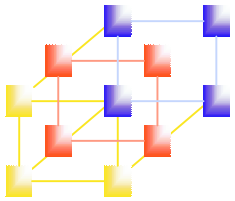
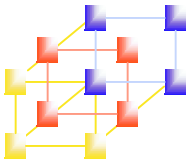


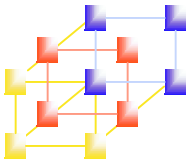
Chapter 4 Macro Processors

-- Macro Processor Design Options



Recursive Macro Expansion
Figure 4.11(a), pp. 200

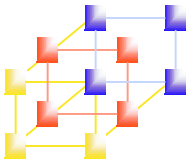
```
10        RDBUFF    MACRO     &BUFADR, &RECLTH, &INDEV
15        .
20        .         MACRO TO READ RECORD INTO BUFFER
25        .
30                  CLEAR     X                 CLEAR LOOP COUNTER
35                  CLEAR     A
40                  CLEAR     S
45                  +LDT      #4096             SET MAXIMUN RECORD LENGTH
50        $LOOP     RDCHAR    &INDEV            READ CHARACTER INTO REG A
65                  COMPR     A, S              TEST FOR END OF RECORD
70                  JEQ       &EXIT             EXIT LOOP IF EOR
75                  STCH      &BUFADR, X        STORE CHARACTER IN BUFFER
80                  TIXR      T                 LOOP UNLESS MAXIMUN LENGTH
85                  JLT       $LOOP             HAS BEEN REACHED
90        $EXIT     STX       &RECLTH           SAVE RECORD LENGTH
95                  MEND
```



Recursive Macro Expansion Figure 4.11(b), pp. 200

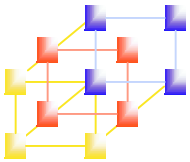
```
5    RDCHAR      MACRO   &IN
10   .
15   .     MACROTO READ CHARACTER INTO REGISTER A
20   .
25                 TD      =X'&IN'                 TEST INPUT DEVICE
30                 JEQ       *-3                   LOOP UNTIL READY
35                 RD      =X'&IN'                 READ CHARACTER
40                 MEND
```

- Recursive macro expansion
  - Invoke a macro by another
- However …



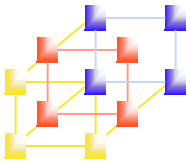
Problem of Recursive Macro Expansion

- Previous macro processor design cannot handle such kind of recursive macro invocation and expansion
  - The procedure EXPAND would be called recursively, thus the invocation arguments in the ARGTAB will be overwritten. (P.201)
  - The Boolean variable EXPANDING would be set to FALSE when the "inner" macro expansion is finished, *i.e.*, the macro process would forget that it had been in the middle of expanding an "outer" macro.
- Solutions
  - Write the macro processor in a programming language that allows recursive calls, thus local variables will be retained.
  - If you are writing in a language without recursion support, use a stack to take care of pushing and popping local variables and return addresses.



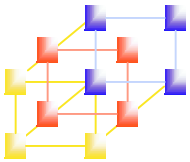
General-Purpose Macro Processors

- Macro processors that do not dependent on any particular programming language, but can be used with a variety of different languages
- Pros
  - Programmers do not need to learn many macro languages.
  - Although its development costs are somewhat greater than those for a language-specific macro processor, this expense does not need to be repeated for each language, thus save substantial overall cost.
- Cons
  - Large number of details must be dealt with in a real programming language
    - `Situations in which normal macro parameter substitution should not occur, e.g., comments.`
    - `Facilities for grouping together terms, expressions, or statements`
    - `Tokens, e.g., identifiers, constants, operators, keywords`
    - `Syntax had better be consistent with the source programming language`



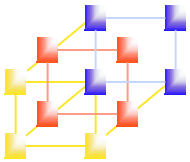
Macro Processing within Language Translators

- The macro processors we discussed are called “Preprocessors”.
  - Process macro definitions
  - Expand macro invocations
  - Produce an expanded version of the source program, which is then used as input to an assembler or compiler
- You may also combine the macro processing functions with the language translator:
  - Line-by-line macro processor
  - Integrated macro processor



