     DS    X             Type X: Type not valid for INCR macro
*
CC      Incr  Day                   Add 1 to Day
DD      Incr  Rate,-3,Reg=15        Decrease rate by 3
        Incr  MyPay,150.50          Add 150.50 to my salary
JJ      Incr  Dist,-3.16227766      Decrease distance by sqrt(10)
KK      Incr  Wt,-2E4,Reg=6         Decrement weight by 10 tons
        Incr  WXY,2                 Test with unsupported type
```

- Use **var**'s assembler type attribute to create compatible literals
  - type of **amt** guaranteed to match type of **var**

Base-Language Type Sensitivity: Simple Polymorphism ...

- Supported types are numeric: H, F, E, D, P

```
        Macro ,                     Increment &V by amount &A (default 1)
&Lab    INCR  &V,&A,&Reg=0          Default work register = 0
&T      SetC  T'&V                  Type attribute of 1st arg
&Op     SetC  '&T'                  Save type of &V for mnemonic suffix
&I      SetC  '1'                   Default increment
        Aif   ('&A' eq '').IncOK    Increment now set OK
&I      SetC  '&A'                  Supplied increment (N.B. Not SETA!)
.IncOK Aif    ('&T' eq 'F').F,('&T' eq 'P').P,  (check base language types)  X
                ('&T' eq 'H' or '&T' eq 'D' or '&T' eq 'E').T   Valid types
        MNote 8,'INCR: Cannot use type ''&T'' of ''&V''.'
        MExit
.F      ANOP  ,                     Type of &V is F
&Op     SetC  ''                    Null opcode suffix for F (no LF opcode)
.T      ANOP  ,                     Register-types D, E, H (and F)
&Lab    L&Op  &Reg,&V               Fetch variable to be incremented
        A&Op  &Reg,=&T.'&I'         Add requested increment as typed literal
        ST&Op &Reg,&V               Store incremented value
        MExit
.P      ANOP  ,                     Type of &V is P
&Lab    AP    &V,=P'&I'             Incr packed variable with P-type literal
        MEnd
```

Base-Language Type Sensitivity: Generated Code

- Code generated by INCR macro (see slide Tech-161)

```
 CC      Incr  Day                  Add 1 to Day
+CC      LH    0,Day                Fetch variable to be increment
+        AH    0,=H'1'              Add requested increment
+        STH   0,Day                Store incremented value
 DD      Incr  Rate,-3,Reg=15       Decrease rate by 3
+DD      L     15,Rate              Fetch variable to be increment
+        A     15,=F'-3'            Add requested increment
+        ST    15,Rate              Store incremented value
         Incr  MyPay,150.50         Add 150.50 to my salary
+        AP    MyPay,=P'150.50'     Increment variable
 JJ      Incr  Dist,-3.16227766     Decrease distance by SQRT(10)
+JJ      LD    0,Dist               Fetch variable to be increment
+        AD    0,=D'-3.16227766'    Add requested increment
+        STD   0,Dist               Store incremented value
 KK      Incr  Wt,-2E4,Reg=6        Decrement weight by 10 tons
+KK      LE    6,Wt                 Fetch variable to be increment
+        AE    6,=E'-2E4'           Add requested increment
+        STE   6,Wt                 Store incremented value

         Incr  WXY,2                Test with unsupported type
+ *** MNOTE ***  8,INCR: Cannot use type 'X' of 'WXY'.
```

Shortcomings of Assembler-Assigned Types

- Suppose **amt** is a variable, not a constant...
  - Need an ADD2 macro: syntax like `ADD2 var,amt`
- What if the assembler types of **var** and **amt** don't conform?
  - Mismatch? Might data type conversions be required? How will we know?

```
Rate   DS    F                  Rate of speed
MyPay  DS    PL6                My salary
       ADD2  MyPay,Rate         Add (binary) Rate to (packed) MyPay ??
```

- Assembler data types know nothing about “meaning” of variables, only their hardware representation; so, typing is very weak!

