Compilers: Collected problems

Mike Spivey

Michaelmas Term 2023

1 Lexical and syntax analysis

1.1 In the lecture, we wrote a regular expression to describe decimal and hexadecimal constants in C, and derived an NFA and a DFA from it. But there was a white lie, because C forbids decimal constants with a leading zero, and allows unsigned octal constants that start with a zero and continue with an arbitrary string of octal digits 0 to 7, a convention beloved of those ancients who programmed the PDP–11. Thus the string 0293 is not an integer constant, because the non-octal digit 9 is inconsistent with the leading zero. Modify the regular expression to reflect this rule, and show what changes result in the NFA and DFA. [1]

1.2 Suppose a lexer is written with one rule for each keyword and a catch-all rule that matches identifiers that are not keywords, like this:

```
rule token =
  parse
      "while"                  { WHILE }
    | "do"                     { DO }
    | "if"                     { IF }
    | "then"                   { THEN }
    | "else"                   { ELSE }
    | "end"                    { END }
      ...
    | ['A'-'Z''a'-'z']+        { IDENT (lexeme lexbuf) }
```

Describe the structure of an NFA and a DFA that correspond to this specification; explain what happens if several keywords share a common prefix. What data structure for sets of strings does the DFA implicitly contain?

1.3 Lex has the conventions that the longest match wins, and that earlier rules have higher priority than later ones in the script. These conventions are

[1] I'm pretty sure these are not the true rules of C, but can't bring myself to wade into the C standard and find out. The exercise is worthwhile independently of that.


```
if expr1 then
  stmts1
elsif expr2 then
  stmts2
else
  stmts3
end
```

```
if expr1 then
  stmts1
else
  if expr2 then
    stmts2
  else
    stmts3
  end
end
```

Figure 1: *Abbreviated syntax for chains of* `else if`*'s*

exploited in the lexer shown in Exercise 1.2 that recognises both keywords and identifiers. Would it be possible to describe the set of identifiers that are not keywords by a regular expression, without relying on these rules? If so, would this be a practical way of building a lexical analyser?

1.4 In C, a comment begins with '/∗' and extends up to the next occurrence of '∗/'. Write a regular expression that matches any comment. What would be the practical advantages and disadvantages of using this regular expression in a lexical analyser for C?

1.5 In Pascal, comments can be nested, so that a comment beginning with (* and ending with *) can have other comments inside it. What is an advantage of this convention? Show how Pascal comments can be handled in a lexical analyser written according to the conventions of *ocamllex* by using either recursion or an explicit counter.

1.6 The following productions for **if** statements appeared in the original definition of Algol 60:

stmt → *basic-stmt*
| **if** *expr* **then** *stmt*
| **if** *expr* **then** *stmt* **else** *stmt*.

Show that these productions lead to an ambiguity in the grammar. Suggest an unambiguous grammar that corresponds to the interpretation that associates each **else** with the closest possible **if**.

Now consider the ambiguous grammar: because it is ambiguous, a shift–reduce parser must have a state where both shifting and reducing lead to a successful conclusion, or a state where it is possible to reduce by two different productions. Find a string with two parse trees according to the grammar, and show the parser state where two actions are possible. Describe the results of shifting and of reducing in this state.

1.7 In the language of Lab 1, **if** statements have an explicit terminator **end** that removes the ambiguity discussed in the preceding exercise. However, this makes it cumbersome to write a chain of **if** tests, since the **end** keyword must be repeated once for each **if**. Show how to change the parser from Lab 1 to allow the syntax shown on the left in Figure 1 as an abbreviation for

the syntax on the right. An arbitrarily long chain of tests written with the keyword **elsif** can have a single **end**; the **else** part remains optional. Arrange for the parser to build the same abstract syntax tree for the abbreviated program as it would for its equivalent written without **elsif**.

1.8 One grammar for lists of identifiers contains the productions,

idlist → **id**
| *idlist* "," **id**

(we call this *left* recursive), and another (*right* recursive) contains the productions,

idlist → **id**
| **id** "," *idlist*

In parsing a list of 100 identifiers, how much space on the parser stack is needed by shift–reduce parsers based on these two grammars? Which grammar is more convenient if we want to build an abstract syntax tree that is a list built with *cons*?

1.9 [part of 2013/1] Hacker Jack decides to make his new programming language more difficult for noobs by writing all expressions in Polish prefix form.[2] In this form, unary and binary operators are written *before* their operands, and there are no parentheses. Thus the expression normally written `b * b - 4 * a * c` would be written

`- * b b * * 4 a c`,

and the expression `(x + y) * (x - y)` would be written

`* + x y - x y`.

After a false start, Jack realises that his language design is doomed if any symbol can be used both as a unary and as a binary operator, so he decides to represent unary minus by ~.

(a) Give a context free grammar for expressions in Jack's language, involving the usual unary and binary operators together with variables and numeric constants. Explain precisely why the grammar would be ambiguous if any operator symbol could be both unary and binary.

(b) In order to simplify the parser for expressions, Jack decides to minimise the number of different tokens that can be returned by the lexical analyser, distinguishing tokens with the same syntactic function by their semantic values alone. Suggest a suitable data type of tokens for use in the parser.

(c) Using this type of tokens and a suitable type of abstract syntax trees, write context free productions with semantic actions for a parser.

1.10 [2011/1 modified] In a file of data about the Oscars, each record contains a sequence of dates and the name of an actor or actress, like this:

`1933, 1967, 1968, 1981, "Katharine Hepburn"`

[2] so called after the Polish logician Jan Łukasiewicz.

```
%token⟨year⟩ YEAR
%token⟨actor⟩ ACTOR
%token COMMA

%type⟨(year list * actor) list⟩ file
%start file

%%

file : /* empty */                  { [] }
  | record file                     { $1 :: $2 } ;

record : years COMMA ACTOR          { ($1, $3) } ;

years :
    YEAR                            { [$1] }
  | YEAR COMMA years                { $1 :: $3 } ;
```

Figure 2: *Ocamlyacc grammar for Oscars data*

```
1975, 1983, 1997, "Jack Nicholson"
2011, "Colin Firth"
```

Figure 2 shows a grammar for such files, for which. *ocamlyacc* reports a shift/reduce conflict.

(a) By showing a parser state where the next action is not determined by the look-ahead, explain why the conflict arises.

(b) Design a grammar for the same language that is accepted by *ocamlyacc* without conflicts. Annotate the grammar with semantic actions that build the same abstract syntax as the grammar shown above.

(c) Can the same language be described by a regular expression over the set of tokens? Briefly justify your answer.

2 Expressions and statements

2.1 Write a program that finds the integer part of $\sqrt{x}$ using binary search, and test it by initially setting x to 200 000 000. Compile your progrm into Keiko code, and work out the purpose of each instruction.

2.2 Some machines have an expression stack implemented in hardware, but with a finite limit on its depth. For these machines, it is important to generate postfix code that makes the maximum stack depth reached during execution as small as possible.

(a) Let the `SWAP` instruction be defined so that it swaps the two top elements of the stack. Show how to use this instruction to evaluate the expression `1/(1+x)` without ever having more than two items on the stack.

(b) Prove that if expression e_1 (containing variables, constants and unary and binary operators) can be evaluated in depth d_1, and e_2 can be eval-

uated in depth d_2, then *Binop* (w, e_1, e_2) can be evaluated in depth

$$min\,(max\,d_1\,(d_2+1))\,(max\,(d_1+1)\,d_2).$$

Write a function *cost* : *expr* → *int* that calculates the stack depth that is needed to evaluate an expression by this method. Show that if e has fewer than 2^N operands, then *cost* $e \le N$.

(c) Write an expression compiler *gen_expr* : *expr* → *code* that generates the code that evaluates an expression e within stack depth *cost* e. [Hint: use *cost* in your definition.]

2.3 Now consider a machine that has a finite stack of depth N. In order to make it possible to evaluate expressions of arbitrary size, the machine is also supplied with a large collection of temporary storage locations numbered 0, 1, 2, ... There are two additional machine instructions:

```
type code = ...
  | PUT of int          (* Save temp (address) *)
  | GET of int          (* Fetch temp (address) *)
```

The instruction PUT n pops a value from the stack and stores it in temporary location n, and the instruction GET n fetches the value previously stored in temporary location n and pushes it on the stack.

Assuming $N \ge 2$, define a new version of *gen_expr* that exploits these new instructions, and places no limit on the size of expressions. The code generated should use as few GET and PUT instructions as possible, but you may ignore the possibility that the source expression contains repeated sub-expressions. There's no need to re-use temps, so you can use a different temp whenever you need to save the value of a sub-expression.

[*Hint*: optimal code for an expression can be generated by a function

gen : *expr* → *code* ∗ *int*

that returns code to evaluate a given expression, together with the number n of stack slots used by the code, with $n \le N$. If both e_1 and e_2 require N slots then evaluation of *Binop* (w, e_1, e_2) will need to use a temporary location.]

2.4 Programs commonly contain nested **if** statements, so that either the **then** part or (more commonly) the **else** part of an **if** statement is another **if** statement. (The latter possibility can be abbreviated using the **elsif** syntax that was the subject of problem 1.6.)

(a) Show the code that is produced for such nested statements by the naive translation scheme that was described in the lectures and used in Lab 1. Point out where this code is untidy and where it is significantly inefficient.

(b) Suggest rules that could be used in a peephole optimiser to improve the code from part (a), tidying it up and ameliorating any inefficiencies.

(c) Consider the problem of generating equally tidy and efficient code directly (without using a peephole optimiser), and if possible define one or more translation functions that produce this code.

2.5 [2013/2] The *scalar product machine* uses an evaluation stack, but replaces the usual floating point addition and multiplication instructions with

a single ADDMUL instruction that, given three numbers x, y and z on the stack, pops all three and replaces them with the quantity $x + y * z$. Thus the expression $b * b + 4 * a * c$ could be computed on this machine with the sequence

```
CONST 0
LOAD b
LOAD b
ADDMUL
CONST 0
CONST 4
LOAD a
ADDMUL
LOAD c
ADDMUL
```

The first ADDMUL instruction computes $t_1 = 0 + b * b$, the second computes $t_2 = 0 + 4 * a$, and the third computes the answer as $t_1 + t_2 * c$. Floating point addition and multiplication may be assumed commutative but not associative, and the distributive law does not hold in general.

(a) Suggest a suitable representation for expressions involving addition, multiplication, constants and (global) variables, and describe in detail a translation process that produces code like that shown in the example, using the smallest possible number of instructions.

(b) The designers of the scalar product machine are planning to include a stack cache whose effectiveness is maximised by keeping the stack small. The code for $b * b + 4 * a * c$ shown above reaches a stack depth of 4 just after the instruction LOAD a. If the machine has an instruction SWAP that exchanges the top two values on the stack, find an alternative translation of the same expression that never exceeds a stack depth of 3.

(c) The designers are willing to add other instructions that permute the top few elements of the stack. Give an example to show that the SWAP instruction on its own is not sufficient to allow every expression to be evaluated in the smallest possible stack space. [You may assume that for each $n \geq 3$ there is an expression e_n with addition at the root that needs a stack depth of n.]

(d) Suggest an additional instruction that, together with SWAP, allows all expressions to be evaluated in the optimal depth, and outline an algorithm that generates code achieving the optimum. There is no need to give code for the algorithm.