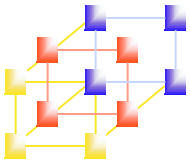
Line-by-Line Macro Processor

- Used as a sort of input routine for the assembler or compiler
  - Read source program
  - Process macro definitions and expand macro invocations
  - Pass output lines to the assembler or compiler
- Benefits
  - Avoid making an extra pass over the source program.
  - Data structures required by the macro processor and the language translator can be combined (e.g., OPTAB and NAMTAB)
  - Utility subroutines can be used by both macro processor and the language translator.
    - `Scanning input lines`
    - `Searching tables`
    - `Data format conversion`
  - It is easier to give diagnostic messages related to the source statements.



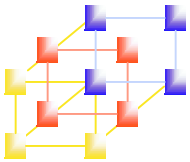
Integrated Macro Processor

- An integrated macro processor can potentially make use of any information about the source program that is extracted by the language translator.
  - Ex (blanks are not significant in FORTRAN)
    - `DO  100  I = 1,20`
      - `a DO statement`
    - `DO 100 I = 1`
      - `An assignment statement`
      - `DO100I:  variable  (blanks are not significant in FORTRAN)`
- An integrated macro processor can support macro instructions that depend upon the context in which they occur.



ANSI C Macro Language

- Definitions and invocations of macros are handled by a preprocessor, which is generally not integrated with the rest of the compiler.
- Example
  - `#define  NULL  0`
  - `#define  EOF   (-1)`
  - `#define  EQ    ==`
    - `/* syntactic modification */`
  - `#define  ABSDIFF (X,Y) ((X)>(Y)?(X)-(Y):(Y)-(X))`



ANSI C Macro Language

- Parameter substitutions are not performed within quoted strings:

```
#define  DISPLAY(EXPR) printf("EXPR= %d\n", EXPR)
```

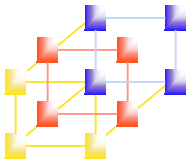
  - Example

```
DISPLAY(I*J+1)  ==>  printf("EXPR= %d\n", I*J+1)
```

- Stringizing" operator, #
  - Used to perform argument substitution in quoted strings

```
#define  DISPLAY(EXPR) printf(#EXPR "= %d\n", EXPR)
```

  - Example
  - `DISPLAY(I*J+1) ==> printf("I*J+1" "= %d\n", I*J+1)`



ANSI C Macro Language

- Recursive macro definitions or invocations
  - After a macro is expanded, the macro processor rescans the text that has been generated, looking for more macro definitions or invocations.
  - Macro cannot invoke or define itself recursively.
- Example

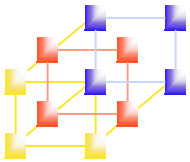
```
DISPLAY(ABSDIFF(3,8))
```

scan

```
printf("ABSDIFF(3,8)"  "= %d\n", ABSDIFF(3,8))
```

rescan

```
printf("ABSDIFF(3,8)"  "= %d\n", ( (3)>(8) ? (3)-
  (8) : (8)-(3) ))
```



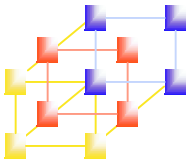
ANSI C Macro Language

- Conditional compilation statements
- Example 1

```
#ifndef   BUFFER_SIZE
   #define  BUFFER_SIZE  1024
#endif
```

- Example 2

```
#define  DEBUG  1
     :
#if DEBUG == 1
   printf(…)              /* debugging outout */
#endif
```



GCC Preprocessor Options

- -E
  - Run only the C preprocessor. Preprocess all the C source files specified and output the results to standard output or to the specified output file.
- -C
  - Tell the preprocessor not to discard comments. Used with the `-E' option.
- -P
  - Tell the preprocessor not to generate `#line' commands. Used with the `-E' option.
- -Dname[=defn]
  - Define macro “name” as “defn”. If “=defn” is omitted, the string “1” is assigned to “name”.

- Reference: http://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc/