University of Oxford
Department of Computer Science

Compilers

Michaelmas Term 2018

Your report on this assignment should be submitted to the Examination Schools, High Street by 12 noon on Friday, 25th January 2019. The assignment is worth 35 marks out of a total of 100 marks for the course.

This assignment is based on the compiler from a Pascal subset to code for the ARM that was the subject of Lab 4 in the course. It asks you to make two extensions, which are specified later in this document.

To carry out the assignment, you will need access to a Unix environment containing version control tools, an OCaml compiler, software tools that enable cross-compiling for the ARM, an ARM emulator, and other support software. Hints and resources for setting up such an environment are provided on the course web page. Lab machines are also available for remote access.

A fresh copy of the practical materials for the course may be obtained (in a directory named `task`) by cloning the Mercurial repository supplied for the lab exercises:

```
$ hg clone http://spivey.oriel.ox.ac.uk/hg/compilers task
```

Candidates are required to submit a written report, describing the changes they made to the compiler in order to support the extensions, and the provisions they have made for testing the modified compiler.

A typical report will consist of five to ten pages of text with embedded code fragments, plus listings of the compiler changes and additional test cases. A sample report for a different exercise is provided on the course web page (`https://www.cs.ox.ac.uk/teaching/materials18-19/com/sample.pdf`) and shows the suggested structure, with a narrative account of the compiler changes stage by stage, followed by difference listings generated with a version control system, and test cases in the format of the existing tests for Lab 4, showing the compiler input, the expected output, and the assembly language code. Candidates are not required or permitted to submit the code of their implementation in machine-readable form.

Your attention is drawn to the University's policy on Plagiarism set out in Appendix A of each Course Handbook and on the University's website. The work submitted for this assignment should be entirely your own and not done in collusion with others. You may make use of written and online sources, but these should be acknowledged. There is no need to acknowledge help given by demonstrating staff during the lab sessions.

The remainder of the document sets out the two tasks that constitute the assignment.

Task 1: Control flow modifier

The first task concerns the implementation of the `continue` statement, which modifies the behaviour of loops. Note that the language of Lab 4 features three loop statements (`for`, `repeat` and `while`). Each of them counts as a loop for the purpose of the assignment.

The semantics of `continue` is described below.

- `continue` ends the current iteration of the innermost enclosing loop.
- Next it triggers the evaluation of the boolean condition that controls the loop. For `for` loops, the loop variable is incremented before the condition is checked (as is the case when the body finishes normally, without `continue`).

Note that the above specification refers to the *innermost enclosing loop*. If the input program contains an occcurence of `continue` that is not inside the body of a loop, the compiler should report an error.

The goal of the task is to add `continue` to the language as a new kind of *statement* and ensure that the compiler will generate code conforming to the above specification.

Task 2: By-name parameters

Recall that if a procedure features a parameter `x` declared using `x:integer` as shown below

```
proc p(..., x:integer,...):type;
```

then, whenever the procedure is called, its argument will be evaluated once at the point of call and the resultant value will be passed to the procedure. In particular, the argument will be evaluated even when the value of the argument might not be needed. Such parameters are called *by-value parameters*.

This task is about defining an alternative parameter-passing mechanism, which will be distinguished by writing `=> x:integer` instead of `x:integer`.

- Parameters declared in this way are *not* evaluated at the point of function call. Instead, they are evaluated only when needed inside the body of the procedure.
- Each use within the procedure body triggers a separate evaluation of the argument, i.e. values obtained earlier are not saved for subsequent evaluations. Consequently, the argument is evaluated as many times as it is needed inside the procedure body.
- In declarations, the use of `=>` should be governed by the same rules as `var` declarations. In particular, `=>` can be followed by a list of identifiers.
- **The assignment asks you to implement the new mechanism for the type `integer` only.** The use of `=>` for any other type should result in a syntax error.
- Procedures that exploit the new kind of parameter passing may *not* be used as arguments to other procedures. This constraint should be enforced by the compiler.

Example

Consider the code below.

```
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;

  (* evaluates 2+3 twice and prints out 10 *)
  println(p(2+3));
  (* g=1 at this point *)

  (* when p needs the value of g, it will be equal to 2, so p will return 4 *)
  println(p(g));
  (* g=2 at this point *)

  (* 28 will be printed out *)
  println(p(p(7)));
  (* g=5 at this point *)

  println(g)
end.
```

- The first call to `p` (with argument `2+3`) will transfer control to the body of `p` without evaluating `2+3` right away. `g` will be incremented next and then `p` will need the value of `x` twice. In each case, `2+3` needs to be evaluated, yielding `5` on each occasion. Consequently, the call will produce `10` and the value of `g` after the call will be `1`.
- Similarly, after the second call (with argument `g`) the argument `g` will not be evaluated right away. Before that, its value will change to `2` due to the statement `g:=g+1` inside `p`. Consequently, when the value is needed, it will be equal to `2` and `p` will return `4` as a result.

- In the next call, `p` will change the value of `g` to `3` and will need to evaluate `p(7)` twice. The first evaluation of `p(7)` will change the value of `g` to 4 and return 14 as a result (evaluating `7` twice). Similarly, the second evaluation of `p(7)` will change `g` to `5` and return `14` too. 28 will be printed out and the value of `g` will reach `5`.
- In the final call, the current value of `g` is printed out. Note that, in our example, it corresponds to the number of times that `p` was called.

Observe that the results would be somewhat different without `=>`. The first call would produce the same result `10` and set `g` to `1`. The second call, however, would produce `2`, because `g` would be evaluated at the point of function call (when its value is still `1`). After the second call, the value of `g` would also be `2`, though.

The third call would also generate `28`. However, `g` would be `4` afterwards, because `p(7)` would be evaluated only once. Accordingly, the last print statement would print `4` instead.

Assessment

Partial marks will be given for implementing only some aspects of the assignment or generating code that is significantly inefficient or overlong. To gain full marks, your submission should include test cases that illustrate the completeness of your implementation and the quality of the code it generates. The following mark scheme will be used, giving a maximum total of 35 marks.

- A basic implementation of Task 1 with simple test cases showing the code.

  (10 marks)

- A basic implementation of Task 2 with simple test cases showing the code.

  (10 marks)

- Tidy object code of reasonable efficiency, demonstrated by test cases.

  (5 marks)

- Further test cases that demonstrate subtle aspects of behaviour for `continue` and by-name parameters.

  (5 marks)

- A written report with good clarity and presentation.

  (5 marks)