

# PLC HW2- Compiler Track

411021314 林芝蓁

---

## Contents

- 1.The problem description
2. Highlight of the way you write the program
3. The program listing
4. Test run results
5. Discussion
6. Notes

# 1. Problem description:

- Augment your  $P$  parser with semantic actions for constructing parse trees, and write a  $C$  code generator for your  $P$  compiler.

```
• typedef struct p_exp {  
•     int exp_id;  
•     char name[16];  
•     int val;  
•     struct p_exp *exp1;  
•     struct p_exp *exp2;  
•     struct p_exp *next;  
• } pEXP;  
•  
• typedef struct p_stm {  
•     int stm_id;  
•     struct p_exp *exp1;  
•     struct p_exp *exp2;  
•     struct p_stm *stm1;  
•     struct p_stm *stm2;  
•     struct p_stm *next;  
• } pSTM;
```

Given the structure of parse tree nodes, we need to complete the code of `p2c_yacc.y` and `p2c_lex.l` and `p2c_tree.c`.

## 2. way to write the program:

- Firstly, we need to specify the interactions between each file.
  - i. For `p2c_lex.l`, it controls the behavior of scanner and the scanned tokens.

```
{STRING} {sscanf(yytext, "%[^\\t\\n]", pas_name);  
          yylval.sr = strdup( pas_name );  
          return STRING;}
```

So we scan the source file and specify tokens by regular expressions, as for the `yylval.sr`, it's defined in the `.y` file.

- ii. For `p2c_yacc.y`, there is a union for classifying the data type of each token.

```
%union{ pSTM* sm; //statement
        pEXP* ex; //expression
        int   nu; //number
        char* sr; //string
      }
```

Thus, we declare the token types as follows:

```
%type <sm> prog
%type <sm> block
%type <ex> type
%type <ex> arrtype
%type <nu> NUM
%type <sr> ID
```

Also, this file controls how the tokens' information is stored into each parse treenode:

```
//A -> X1 X2 ... Xk
//$$ return value of A
//$k return value of Xk
block : vardecs prodecs stmts
      { $$ = create_stm();
        $$->stm_id = sBLOCK;
        $$->stm1 = $1; //use $$ to store what returned from vardecs
        $$->stm2 = $2; //..what returned from prodecs
        $$->next = $3; //..what returned from stmts
      }
;
```

For example, `$$` is the pointer to the current treenode( that is, for the non-terminal block ), and we mark the treenode by the label "sBlock". After that, we let the pointers `stm1`, `stm2`, and `next` point to the treenodes of the three transforming non-terminals of the current treenode.

- iii. For p2c\_tree.c, it defines how the information in the parse treenodes is represented to generate the C code corresponding to the source code in P language:

```
vardec : ID moreid COLON type
{ $$ = create_stm();
  //printf("vardec\n") ;
  $$->stm_id = sVARDEC;
  $$->exp1 = create_exp();
  $$->exp1->exp_id = eVARDEC;
  strcpy($$->exp1->name, $1);
  $$->exp1->exp1 = $2;
  $$->exp2 = $4;
}
```

For example, for the treenode labeled with “eVARDEC”, which is for the statement of variable declaration. Since the variable name is stored in the \$\$ -> exp1 -> name, and its type is stored in the \$\$ -> exp2, so we need to generate the statement of variable declaration by the C syntax, e.g., <type> <var\_name> ;

```
void print_exp ( pEXP* p ) {
    pEXP* te;
    if( p ) {
```

In the p2c\_tree.c file, we specify the type of a treenode by its label, and p is the \$\$ in the p2c\_yacc.y file.

Since the declaration of a simple integer and an array is quite different in C language and in P grammar. It's a little complicated to deal with the generating order for the C code.

Here's the example:

```
A: array[1..10] of integer;
int A[ 10 ] ;
```

The variable name can be generated first, then the info type latter including the data type and array size in P grammar, but **NOT** in C language, we need to generate the data type in \$\$ -> exp2 and go to the \$\$ -> exp1 to generate the variable name, then finally go back to \$\$ -> exp2 to generate the size of array.

