Compilers Assignment

Report

This report presents the two parts of the assignment for Compilers. Below there is the descriprion of how each task was solved, as well as four test cases for each of the two constructs. Moreover, the appendices show the detailed changes made (sample.diff).

1. Task 1

The task was to implement the continue statement, which modifies the behavior of the loops. This report describes briefly the changes made to the compiler and the new test cases that were written.

1.1 Abstract syntax

A representation of the new kind of statement needs to be added to the type of abstract syntax tree. In the file tree.mli:

```
type stmt_guts = ...
        | ContinueStmt
```

The same changes need to be done in the file tree.ml and the pretty-printer for abstract syntax trees in the same file was also extended to also print the new type of statement. Note that the continue statement does not take any parameters. This is because the statement has the same behaviour regardless of the context in which it is used.

1.2 Concrete syntax

The lexer and parser are easily modified to implement the new construct. For the lexer, all that is needed is to add the “continue” keyword in the hash table used to look up each identifier.

```
let symtable = Util.make_hash 100 [ ...; ("continue", CONTINUE); ... ]
```

In the parser we add CONTINUE as a new token type.

```
%token                  CONTINUE
```

We also need to add the new type of statement, in order to allow the production of the appropriate syntax tree.

```
stmt1: ...
        | CONTINUE                  { ContinueStmt };
```

1.3 Semantic checks

Following the patterns of other constructs, such as Skip, the ContinueStmt does not have to check any patterns, as it does not take any other parameters. In comparison, the while statement needs to check the patterns for both the body and the condition.

```
(∗ | check_stmt| – check and annotate a statement ∗)
```

```
let rec check_stmt s env alloc =
        err_line := s.s_line;
        match s.s_guts with ...
                | ContinueStmt -> ()
```

1.4 Translation

The continue construct can be implemented by jumping code that ends the current iteration of the loop and triggers the evaluation of the Boolean condition which controls the loop. For for loops, the loop variables are incremented before the condition is checked. However, this is also the case when the continue statements are not implemented. To make the implementation of the ContinueStmt easier, we need to pass an extra parameter to the function gen_stmt, in order to keep track of where exactly in the code sequence we need to jump. This will be a label parameter, which is going to be passed around by all functions apart from CaseStmt, which will jump to that respective label.

```
(∗ |gen_stmt| – generate code for a statement ∗)
let rec gen_stmt s j_lab = match s.s_guts with ...
        | ContinueStmt ->
                 if j_lab = !retlab then failwith "continue statement outside of loop"
                else <JUMP j_lab>
```

For each loop, I created a new label right before the condition was checked. The gen_stmt for body was then called like this : gen_smt body l3 , where l3 is the label right before the condition check. The rest of the code remains unchanged.

1.5 Test cases

All the existing test cases continue to pass. In addition, four tests for the new construct are provided:

- A for loop which uses continue (continue1.p)
- A test for nested loops that uses a for and a while loop (continue2.p)
- A continue statement which doesn't affect a repeat… until loop (continue3.p)
- A test where continue is outside of any loops (continue4.p)

In each case, compiler output embedded in the test case shows that good code is generated. The code shows that continue statements interact well with the rest of the program and the continue statement behaves as expected.

continue1.p

```
(* Check that the continue construct works properly on for loops  *)
var i : integer;

begin
        for i := 1 to 5 do
                if i = 3 then
                        continue
                end;
                print_num(i);
                newline()
        end
end.

(*<<
1
2
4
5
>>*)
```

```
(*[[
@ picoPascal compiler output
        .include "fixup.s"
        .global pmain

        .text
pmain:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@   for i := 1 to 5 do
        mov r0, #1
        set r1, _i
        str r0, [r1]
        mov r4, #5
.L2:
        set r0, _i
        ldr r5, [r0]
        cmp r5, r4
        bgt .L1
@     if i = 3 then
                cmp r5, #3
                beq .L4
@     print_num(i);
        mov r0, r5
        bl print_num
@     newline()
        bl newline
.L4:
        set r5, _i
        ldr r0, [r5]
        add r0, r0, #1
        str r0, [r5]
        b .L2
.L1:
        ldmfd fp, {r4-r10, fp, sp, pc}
        .ltorg

        .comm _i, 4, 4
@ End
]]*)
```

