Programming Language Processor
Report

slackhideo

February 9, 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.
(Answer)

We add the `case Do:` part to the `isStBeginKey` function because the statement we are adding begin with the `do` keyword.

Question 1-2

Modify a function `statement` in `compile.c` so that your PL/0' compiler can output object codes of Fig.1 for do-while statements. Explain the modificaiton in your report.

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

Figure 1: Object codes for a do-while statement

(Answer)

We include the following code to `statement` function:

```
case Do:                            /* do-while statement */
    token = nextToken();            /* gets the next token */
    backP2 = nextCode();            /* target address for the jump at the end */
    statement();                    /* a statement */
    token = checkGet(token, While); /* next token must be "while" */
    condition();                    /* a condition */
    backP = genCodeV(jpc, 0);       /* a conditional jump to the end */
    genCodeV(jmp, backP2);          /* a jump to the beginning of do-while
                                                                statement */
    backPatch(backP);               /* adjusts the jpc target address */
    return;
```

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. 2?

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

Figure 2: A test program `do.pl0`

(Answer)

It outputs:

```
; start compilation
; start execution
0
1
2
```

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.1.
(Answer)

```
label1:   Object code of statement
          Object code of condition
          jpc label1
```

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.
(Answer)

In `getSource.h` we add the following line;

```
Repeat, Until,
```

in the `typedef enum keys KeyId` block. And in `getSource.c`, we add:

```
{"repeat", Repeat},
{"until", Until},
```

to add `repeat` and `until` as reserved words and make the compiler recognize them.

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.

(Answer)

We add the `case Repeat:` part to the `isStBeginKey` function because the statement we are adding begin with the `Repeat` keyword.

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.

(Answer)

We include the following code to `statement` function:

```
case Repeat:                        /* repeat-until statement */
    token = nextToken();            /* gets the next token */
    backP = nextCode();             /* target address for the jump at the end */
    statement();                    /* a statement */
    token = checkGet(token, Until); /* next token must be "until" */
    condition();                    /* a condition */
    genCodeV(jpc, backP);           /* a conditional jump to the beginning of
                                                     repeat-until statement */
    return;
```

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.3?

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

Figure 3: A test program `repeat.pl0`

(Answer)

It outputs:

```
; start compilation
; start execution
0
1
2
```

Question 3

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

Production rule *statement* → **if** *condition* **then** $statement_1$ (**else** $statement_2$ | ϵ)

Action A statement '**if** *condition* **then** $statement_1$ (**else** $statement_2$ | ϵ)' 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.1.

(Answer)

```
         Object code of condition
         jpc label1
         Object code of statement1
         jmp label2
label1:  Object code of statement2
label2:
```

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.

(Answer)

In `getSource.h` we add the following line;

```
Else,
```

in the `typedef enum keys KeyId` block. And in `getSource.c`, we add:

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

to add `else` as reserved words and make the compiler recognize them.

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.

(Answer)

We include some code in the `case If` part of `statement` function, as follows:

```
case If:                           /* if-then-else statement */
    token = nextToken();           /* gets the next token */
    condition();                   /* a conditional expression */
    token = checkGet(token, Then); /* next token must be "then" */
    backP = genCodeV(jpc, 0);      /* a conditional jump (to the end if it is
                                      an if-then statement, or to the second
                                      statement if it is an if-then-else
                                                                  statement */

    statement();                   /* a statement just after "then" */
    if(token.kind == Else) {       /* verifies if it is an if-then-else
                                                                  statement */
        token = nextToken();       /* gets the next token */
        backP2 = genCodeV(jmp, 0); /* a jump to the end of the statement */
        backPatch(backP);          /* adjusts the jpc target address */
        statement();               /* a statement after "else" */
        backPatch(backP2);         /* adjusts the jmp target address */
    }
    else {
        backPatch(backP);          /* adjusts the jpc target address */
    }
    return;
```

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.4?

```
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 4: A test program else.pl0

(Answer)

It outputs:

```
; start compilation
; start execution
0
1
2
```

Question 4

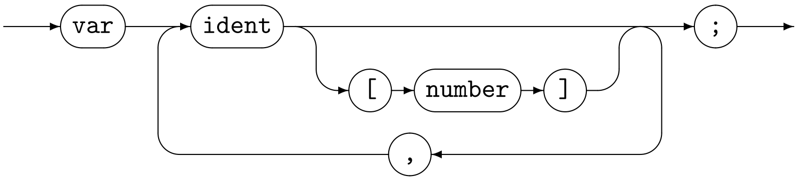
Answer the following questions to introduce one-dimensional array to PL/0'.

Question 4-1

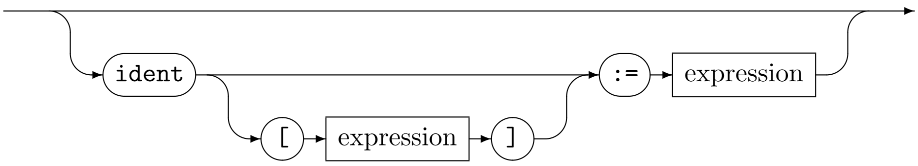
Explain how to modify the grammar of PL/0' to introduce one-dimensional array to PL/0'.
(Answer)

In order to add one-dimensional array feature in PL/0', we can do the following changes:

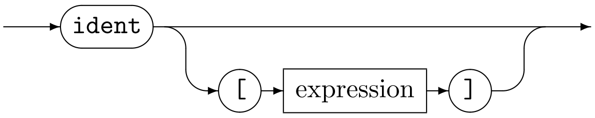
varDecl



statement



factor



The *varDecl* modification is needed to declare arrays. It accepts an `ident` as an identifier and an `number` as the size.
The *statement* modification lets the array (an element of the array) to receive a value.
The *factor* modification is for getting a value from the array (from a specified position of the array).
Please note that are only shown the modified parts. The rest of *statement* diagram, *factor* diagram and the other needed diagrams are identical to the original ones.

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.

(Answer)

Yes, I needed to add two more instructions to the PL/0' virtual machine. They are:

lot

Overview

Push a value of a variable (especially useful for arrays) to the stack considering the value of stack top.

Mnemonic

```
lot,LEVEL,ADDR
```

Description

LEVEL is a nesting level in a source PL/0' program. ADDR is a relative address.

Details

```
top--;
stack[top] = stack[display[LEVEL] + ADDR + stack[top]];
top++;
```

stt

Overview

Store a value on the stack top to a variable on the stack, considering the value below of stack top.

Mnemonic

```
stt,LEVEL,ADDR
```

Description

LEVEL is a nesting level in a source PL/0' program. ADDR is a relative address.

Details

```
stack[display[LEVEL] + ADDR + stack[top - 2]] = stack[--top];
```

Question 4-3

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

(Answer)

In addition to the new instructions added in `codegen.h` and `codegen.c`, I also modified the `compile.c` file:

- In `varDecl` function, I introduced the array declaration method described in Question 4-1, getting the array size as a number (I tried to use `expression`, but determining the starting addresses of arrays dynamically is quite complex). To store the array in the name table, I use a function `enterTarray`, which is similar to `enterTvar` but uses the array size as a parameter.
- In `statement` function, I introduced the value assignment method, as described in Question 4-1. Here I use the `stt` instruction.
- In `factor` function, I introduced the value retrieval method, as described in Question 4-1. Here I use the `lot` instruction.

In `getSource.h` file, I registered `Lbrack` and `Rbrack` in the `typedef enum keys KeyId`.

In `getSource.c` file, I added the following lines to `struct keyWd KeyWdT[]`:

```
{"[", Lbrack}, /* added left square brackets (used in array declaration) */
{"]", Rbrack}, /* added right square brackets (used in array declaration) */
```

and the following line to `initCharClassT` function:

```
charClassT['['] = Lbrack; charClassT[']'] = Rbrack;
```

Also, I added support for printing the arrays in HTML (although they are shown in HTML files as `varId`, I included a kind `arrayId`, used in internals).

In `table.h` file, I included the kind `arrayId`.

In `table.c` file, I included the function `enterTarray`, which registers an array into the name table.

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.

(Answer)

A simple test program is:

```
var small[2], another[3];
begin
    small[0] := 5 + 7;
    small[1] := small[0];
    another[4 / 2] := 2 * small[1];
    write small[1];
    writeln;
    write another[1 + 1];
    writeln
end.
```

This test program exercises creation of multiple arrays, assignment of expressions of numbers, expressions using arrays and expressions to calculate the index.
My PL/0' compiler outputs:

