```
1       prog1:  procedure options (main) ;
2
3
4          /*************************************************************/
5          /*      PURPOSE:  This program will convert equations in simple */
6          /*                infix, suffix and prefix notation into fully  */
7          /*                parenthesized infix equations.                */
8          /*        INPUT:  prog1.d01 = input equations                   */
9          /*       OUTPUT:  prog1.p01 = Report of converted equations     */
10         /*       AUTHOR:  Garry R. Daly                                 */
11         /*************************************************************/
12
13
14         /* The following represents the Data flow diagram of this program:
15
16                                  *************
17                                  *           *
18                                  *   PROG1   *
19                                  *           *
20                                  *************
21               /------------------/  /   |  \  \--------------------\
22              |                     /    |   \                       |
23         *************            ----    |    ----            *************
24         *           *           /        |        \           *           *
25         *  VALIDTE  *          /         |         \          *  PRNTRES  *
26         *           *         /          |          \         *           *
27         *************        /           |           \        *************
28                       |                  |            |
29                    *************         |         *************
30                    *           *         |         *           *
31                    *  TRANPRE  *         |         *  TRANSUF  *
32                    *           *         |         *           *
33                    *************         |         *************
34                        |     |           |           |     |
35                        |     |   *************       |     |
36                        |     |   *           *       |     |
37                        |     |   *  TRANINF  *       |     |
38                        |     |   *           *       |     |
39                        |     |   *************       |     |
40                         \    |                     |    /
41                          \    \------\   /------/    /
42                           \           \ /           /
43                            \          X          /
44                              |     |---/ \---|     |
45                           *************   *************
46                           *           *   *           *
47                           *   PUSH    *   *    POP    *
48                           *           *   *           *
49                           *************   *************
50                                                                    */
51          /* This program makes the following assumptions:      */
52          /*   Valid data may only consist of the operators     */
```

```
53          /*   *,/,+,- and of upper and lower case letters of   */
54          /*   the alphabet.  Input strings are not to exceed   */
55          /*   20 characters and are required to have a minimum */
56          /*   of 3 characters to be valid.                     */
57
58        dcl   sysin              file input stream ,
59              sysprint           print file ;
60
61        dcl   verify             builtin ,
62              length             builtin ,
63              substr             builtin ;
64
65        dcl   operators          char(4)       init ('*/+-') ,
66              invalid_str        char(5)       init ('.....') ;
67
68        dcl   type               char(6) ,
69              infix              char(6)       init ('INFIX ') ,
70              suffix             char(6)       init ('SUFFIX') ,
71              prefix             char(6)       init ('PREFIX') ,
72              invalid            char(15)      init
73                                               ('*** INVALID ***') ,
74              error              char(6)       init ('ERROR ') ;
75
76        dcl   continue           char(3) ,
77              yes                char(3)       init ('yes') ,
78              no                 char(3)       init ('no ') ;
79
80        dcl   expression_in      char(100)     varying ,
81              expression_out     char(100)     varying ,
82              stack_pntr         pointer ;
83
84        dcl 1 eoj_hdr ,
85              2 info             char(17)      init
86                                               ('*** END OF REPORT') ;
87
88        /* Set up a loop to read the data and print out the    */
89        /*   results in a stream process.                     */
90
91        on endfile (sysin)
92           continue = no ;
93
94        continue = yes ;
95        do while (continue = yes) ;
96          get file (sysin) edit (expression_in) (a(100)) ;
97          if continue = no
98            then goto wrapup ;        /* Std PLI allows leave */
99
100         /* Validate incomming expression --                  */
101         /*   If the expression is valid, then expression_out */
102         /*   will equal expression_in (or its length > 0)    */
103
104         call validte(expression_in,expression_out) ;
```

```