continue2.p

```
var i : integer;

proc nested (x: integer);
var j : integer;
begin
   for x := 1 to 5 do
    j := 0;
    while j < 5 do
        j := j+1;
        if (x mod 2 = 0) and (j < 5) then
          continue
        end;
        print_num(x)
    end;
    newline()
  end
end;

begin
  nested(i)
end.

(*<<
11111
2
33333
4
55555
>>*)

(*[[
@ picoPascal compiler output
        .include "fixup.s"
        .global pmain

@ proc nested (x: integer);
        .text
_nested:
        mov ip, sp
        stmfd sp!, {r0-r1}
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@    for x := 1 to 5 do
        mov r0, #1
        str r0, [fp, #40]
        mov r5, #5
.L2:
        ldr r0, [fp, #40]
        cmp r0, r5
        bgt .L1
@     j := 0;
        mov r4, #0
.L5:
@     while j < 5 do
        cmp r4, #5
        bge .L7
@       j := j+1;
        add r4, r4, #1
@       if (x mod 2 = 0) and (j < 5) then
        mov r1, #2
        ldr r0, [fp, #40]
        bl int_mod
        cmp r0, #0
        bne .L10
        cmp r4, #5
        blt .L5
.L10:
@       print_num(x)
        ldr r0, [fp, #40]
        bl print_num
        b .L5
.L7:
@     newline()
        bl newline
        ldr r0, [fp, #40]
        add r0, r0, #1
        str r0, [fp, #40]
        b .L2
.L1:
        ldmfd fp, {r4-r10, fp, sp, pc}
        .ltorg

pmain:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@   nested(i)
        set r0, _i
        ldr r0, [r0]
        bl _nested
        ldmfd fp, {r4-r10, fp, sp, pc}
        .ltorg

        .comm _i, 4, 4
@ End
]]*)
```

continue3.p

```
var x : integer;

begin
  x := 10;
  repeat
        x := x-1;
        if x mod 10 = 0 then continue end;
        print_num(x)
  until x = 1;
  newline()
end.

(*<<
987654321
>>*)
```

```
(*[[
@ picoPascal compiler output
        .include "fixup.s"
        .global pmain

        .text
pmain:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@   x := 10;
        mov r0, #10
        set r1, _x
        str r0, [r1]
.L2:
@       x := x-1;
        set r4, _x
        ldr r0, [r4]
        sub r5, r0, #1
        str r5, [r4]
@       if x mod 10 = 0 then continue end;
        mov r1, #10
        mov r0, r5
        bl int_mod
        cmp r0, #0
        beq .L4
@       print_num(x)
        ldr r0, [r4]
        bl print_num
.L4:
        set r0, _x
        ldr r0, [r0]
        cmp r0, #1
        bne .L2
@   newline()
        bl newline
        ldmfd fp, {r4-r10, fp, sp, pc}
        .ltorg

        .comm _x, 4, 4
@ End
]]*)
```

continue4.p

```
(*
This test gives back the following error:
*** Test continue4.p
./ppc -O2 test/continue4.p >b.s
Fatal error: exception Failure("continue statement outside of loop")
Makefile:56: recipe for target 'test0-continue4' failed
make: *** [test0-continue4] Error 2
*)

var x : integer;

begin
  x := 2;
  continue;
  print_num(x)
end.

(*<<
2
>>*)
```

2. Task 2

The task was to implement by-name parameters, by adding the symbol “=>” which will allow the user to call a parameter by name instead of by value. This way, the parameter will be evaluated once for every call of the procedure.

2.1 Abstract syntax

A representation of the new kind of parameter needs to be added to the dictionary. Therefore, the files dict.ml and dict.mli need a new type of def_kind, for the new kind o parameter. Let’s call it NParamDef and this will denote the parameters given by name.

```
(* |def_kind| -- kinds of definition *)
type def_kind =
   NParamDef of int            (* Name parameter *)
```

Once this statement is added to both dict.ml and dict.mli, the syntax tree can be created with the new type of parameter.

2.2 Concrete syntax

The lexer and parser are easily modified to implement the new type of parameter. For the lexer, we need to add “=>” in the list of tokens (call it EQUIV) :

```
rule token =
  parse
            | "=>"            { EQUIV }
```

This will allow the lexer to recognize the definition of a by-name parameter and apply the respective rules on it. In the parser we add EQUIV as a new punctuation token.

```
%token          EQUIV
```

We also need to add a new type of formal declaration in the parser, so the compiler knows what rules to follow when such a declaration is made in the program. This will ensure the use of the VarDecl declaration type, using a new case, the case for NParamDef.

```
formal_decl :
   | EQUIV ident_list COLON typexpr      { VarDecl (NParamDef, $2, $4) }
```

2.3 Semantic checks

Following the pattern for other variable declarations, such as VParamDef, the semantic analyser just needs to check if the argument in the declaration are valid. Therefore, we just need to add a guard for NParamDef which has the same behaviour as the others. However, the by-name parameters should work only for integer, so another if-statement is needed for this check. Also, we need to add a guard fot has_value and for check_var that recognizes the new parameter type.

```
and check_arg formal arg env =
  match formal.d_kind with
      CParamDef | VParamDef | NParamDef ->
        let t1 = check_expr arg env in
        if not (same_type formal.d_type t1) then
          sem_error "argument has wrong type" [];
        if formal.d_kind = VParamDef then
          check_var arg true;
       if formal.d_kind = NParamDef then
          if not (same_type formal.d_type integer) then
                sem_error "by-name parameters must be integers" [];
          check_var arg true
```

