Programming Language Processor
Assignment 3

XL14608 Thiago Machado da Silva

January 26, 2015

Question 1

To introduce the following do-while statement to PL/0', answer the following questions.

Production rule *statement* → **do** *statement* **while** *condition*

Action A statement '**do** *statement* **while** *condition*' works as follows

1. Execute *statement*.
2. If the value of *condition* is true, go to the step 1. Otherwise, exit this loop.

Question 1-1

To add a token `do` to a set of starting tokens of *statement*, modify a function `isStBeginKey` in `compile.c` and explain the modification in your report.

Line 253 was added, as shown in Figure 1. Now after a "begin", a "do" can be a next token.

```
252  case While: case Write: case WriteLn:
253  case Do:
254    return 1;
255  default:
256    return 0;
```

Figure 1: `isStBeginKey` in `compile.c`.

Question 1-2

Modify a function **statement** in **compile.c** so that your PL/0' compiler can output object codes of Fig.2 for do-while statements. Explain the modification in your report.

label1:	Object codes of *statement*
	Object codes of *condition*
	jpc label2
	jmp label1
label2:	

Figure 2: Object codes for a do-while statement

Now there is a "do" case in statement function. It was derived from the "while" case, with a few changes. The code is shown in Figure 3.

```
case Do: /*A do-while-statement, based on the Write-statement */
  token = nextToken();
  backP2 = nextCode(); /* The target address of the jump at the end of the do-while-statment. */
  statement(); /* A statement (the body of the do-while-statement) */
  token = checkGet(token, While); /* It must be "while". */
  condition(); /* A conditional expression */
  backP = genCodeV(jpc, 0); /* A conditional jump which jumps when the condition is false */
  genCodeV(jmp, backP2); /* A jump to the beginning of the do-while-statement */
  backPatch(backP); /* It adjusts the target address of the conditional jump */
  return;
```

Figure 3: `do-while` case in `statement` in `compile.c`.

Question 1-3

What does your PL/0' compiler outputs when your PL/0' compiler compiles and executes a PL/0' program do.pl0 of Fig. 4?

The output is shown in Figure 5. The left side is the output itself, and the right side is the pl0 code (I decided to show the code again).

```
var x;
begin
   x := 0;
   do begin
      write x;
      writeln;
      x := x + 1
   end
   while x < 3
end.
```

Figure 4: A test program `do.pl0`

```
pl0d/pl0 - [a3c082e...•] » ./pl0d do.pl0
; start compilation
; start execution
0
1
2
pl0d/pl0 - [a3c082e...•] »
```

```
pl0d/pl0 - [a3c082e...•] » cat do.pl0
var x; begin
   x := 0;
   do begin
      write x;
      writeln;
      x := x + 1
end
   while x < 3
end.
pl0d/pl0 - [a3c082e...•] »
```

Figure 5: output from `do.pl0`.

Question 2

Answer the following questions to add the following repeat-until statement to PL/0'.

Production rule *statement* → **repeat** *statement* **until** *condition*

Action A statement '**repeat** *statement* **until** *condition*' works as follows.

1. Execute *statement*.
2. If the value of *condition* is false, go to the step 1. Otherwise, exit this loop.

Question 2-1

Write object codes for the repeat-until statement like object codes for the do-while statement of Fig.2.

label1: Object codes of *statement*
Object codes of *condition*
jpc label1

Figure 6: Object codes for a repeat-until statement.

Question 2-2

Modify `getSource.h` and `getSource.c` to register two tokens `repeat` and `until` to your PL/0' compiler. Explain the modification in your report.

The modification is shown in Figure 7. The left side is the `getSource.h` and the right side is the `getSource.c`. In the `.h` file the change is in line 15 in the "keys" enum, and in the `.c` file the changes are in lines 47 and 48 in "KeyWdT" vector. Those are needed so they can be valid tokens when reading the code.

```
14  While, Do,
15  Repeat, Until,
16  Ret, Func,
```

```
46  {"do", Do},
47  {"repeat", Repeat},
48  {"until", Until},
```

Figure 7: `getSource.h` (left side) and `getSource.c` (right side).

Question 2-3

To add a token `repeat` to a set of starting tokens of *statement*, modify a function `isStBeginKey` in `compile.c` and explain the modification in you report.

Line 253 was modified, as shown in Figure 8. The same reason from Question 1-1 apply here.

```
253  case Do: case Repeat: case Until:
```

Figure 8: `isStBeginKey` in `compile.c`.

Question 2-4

Modify a function `statement` in `compile.c` so that your PL/0' compiler can output object codes for repeat-until statements. Explain the modificaiton in your report.