105          if length(expression_out) > 0
106            then do ;
107
108              /* Determine the type of valid expression --     */
109              /*   An operator in position 1 = prefix          */
110              /*   An operator in the end position = suffix    */
111              /*   Neither prefix or suffix = infix            */
112
113               if verify(substr(expression_in,1,1),operators) = 0
114                then do ;     /* PREFIX */
115                  if verify(substr(expression_in,
116                            length(expression_in),1),'*/+-') = 0
117                    then do ;    /* ERROR type */
118                      expression_out = invalid ;
119                      type = error ;
120                      end ;
121                    else do ;        /* Truly prefix */
122                      call tranpre(expression_in,
123                                   expression_out,
124                                   stack_pntr) ;
125                      type = prefix ;
126
127                      /* Check the length of expression_out --   */
128                      /* -- if zero = invalid prefix expression. */
129
130                      if length(expression_out) = 0
131                        then do ;     /* free the unused nodes */
132                          do while (stack_pntr ^= null()) ;
133                             stack_pntr = pop(stack_pntr) ;
134                             end ;
135                          expression_out = invalid ;
136                          end ;
137                      end ;
138                end ;
139
140               else if verify(substr(expression_in,
141                              length(expression_in),1),
142                              operators) = 0
143                   then do ;     /* SUFFIX */
144                     call transuf(expression_in,
145                                  expression_out,
146                                  stack_pntr) ;
147
148                     /* Again -- check the length of         */
149                     /*   expression_out.  Zero length will  */
150                     /*   indicate an invalid expression.    */
151
152                     if length(expression_out) = 0
153                       then do ;     /* free the unused nodes */
154                         do while (stack_pntr ^= null()) ;
155                            stack_pntr = pop(stack_pntr) ;
156                            end ;
```

```
                        expression_out = invalid ;
                        end ;
                    type = suffix ;
                    end ;
                  else do ;     /* INFIX */
                    expression_out = traninf(expression_in) ;
                    type = infix ;

                    /* Infix resolution is done by recursion */
                    /* The program will return a string      */
                    /* namely: '.....' if during the         */
                    /* recursion an error was detected.      */

                    if verify(invalid_str,expression_out) = 0
                      then expression_out = invalid ;
                    end ;
          end ;     /* end of valid expression */
          else do ;
            type = error ;
            expression_out = invalid ;
            end ;

        /* Print the results of the expression read in */

        call prntres(expression_in,expression_out,type) ;
        end ;

      /* Print the header signifying the end of job */

      wrapup:
          put file (sysprint) skip(3) edit (eoj_hdr) (a) ;

      /*****************************/

    validte:  procedure(expression_in,expression_out) ;

      /* This procedure will attempt to validate an input    */
      /*   string namely expression_in, as being a subset    */
      /*   of upper and lower case alphabetic characters and */
      /*   a set of four operators (*,/,+,-).  If unable to  */
      /*   do so, expression_out will be set to null.        */

      dcl   verify              builtin ,
            length              builtin ;

      dcl   expression_in       char(*)       varying ,
            expression_out      char(*)       varying ,
            opreators           char(4)       init('*/+-') ,
            upper_letters       char(26)      init
                                ('ABCDEFGHIJKLMNOPQRSTUVWXYZ') ,
            lower_letters       char(26)      init
                                ('abcdefghijklmnopqrstuvwxyz') ;
```

```
209
210         /* if expression_in is not a subset of the character   */
211         /*   set: upper_letters or lower_letters or operators, */
212         /*   then return a null expression_out                 */
213
214         if length(expression_in) < 3
215            | length(expression_in) > 20
216            |  verify(expression_in,upper_letters ||
217                                    lower_letters || operators ) > 0
218
219            then expression_out = '' ;
220            else expression_out = expression_in ;
221
222         return ;
223
224      end validte ;
225
226          /*****************************/
227
228      tranpre:  procedure(expression_in,
229                          expression_out,
230                          stack_pntr) ;
231
232         /* This procedure will attempt to translate an prefix  */
233         /*   expression (expression_in) and translate it into  */
234         /*   a well balanced parenthesized expression.  If it  */