2.4 Translation

In order to implement the by-name parameters in the compiler, the translator needs to create a procedure containing a sequence of instructions. These, when called, need to compute the address and/or the value of the expression in the argument. This implementation will be very similar to that of a parametric procedure. The argument needs to store the address of the new (created) procedure and the environment in which this procedure has been created (so create a closure). Therefore, if we have a procedure let add1 x = x+1 in add1(1+2), where x is a by-name variable, the compiler will have to turn this into a separate function, with a behaviour similar to : let add1 x = x() + 1 in add1 (fun () -> 1+2). Therefore, we need to add a guard in gen_addr, gen_closure and gen_args that deals with the new type of parameter:

```
(* |gen_addr| -- code for the address of a variable *)
let rec gen_addr v =
  match v.e_guts with
      Variable x ->
        let d = get_def x in
        begin
          match d.d_kind with…
            | NParamDef ->
                <LOADW, address d>

(* |gen_closure| -- two trees for a (code, envt) pair *)
let gen_closure d =
  match d.d_kind with …
   | NParamDef ->
        (<LOADW, address d>,
          <LOADW, <OFFSET, address d, <CONST addr_size>>>)

(* |gen_arg| -- generate code for a procedure argument *)
and gen_arg f a =
  match f.d_kind with …
    | NParamDef ->
```

```
begin
  match a.e_guts with
        Variable x ->
         let (proc, env) = gen_closure (get_def x) in [proc; env]
    | _ -> [gen_expr a]
end
```

This will make sure the code is evaluated as a function which modifies the environment, which is then used to evaluate the procedure further.

2.5 Tests

All existing tests continue to pass. In addition, I've added the following tests to check that the new construct works properly. The tests are shown in the next few pages.

- The example test given in the assignment (byname1.p)
- Defining a constant function in the call-by-name style (byname2.p)
- A test for swapping two numbers, used on an array (byname3.p)
- A test which takes a parameter of type string (not an integer) (byname4.p)

byname1.p

```
var g: integer;

proc println (x:integer);
begin
  print_num(x);
  newline()
end;

proc p (=> x:integer): integer;
begin
  g := g+1;
  return x+x
end;

begin
  g:=0;
  println(p(2+3));
 println(p(g));
 println(p(p(7)));
 println(g);
end.

(*<<
10
4
28
5
>>*)
```

byname2.p

```
(*
The second print statement gives back
an error as you cannot assign a
constant to another constant=> this
returns an "invalid assignment" error
*)
var x: integer;

proc constant(=> y:integer): integer;
begin
   y := 1;
   return y
end;

begin
   x:=0;
   print_num(constant(x));
  newline();
  print_num(constant(2))
end.

(*<<
1
error
>>*)
```

byname3.p

```
(*
This test shows how defining a swap function
using by-name parameters will have a different
behaviour than the one we are expecting:
instead of swapping the values of i and a[i], i is
reevaluated in the swap function. Therefore the
modified value is the one of a[a[i]] rather than
a[i].
*)
var a: array 4 of integer;
var i: integer;

proc swap (=> a:integer; => b:integer);
var temp; integer;
begin
  temp := a;
  a :=b;
 b := temp
end;

begin
  a[0] := 1; a[1] := 2;
 a[2] := 3; a[3] := 0;
i := 0;
swap(i, a[i]);
print_num(i); newline();
print_num(a[0]); newline();
print_num(a[1]);
end.

(*<<
1
1
0
>>*)
```

byname4.p

```
(*
Thist test creates a procedure that uses a by-name
parameter of type string (other than integer).
Therefore, the compiler will give back the following
semantic error :
*** Test byname4.p
./ppc -O2 test/byname4.p >b.s
"test/byname4.p", line 10: by-name parameters must
be integers
Makefile:56: recipe for target 'test0-byname4' failed
make: *** [test0-byname4] Error 1
*)
var s : boolean;

proc isTrue (=> x:boolean ): boolean;
begin
   return x
end;

begin
  s = true;
  s = isTrue(s)
end.
```

3. References

- Sample Assignment Report: https://www.cs.ox.ac.uk/teaching/materials18-19/com/sample.pdf
- Compilers Coursebook by Mike Spivey:https://www.cs.ox.ac.uk/teaching/materials18-19/com/book.pdf
- A W Appel, Modern Compiler Implementation in ML, Cambridge University Press.
- A V Aho, R Sethi and J D Ullman, Compilers: principles, techniques and tools, 3rd edition, Addison-Wesley.