University of Oxford

Department of Computer Science

Compilers – Practical Assignment

December 2017

Your report on this assignment should be submitted to the Examination Schools, High Street by 12 noon on Friday, 26th January. 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 replace the **for** loop that is supported by that compiler with a replica of the **for** loop of ALGOL 60, as described in the part of the defining document reproduced below.

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.

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, with the command

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

or the identical Git repository with

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

The materials are also available on the course web page.

Candidates are required to submit a written report, describing the changes they made to the compiler in order to support the enhanced **for** loop, 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, 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.

Section 1 of this document sets out the specification for the enhanced **for** loop that should be added to the compiler, and Section 2 gives some suggestions for implementation that you may find useful.

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.

1 Specification

The programming language ALGOL 60 included only one looping construct, the **for** loop, but this could take different forms. A **for** loop had a 'controlled variable' that would be assigned a sequence of values as the loop body was executed. These values could be listed explicitly, as in

for $k := 1, 2, 3, 5$ **do** $print(k)$.

It was also possible to specify an arithmetic progression, as in this example:

for $k := 10$ **step** 10 **until** 100 **do** $print(k)$.

Here k takes the values 10, 20, 30, ..., 100 in successive executions of the loop body. A third form of loop allowed more that just arithmetic progressions. Here is an example:

$k := 1$;
for $k := 2 * k$ **while** $k < 1000$ **do** $print(k)$.

This prints 2, 4, 8, ..., 512, stopping there because the next value, 1024, does not satisfy the test. These three forms – simple expressions, **step-until** elements, and **while** elements – could be combined, like this:

for $k := 1, 2, 3, 5, 10$ **step** 10 **until** $100, 2 * k$ **while** $k < 1000$ **do**
  $print(k)$.

Perhaps unexpectedly, this program prints 1, 2, 3, 5, 10, 20, 30, ..., 100, 220, 440, 880. The next value after 100 is 220, because (as the formal definition below makes explicit), k is set to 110 before the test $k \leq 100$ fails, and the **while** element begins by computing $2 * k$ with this value of k.

As a more meaningful example, the following program computes $a = \lfloor \sqrt{x} \rfloor$ using binary search. A single loop encompasses two phases, one where d is growing until $(a + d)^2 > x$, and another where d is shrinking again.

$a := 0$;
for $d := 1, 2 * d$ **while** $square(a + d) \leq x, d$ **div** 2 **while** $d \geq 1$ **do**
  if $square(a + d) \leq x$ **then** $a := a + d$.

The meaning of **for** loops is defined in the following edited extract from the Revised Report on ALGOL 60.[1] [2]

Description of the reference language

4. Statements

4.6. For Statements

4.6.1. *Syntax [edited]*

⟨for list element⟩ ::= ⟨arithmetic expression⟩ |
  ⟨arithmetic expression⟩ **step** ⟨arithmetic expression⟩
  until ⟨arithmetic expression⟩ |
  ⟨arithmetic expression⟩ **while** ⟨Boolean expression⟩
⟨for list⟩ ::= ⟨for list element⟩ |
  ⟨for list⟩, ⟨for list element⟩
⟨for clause⟩ ::= **for** ⟨variable⟩ := ⟨for list⟩ **do**
⟨for statement⟩ ::= ⟨for clause⟩ ⟨statement⟩

[1] Naur, P. (Ed.), *Revised report on the algorithmic language ALGOL 60*, Computer Journal, **5** (4), January 1963, pp. 349–67. *Copies of this report and the paper by Knuth are linked from the course web page.*

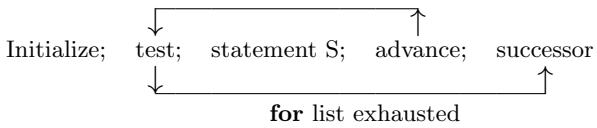
[2] The extract has been edited to remove references to **go to** statements and labels and an unhelpful reference to the function *sign*.

4.6.2 *Examples*[3]

for $q := 1$ **step** s **until** n **do** $A[q] := B[q]$

for $k := 1, k \times 2$ **while** $k < B$ **do**
    for $j := I + G, L, 1$ **step** 1 **until** $N, C + D$ **do**
        $A[k, j] := B[k, j]$

4.6.3 *Semantics*
A **for** clause causes the statement S which it precedes to be repeatedly executed zero or more times. In addition it performs a sequence of assignments to its controlled variable. The process may be visualized by means of the following picture:



In this picture the word initialize means: perform the first assignment of the **for** clause. Advance means: perform the next assignment of the **for** clause. Test determines if the last assignment has been done. If so, the execution continues with the successor of the **for** statement. If not, the statement following the **for** clause is executed.

4.6.4. *The* **for** *list elements*
The **for** list gives a rule for obtaining the values which are consecutively assigned to the controlled variable. This sequence of values is obtained from the **for** list elements by taking these one by one in the order in which they are written. The sequence of values generated by each of the three species of **for** list elements and corresponding execution of the statement S are given by the following rules:

4.6.4.1. *Arithmetic expression.* This element gives rise to one value, namely the value of the given arithmetic expression as calculated immediately before the corresponding execution of the statement S.

4.6.4.2. **Step-until**-*element.* An element of the form A **step** B **until** C, where A, B, and C are arithmetic expressions, gives rise to an execution which may be described most concisely in terms of additional ALGOL statements as follows [edited]:

    $V := A$;
$L1:$ **if** $B > 0$ **then begin if** $V > C$ **then go to** *Element exhausted* **end**
    else begin if $V < C$ **then go to** *Element exhausted* **end**;
    statement S;
    $V := V + B$;
    go to $L1$;

where V is the controlled variable of the **for** clause and *Element exhausted* points to the evaluation according to the next element of the **for** list, or if the **step-until**-element is the last of the list, to the next statement in the program.