So the part of variable declaration is implemented as follows:

```
case sVARDEC:
    print_exp( p -> exp2 );
    if( p -> exp2 -> exp_id == eARRTYPE )
    {
        fprintf( yyout , p -> exp1 -> name ) ;
        print_exp( p -> exp1 ) ;
        fprintf( yyout , "[ %d ] " , aind ) ;
    }else{
        print_exp( p -> exp1 );//eVARDEC
    }//p -> exp1 -> exp1 is eMOREID
    fprintf(yyout, ";\n");
    amode = 0;
    break;
```

```
case eVARDEC:
    if( !amode ) fprintf( yyout , p -> name ) ;
    print_exp( p -> exp1 ) ;
    fprintf( yyout , " " ) ;
    break;
```

- The encountered problem and the resolution:

Except the previous problem encountered as I wrote the program, the implementation for write statement was also a trouble. Since the data type might be integer or string, so I had to check the outvalue before generate the C code.

```
case sWRISTATE:/*******/
    // Write your own code generation
    if( p -> exp1 && p -> exp1 -> exp_id == eSTRING )
    {
        fprintf( yyout , "printf(" ) ;
        print_exp( p -> exp1 ) ;
        fprintf( yyout , ");\n" ) ;
    }else{
        fprintf( yyout , "printf(\"%d\", " ) ;
        print_exp( p -> exp1 ) ;
        fprintf( yyout ,");\n" ) ;
        print_stm( p -> stm1 ) ;
    }
    break;
```

And I also created a new token in p2c\_lex.l to specify the string:

```
{STRING} {sscanf(yytext, "%[^\\t\\n]", pas_name);  
    yylval.sr = strdup( pas_name );  
    return STRING;}
```

```
wristate:  WRITE LP outvalue moreoutval RP  
    { // ***** Write your own semantic action  
    $$ = create_stm() ;  
    $$ -> stm_id = SWRISTATE ;  
    $$ -> exp1 = $3 ;  
    $$ -> exp2 = $4 ;  
    }  
|        WRITE LP STRING RP  
    {  
    $$ = create_stm() ;  
    $$ -> stm_id = SWRISTATE ;  
    $$ -> exp1 = create_exp() ;  
    $$ -> exp1 -> exp_id = eSTRING ;  
    strcpy( $$ -> exp1 -> name , $3 ) ;  
    }  
;
```

The above script is the parsing rule in the p2c\_yacc.y file.

### 3. The program listing

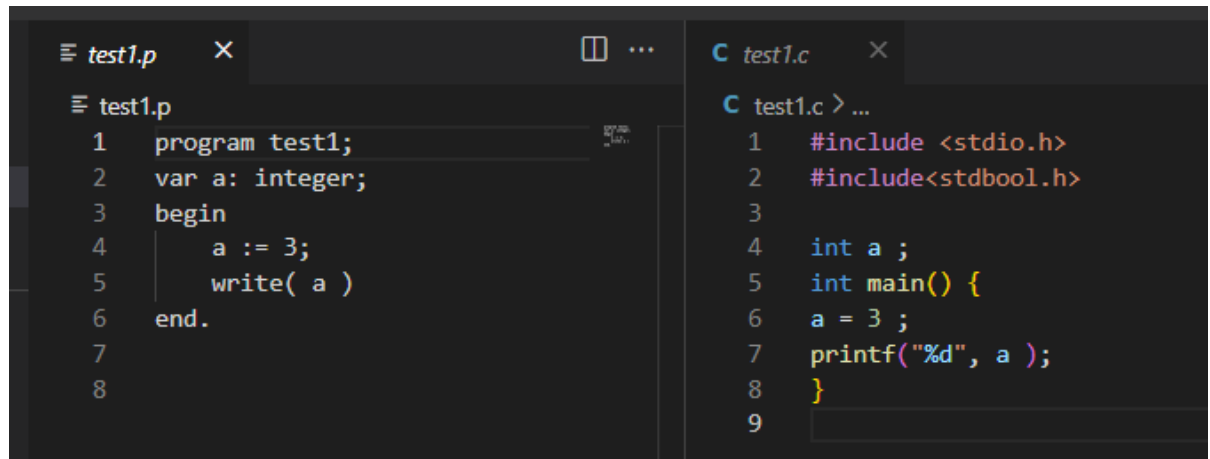
- There are 6 coding files in the .rar file:

p2c\_lex.l  
p2c\_tree.c  
p2c\_tree.h  
p2c\_yacc.y  
p2c.c  
p2c.h

And there are 3 files that I modified to augment the P parser and generate the C code, which are p2c\_lex.l, p2c\_tree.c, and p2c\_yacc.y.

## 4. Test run results

- For test case 1:

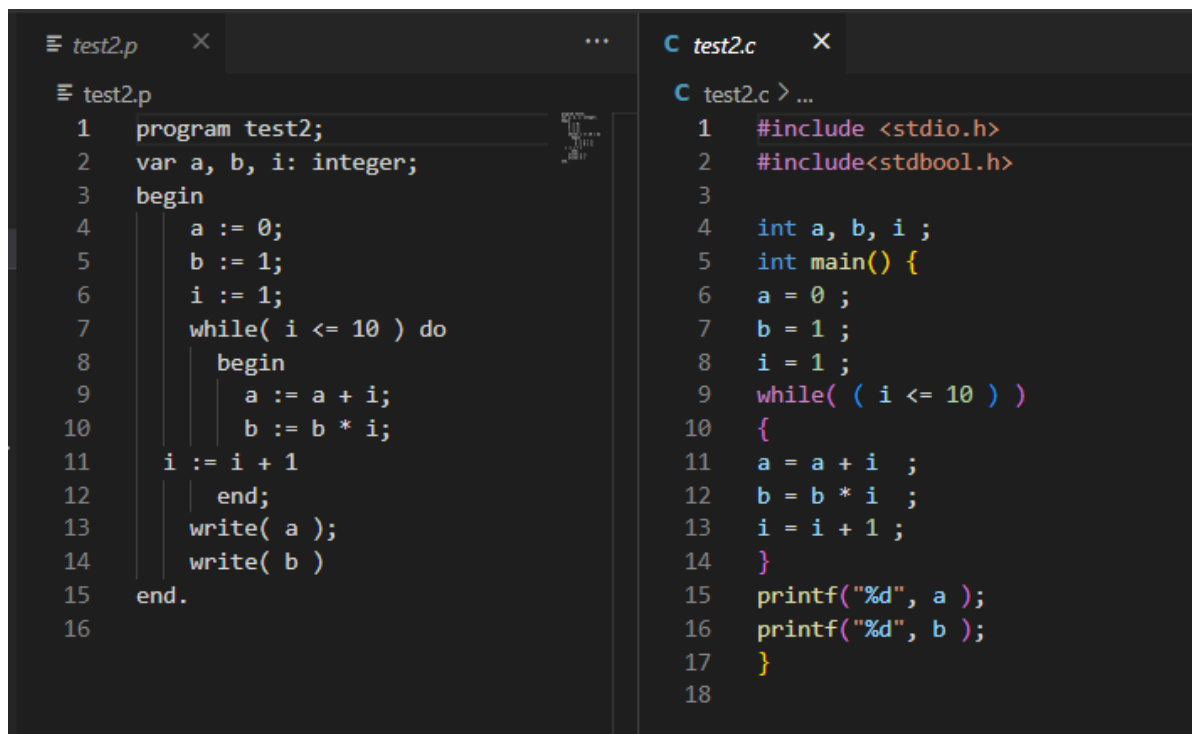


The screenshot shows a code editor with two files open: test1.p and test1.c. The test1.p file contains a Pascal program that declares an integer variable 'a', assigns it the value 3, and prints it. The test1.c file contains the corresponding C implementation, including headers for stdio and stdbool, and a main function that performs the same operations.

```
test1.p
1 program test1;
2 var a: integer;
3 begin
4     a := 3;
5     write( a )
6 end.
7
8

test1.c
1 #include <stdio.h>
2 #include<stdbool.h>
3
4 int a ;
5 int main() {
6     a = 3 ;
7     printf("%d", a );
8 }
9
```