```
Day    DS    H                  Day of the week
Rate   DS    F                  Rate of speed
Dist   DS    D                  A distance
Wt     DS    E                  A weight

*      Following assembler types conform (but not their logical types)!
*      Numeric consistency, logical inconsistency

       ADD2  Rate,Day           Add binary Day to Rate (??)
       ADD2  Dist,Wt            Add floating Distance to Weight (??)
```

Symbol Attributes and Lookahead Mode

- Attributes entered in the symbol table when symbol is defined
- Attribute references are resolved during conditional assembly by
  1. Finding them in the symbol table, or
  2. Forward-scanning source file (“Lookahead Mode”) for symbol's definition
     - No macro definition/generation, no substitution, no AGO/AIF
     - Symbol attributes *may change* during final assembly
     - Scanned records are saved (SYSIN is read only once!)
- Symbols generated by macros can't be found in Lookahead Mode
  - Unknown or partially-defined symbols assigned type attribute `'U'`
- Symbol attributes needed for desired conditional assembly results must be defined before they are referenced
- Use LOCTR instruction to “group” code and data separately
  - Data declarations can precede code in source, but follow it in storage

Case Study 8b: Simple Instruction-Operand Type Checking

- Check the second operand of the A instruction
  - Accept type attributes type F, A, or Q; note others and proceed
- First, save the assembler's definition of instruction “A”

```
My_A     OpSyn  A          Save definition of A as My_A
A        OpSyn  ,          Nullify assembler's definition of A
```

- Define a macro named “A” that eventually calls My_A
- Macro “A” checks the second operand for type F, A, or Q

```
         Macro
&L       A      &R,&X
         AIF    (T'&X eq 'F' or T'&X eq 'A' or T'&X eq 'Q').OK
         MNote  1,'Note! Second operand type not F, A, or Q.'
.OK      ANop
&L       My_A   &R,&X
         MEnd
```

- Allowed types are “hard coded” in this macro

Base-Language Type Sensitivity: General Type Checking

- Intent: compatibility checking between instruction and operand types
- Define `TypeChek` macro to request type checking

  Syntax: `TypeChek mnemonic,valid_types`
- Call `TypeChek` with mnemonic to check and its allowed types

```
TypeChek  L,'ADFQVX'    Allowed types: AQV (adcons), D, F, X
```

- Sketch of macro to initiate type checking for one mnemonic:

```
        Macro
        TypeChek  &Op,&Valid    Mnemonic, set of valid types
        GblC  &(TypeCheck_&Op._Valid),&(TypeCheck_&Op)
&(TypeCheck_&Op._Valid)  SetC  '&Valid'   Save valid types for this opcode
TypeCheck_&Op.           OpSyn &Op.    Save original opcode definition
&Op     OpSyn ,                  Disable previous definition of &Op
.*      MNote *,'Mnemonic ''&Op.'' valid types are ''&(TypeCheck_&Op._Valid).''.'
        MEnd
```

- Can be generalized to multiple mnemonics
  - But: requires creating macros for each mnemonic...

Base-Language Type Sensitivity: General Type Checking ...

- First, install L macro in the macro library:

```
        Macro
&Lab    L     &Reg,&Operand
        GblC  &(TypeCheck_L_Valid) List of valid types for L
&TypOp  SetC  T'&Operand          Type attribute of &Operand
&Test   SetA  Find('&(TypeCheck_L_Valid)','&TypOp')  Check validity
        AIf   (&Test ne 0).OK        Skip if valid
        MNote 1,'Possible type incompatibility between L and ''&Operand.''?'
.OK     ANop                         Now, do the original L instruction
&Lab    TypeCheck_L  &Reg,&Operand
.*      TypeCheck_L was OPSYN'd to L by the TypeChek macro
        MEnd
```

- Now, use L "instruction" as usual:

```
000084                5 A       DS    F           A has type attribute F
000088                6 B       DS    H           B has type attribute H
                                _ _ _
0001E4 5810F084      23         L     1,A         Load from fullword
0001E8 5820F088      24         L     2,B         Load from halfword
         *** MNOTE *** + 1,Possible type incompatibility between L and 'B'?
```

- Inconvenient: we must write a macro for each checked mnemonic

Base-Language Type Checking: Extensions

- Previous technique requires a macro for each checked instruction
  - Not difficult to write, just a lot of repetitive work
  - Macros must be available in a library
    - If not using **TypeChek**, don't use the instruction-replacement macros!
- Better:
  - Specify a list of instructions to be checked, such as

```
TypeChek  (L,ST,A,AL,S,SL,N,X,O),'ADFQVX'
```

  - The **TypeChek** macro generates the replacement macros as needed
    - Using AINSERT!

The AINSERT Statement

- AINSERT allows generation of fully parameterized records

```
AINSERT 'string',[FRONT|BACK]
```

- Placed at front or back of assembler's internal buffer queue
  - String padded or truncated to form 80-byte record
- HLASM reads from the FRONT of the buffer before reading from SYSIN
  - Input from SYSIN resumes when the buffer is empty
- Operand string may contain “almost anything”

```
          AInsert  '* comment about &SysAsm. &SysVer.',BACK
>* comment about HIGH LEVEL ASSEMBLER 9.7.0
```

  - The '>' character in “column 0” indicates an AINSERTed statement
- Use AINSERT to generate tailored, parameterized macro definitions

Base-Language Type Checking: Generated Macros

- Generate each type-checking macro using AINSERT

```
TypeChek  (L,ST,A,AL,S,SL,N,X,O),'ADFQVX'   Desired style
```

- Sketch of revised inner loop of TypeChek macro:

```
&Op    SetC  '&Ops(&K)'          Pick off K-th mnemonic
&Op    OpSyn ,                   Disable previous definition of &Op
.*   Generate macro to redefine &Op for type checking (note paired ', &)
 AInsert ' Macro ',BACK
 AInsert '&&Lab &Op. &&Reg,&&Opd',BACK
 AInsert ' GblC &&(TypeCheck_&Op._Valid)',BACK
 AInsert '&&TO SetC T''&&Opd ',BACK
 AInsert '&&T SetA Find(''&&(TypeCheck_&Op._Valid)'',''&&TO'')',BACK
 AInsert ' AIf (&&T ne 0).OK ',BACK
 AInsert ' MNote 1,''Possible type conflict between &Op and &&Opd?''',B*
               ACK
 AInsert '.OK ANop ',BACK
 AInsert '&&Lab TypeCheck_&Op &&Reg,&&Opd ',BACK  Assumes previous OPSYN
 AInsert ' MEnd ',BACK
.*    End of macro generation
```

- Compare to “hand-coded” L macro (slide Tech-168)

User-Assigned Assembler Type Attributes

- We can utilize third operand of EQU statement for type assignment:

```
symbol  EQU   expression,length,type
```

  - Assembler's “native” types are upper case letters, $, and '@'
  - We can use lower case letters for user-assigned types

- Example (extend the REGS macro, slide Tech-104) to create a TYPEREGS macro:

```
GR&N    EQU   &N,,C'g'      Assign value and type attribute 'g' for GPR
FR&N    EQU   &N,,C'f'      Assign value and type attribute 'f' for FPR
```

- GRnn symbols have type attribute **'g'**, FRnn have **'f'**
- Can use type attribute to check symbols used in register operands

Instruction-Operand-Register Type Checking

- Intent: check “typed” register names in type-checking macros
- Example: extend L macro (see slides Tech-167 and Tech-168)

```
        Macro
&Lab    L     &Reg,&Operand
        GblC  &(TypeCheck_L_Valid),&(TypeCheck_L_RegType)
&TypOp SetC   T'&Operand          Type attribute of &Operand
&Test   SetA  Find('&(TypeCheck_L_Valid)','&TypOp')  Check validity
        AIf   (&Test ne 0).OK_Op      Skip if valid
   MNote 1,'Possible type incompatibility between L and ''&Operand.''?'
.OK_Op ANop                          Now, do the original L instruction
.*      Added checking for register type:
&TypRg SetC   T'&Reg              Type attribute of &Reg
&Test   SetA  Find('&(TypeCheck_L_RegType)','&TypRg')  Check validity
        AIf   (&Test ne 0).OKReg     Skip if valid
   MNote 1,'Possible register incompatibility between L and ''&Reg.''?'
.OKReg ANop                          Now, do the original L instruction
&Lab    TypeCheck_L  &Reg,&Operand
        MEnd
```

- Typical output (assuming F names a floating-point constant):

```
          L     FR4,F
*** MNOTE *** 1,Possible type incompatibility between L and 'F'?
*** MNOTE *** 1,Possible register incompatibility between L and 'FR4'?
```

Case Study 8c: Encapsulated Abstract Data Types

- Intent: declare two user types, and define operations on them
- Types: Date and Duration (or Interval) between 2 Dates
  - Unfortunately, both Date and Duration start with D
  - So, we'll use “Interval” as the safer (if less intuitive) term for “duration”
    - A measure of elapsed time, in days
  - We will use lower case letters **'d'** and **'i'** for our types!
- DCLDATE and DCLNTVL macros declare variables (abstract data types):

```
DCLDATE  Birth,Graduation,Marry,Hire,Retire,Expire

DCLNTVL  Training,Employment,Retirement,LoanPeriod
```

User-Assigned Type Attributes: DCLDATE Macro

- Declaration of DATE types made by `DclDate` macro

```
          Macro ,                          Args = list of names
          DCLDATE   &Len=4                 Default data length = 4
          GblC  &DateTyp                   Type attr of Date variable
 &DateTyp SetC  'd'                        User type attr is lower case 'd'
 .*       Length of a DATE type could also be a global variable
 &NV      SetA  N'&SysList                 Number of arguments to declare
 &K       SetA  0                          Counter
 .Test    Aif   (&K ge &NV).Done           Check for finished
 &K       SetA  &K+1                       Increment argument counter
          DC    PL&Len.'0'                 Define storage as packed decimal
 &SysList(&K) Equ *-&Len.,&Len.,C'&DateTyp'  Define name, length, type
          Ago   .Test
 .Done    MEnd

          DclDate  LoanStart,LoanEnd    Declare 2 date fields
+         DC  PL4'0'                   Define storage as packed decimal
+LoanStart  Equ   *-4,4,C'd'           Define name, length, type
+         DC  PL4'0'                   Define storage as packed decimal
+LoanEnd    Equ   *-4,4,C'd'           Define name, length, type
```

User-Assigned Type Attributes: DCLNTVL Macro

- Declaration of INTERVAL types made by `DclNtvl` macro
  - Initial value can be specified with `Init=` keyword

```
          Macro ,                              Args = list of names
          DCLNTVL  &Init=0,&Len=3              Optional initialization value
          GblC  &NtvlTyp                       Type attr of Interval variable
          LclA  &NtvlLen                       Length of an Interval variable
&NtvlTyp  SetC  'i'                            User type attr is lower case 'i'
.*        Length of an INTERVAL type could also be a global variable
&NV       SetA  N'&SysList                     Number of arguments to declare
&K        SetA  0                              Counter
.Test     Aif   (&K ge &NV).Done               Check for finish
&K        SetA  &K+1                           Increment argument count
          DC    PL&Len.'&Init.'                Define storage
&SysList(&K) Equ *-&Len.,&Len.,C'&NtvlTyp'  Declare name, length, type
          Ago   .Test
.Done     MEnd

          DclNtvl  Week,Init=7
+         DC    PL3'7'                 Define storage
+Week     Equ   *-3,3,C'i'             Name, length, type
```

Calculating With Date Variables: CalcDat Macro

- Now, define operations on DATEs and INTERVALs
- User-callable `CalcDat` macro calculates dates:

```
&AnsDate CalcDat  &Arg1,Op,&Arg2      Calculate a Date variable
```

- Allowed forms are:

```
ResultDate    CalcDat  Date,+,Interval        Date = Date + Interval
ResultDate    CalcDat  Date,-,Interval        Date = Date - Interval
ResultDate    CalcDat  Interval,+,Date        Date = Interval + Date
```

- `CalcDat` validates arguments, calls auxiliary macros

```
DATEADDI  Date1,LDat,Interval,LNvl,AnsDate,AnsLen  Date = Date+Interval
DATESUBI  Date1,LDat,Interval,LNvl,AnsDate,AnsLen  Date = Date-Interval
```

  - Auxiliary service macros (“private methods”) understand actual data representations (“encapsulation”)

Calculating With Date Variables: CalcDat Macro ...

- Calculate `Date=Date± Interval` or `Date=Interval+Date`
  - DATESUBI and DATEADDI are “private methods” (called subroutines)

```
         Macro ,                          Most error checks omitted!!
&Ans     CALCDAT &Arg1,&Op,&Arg2          Calculate a date in &Ans
         GblC  &NtvlTyp,&DateTyp          Type attributes
&T1      SetC  T'&Arg1                    Save type of &Arg1
&T2      SetC  T'&Arg2                    And of &Arg2
         Aif  ('&T1&T2' ne '&DateTyp&NtvlTyp' and                        X
               '&T1&T2' ne '&NtvlTyp&DateTyp').Err4  Validate types
         Aif   ('&Op' eq '+').Add         Check for add operation
         DATESUBI &Arg1,L'&Arg1,&Arg2,L'&Arg2,&Ans,L'&Ans   D = D-I
         MExit
.Add     AIF   ('&T1' eq '&NtvlTyp').Add2 1st opnd is interval of days
         DATEADDI &Arg1,L'&Arg1,&Arg2,L'&Arg2,&Ans,L'&Ans   D = D+I
         MExit
.Add2    DATEADDI &Arg2,L'&Arg2,&Arg1,L'&Arg1,&Ans,L'&Ans   D = I+D
         MExit
.Err4    MNote 8,'CALCDAT: Incorrect declaration of Date or Interval?'
         MEnd
```

Calculating Interval Variables: CalcNvl Macro

- Define user-called `CalcNvl` macro to calculate intervals
- Allowed forms are:

```
ResultInterval  CalcNvl  Date,-,Date         Difference of two date variables
ResultInterval  CalcNvl  Interval,+,Interval Sum of two interval variables
ResultInterval  CalcNvl  Interval,-,Interval Difference of two intervals
ResultInterval  CalcNvl  Interval,*,Number   Product of interval, number
ResultInterval  CalcNvl  Interval,/,Number   Quotient of interval, number
```

- `CalcNvl` validates declared types of arguments, and calls one of five auxiliary macros (more “private methods”):

```
DATESUBD  Date1,LDat1,Date2,LDat2,AnsI,AnsLen    Nvl = Date-Date
NTVLADDI  Nvl1,Len1,Nvl2,Len2,AnsI,AnsLen        Nvl = Nvl + Nvl
NTVLSUBI  Nvl1,Len1,Nvl2,Len2,AnsI,AnsLen        Nvl = Nvl - Nvl
NTVLMULI  Nvl1,Len1,Nvl2,Len2,AnsI,AnsLen        Nvl = Nvl * Num
NTVLDIVI  Nvl1,Len1,Nvl2,Len2,AnsI,AnsLen        Nvl = Nvl / Num
```

- Date arithmetic done by called subroutines

Calculating Interval Variables: CalcNvl Macro ...

```
         Macro ,                       Most error checks omitted!
&Ans     CALCNVL  &Arg1,&Op,&Arg2
         GblC  &NtvlTyp,&DateTyp       Type attributes
&X(C'+') SetC  'ADD'                   Name for ADD routine
&X(C'-') SetC  'SUB'                   Name for SUB routine
&X(C'*') SetC  'MUL'                   Name for MUL routine
&X(C'/') SetC  'DIV'                   Name for DIV routine
&Z       SetC  'C''&Op'''              Convert &Op char to self-def term
&T1      SetC  T'&Arg1                 Type of Arg1
&T2      SetC  T'&Arg2                 Type of Arg2
         Aif   ('&T1&T2&Op' eq '&DateTyp&DateTyp.-').DD  Chk date-date
         Aif   ('&T2' ne 'N').II          Second operand nonnumeric
         NTVL&X(&Z).I Arg1,L'&Arg1,=PL3'&Arg2',3,&Ans,L'&Ans I op const
         MExit
.II      NTVL&X(&Z).I &Arg1,L'&Arg1,&Arg2,L'&Arg2,&Ans,L'&Ans I op I
         MExit
.DD      DATESUBD  &Arg1,L'&Arg1,&Arg2,L'&Arg2,&Ans,L'&Ans  date-date
         MEnd

 Days     CALCNVL  Days,+,Days         Interval + Interval
+         NTVLADDI     Days,L'Days,Days,L'Days,Days,L'Days     I op I
 Days     CALCNVL  Hire,-,Degree       Date - Date
+         DATESUBD  Hire,L'Hire,Degree,L'Degree,Days,L'Days  date-date
```

Example of an Interval-Calculation Macro

- Macro NTVLADDI adds intervals to intervals

```
         Macro
&L       NTVLADDI  &Arg1,&L1,&Arg2,&L2,&Ans,&LAns
         AIf   ('&Arg1' ne '&Ans').T1    Check for Ans being Arg1
         AIf   (&L1 ne &LAns).Error      Same field, different lengths
&L       AP    &Ans.(&Lans),&Arg2.(&L2)  Add Arg2 to Answer
         MExit
.T1      AIf   ('&Arg2' ne '&Ans').T2    Check for Ans being Arg2
         AIf   (&L2 ne &LAns).Error      Same field, different lengths
&L       AP    &Ans.(&Lans),&Arg1.(&L1)  Add Arg1 to Answer
         MEXit
.T2      ANop  ,
&L       ZAP   &Ans.(&Lans),&Arg1.(&L1)  Move Arg1 to Answer
         AP    &Ans.(&Lans),&Arg2.(&L2)  Add Arg2 to Arg1
         MExit
.Error   MNote 8,'NTVLADDI: Target variable ''&Ans'' has same name as,*
               but different length than, a source operand'
         MEnd