```
; start compilation
; start execution
12
24
```

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)

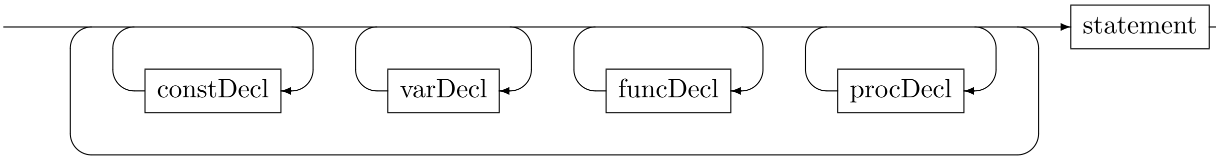
Question 5-1

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

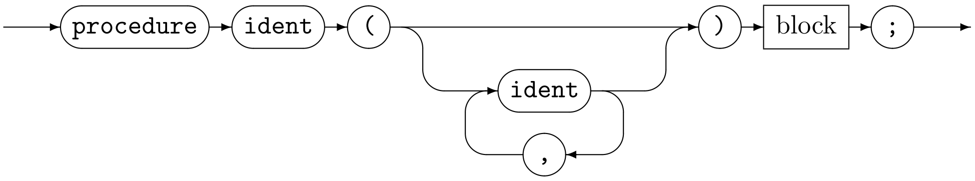
(Answer)

In order to add procedure declarations and procedure calls features in PL/0', we can do the following changes:

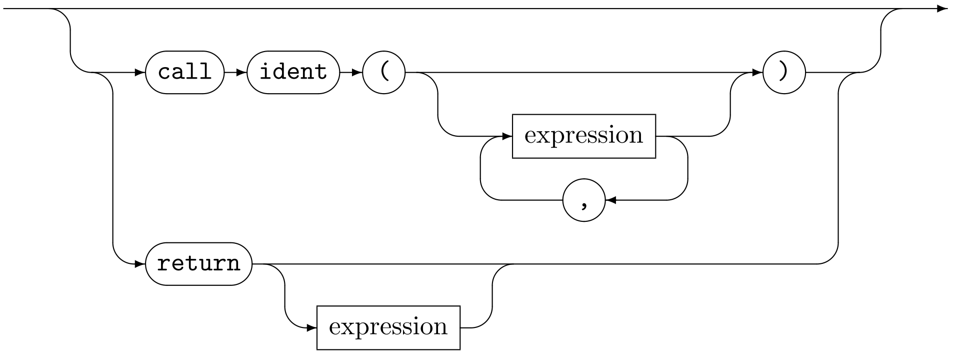
block



procDecl



statement



The *block* modification is needed to declare procedures. It adds a `procDecl` (procedure declaration) section.
The *procDecl* is similar to *funcDecl*, but it uses the `procedure` keyword.
The *statement* introduces the `call` statement, which accepts the `call` keyword, then an `ident` as the procedure identifier and then the parameters, if there is any. It makes possible to call procedures. Also, `return` statement was modified to accept an `expression` or not.
Please note that are only shown the modified parts. The other sections are the same as the original diagram.

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.
(Answer)

No, I did not need any new instructions.

Question 5-3

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

I did the following modifications:

- In `getSource.h` file, I added `Proc` and `Call` in the `typedef enum keys KeyId`.
- In `getSource.c` file, I added {`"procedure", Proc`} and {`"call", Call`} entries in the `struct keyWd KeyWdT[]`. Also, I added support for printing the procedures in HTML.
- In `table.h` file, I included the kind `procId`.
- In `table.c` file, I included the function `enterTproc`, which registers a procedure into the name table.
- In `compile.c` file, I did the following changes:
  - In `block` function, I included the `procDecl()` function call, as shown in the *block* syntax diagram of Question 5-1.
  - Added the `procDecl()` function, which compiles procedure declarations, as shown in the *procDecl* syntax diagram of Question 5-1.

- In `statement` function, I modified the `return` statement to accept an `expression` or no return value (used in the case of procedures). Also, I introduced the `call` statement, as shown in the *statement* syntax diagram of Question 5-1.

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)

A simple test program is:

```
var primes[5];

procedure search(x, size)
begin
    if size = 0 then
        return;

    if primes[size - 1] = x then
        begin
            write size - 1;
            writeln
        end
    else
        call search(x, size - 1)
end;


begin
    primes[0] := 2;
    primes[1] := 3;
    primes[2] := 5;
    primes[3] := 7;
    primes[4] := 11;
    call search(7,5);
end.
```

This test program exercises procedures using multiple parameters and recursive procedure calls. The procedure takes a number to search for and an array size and prints the array index of the number if it is in the array, otherwise it does nothing.

My PL/0' compiler outputs:

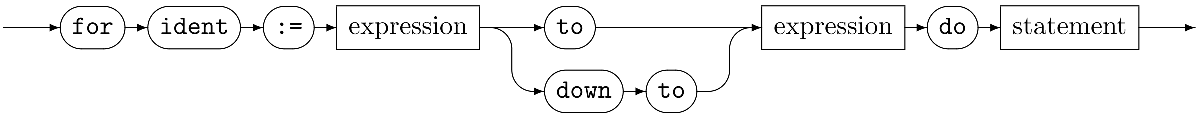
```
; start compilation
; start execution
3
```

Question 6

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

I introduced the *for-do statement* into my PL/0' compiler. The corresponding syntax diagram is as follows:

statement



The object code for *for-do statement* is:

```
         Object code for the variable initialisation
label1:  Object code for the condition
         jpc label2
         Object code of statement
         Object code for updating the "for variable"
         jmp label1
label2:
```

I introduced two more instructions to the PL/0' virtual machine just to make the code cleaner and more concise. They are:

uad

Overview
Increment by one the value of a variable. It is a conjunction of a lod, a lit, an opr,add and a sto instructions.

Mnemonic
`uad,LEVEL,ADDR`

Description
LEVEL is a nesting level in a source PL/0' program. ADDR is a relative address.

Details

```
stack[top++] = stack[display[i.u.addr.level] + i.u.addr.addr];
stack[top] = 1;
stack[top-1] += stack[top];
stack[display[i.u.addr.level] + i.u.addr.addr] = stack[--top];
```

usb

Overview

Decrement by one the value of a variable. It is a conjunction of a lod, a lit, an opr,sub and a sto instructions.

Mnemonic

```
usb,LEVEL,ADDR
```

Description

LEVEL is a nesting level in a source PL/0' program. ADDR is a relative address.

Details

```
stack[top++] = stack[display[i.u.addr.level] + i.u.addr.addr];
stack[top] = 1;
stack[top-1] -= stack[top];
stack[display[i.u.addr.level] + i.u.addr.addr] = stack[--top];
```

To implement the *for-do statement*, I did the following modifications:

- In `codegen.h` file, I added `uad` and `usb` instructions in the `typedef enum codes OpCode`.
- In `codegen.c` file, I implemented the new instructions's behaviour in the `execute` function.
- In `getSource.h` file, I added `For`, `Down` and `To` keywords in the `typedef enum keys KeyId`.
- In `getSource.c` file, I added {`"for", For`}, {`"down", Down`} and {`"to", To`} entries in the `struct keyWd KeyWdT[]`.
- In `compile.c` file, I modified the `statement` function, including a For case in the main switch, implementing the compilation of the *for-do statement*, supporting both incrementing and decrementing for loops. This implementation follows the syntax diagram shown above and the object code. It also uses the two new instructions `uad` (for incrementing) and `usb` (for decrementing) for updating the value of the for loop variable. And the `For` token was added in the `isStBeginKey` function as a `statement` beginning token.

A simple test program is:

```
var i, a[10];

begin
    for i := 0 to 9 do
        a[i] := 2 * (i + 1);

    for i := 0 to 9 do begin
        write a[i];
        writeln
    end;

    writeln;

    for i := 9 down to 0 do begin
        write a[i];
        writeln
    end
end.
```

This test program exercises *for-do statement* in both incrementing and decrementing versions, as well its combination with arrays.

My PL/0' compiler outputs:

```
; start compilation
; start execution
2
4
6
8
10
12
14
16
18
20

20
18
16
14
12
10
8
6
4
2
```