It's similar to the while or do-while from Question 1-2. Basically, the code executes a statement, and jump to the begginning if the codition is false. The code is shown in Figure 9.

```
    case Repeat: /*A repeat-until-statement, based on the Write-statement */
      token = nextToken();
      backP = nextCode(); /*The target address of the jump at the end of the repeat-statement. */
      statement(); /* A statement (the body of the repeat-until-statement) */
      token = checkGet(token, Until); /* It must be "until". */
      condition(); /* A conditional expression */
      genCodeV(jpc, backP); /* A conditional jump which jumps when the condition is false */
      return;
```

Figure 9: `repeat-until` case in `statement` in `compile.c`.

Question 2-5

What does your PL/0' compiler outputs when your PL/0' compiler compiles and executes a PL/0' program `repeat.pl0` of Fig.10?

```
var x;
begin
   x := 0;
   repeat begin
      write x;
      writeln;
      x := x + 1
   end
   until x=3
end.
```

Figure 10: A test program `repeat.pl0`

The output is shown in Figure 11. The left side is the output itself, and the right side is the pl0 code.

```
pl0d/pl0 - [41411b5...•] » ./pl0d repeat.pl0
; start compilation
; start execution
0
1
2
pl0d/pl0 - [41411b5...•] »
```

```
|pl0d/pl0 - [41411b5...•] » cat repeat.pl0
|var x; begin
|  x := 0;
|  repeat begin
|    write x;
|    writeln;
|    x := x + 1
|  end
|  until x=3
|end.
|pl0d/pl0 - [41411b5...•] »
```

Figure 11: output from `do.pl0`.

Question 3

Answer the following questions to add the following if-then-else statement to PL/0'.

Production rule *statement* $\rightarrow$ **if** *condition* **then** $statement_1$ (**else** $statement_2$ $\mid \epsilon$)

Action A statement '**if** *condition* **then** $statement_1$ (**else** $statement_2$ $\mid \epsilon$)' works as follows.

1. Evaluate *condition*.
2. If the value of *condition* is true, execute $statement_1$.
3. If the value of *condition* is false and $statement_2$ exists, execute $statement_2$.

Description To resolve ambiguity of the grammar of PL/0', we use the following rule.

- When we find an **else**, we relate the **else** to the nearest **then** which has not be related to any **else** yet.

Question 3-1

Write object codes for a statement '**if** *condition* **then** $statement_1$ **else** $statement_2$' like object codes for a do-while statement of Fig.2.

	Object codes of *condition*
	jpc label1
	Object codes of *statement1*
	jmp label2
label1:	If "else": Object codes of *statement2*
label2:	

Figure 12: Object codes for a if-else statement.

Question 3-2

Modify `getSource.h` and `getSource.c` to register a token `else` to your PL/0' compiler. Explain the modification in your report.

The modification is shown in Figure 13, just like Question 2-2. The left side is the `getSource.h` and the right side is the `getSource.c`. In the `.h` file the change is in line 13 in the "keys" enum, and in the `.c` file the changes are in lines 45 in "KeyWdT" vector. Those are needed so it can be a valid token when reading the code.

```
13  If, Then, Else,
```

```
45  {"else", Else},
```

Figure 13: `getSource.h` (left side) and `getSource.c` (right side).

Question 3-3

Modify a function `statement` in `compile.c` so that your PL/0' compiler can output object codes for if-then-else statements. Explain the modificaiton in your report.

It's a modification from the original "if" case. After the execution of the statement1, the execution should jump to the end of the case. Before that, if the condition evaluation is false, the execution should jump to the statement2, if there is a "else" token. The code is shown in Figure 14.

```
case If: /* An if-else-statement */
  token = nextToken();
  condition(); /* A conditional expression */
  token = checkGet(token, Then); /* It must be "then". */
  backP = genCodeV(jpc, 0); /* A conditional jump */
  statement(); /* A statement just after "then" */
  backP2 = genCodeV(jmp, backP2); /* A jump */
  backPatch(backP); /* It adjusts the target address of the conditional jump. */
  if (token.kind == Else) {
    token = nextToken();
    statement(); /* A statement just after "then" */
  }
  backPatch(backP2); /* It adjusts the target address of the conditional jump. */
  return;
```

Figure 14: `if-then-else` case in `statement` in `compile.c`.