 A       NTVLADDI  X,3,=P'5',1,X,3
+A       AP    X(3),=P'5'(1)             Add Arg2 to Answer
```

Comparison Operators for Dates and Intervals

- Define comparison macros CompDate and CompNtvl

```
&Label  CompDate  &Date1,&Op,&Date2,&True  Compare two dates
&Label  CompNtvl  &Ntvl1,&Op,&Ntvl2,&True  Compare two intervals
```

  - &Op is any useful comparison operator (EQ, NEQ, GT, LE, etc.)
  - &True is the branch target for true compares

```
         Macro
&Label   CompDate  &Date1,&Op,&Date2,&True
         GblA  &DateLen           Length of Date variables
&Mask(1) SetA  8,7,2,13,4,11,10,5,12,3  BC Masks
&T       SetC  ' EQ  NEQ GT  NGT LT  NLT GE  NGE LE  NLE ' Operators
&C       SetC  Upper('&Op')       Convert to Upper Case
&N       SetA  Index('&T','&C')   Find operator
         AIf   (&N eq 0).BadOp
&N       SetA  (&N+3)/4           Calculate mask index
&Label   CP    &Date1.(&DateLen),&Date2.(&DateLen)
         BC    &Mask(&N),&True    Branch to 'True Target'
         MExit
.BadOp   MNote 8,'&SysMac: Bad Comparison Operator ''&Op.'''
         MEnd
```

Case Study 9: Program Attributes

- Program variables typically have attributes like these:
  1. ***Storage representation***: how bits and bytes are used by the machine
     - Binary integers, characters, floating point, bit flags, etc.
  2. Data representation naming: describe the ***storage representation***
     - Assembler constants of types A, C, D, F, X, etc.
       - ... and attributes of the name of the storage representation
         - Traditionally, the assembler's traditional T' type attribute
  3. The ***computational abstraction*** represented by the data item
     - Date, Name, Age, Time, Weight, Height, Distance, Area, Volume, Speed, etc.
  4. The ***measures and units*** used to describe the data abstraction
     - Distances can be Inches, Centimeters, Miles, Kilometers, Light-Years, ...

- Traditional high-level languages generally don't handle the last two
  - ***Program attributes can and do!***

Program Attributes, Variation 1

- Suppose macros EVAL1-EVAL5 assign one variable to another:

```
         EVALn  var1,=,var2        Assigns var2 to var1
```

- A simple form (EVAL1) with no checking; types F, H, E, D are OK

```
         Macro ,                  Some checking omitted
&L       EVAL1 &T,&Op,&S          Target variable, operator, source
         LclC  &V                 Operation variant
&TS      SetC  T'&S               Type of source variable
&TT      SetC  T'&T               Type of target variable
         AIF   ('&TS' eq '&TT').OKType
         MNote 8,'&SysMac. — Incompatible argument types'
         MExit
.OKType  AIF   ('&TS' eq 'F').NoVar
&V       SetC  '&TS'              Type variant
.NoVar   ANop  ,
&L       L&V   0,&S
         ST&V  0,&T
         MEnd
```

- Traditional T' type attribute selects instructions

Program Attributes, Variation 2

- Let EVAL2 support mixed integer/floating point operands.

```
DS    DC   D'0.2'       Doubleword hex float source
DT    DS   D            Doubleword target variable
ES    DC   E'0.3'       Short hex float source
ET    DS   E            Short hex float target variable
FS    DC   F'45678901'  Fullword binary integer source
FT    DS   F            Fullword target variable
HS    DC   H'23456'     Halfword binary integer source
HT    DS   H            Halfword target variable
```

- Allow mixed-type assignments like

```
EVAL2 FT,=,HS     Halfword binary assigned to fullword
EVAL2 DT,=,ES     Short hex float assigned to long
EVAL2 DT,=,FS     Fullword binary assigned to long hex float
EVAL2 ET,=,HS     Halfword binary assigned to short hex float
```

  - The third and fourth assignments require type conversion!

- Some assignments are flagged (e.g. float hex to halfword binary)

Program Attributes, Variation 2 ...

- Examples of output generated by EVAL2:

```
                          63         EVAL2 DT,=,DS  OK
000024 6800 F000          64+        LD    0,DS
000028 6000 F008          65+        STD   0,DT