235         /*   is unable to do so, (due to expression_in being   */
236         /*   invalid), expression_out will be returned as a    */
237         /*   null value.                                       */
238
239         dcl   substr             builtin ,
240               null               builtin ,
241               length             builtin ,
242               verify             builtin ;
243
244         dcl   pos_in_exp         fixed bin(15) ,
245               expression_in      char(*)       varying ,
246               expression_out     char(*)       varying ,
247               temp               char(100)     varying ;
248
249         dcl   stack_pntr        pointer ;
250
251         dcl 1 stack             based(stack_pntr) ,
252               2 parsed_exp      char(100)      varying ,
253               2 last_parsed_exp pointer ;
254
255         /* Initialize the first member of the stack */
256
257         expression_out = '' ;
258         stack_pntr = push(substr(expression_in,1,1),null()) ;
259
260         /* Loop from position 2 thru the length of the string */
```

```
/*   pushing onto the stack until two constants are   */
/*   side by side.  In this case you should pop off   */
/*   two constants and an operator -- otherwise the   */
/*   expression is invalid.                           */

do pos_in_exp=2 to length(expression_in) ;
  if verify(substr(expression_in,pos_in_exp,1),'*/+-') > 0
     then do ;     /* A constant was just found */

       /* Check if there is anything in the stack */

       if stack_pntr = null()
         then stack_pntr = push(substr(expression_in,
                                pos_in_exp,1),stack_pntr) ;
         else if verify(stack_pntr->parsed_exp,'*/+-') > 0
              then do ;     /* constant on the stack */
                temp = '(' || stack_pntr->parsed_exp ||
                       ' ' ;
                stack_pntr = pop(stack_pntr) ;

                /* Find out if you have operator in    */
                /* the stack -- if not = error.        */

                if stack_pntr = null() |
                   verify(stack_pntr->parsed_exp,'*/+-') >0
                   then do ;
                     expression_out = '' ;
                     return ;
                     end ;
                temp = temp || stack_pntr->parsed_exp ||
                       ' '  || substr(expression_in,
                         pos_in_exp,1) || ')' ;
                stack_pntr = pop(stack_pntr) ;
                stack_pntr = push(temp,stack_pntr) ;
                end ;

              else stack_pntr = push(substr(expression_in,
                                 pos_in_exp,1),stack_pntr) ;
       end ;     /* end found constant in expression_in */
     else stack_pntr = push(substr(expression_in,pos_in_exp,
                            1),stack_pntr) ;
  end ;
/* Finally, check the stack.  If there is only one */
/*   value in it, your finished.  Otherwise, you   */
/*   you have an error condition.                  */

if stack_pntr = null() |
   stack_pntr->last_parsed_exp ^= null()
   then do ;
     expression_out = '' ;
     return ;
     end ;
```

```
313          else do ;
314            expression_out = stack_pntr->parsed_exp ;
315            stack_pntr = pop(stack_pntr) ;
316            return ;
317            end ;
318
319     end tranpre ;
320
321        /*****************************/
322
323     traninf:  procedure (expression_in) recursive
324                    returns (char(100) varying) ;
325
326        /* This procedure will attempt to translate an infix   */
327        /*   expression (expression_in) and translate it into  */
328        /*   a well balanced parenthesized expression.  If it  */
329        /*   is unable to do so, (due to expression_in being   */
330        /*   invalid), the string '.....' will be returned to  */
331        /*   signify that an error has occurred.               */
332        /*   The main program will check the string returned   */
333        /*   for the occurrance of '.....'.  This was done     */
334        /*   because during recursive programming, there is    */
335        /*   no clear method of returning when nested.         */
336
337        dcl   length             builtin ,
338              verify             builtin ,
339              index              builtin ,
340              substr             builtin ;
341
342        dcl   expression_in      char(*)       varying ,
343              operator(4)        char(1)        init ('-','+','/','*')
344                                                static ,
345              oper_pos           fixed bin(15) ,
346              oper_index         fixed bin(15) ;
347
348        if length(expression_in) = 3       /* i.e A*B */
349          then return ('(' || substr(expression_in,1,1) || ' '
350                           || substr(expression_in,2,1) || ' '
351                           || substr(expression_in,3,1) || ')' ) ;
352        if length(expression_in) = 1     /* i.e. a constant */
353          then do ;
354
355             /* Check if the expression is an operator --      */
356             /*  -- if so, add a blank before and after it.    */
357
358             if verify(substr(expression_in,1,1),'*/+-') = 0
359                then return (' ' || expression_in || ' ') ;
360                else return (expression_in) ;
361             end ;     /* end constant or operator */