- For test case 2:



The screenshot shows a code editor with two files open: test2.p and test2.c. The test2.p file contains a Pascal program that declares three integer variables 'a', 'b', and 'i'. It initializes 'a' to 0, 'b' to 1, and 'i' to 1. It then enters a while loop that runs as long as 'i' is less than or equal to 10. Inside the loop, 'a' is incremented by 'i', 'b' is multiplied by 'i', and 'i' is incremented by 1. After the loop, it prints the values of 'a' and 'b'. The test2.c file contains the corresponding C implementation, including headers for stdio and stdbool, and a main function that performs the same operations.

```
test2.p
1 program test2;
2 var a, b, i: integer;
3 begin
4     a := 0;
5     b := 1;
6     i := 1;
7     while( i <= 10 ) do
8     begin
9         a := a + i;
10        b := b * i;
11    i := i + 1
12    end;
13    write( a );
14    write( b )
15 end.
16

test2.c
1 #include <stdio.h>
2 #include<stdbool.h>
3
4 int a, b, i ;
5 int main() {
6     a = 0 ;
7     b = 1 ;
8     i = 1 ;
9     while( ( i <= 10 ) )
10    {
11        a = a + i ;
12        b = b * i ;
13        i = i + 1 ;
14    }
15    printf("%d", a );
16    printf("%d", b );
17 }
18
```

- For test case 3:

<pre> ≡ test3.p 1  program test3; 2 3  var a, b, i, x: integer; 4      t, f: Boolean; 5      A: array[1..10] of integer; 6 7  procedure hello; 8  begin 9      write( 1 ) 10 end; 11 12 begin 13     a := 0; 14     b := 1; 15     i := 1; 16     t := true; 17     f := false; 18     hello; 19     read( x ); 20     if ( x &gt; 10 ) then x := 10 21     else if ( x &lt; 1 ) then x := 1; 22     while( i &lt;= x ) do 23     begin 24         a := a + i; 25         b := b * i; 26         A[i] := a + b; 27         write( A[i] ); 28         i := i + 1 29     end; 30     write( a ); 31     write( b ) 32 end. 33 </pre>	<pre> C test3.c &gt; main() 1  #include &lt;stdio.h&gt; 2  #include&lt;stdbool.h&gt; 3 4  int a, b, i, x ; 5  bool t, f ; 6  int A [ 9 ] ; 7  void hello () 8  { 9      printf("%d", 1); 10 } 11 int main() { 12     a = 0 ; 13     b = 1 ; 14     i = 1 ; 15     t = true ; 16     f = false ; 17     hello (); 18     scanf("%d", &amp;x ); 19     if( ( x &gt; 10 ) ) 20     { 21         x = 10 ; 22     }else{ 23         if( ( x &lt; 1 ) ) 24         { 25             x = 1 ; 26         } 27     } 28     while( ( i &lt;= x ) ) 29     { 30         a = a + i ; 31         b = b * i ; 32         A[ i ]= a + b ; 33         printf("%d", A[ i ] ); 34         i = i + 1 ; 35     } 36     printf("%d", a ); 37     printf("%d", b ); 38 } 39 </pre>
--	--

- For test case 4:

<pre> ≡ test4.p  × ≡ test4.p 1  program test4; 2  begin 3      a := 3 4  end. 5 </pre>	<pre> C test4.c 1 × C test4.c &gt; ... 1  #include &lt;stdio.h&gt; 2  #include&lt;stdbool.h&gt; 3 4  int main() { 5      a = 3 ; 6  } 7 </pre>
--	--