4.6.4.3. **While**-*element.* The execution governed by a **for** list element of the form E **while** F, where E is an arithmetic and F a Boolean expression, is most concisely described in terms of additional ALGOL statements as follows:

$L3:$ $V := E$;
    if $\neg F$ **then go to** *Element exhausted*;
    Statement S;
    go to $L3$;

where the notation is the same as in 4.6.4.2 above.

[3] The second example incorporates a correction proposed in Knuth, D. E., *The remaining trouble spots in ALGOL 60*, Comm. ACM, **10** (10), October 1967, pp. 611–18.

4.6.5. The value of the controlled variable upon exit
[Upon exit from a **for** statement owing] to the exhaustion of the **for** list, . . . the value of the controlled variable is undefined[4] after the exit.

The goal of this assignment is to produce a modern implementation of this looping construct by replacing the existing **for** loop in the compiler we used in Lab 4. To fit in with the rest of the language implemented by that compiler, you should implement ALGOL 60-style **for** loops with the following differences:

- In accord with the syntactic style of the rest of the language, a ⟨for clause⟩ should govern a sequence of statements terminated by **end**; in ALGOL 60, a similar effect would be obtained by enclosing the sequence of statements in **begin** . . . **end** brackets.
- A number of compiler test cases contain **for** loops with the existing syntax. To allow these to continue to pass, you should allow *lo* **to** *hi* as an abbreviation for *lo* **step** 1 **until** *hi*.
- In ALGOL 60, **for** loops may have controlled variables with either real or integer type. Since real variables and arithmetic are not supported by the compiler in Lab 4, you should allow only integer variables and expressions.

Partial credit will be given for implementations that permit only a single element in each for list, or implement only some of the three kinds of element, or generate code that is significantly inefficient or over-long. 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 allowing single **step-until** and **while** elements, with test cases showing the code. *10 marks*
- An implementation that allows multiple elements of all three kinds, with test cases. *10 marks*
- Tidy object code of reasonable efficiency, demonstrated by test cases. *5 marks*
- Further test cases that demonstrate detailed aspects of loop behaviour. *5 marks*
- A written report with good clarity and presentation. *5 marks*

2 Implementation hints

(a) To implement the new style of **for** loop, you will need to consider what changes are needed in each phase of the compiler. Study carefully the description of loops given in the extract from the Revised Report, and note that the step and upper bound of a **step-until** element are re-evaluated before each iteration of the loop. The equivalent code given for **step-until** elements suggests that the step is evaluated *twice* in each iteration. Your implementation should be faithful to this interpretation, and you should include a test case that verifies that the step is indeed evaluated twice where doing so makes a difference.[5]

---

[4] "Whenever . . . the outcome of a certain process is left undefined or said to be undefined, this is to be interpreted in the sense that a program only fully defines a computational process if the accompanying information specifies . . . the course of action to be taken in all such cases as may occur during the execution of the computation." (Naur, *op. cit.*, footnote on p. 352.)

[5] We are therefore taking the 'conservative' interpretation of the semantics, as described by Knuth (*op. cit.*).

(b) To ensure compact code, you should design a code generation scheme such that code for the loop body is generated only once. This means setting things up so that, when the body finishes, control returns to the correct element of the for list. It is convenient to do this by introducing a hidden variable associated with the loop that contains 0 when the first element of the for list is active, 1 when the second element is active, and so on. Following the loop body, the code can use the value of this variable to branch back to the code for the appropriate list element.

(c) The existing implementation of **for** statements uses a hidden variable to hold the upper bound for the iteration, making sure this upper bound is fixed when the loop begins, and avoiding re-evaluation of the upper bound expression on each iteration of the loop. Space for this variable is allocated in the stack frame of the enclosing procedure during semantic analysis, and it is used in generating code for the loop. You will observe that the upper bound is not fixed in ALGOL 60, but is re-evaluated on each iteration. Nevertheless, the existing code to support a hidden variable will be useful to you, because your implementation will need a variable to keep track of which element of the for list is currently active.

(d) The code for each element of the for list has two entry points: one at the top that is entered when the **for** statement starts, or when the previous element is finished, and another (with a label) where control returns after executing the loop body. For a **step-until** element, one of these entry points sets the controlled variable to its initial value, and the other increments it, before in each case evaluating the termination test. When each element is exhausted, execution falls through to the next element.

(e) For the multi-way branch that follows the loop body, you can exploit an operation *JCASE* that generates jump tables. As an example, the statement optree

$$\langle JCASE\ ([lab_0;\ lab_1;\ lab_2], lab_3), t\rangle$$

examines the value of t, branching to lab_0, lab_1 or lab_2 respectively if the value of t is 0, 1 or 2, and branching to lab_3 if the value of t is either less than 0 or greater than 2. This operation is used in the existing compiler to implement a simple form of **case** statement where the case labels are assumed to lie in a fairly compact range.

(f) Most **for** loops will have only a single element in the for list, and you should ensure that your implementation generates code for them that is, as nearly as possible, equal in efficiency to the code generated for equivalent loops by the existing compiler.

(g) Where the step size is fixed and known at compile time, code that is directly based on the outline given in the Revised Report will contain one or more conditional branches that can be seen as always taken or never taken. Better code will result if these are optimised away before the intermediate form is translated into machine code.

(h) In writing the report, you can use Mercurial or Git to track what changes you have made to the compiler, making sure you include and explain all the code you have added. You should also include test cases to demonstrate that your implementation works properly, and to display the generated code, showing that it is compact and efficient. Think of possible bugs in your colleagues' implementations, and include test cases to show that your compiler does not suffer from them.