2.6 [2012/2] The programming language Oberon07 contains a new form of loop construct, illustrated by the following example:

```
while x > y do
  x := x - y
elsif x < y do
  y := y - x
end
```

The loop has a number of clauses, each containing a condition and a corresponding list of statements. In each iteration of the loop, the conditions are evaluated one after another until one of them evaluates to true; the corresponding statements are then executed, and then the loop begins its next iteration. If all the conditions evaluate to false, the loop terminates. In the example, if initially `x` and `y` are positive integers, then the loop will continue to subtract the smaller of them from the larger until they become equal. The loop thus implements Euclid's algorithm for the greatest common divisor of two numbers.

Previous versions of Oberon included a form of loop with embedded `exit` statements. The multi-branch `while` shown above is equivalent to the following `loop` statement:

```
loop
  if x > y then
    x := x - y
  elsif x < y then
    y := y - x
  else
    exit
  end
end
```

In general, a `loop` statement executes its body repeatedly, until this leads to one of the embedded `exit` statements; at that point, the whole loop construct terminates immediately.

(a) Suggest an abstract syntax for both these loop constructs, including the `exit` statement, and write production rules suitable for inclusion in an *ocamlyacc* parser for the language.

(b) The two kinds of loop are both to be implemented in a compiler that generates code for a virtual stack machine. Write the appropriate parts of a function that generates code for the two constructs by a syntax-directed translation.

(c) Show the code that would be generated by your implementation for the two examples given above. Assume that `x` and `y` are local variables at offsets −4 and −8 in the stack frame for the current procedure.

(d) The code that is generated for the multi-branch `while` loop is marginally more efficient than that for the equivalent `loop` statement. Suggest rules for inclusion in a peephole optimiser that would remove the difference in efficiency.

2.7 [2014/1, edited] Some programming languages provide conditional expressions such as

```
if i >= 0 then a[i] else 0
```

which evaluates to `a[i]` if `i >= 0`, and otherwise evaluates to zero without attempting to access the array element `a[i]`.

(a) Suggest an abstract syntax for this construct, and suggest a way of incorporating the construct into an *ocamlyacc* parser for a simple pro-

gramming language so as to provide maximum flexibility without introducing ambiguity. Make sure that an expression like

```
if x then y else p+q
```

has `p+q` as a subexpression.

In a compiler for the language, postfix code for expressions is generated by a function

gen_expr : *expr* → *code*.

Control structures are translated using a function

gen_cond : *expr* → *codelab* → *codelab* → *code*,

defined so that *gen_cond e tlab flab* generates code that jumps to label *tlab* if expression *e* has boolean value **true**, and the label *flab* if it has value **false**.

(b) Show how to enhance *gen_expr* and *gen_cond* to deal appropriately with conditional expressions.

It is suggested that short-circuit boolean `and` could be translated by getting the parser to treat e_1 `and` e_2 as an abbreviation for the conditional expression

`if` e_1 `then` e_2 `else false`,

expanding the abbreviation in creating the abstract syntax tree.

(c) Show the code that would be generated for the statement

```
if (i >= 0) and (a[i] > x) then i := i+1 end
```

according to your translation, assuming both `i` and `x` are global integer variables, and `a` is a global array of integers. Omit array bound checks.

If the resulting code is longer or slower than that produced by translating the `and` operator directly, suggest rules for post-processing the code so that it is equally good.

3 Data structures

3.1 Assume the following declarations.

```
type dogptr = pointer to dogrec;
  dogrec = record name: array 12 of char; age: integer; next: dogptr; end;

var q: dogptr; s: integer;
```

The following two statements form the body of a loop that sums the ages in a linked list of dogs.

```
s := s + q↑.age;
q := q↑.next
```

Show Keiko code for these two statements, omitting the run-time check that `q` is non-null.

3.2 A small extension to the language of Lab 2 would be to allow blocks with local variables. We can extend the syntax by adding a new kind of statement:

stmt → `local` *decls* `in` *stmts* `end`

For example, here is a program that prints 53:

```
var x, y: integer;
begin
  y := 4;
  local
    var y: integer;
  in
    y := 3 + 4; x := y * y
  end;
  print x + y
end.
```

As the example shows, variables in an inner block can have the same name as others in an outer block. Space for the local variables can be allocated statically, together with the space for global variables. Sketch the changes needed in our compiler to add this extension.