                          75         EVAL2 FT,=,HS  Halfword to fullword
000044 4800 F020          76+        LH    0,HS
000048 5000 F01C          77+        ST    0,FT

                          79         EVAL2 DT,=,ES  Short hex float to long
00004C 2F00               80+        LZER  0
00004E 7800 F010          81+        LE    0,ES
000052 6000 F008          82+        STD   0,DT

                          84         EVAL2 DT,=,FS  Convert fixed to float
000056 5800 F018          85+        L     0,FS
00005A B3B5 0000          86+        CDFR  0,0
00005E 6000 F008          87+        STD   0,DT

                         106         EVAL2 HT,=,FS  Fullword to halfword (?)
** ASMA254I *** MNOTE *** 107+     4,EVAL2 — Possible precision loss
00008E 5800 F018         108+        L     0,FS
000092 4000 F022         109+        STH   0,HT
```

Program Attributes, Variation 3: Declaration

- Assign attributes to data having two *items*:
  the physical properties (**D**istance, **W**eight)
- The program attribute uses letters for readability; other forms work as well
  - First character is data representation (Float or Integer)
  - Second character is the Item type (Distance or Weight)
  - Remaining characters are blank; will be used later for other attributes
- No units or measures specified; assumed to be compatible
- Use a **DCL3** macro to declare the data (instead of doing it manually)

```
         DCL3  names,Item=[W|D],Rep=[F|I]
.*        ... and a name can also be (name,initial_value)
```

- Data declared by **DCL3** is used by **EVAL3**

Program Attributes, Variation 3: Evaluation

- **EVAL3** macro checks types and representations, generate instructions:

```
                          147 T1     EVAL3 w,=,m   IW ← IW
000090 5800 F070          148+T1     L     0,m
000094 5000 F000          149+       ST    0,w
                          150 T2     EVAL3 x,=,j   FW ← FW
000098 6800 F058          151+T2     LD    0,j
00009C 6000 F008          152+       STD   0,x
                          160 T3     EVAL3 x,=,q   FW ← IW
0000B0 5800 F07C          161+T3     L     0,q
0000C0 B3B5 0000          162+       CDFR  0,0
0000B8 6000 F008          163+       STD   0,x
                          164 T4     EVAL3 w,=,j   IW ← FW
0000BC 6800 F058          165+T4     LD    0,j
0000B4 B3B9 4000          166+       CFDR  0,4,0
** ASMA254I *** MNOTE *** 167+  4,EVAL3 — Possible float←int precision loss
0000C4 5000 F000          168+       ST    0,w
```

Program Attributes, Variation 4

- Program attributes can assign “application meaning” to data items
  - Weights: Metric kilograms and English pounds:

```
000000 42A5000000000000     44 WtE    DC    DP(C'WE  ')'165'  English: Pounds
000008 424ACCCCCCCCCCD      45 WtM    DC    DP(C'WM  ')'74.8' Metric: Kilograms
000010                      46 NewE   DS    DP(C'WE  ')       English units
000018                      47 NewM   DS    DP(C'WM  ')       Metric units
```

  - Single storage representation: long hex floating point
- Final segment of the EVAL4 macro converts measures, e.g.

```
          – – –
&L        LD    0,&S
          AIF   ('&PAS'(2,1) eq '&PAT'(2,1)).DoSt
          AIF   ('&PAS'(2,1) eq 'E').ESrc  Source units English
.*              Source units metric:
          MD    0,=D'2.20462'       ◄— pounds per kilogram
          AGO   .DoSt
.ESrc     DD    0,=D'2.20462'       ◄— kilograms per pound
.DoSt     STD   0,&T
          – – –
```

Program Attributes, Variation 4: Evaluation

- Sample results from EVAL4:

```
                          49         EVAL4 NewE,=,WtE  English←English
000020 6800 F000          50+        LD    0,WtE
000024 6000 F010          51+        STD   0,NewE
                          52         EVAL4 NewM,=,WtM  Metric←Metric
000028 6800 F008          53+        LD    0,WtM
00002C 6000 F018          54+        STD   0,NewM
                          55         EVAL4 NewE,=,WtM  English←Metric
000030 6800 F008          56+        LD    0,WtM
000034 6C00 F068          57+        MD    0,=D'2.20462'
000038 6000 F010          58+        STD   0,NewE
                          59         EVAL4 NewM,=,WtE  Metric←English
00003C 6800 F000          60+        LD    0,WtE
000040 6D00 F068          61+        DD    0,=D'2.20462'
000044 6000 F018          62+        STD   0,NewM
```

- Sensitivity to extended types: ***No HLL can do this!*** (As far as I know)

Program Attributes, Variation 5: Declaration

- First, we use a **DCL5** macro to declare variables
  - Representations: Float, Integer (F, I)
  - Items: Weight, Distance (W, D)
  - Measures: English, Metric (E, M)
  - Units: Kilogram(Kg,K), Pound(P), Mile(Mi,M), Foot(Ft,F), Meter(M), Kilometer(Km,K)
- Some variables declared as Weights:

```
DCL5  iwmk,Rep=I,Item=W,Unit=M,Meas=kg
DCL5  fwmk,Rep=F,Item=W,Unit=m,meas=k
```

- Some variables declared as Distances:

```
DCL5  idef,Rep=I,Item=D,unit=e,meas=ft
DCL5  idem,Rep=I,Item=D,unit=e,meas=mi
DCL5  fdem,Rep=F,Item=D,unit=e,meas=m
DCL5  idmm,Rep=I,Item=D,unit=m,meas=m
DCL5  fdmk,Rep=F,Item=D,unit=m,meas=k
```

- Choice of letters used for Program Attribute values is arbitrary

Program Attributes, Variation 5: Evaluation

- Use an **EVAL5** macro to do assignments:

- Pseudo-code description:

```
Verify presence of non-null program attributes
Extract source and target variable Rep, Item, Unit, Meas
Verify units match (can't mix weight/distance assignment)
For units W and D:
  Create a table of actions for each attribute combination
  Construct source/target-variable "selection" and "action" matrix
  Select action for source/target variables in matching matrix entry
  Define appropriate conversion operations and constants
Generate code
```

- Other methods of selecting actions could be used

Program Attributes, Variation 5: Evaluation Examples

- Examples of statements generated by the EVAL5 macro:

```
                    317         EVAL5 fwep,=,iwmk   Kg to Pounds
5800 F000           318+        L     0,iwmk
B3B5 0000           319+        CDFR  0,0
6C00 F360           320+        MD    0,=D'2.20422'
6000 F010           321+        STD   0,fwep

                    417         EVAL5 idef,=,idem   Miles to Feet
5800 F028           418+        L     0,idem
A70C 14A0           419+        MHI   0,5280
5000 F018           420+        ST    0,idef

                    442         EVAL5 fdmk,=,idem   Miles to Km
5800 F028           443+        L     0,idem
B3B5 0000           444+        CDFR  0,0
6C00 F388           445+        MD    0,=D'1.60935'
6000 F050           446+        STD   0,fdmk
```

- Automatic conversion of representation, unit, and measure

Assembler and Program Attribute Summary

- Assembler attributes help ensure correct register usage
- Program attributes provide tremendous flexibility
  - Detailed descriptions of individual data items
  - Verifiable interactions between instructions and data
  - Enables conversions, diagnostics, etc.
- Adds more power to the macro language
- ***No HLL can do such powerful and useful things as this!***
  (As far as I know)

Case Study 10: “Front-Ending” a Macro

- Put your code “around” a call to a library macro, to:
  - Validate arguments to the library macro
  - Generate your own code before/after the library macro's
- Use OPSYN for dynamic renaming of opcodes:
  1. Define your “wrapper” macro with the same name
  2. OPSYN the name to a temp, then nullify itself (!)
  3. Do “front-end” processing, then call the library macro
  4. Do “back-end” processing
  5. Re-establish the “wrapper” definition from the temp name
- Example: “Wrapper” for a “READ” macro

```
         Macro ,                Defined in the source program
&L       READ   &A,&B,&C
READ_XX  OpSyn  READ            Save Wrapper's definition as READ_XX
READ     OpSyn  ,               Nullify this macro's definition
         — — —                  ...perform 'front—end' processing
&L       READ   &A1,&B1,&C1     Call system version of READ
         — — —                  ...perform 'back—end' processing
READ     OpSyn  READ_XX         Re—establish Wrapper's definition
         MEnd
```

Summary

- Easy to implement “High-Level Language” features in **your** Assembler Language
- Start with simple, concrete, useful forms
- Build new “language” elements incrementally
- Useful results directly proportional to implementation effort
  - Create as few or as many capabilities as needed
  - Checking and diagnostics as simple or elaborate as desired
- New language can precisely match application requirements
- Best of all: it's fun!