University of Oxford
Department of Computer Science

Compilers

Michaelmas Term 2021

Your report on this assignment should be submitted online via Inspera by 12 noon on 28th January 2022. The assignment is worth 35 marks out of a total of 100 marks for the course.

This assignment is based on compilers provided with the lab materials. It asks you to make a number of extensions 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 as described on the course website, e.g. by cloning the Git repository supplied for the lab exercises to a directory named `task`:

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

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

Task 1: Block expressions

The programming language CBPL provides a dedicated construct for creating an expression out of a block of statements (a block expression). This task concerns adding a similar feature to the compiler found in the directory `ppc` of the lab materials that generates Keiko code for the whole picoPascal language.

- Block expressions have the form "`valof` *stmts* `end`". They may be nested.
- The expressions should be implemented by executing *stmts* until a statement of the form "`resultis` *expr*" (also to be added to the syntax) is encountered. *expr* should then be evaluated to provide the value of the (innermost) block expression enclosing `resultis`. When execution reaches the end of *stmts* without encountering `resultis`, the expression is also deemed to produce a result, but the exact value is unspecified and may be implemented arbitrarily.
- In valid programs, each occurrence of `resultis` must be surrounded by at least one "`valof` *stmts* `end`" block and all occurrences of `resultis` associated with the same block expression should return values of the same type. The results can be of any scalar type.
- Block expressions may not be used inside arguments of comparison operators ($=$, $<$, $>$, $\leq$, $\geq$, $\neq$) or in guards of `case` statements.

The goal of Task 1 is to extend the `ppc` compiler so that it conforms to the specification given above. Whenever programs are syntactically invalid, the compiler should issue an error message.

Example

```
var b: boolean;
var i: integer;

begin
  i:=0;
  b:= valof while true do
                if i > 2021 then resultis (i <= 2021) else i:=i+1 end
             end
       end;
  (* At this point i=2022 and b=false. *)

  print_num(valof i:=i-1; if (i=2021) and not b then resultis i+1 end end);
  newline(); print_num(i); newline()
  (* The program should print 2022 and 2021. *)
 end.
```

Task 2: Instruction folding

The Lab 4 compiler finds common subexpressions blindly and computes them into registers, even if the subexpressions could be recomputed at no cost, for example using the addressing modes of load and store instructions. For example, consider the following program.

```
var a, b: array 10 of integer;

proc swap(i: integer);
  var x: integer;
begin
  x := a[i]; a[i] := b[i]; b[i] := x
end;

begin
  swap(3)
end.
```

Even with a version of the compiler that can use the `reg+reg` addressing mode, the body of `swap` compiles into the following code, where the addresses of `a[i]` and `b[i]` are computed into registers `r5` and `r6` using `add` instructions.

```
    ldr r0, [fp, #40]
    lsl r5, r0, #2
    set r0, _a
    add r6, r0, r5
    ldr r4, [r6]
    set r0, _b
    add r5, r0, r5
    ldr r0, [r5]
    str r0, [r6]
    str r4, [r5]
```

Better code eliminates two instructions by performing the additions as part of the load and store instructions.

```
    ldr r0, [fp, #40]
    lsl r5, r0, #2
    set r0, _a
    ldr r4, [r0, r5]
    set r1, _b
    ldr r0, [r1, r5]
    str r0, [r0, r5]
    str r4, [r1, r5]
```

Design an enhancement to the common subexpression elimination pass of the Lab 4 compiler that reverses the process of identifying common subexpressions if the operations are likely to be folded into load or store instructions.

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 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)