Question 3-4

What does your PL/0' compiler outputs when your PL/0' compiler compiles and executes a PL/0' program else.pl0 of Fig.15?

```
var x;
begin
   x := 0;
   while x<3 do begin
      if x < 1 then write 0
      else if x < 2 then write 1
      else write 2;
      writeln;
      x := x+1;
   end;
end.
```

Figure 15: A test program else.pl0

The output is shown in Figure 16. The left side is the output itself, and the right side is the pl0 code.

```
pl0d/pl0 - [64113b0...•] » ./pl0d else.pl0
; start compilation
; start execution
0
1
2
pl0d/pl0 - [64113b0...•] »
```

```
pl0d/pl0 - [64113b0...•] » cat else.pl0
var x; begin
   x := 0;
   while x<3 do begin
      if x < 1 then write 0
      else if x < 2 then write 1
      else write 2;
      writeln;
      x := x+1;
end; end.
pl0d/pl0 - [64113b0...•] »
```

Figure 16: output from `do.pl0`.

Question 4

Answer the following questions to introduce one-dimensional array to PL/0'.

Question 4-1

Explain how to modify the grammar of PL/0' to introduce one-dimensional array to PL/0'.

The grammar needs to identify the brackets when declaring variables or when acessing variables values, when this variable is an array. In array declaration, inside the brackets there is a constant number referrering to the length of this array. In array access, inside the brackets there is an expression that evaluates to a number, the array index.

Question 4-2

Do you need new instructions to the PL/0' virtual machine for one-dimensional array? If you need new instructions, define thier mnemonics and their actions.

Yes, two instructions: ”ldr” and ”str”. ldr refers to ”relative load”, and str to ”relative store”.
ldr pushes to the top of the stack some value residing in another position in the same stack. This position is evaluated using the array ”base position” (this information is with the ldr instruction) and the ”index” (this information should be at the top of the stack).
str is similar to ldr: changes the value from some position in the stack. The new value comes from the top of the stack, and the ”index” comes right below it. The ”index” is relative to the array ”base position” (information contained in the str instruction).
In the stack, the array position works as follows: The base position is referred by the array name (in the nameTable). Every increment in the ”index” decrements the ”absolute position” in the stack. Also the first (zero index) position is not at the same position as the array base, but it's position minus one.

Question 4-3

Modify your PL/0' compiler so that it can support one-dimensional array. Explain the modification in your report.

To register the new instructions the `codegen.h` (line 5 in `codes enum`) and `codegen.c` (lines 113 and 115 in `updateRef` for flagging, 139 and 141 in `printCode` for printing, 201 to 203 and 206 to 209 in `execute` for the actions) files were changed, as shown in Figures 17 and 18.

```
4 typedef enum codes{      /* Constants for operation codes (opecodes) */
5  lit, opr, lod, ldr, sto, str, cal, ret, ict, jmp, jpc
6 }OpCode;
```

Figure 17: `codegen.h`.

```
113   case ldr: flag=2; break;
114   case sto: flag=2; break;
115   case str: flag=2; break;
 NORMAL > PASTE >> <trailing[130] mixed-indent[200]
```

```
139   case ldr: printf("ldr"); flag=2; break;
140   case sto: printf("sto"); flag=2; break;
141   case str: printf("str"); flag=2; break;
 NORMAL > PASTE >> <railing[181] mixed-indent[200]
```

```
201     case ldr:
202          stack[top - 1] = stack[display[i.u.addr.level] + i.u.addr.addr - stack[top - 1] - 1];
203          break;
 NORMAL > PASTE > master > <.c   c < latin1[unix] <  80% : 201:   1 < ! trailing[201] mixed-indent[203]
```

```
206     case str:
207          stack[display[i.u.addr.level] + i.u.addr.addr - stack[top - 2] - 1] = stack[top - 1];
208          top -= 2;
209          break;
```

Figure 18: `codegen.c`.

To register the brackets as valid tokens, there is changes in the `getSource.c` (lines 65 and 66 in `KeyWdT array` and line 108 in `initCharClassT`) and in the `getSource.h` (line 23), as shown in the Figure 19.

```
65   {"[", Lbrackets},
66   {"]", Rbrackets},
67   {"=", Equal},
 NORMAL > PASTE >> <] mixed-indent[204]
```

```
108   charClassT['['] = Lbrackets; charClassT[']'] = Rbrackets;
109   charClassT['='] = Equal; charClassT['<'] = Lss;
110   charClassT['>'] = Gtr; charClassT[','] = Comma;
 NORMAL > PASTE >> <8:  2 < ! trailing[133] mixed-indent[204]
```

```
23  Lbrackets, Rbrackets,
```

Figure 19: `getSource.c` (upper side) and `getSource.h` (down side).