362
363        /* if expression is neither a constant or a simple  */
364        /* infix expression -- you have to break the        */
```

```
365         /* expression down further so that it becomes one.  */
366
367         oper_pos = 0 ;
368         do oper_index=1 to 4 while (oper_pos = 0) ;
369           oper_pos = index(expression_in,operator(oper_index)) ;
370           end ;
371
372         /* Check oper_pos -- it will equal zero when the       */
373         /*   infix string passed (expression_in) is invalid    */
374
375         if oper_pos = 0
376           then return ('.....') ;
377           else return(
378                       '('
379                     || traninf(substr(expression_in,1,oper_pos-1))
380                     || traninf(substr(expression_in,oper_pos,1))
381                     || traninf(substr(expression_in,oper_pos+1,
382                           length(expression_in) - (oper_pos)))
383                     || ')' ) ;
384
385       end traninf ;
386
387         /*****************************/
388
389       transuf:  procedure(expression_in,
390                           expression_out,
391                           stack_pntr) ;
392
393         /* This procedure will attempt to translate an suffix  */
394         /*   expression (expression_in) and translate it into  */
395         /*   a well balanced parenthesized expression.  If it  */
396         /*   is unable to do so, (due to expression_in being   */
397         /*   invalid), expression_out will be returned as a    */
398         /*   null value.                                       */
399
400         dcl   substr             builtin ,
401               null               builtin ,
402               verify             builtin ,
403               length             builtin ;
404
405         dcl   expression_in     char(*)       varying ,
406               expression_out    char(*)       varying ,
407               temp              char(100)     varying ,
408               pos_in_exp        fixed bin(15) ;
409
410         dcl   stack_pntr        pointer ;
411
412         dcl 1 stack             based(stack_pntr) ,
413               2 parsed_exp      char(100)     varying ,
414               2 last_parsed_exp pointer ;
415
416         /* Insert the first member on top of the stack */
```

```
417
418         expression_out = '' ;
419         stack_pntr = push(substr(expression_in,1,1),null()) ;
420
421         /* Set up the loop to search until an operator is  */
422         /*   found and then you can pop two constants off  */
423         /*   the stack if it truly is a valid expression.  */
424
425         do pos_in_exp=2 to length(expression_in) ;
426           if verify(substr(expression_in,pos_in_exp,1),'*/+-') = 0
427             then do ;    /* Current position has operator */
428               if stack_pntr = null() |
429                  stack_pntr->last_parsed_exp = null()
430                  then do ;
431
432                    /* You can't pop two values = error */
433
434                    expression_out = '' ;
435                    return ;
436                    end ;
437
438               /* Check both values on the stack to be certain */
439               /*   that they are both constants or this is an */
440               /*   error.                                     */
441
442               if length(stack_pntr->parsed_exp) < 2
443                & verify(stack_pntr->parsed_exp,'*/+-') = 0
444                 then do ;    /* Case 1: First value on stack */
445                   expression_out = '' ;
446                   return ;
447                   end ;
448
449               temp = substr(expression_in,pos_in_exp,1) || ' '
450                      || stack_pntr->parsed_exp || ')' ;
451               stack_pntr = pop(stack_pntr) ;
452
453               /* Check if you have anything in your stack */
454
455               if stack_pntr = null() |
456                  length(stack_pntr->parsed_exp) < 2
457                  & verify(stack_pntr->parsed_exp,'*/+-') = 0
458
459                  then do ;   /* Case 2: Second value on stack */
460                    expression_out = '' ;
461                    return ;
462                    end ;
463
464               temp = '(' || stack_pntr->parsed_exp || ' '
465                      || temp ;
466
467               /* Remove the old node and insert the new node */
468
```

```
          stack_pntr = pop(stack_pntr) ;
          stack_pntr = push(temp,stack_pntr) ;
          end ;     /* end found an operator in pos_in_exp */

        else stack_pntr = push(substr(expression_in,pos_in_exp,
                               1),stack_pntr) ;
        end ;     /* end do pos... */

   /* There should only be one node left if this is a     */
   /*   valid expression.                                 */

   if stack_pntr = null() |
      stack_pntr->last_parsed_exp ^= null()
      then do ;
        expression_out = '' ;
        return ;
        end ;

        else do ;
          expression_out = stack_pntr->parsed_exp ;
          stack_pntr = pop(stack_pntr) ;
          return ;
          end ;

end transuf ;

    /*****************************/

push:  procedure(expression_in,last_stack_mbr)
                 returns (pointer) ;

   /* This program will accept an expression along with */
   /*   a pointer.  It will only allocate a stack       */
