University of Oxford
Department of Computer Science

Compilers

Michaelmas Term 2019

The completed assignment is due by 12 noon on Friday 31st January 2020. You should submit it online through WebLearn. 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 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, 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: Array loops

The language of Lab 4 features **for**-loops that adhere to the pattern below.

for *name* **:=** *expr* **to** *expr* **do** *stmts* **end**

In particular, *name* must be an integer-valued variable.

Many other programming languages (e.g. Java, Rust, Scala) provide more elaborate constructs for looping over collections of data. The first task is to to extend the existing compiler to handle a looping construct for arrays. It should have the shape

for *name* **in** *array* **do** *stmts* **end**

and conform to the following conditions.

- *array* must be a program phrase of array type.
- Unlike in the original **for**-construct, the variable *name* need not be declared in advance. Instead, "**for** *name* **in**" should be interpreted as a declaration of a reference parameter *name* that is local to the "**do** *stmts* **end**" block.
- The intended effect of the construct is to execute *stmts* for each element of *array* in turn (in the same order as the elements appear in the array) on the understanding that, at each iteration, *name* refers to the current element.

Make sure that your solution covers all kinds of arrays definable in the language.

Example

```
type point = record x,y:integer end;
var a: array 10 of point;
var i, sum: integer;

begin
  i := 0;
  for r in a do
     r.x := i mod 3;
     r.y := i mod 5;
     i := i+1
  end;

  (* x values: 0, 1, 2, 0, 1, 2, 0, 1, 2, 0 *)
  (* y values: 0, 1, 2, 3, 4, 0, 1, 2, 3, 4 *)

  sum := 0;
```

```
  for r in a do
     sum := sum + r.x * r.y
  end;

  (* sum = 0*0 + 1*1 + 2*2 + 0*3 + 1*4 + 2*0 + 0*1 + 1*2 + 2*3 + 0*4 = 17 *)

  print_num(sum); newline()
end.
```

Task 2: Commuting code

In preparation for the next generation of compiler optimisations, programmers were asked to identify fragments of code that could be executed in arbitrary order. To that end, a new kind of statement, namely [$stmts_1$: $stmts_2$], was added to the language with the expectation that programmers will write $[stmts_1 : stmts_2]$ only if they are sure that it does not matter whether the code is translated as $stmts_1; stmts_2$ or $stmts_2; stmts_1$.

The following program illustrates correct usage

```
var x,y: integer;
begin
  [ x:=1 : y:=2 ];
  print_num(x); print_num(y)
end.
```

whereas below the construct is not used correctly.

```
var x: integer;
begin
  [ x:=1 : x:=2 ];
  (* swapping would lead to a different answer *)
print_num(x);
end.
```

The second task is to extend the lexer, parser and semantic analyser to accommodate the new construct in such a way that only programs that use the construct correctly are passed on to the code generator. For the purposes of the assignment, in such cases the compiler should halt and print *"Correct"*. Otherwise it should halt and output the message *"Possibly incorrect"*.

Note that you are not asked to detect all instances of correct usage, but rather to propose and implement checks that will reliably identify a range of non-trivial correct instances. Please make sure to explain in the report what features of the language are and are not covered by your method. You may wish to start off by considering statements that do not feature procedure calls and manipulate global variables only.

Assessment

Partial marks will be given for implementing only some aspects of the assignment or generating code that is significantly inefficient or over-long. To gain full marks, your submission should include test cases that illustrate the quality of your implementation and the generated code (the latter is relevant to Task 1 only). 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 the solution.

  (5 marks)

- A written report with good clarity and presentation.

  (5 marks)