To register arrId (array kind of identifier), `table.h` (line 5) and `table.c` (line 42) files are modified. Also in `table.c` the `enterTarr` function is created (lines 125 to 139), it creates a new array identifier and reserves space in the stack according to the array length. Both files are shown in Figure 20.

```
typedef enum kindT {
  varId, funcId, parId, constId, arrId
}KindT;
NORMAL > PASTE >> <]   <<< ! trailing[10]
```

```
  case funcId: return "func";
  case constId: return "const";
  case arrId: return "array";
NORMAL > PASTE >> <:  1  < ! trailing[84] mixed-indent[129]
```

```
static const *NULLCHAR = "\0";
int enterTarr(char *id, int l)    /* It records a constant and its length in the name table. */
{
  for (int i = 0; i < l; i++) {
    enterT(NULLCHAR);
    nameTable[tIndex].kind = varId;
    nameTable[tIndex].u.raddr.level = level;
    nameTable[tIndex].u.raddr.addr = localAddr++;
  }
  enterT(id);
  nameTable[tIndex].kind = arrId;
  nameTable[tIndex].u.raddr.level = level;
  nameTable[tIndex].u.raddr.addr = localAddr++;
  return tIndex;
}
```

Figure 20: `table.h` (upper-left side) and `table.c` (upper-right and down side).

The `compile.c` file is also modified. There is a modification in `varDecl` function (when `token.kind` equals `Id`, lines 102 to 118, to declare a new array). This is for the array declaration, and a constant number is required between the brackets. See Figure 21.

The next change is in `statement` function (in `case Id`, lines 176 to 192). This is for array access and it supports an expression between the brackets. See Figure 22.

The last one is in `factor` function (added `case arrId`, lines 368 to 374). This is also for array access, and an expression is supported between the brackets. See Figure 23.

```
if (token.kind==Id){
  Token temp = token;
  token = nextToken();
  if (token.kind == Lbrackets) {
    token = nextToken();
    if (token.kind==Num) {
      enterTarr(temp.u.id, token.u.value);
    } else {
      errorType("number");
    }
    token = checkGet(nextToken(), Rbrackets); /* The next token should be "]". */
  }  else {
    setIdKind(varId);
    enterTvar(temp.u.id);
  }
}else
  errorMissingId();
```

Figure 21: `varDecl` in `compile.c`.

```
case Id:
  tIndex = searchT(token.u.id, varId);
  k = kindT(tIndex);
  if (k == arrId) {
    Token temp = token;
    token = nextToken();
    token = checkGet(token, Lbrackets); /* It must be "[". */
    expression();
    token = checkGet(token, Rbrackets); /* It must be "]". */
  } else token = nextToken();
  setIdKind(k);
  if (k != varId && k != parId && k != arrId)
    errorType("var/par/arr");
  token = checkGet(token, Assign);
  expression();
  genCodeT((k == arrId) ? str : sto, tIndex);
  return;
```

Figure 22: `statement` in `compile.c`.

```
case arrId:
  token = nextToken();
  token = checkGet(token, Lbrackets);
  expression();
  token = checkGet(token, Rbrackets);
  genCodeT(ldr, tIndex);
break;
```

Figure 23: `factor` in `compile.c`.

Question 4-4

Write a simple test program `array.pl0` for one-dimensional array. Explain the test program and what your PL/0' compiler outputs when it compiles and executes the test program.

The output is shown in Figure 24. The left side is the output itself, and the right side is the pl0 code.
Line 1: the array is created, it's length equals 2.
Line 2: the variable's first index value is changed (statement's code).
Line 4: the variable's first index value is printed (factor's code).

```
pl0d/pl0 - [master•] » ./pl0d arr.pl0
; start compilation
; start execution
3 4
8 32 %
pl0d/pl0 - [master•] »
```

```
1  var x[2]; begin
2    x[0] := 3;
3    x[1] := 4;
4    write x[0];
5    write x[1];
6    writeln;
7    x[0] := x[0] + x[1] + 1;
8    x[1] := x[0] * x[1];
9    write x[0];
10   write x[1];
11 end.
```

Figure 24: output from `do.pl0`.

Question 5

Answer the following questions to introduce procedures (functions withaout any return values) to PL/0'.

We use the following statement to call a procedure with n arguments.

call *procedure*(arg_1, arg_2, $\ldots$, arg_n)

Question 5-1

Explain how to modify the grammar of PL/0' to introduce procedures to PL/0'.

Question 5-2

Do you need new instructions to the PL/0' virtual machine for procedures? If you need new instructions, define thier mnemonics and their actions.

Question 5-3

Modify your PL/0' compiler so that it can support procedures. Explain the modification in your report.

(Answer)

Question 5-4

Write a simple test program `proc.pl0` for procedures. Explain the test program and what your PL/0' compiler outputs when it compiles and executes the test program.

(Answer)

Question 6

Introduce your own idea to your PL/0' compiler.
(Answer)