   /*   initializing it to the expression_in and the    */
   /*   the previous node in a chain.                   */

   dcl   expression_in         char(*)       varying ,
         last_stack_mbr        pointer ,
         stack_pntr            pointer ;
   dcl 1 stack                 based(stack_pntr) ,
         2 parsed_exp          char(100)     varying ,
         2 last_parsed_exp     pointer ;

   allocate stack ;
   stack_pntr->parsed_exp = expression_in ;
   stack_pntr->last_parsed_exp = last_stack_mbr ;
   return(stack_pntr) ;

end push ;

   /*****************************/
```

```
pop:  procedure(stack_pntr)  returns (pointer) ;

   /* This program accepts a pointer and frees the    */
   /*   current node returning the previous member    */
   /*   in the chain.                                 */

   dcl   stack_pntr         pointer ,
         last_stack_mbr     pointer ;

   dcl 1 stack              based(stack_pntr) ,
         2 parsed_exp       char(100)       varying ,
         2 last_parsed_exp  pointer ;

   last_stack_mbr = stack_pntr->last_parsed_exp ;
   free stack ;
   return(last_stack_mbr) ;

end pop ;

   /*****************************/

prntres:  procedure(expression_in,expression_out,
                    type) ;

   /* This program will print the results of the trans- */
   /*   formation of expressions to well balanaced      */
   /*   parenthesized expressions.                      */

   dcl   date               builtin ,
         substr             builtin ;

   dcl   expression_in      char(*)       varying ,
         expression_out     char(*)       varying ,
         type               char(*) ;

   dcl   lines_printed      fixed bin(15) init (0) static ,
         page_no            fixed bin(15) init (0) static ;

   dcl 1 head1 ,
         2 dheader          char(5)       init ('Page ') ,
         2 current_page     fixed bin(15) ;
   dcl 1 head2 ,
         2 spaces           char(23)      init (' ') ,
         2 dheader          char(17)      init
                            ('DEPAUL UNIVERSITY') ;

   dcl 1 head3 ,
         2 spaces           char(18)      init (' ') ,
         2 dheader          char(27)      init
                            ('EQUATION CONVERSION PROGRAM') ;

   dcl 1 head4 ,
```

```
573             2 spaces           char(23)      init (' ') ,
574             2 dheader          char(10)      init
575                                ('CSC311    ') ,
576             2 todays_date      char(8) ;
577
578       dcl 1 head5 ,
579             2 spaces           char(25)      init (' ') ,
580             2 dheader          char(13)      init
581                                ('Garry R. Daly') ;
582       dcl 1 head6 ,
583             2 dheader1         char(17)      init
584                                ('ORIGINAL EQUATION') ,
585             2 space1           char(11)      init (' ') ,
586             2 dheader2         char(4)       init ('TYPE') ,
587             2 space2           char(13)      init(' ') ,
588             2 dheader3         char(6)       init('RESULT') ;
589
590       dcl 1 head7 ,
591             2 under1           char(25)      init
592                                ('-------------------------') ,
593             2 under2           char(25)      init
594                                ('-------------------------') ,
595             2 under3           char(25)      init
596                                ('-------------------------') ,
597             2 under4           char(15)      init
598                                ('---------------') ;
599
600       dcl 1 printed_results ,
601             2 equation_in      char(20) ,
602             2 space1           char(7)       init (' ') ,
603             2 equation_type    char(6) ,
604             2 space2           char(12)      init (' ') ,
605             2 results          char(45) ;
606
607       /* Check to see if have to perform a page eject */
608
609       if lines_printed > 60 | page_no = 0
610         then do ;
611           lines_printed = 8 ;     /* 8 lines of header info */
612           page_no = page_no + 1 ;
613           if page_no = 1
614             then do ;             /* Capture system date */
615               substr(head4.todays_date,1,2) = substr(date(),3,2) ;
616               substr(head4.todays_date,3,1) = '/' ;
617               substr(head4.todays_date,4,2) = substr(date(),5,2) ;
618               substr(head4.todays_date,6,1) = '/' ;
619               substr(head4.todays_date,7,2) = substr(date(),1,2) ;
620               end ;
621           head1.current_page = page_no ;
622           put file (sysprint) page ;
623           put file (sysprint) skip(1) edit (head1) (a) ;
624           put file (sysprint) skip(2) edit (head2) (a) ;
```

```
625            put file (sysprint) skip(1) edit (head3) (a) ;
626            put file (sysprint) skip(1) edit (head4) (a) ;
627            put file (sysprint) skip(1) edit (head5) (a) ;
628            put file (sysprint) skip(2) edit (head6) (a) ;
629            put file (sysprint) skip(1) edit (head7) (a) ;
630            end ;
631
632         printed_results.equation_in = expression_in ;
633         printed_results.equation_type = type ;
634         printed_results.results       = expression_out ;
635         put file (sysprint) skip(2) edit (printed_results) (a) ;
636         lines_printed = lines_printed + 2 ;
637
638      end prntres ;
639
640      end prog1 ;
```