3.3 A certain imperative programming language contains a looping construct that consists of named loops with `exit` and `next` statements. Here is an example program:

```
loop outer:
  loop inner:
    if x = 1 then exit outer end;
    if even(x) then x := x/2; next inner end;
    exit inner
  end;
  x := 3*x+1
end
```

Each loop of the form `loop L:` ... `end` has a label `L`; its body may contain statements of the form `next L` or `exit L`, which may be nested inside inner loops. A loop is executed by executing its body repeatedly, until a statement `exit L` is encountered. The statement `next L` has the effect of beginning the next iteration of the loop labelled `L` immediately.

(a) Suggest an abstract syntax for this construct.

(b) Suggest what information should be held about each loop name in a compiler's symbol table.

(c) Briefly discuss the checks that the semantic analysis phase of a compiler should make for the loop construct, and the annotations it should add to the abstract syntax tree to support code generation. Give ML code for parts of a suitable analysis function.

(d) Show how the construct can be translated into a suitable intermediate code, and give ML code for the relevant parts of a translation function.

3.4 In some programming languages, it is a mistake to use the value of a variable if it has not first been initialised by assigning to it. Write a function

that, for the language of Lab 1, tries to identify uses of variables that may be subject to this mistake. Discuss whether it is possible to do a perfect job, and if not, what sort of approximation to the truth it is best to make.

4 Procedures

Note: Questions on this sheet ask for Keiko code for programs in a typed language with procedures. For experimentation, I recommend the picoPascal compiler in the ppc4 subdirectory of the lab materials.

4.1 [See pp/test/prob4-1.p] Show the Keiko code for the following program, explaining the purpose of each instruction.

```
proc double(x: integer): integer;
begin
  return x + x
end;

proc apply3(proc f(x:integer): integer): integer;
begin
  return f(3)
end;

begin
  print_num(apply3(double));
  newline()
end.
```

4.2 Here is a procedure that combines nesting and recursion:

```
proc flip(x: integer): integer;
  proc flop(y: integer): integer;
  begin
    if y = 0 then return 1 else return flip(y–1) + x end
  end;
begin
  if x = 0 then return 1 else return 2 * flop(x–1) end
end;
```

(a) Copy out the program text, annotating each applied occurrence with its level number.

(b) If the main program contains the call flip(4), show the layout of the stack (including static and dynamic links) at the point where procedure calls are most deeply nested.

4.3 [See ppc4/test/cpsfac.p] The following PICOPASCAL program is written in what is called 'continuation-passing style':

```
proc fac(n: integer;
           proc k(r: integer): integer): integer;
  proc k1(r: integer): integer;
  begin
```

```
    return k(n * r)
  end;
begin
  if n = 0 then
    return k(1)
  else
    return fac(n-1, k1)
  end
end;

proc id(r: integer): integer;
begin
  return r
end;

begin
  print_num(fac(3, id));
  newline()
end.
```

When this program runs, it eventually makes a call to `id`.

(a) Draw a diagram of the stack layout at that point, showing the static and dynamic links.

(b) Show Keiko code for the procedure calls `k(n * r)` and `fac(n-1, k1)`.

4.4 [2013/3; see ppc4/test/sumarray.p] Figure 3 shows a program that computes

$$\sum_{0 \le i < 10} (i + 1)^2 = 385$$

by filling an array a so that $a[i] = (i + 1)^2$, then calling a procedure that sums the vector by using the higher-order procedure *dovec* to iterate over its elements. The parameter v to the procedures *sum* and *dovec* is passed by reference.

(a) Draw the layout of the subroutine stack at a time when the procedure *add* is active, showing the layout of the stack frames for each procedure and all the links between them.