- For test case 5:

```

test5.p
1 program test5;
2
3 var a, b, i, x: integer;
4     t, f: Boolean;
5     A: array[1..10] of integer;
6
7 procedure hello;
8 begin
9     write( "Input ?" )
10 end;
11
12 begin
13     a := 0;
14     b := 1;
15     i := 1;
16     t := true;
17     f := false;
18     hello;
19     read( x );
20     if ( x > 10 ) then x := 10
21     else if ( x < 1 ) then x := 1;
22     while( i <= x ) do
23     begin
24         a := a + i;
25         b := b * i;
26         A[i] := a + b;
27         write( A[i] );
28     end;
29     i := i + 1;
30     write( a, b );
31 end.
32
C test5.c
1 #include <stdio.h>
2 #include<stdbool.h>
3
4 int a, b, i, x ;
5 bool t, f ;
6 int A [ 9 ] ;
7 void hello ()
8 {
9     printf("Input ?");
10 }
11 int main() {
12     a = 0 ;
13     b = 1 ;
14     i = 1 ;
15     t = true ;
16     f = false ;
17     hello ();
18     scanf("%d", &x );
19     if( ( x > 10 ) )
20     {
21         x = 10 ;
22     }else{
23         if( ( x < 1 ) )
24         {
25             x = 1 ;
26         }
27     }
28     while( ( i <= x ) )
29     {
30         a = a + i ;
31         b = b * i ;
32         A[ i ]= a + b ;
33         printf("%d", A[ i ]);
34         i = i + 1 ;
35     }
36     printf("%d", a );
37 }
38

```

## 5. Discussion

- We can see from the run result, the generated C code is not that neat as we write the code as usual, here's an example:

```
#include <stdio.h>
#include<stdbool.h>

int a ;
int main() {
a = 3 ;
printf("%d", a );
}
```

And the better coding style for the script would be:

```
#include <stdio.h>
#include<stdbool.h>

int a ;
int main() {
    a = 3 ;
    printf("%d", a );
}
```

To generate a more common C code, we may try to record the current state of the code(where the code is in the whole program structure).

- Actually, the completion of the files is quite complete for the P Grammar, however, for other grammar, the implementation would be more complicated than the current one. For example, if we try to read data with character type(assume the grammar has the character data type), we need to deal with the format specifier by checking the data type of the variable then decide which to be given as a specifier: %c or %d.
- To generate a C program from a P source file, is there any constraint? Take the run result of the test case 4 as an example, the generated C code has compile error since the variable a has NOT been declared, however, the source file in P grammar is correct for P syntax, what if we want to accomplish the goal that the parser can not only transform the code but also add some necessary keywords for the generated code? Like if the parser found that a variable

have not declared, then it try to fix the error by checking the data type of it. The parser would be more general but it's a bit tricky to do that.

## 6. Notes

```
/*STRING \"[^\"]*\"*/  
/*Example: [^0-9]+ means Except 0-9*/
```

```
/**yytext pointing the soruce code of the input program**/  
/**yyval is agreed by Bison and flex if they want to  
communicate with the global variable**/
```

```
//the type for symbols are declare this way  
//sm, wx, nu, sr are just their names
```

```
%union{ pSTM* sm; //statement  
        pEXP* ex; //expression  
        int   nu; //number  
        char* sr; //string  
}  
%type <sm> prog  
%type <sm> block
```

In the p2c.c file, the main function is

```
int main(int argc, char *argv[]) {
```

### #1) Argument Count (ARGC)

This is a non-negative integer argument that holds the number of command line arguments including the program name. Thus if pass a program name is passed then argc will have the value of 1.

### #2) Argument Vector (ARGV)

Argv is an array of character pointers that contains all the command line arguments passed to the main function. If ARGC is greater than zero, then Argv[0] will contain the name of the program. Argv [1] to argv [argc -1] will contain the other command line arguments.

(reference from Internet: <https://www.softwaretestinghelp.com/command-line-arguments-in-cpp/#:~:text=We%20know%20the%20basic%20prototype,arguments%20are%20passed%20to%20it.&text=However%2C%20we%20can%20also%20pass,known%20as%20Command%20Line%20Arguments.>