(b) Show Keiko code that implements each of the following statements in the program, with comments to clarify the purpose of each instruction.

  (i) The statement `f(v[i])` in `dovec`.

  (ii) The statement `s := s + x` in `add`.

  (iii) The statement `dovec(add, v)` in `sum`.

(c) Briefly discuss the changes in the object code and in the organisation of storage that would be needed if the parameter v in *sum* and *dovec* were passed by value instead of by reference. Under what circumstances would a subroutine be faster with an array parameter passed by value instead of by reference? On a register machine, what optimisations to the procedure body might remove this advantage?

```
type vector = array 10 of integer;

(* dovec – call f on each element of array v *)
proc dovec(proc f(x: integer); var v: vector);
  var i: integer;
begin
  i := 0;
  while i < 10 do
    f(v[i]); i := i+1
  end
end;

(* sum – sum the elements of v *)
proc sum(var v: vector): integer;
  var s: integer;

  (* add – add an integer to s *)
  proc add(x: integer);
  begin
    s := s + x
  end;

begin
  s := 0;
  dovec(add, v);
  return s
end;

var a: vector; i: integer;

begin
  i := 0;
  while i < 10 do
    a[i] := (i+1)*(i+1);
    i := i+1
  end;

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

Figure 3: *Program for exercise 4.4*

4.5 [2014/2] The following Pascal-style program declares a record type and two procedures, one of which takes a parameter of record type that is passed by reference.

```
type rec = record c1, c2: char; n: integer end;

proc f(var r: rec);
begin
  r.n := r.n + 1
end;

proc g();
  var s: rec;
begin
  ...
  f(s)
  ...
end;
```

(a) Briefly explain why the semantic analysis phase of a compiler must take into account both the size and the alignment of data types, and give an example where two types would (on a typical machine) have the same size but different alignment.

(b) Making reasonable assumptions about the size and alignment of the character and integer types, show the layout that would be used for the record type `rec`.

(c) Sketch the frame layouts of procedures `f` and `g` in the program, and (briefly defining the instructions you use) give postfix code for the assignment `r.n := r.n + 1` and the procedure call `f(s)` in the program.

In a different programming language, values of record type are pointers to dynamically allocated storage for a record and these pointers are passed by value, rather like values of class type in Java. Dereferencing of the pointer is implicit in the expression `r.n`.

(d) Show what code would be generated from such a language for the assignment `r.n := r.n + 1` and the procedure call `f(s)`, assuming the parameter `r` is passed by value.

(e) For the Java-like language, give an example of a program demonstrating that parameters are passed by value and not by reference, and state what results are expected from the program in each case.

5 Machine code

5.1 Figures 8.4 and 8.5 show two tilings of the same tree for `x := a[i]`. Under reasonable assumptions, how many distinct tilings does this tree have, and what is the range of their costs in terms of the number of instructions generated? (Relevant rules are numbered 1, 4, 6, 9, 16, 21, 36–40, 42–44 and 49 in Appendix D.)

5.2 The ARM has a multiply instruction `mul r1, r2, r3` that, unlike other arithmetic instructions, demands that both operands be in registers, and

does not allow an immediate operand. How is this restriction reflected in the code generator?

5.3 A previous version of the machine grammar for ARM covered the left-shift operation with the rule,

$$reg \rightarrow \langle \textsc{Binop}\ Lsl, reg_1, rand \rangle \qquad \{\ \mathsf{lsl}\ reg,\ reg_1,\ rand\ \}$$

where *rand* is the same non-terminal that describes the second operand of arithmetic instructions line `add`. Identify a source program that would be wrongly translated by a compiler incorporating this rule. What goes wrong, how does the grammar in Appendix D avoid the problem?

5.4 Consider the following data type and procedure:

```
type dogptr = pointer to dogrec;
  dogrec = record name: array 12 of char; age: integer; next: dog-
ptr; end;

proc sum(p: dogptr): integer;
  var q: dogptr; s: integer;
begin
  q := p; s := 0;
  while q <> nil do
    s := s + q↑.age;
    q := q↑.next
  end;
  return s
end;
```

Making appropriate assumptions, describe possible layouts of the record type `rec` and the stack frame for `sum`, assuming that all local variables are held in the frame.

5.5 Using the layout from the previous exercise, show the sequence of trees that would be generated by a syntax-directed translation of the statements

```
s := s + q↑.age;
q := q↑.next
```

in the loop body. Omit the run-time check that `q` is not null. (In contrast to Exercise 3.1, both `s` and `q` are local variables here.)

5.6 Suggest a set of tiles that could be used to cover the trees, and show the object code that would result.

5.7 The code that results from direct translation of the trees is sub-optimal. Considering just the loop body in isolation, suggest an optimisation that could be expressed as a transformation of the sequence of trees, show the trees that would result, and explain the consequent improvements to the object code.

5.8 If a compiler were able to consider the whole loop instead of just its body, suggest a further optimisation that would be possible, and explain what improvements to the object code that would result from it.

5.9 Suppose that the ARM is enhanced by a memory-to-memory move instruction

```
movm [r1], [r2]
```

with the effect $mem_4[r_1] \leftarrow mem_4[r_2]$; the two addresses must appear in registers.

(a) Use this instruction to translate the assignment `x := y`, where `x` and `y` are local variables in the stack frame. Assuming each instruction has unit cost, compare the cost of this sequence with the cost of a sequence that uses existing instructions.

(b) Find a statement that can be translated into better code if the new instruction is used.

(c) Write one or more rules that could be added to a tree grammar to describe the new instruction.

(d) Explain, by showing examples, why optimal code for the new machine cannot be generated by a code generator that simply selects the instruction that matches the biggest part of the tree.

(e) [Not covered in lectures.] Label each node with its cost vector, and show how optimal code for `x := y` and for your example in part (b) could be generated by the dynamic programming algorithm.

5.10 [part of 2012/3, edited]

(a) Show the trees that represent the statement

```
a[a[i]] := a[i]+i
```

before and after eliminating common sub-expressions, if `a` is a global array, and `i` is a local variable stored in the stack frame of the current procedure. Show also the machine code that would be generated for a typical RISC machine. If the target machine had an addressing mode that added together a register and the address of a global like `a`, how would that affect the decision which sub-expressions should be shared?

(b) Show the process and results of applying common sub-expression elimination to the sequence,

```
x := x - y;  y := x - y;  z := x - y
```

where all of `x`, `y` and `z` are locals stored in the stack frame. Show also the resulting machine code.

6 Revision

This is a selection of past exam questions, edited in some cases to fit better with the course as I gave it this year.

6.1 A certain programming language has the following abstract syntax for expressions and assignment statements:

type *stmt* =
    Assign **of** *expr* $*$ *expr* ($*$ Assignment $e_1 := e_2$ $*$)

and *expr* = { *e_guts* : *expr_guts*; *e_size* : *int* }

and *expr_guts* =
  Var **of** *name* (∗ Variable (name, address) ∗)
| *Sub* **of** *expr* ∗ *expr* (∗ Subscript $e_1[e_2]$ ∗)
| *Binop* **of** *op* ∗ *expr* ∗ *expr* (∗ Binary operator e_1 *op* e_2 ∗)

and *name* = { *x_name* : *ident*; *x_addr* : *symbol* }

and *op* = *Plus* | *Minus* | *Times* | *Divide*

Each expression is represented by a record *e* with a component *e.e_guts* that indicates the kind of expression, and a component *e.e_size* that indicates the size of its value. Each variable *Var x* is annotated with its address *x.x_name*.

You may assume that syntactic and semantic analysis phases of a compiler have built a syntactically well-formed abstract syntax tree, in which only variables and subscript expressions appear as the left hand side e_1 of each assignment e_1 **:=** e_2 and the array e_1 in each subscripted expression $e_1[e_2]$.

The task now is to translate expressions and assignment statements into postfix intermediate code, using the following instructions:

type *code* =
  CONST **of** *int* (∗ Push constant ∗)
| *GLOBAL* **of** *symbol* (∗ Push symbolic address ∗)
| *LOAD* (∗ Pop address, push contents ∗)
| *STORE* (∗ Pop address, pop value, store ∗)
| *BINOP* **of** *op* (∗ Pop two operands, push result ∗)
| *SEQ* **of** *code list* (∗ Sequence of code fragments ∗)

(a) Defining whatever auxiliary functions are needed, give the definition of a function *gen_stmt* : *stmt* → *code* that returns the code for an assignment statement. Do not attempt any optimisation at this stage.

(b) Show the code that would be generated for the assignment

```
a[i,j] := b[i,j] * b[1,1]
```

where the variables `a`, `b`, `i`, `j` are declared by

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

Assume that integers have size 1 in the addressing units of the target machine, and array `a` has elements `a[0,0]` up to `a[9,9]`.

(c) Suggest two ways in which the code you showed in part (b) could be optimised.

6.2 (a) Briefly explain the distinction between value and reference parameters, and give an example of a program in an Algol-like language that behaves differently with these two parameter modes.

(b) Describe how both value and reference parameters may be implemented by a compiler that generates postfix code, showing the code that would be generated for your example program from part (a) with each kind of parameter.

A procedure with a *value-result* parameter requires the actual parameter to be a variable; the procedure maintains its own copy of the parameter, which is initialised from the actual parameter when the procedure is called, and has its final value copied back to the actual parameter when the procedure exits.

(c) Give an example of a program in an Algol-like language that behaves differently when parameters are passed by reference and by value-result.

(d) Suggest an implementation for value-result parameters, and show the code that would be generated for your example program from part (c) with value-result parameters.

6.3 The following program is written in a Pascal-like language with arrays and nested procedures:

```
var A: array 10 of array 10 of integer;

procedure P(i: integer);

    var x: integer;

    procedure Q();
        var j: integer;
    begin
        A[i][j] := x
    end;

    procedure R();
    begin
        Q()
    end;

begin (* P *)
    R()
end

begin (* main program *)
    P(3)
end.
```

The array A declared on the first line has 100 elements A[0][0], . . . , A[9][9].

(a) Describe the layout of the subroutine stack when procedure Q is active, including the layout of each stack frame and the links between them.

(b) Using a suitable stack-based abstract machine code, give code for the procedure Q. For those instructions in your code that are connected with memory addressing, explain the effect of the instructions in terms of the memory layout from part (a).

(c) Similarly, give code for procedure R.

6.4 (a) Explain how the run-time environment for static binding can be represented using chains of static and dynamic links. In particular, explain the function of the following elements, and how the elements are modified and restored during procedure call and return: frame pointer, static link, dynamic link, return address.

(b) Explain how functional parameters may be represented, and how this representation may be computed when a local procedure is passed to another procedure that takes a functional parameter. [A functional parameter of a procedure P is one where the corresponding actual parameter is the name of another procedure Q, and that procedure may be called from within the body of P.]

(c) The following program is written in a Pascal-like language with static binding that includes functional parameters:

```
proc sum(n: int; proc f(x: int): int): int;
begin
  if n = 0 then
    return 0
  else
    return sum(n-1, f) + f(n)
  end
end;

proc sumpowers(n, k: int): int;
  proc power(x: int): int;
    var i, p: int;
  begin
    i := 0; p := 1;
    while i < k do
      p := p * x; i := i + 1
    end;
    return p
  end
begin
  return sum(n, power)
end;

begin (* Main program *)
  print_num(sumpowers(3, 3))
end.
```

During execution of the program, a call is made to the procedure `power` in which the parameter `x` takes the value 1. Draw a diagram of the stack layout at this point, showing all the static links, including those that form part of a functional parameter.