Compilers: Collected problems (with solutions)

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]

Answer: We want the regular expression

```
-?[1-9][0-9]∗ | 0x[0-9a-f]+ | 0[0-7]∗
```

with three alternatives, one for each base. I've chosen to disallow a leadig minus sign in front of an octal constant, though actually I think the rules of C don't make the minus sign part of the constant at all. It turns out that the common constant 0 is, according to this account, written in octal – and the form -0 isn't allowed at all.

For the NFA (Figure 1), we can simply add a third track to the machine, adjusting some of the other transitions appropriately. Turning this into a DFA (Figure 2) reveals that constants can be classified by looking at the first two characters, and after that the allowable continuation characters are determined.

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 }
```

[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.

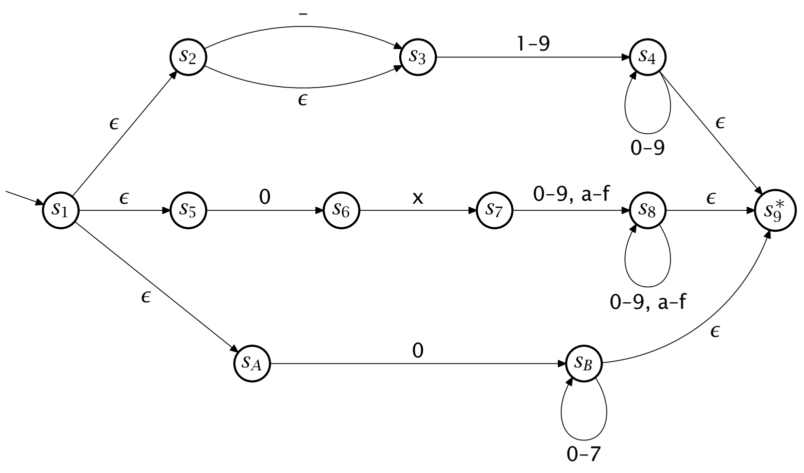


Figure 1: *NFA for problem 1.1*

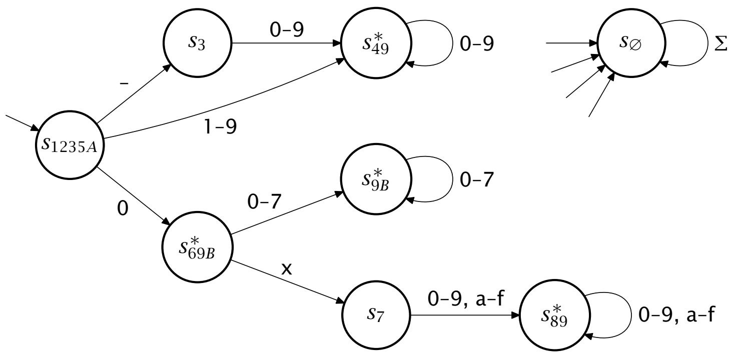


Figure 2: *DFA for problem 1.1*

```
| "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?

Answer: In an NFA, each keyword can correspond to a chain of states, linked by transitions that spell out the keyword. The catch-all corresponds to a loop that allows all letters. The complete NFA has all these parts linked by ϵ-transitions to common starting and finishing states.

In a DFA, it is necessary to have a tree of states corresponding to prefixes of the keywords, with transitions linking each state to successors that correspond to prefixes that are longer by one letter. Letters that do not appear in some keyword lead to a state with a loop that spells out any identifier, and there is an additional dustbin state reached by any non-letter. The structure for recognising keywords via their prefixes is equivalent to a trie.

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 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?

Answer: Automata theory shows that the class of regular languages contains all finite sets of strings and is closed under complementation and set intersection. So there is a regular expression that describes the set of identifiers that are not keywords. However, for any practical language, this regular expression is huge; this is not 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?

Answer: A suitable regexp is

```
"/*"("*"*[^*/]|"/")*"*"+"/"
```

It becomes clearer if we replace /, ∗ and other characters [ˆ∗/] with a, b and c respectively, and expand b+ to bb∗ – then the expression becomes

```
ab(b*c|a)*bb*a
```

This regexp could be used to scan comments, and would pass over an entire comment quickly. But if comments are very long, there is a danger of overflowing the input buffer of the generated lexer, or at least requiring sufficient memory to store the entire comment before it is discarded. Also, comments can contain embedded newlines, and this would cause the lexer to lose track of the line number in the input.

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.

Answer: Nested comments allow you to comment out code easily, even if the code contains comments itself. Here's how to handle nested comments using the feature of *ocamllex* that allows lexers to have arguments:

```
rule token =
  parse
      ...
    | "(*"                     { comment 0 lexbuf; token lexbuf }
      ...

and comment n =
  parse
      "(*"                     { comment (n + 1) lexbuf }
    | "*)"                     { if n = 0 then () else comment (n−1) lexbuf }
    | "\n"                     { incr lineno; comment lexbuf }
    | _                        { comment n lexbuf }
    | eof                      { error "unterminated comment" }
```

The scanner *comment* consumes the part of a comment after the initial (∗. If it finds a nested (∗, then it calls itself with an argument that is incremented by 1, and if it finds a ∗), the counter is decremented by 1, until it reaches 0.

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.

Answer: The proposed grammar is ambiguous because, if *e* is an expression and *b* a basic statement, the string

`if` *e* `then` `if` *e* `then` *b* `else` *b*

can be parsed both in a way that associates the `else` with the first `if` and in a way that associates it with the second `if`.[i]

For an unambiguous grammar, we can insist that the statement that appears between `then` and `else` should be 'saturated', in the sense that is cannot have another `else` added to it. That means that each `else` is paired with the closest possible `if`.

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

(We can verify that this grammar is unambiguous by submitting it to yacc and observing that there are no conflicts, since any *LALR*(1) grammar is also unambiguous.)

Returning to the original grammar, and the string that reveals the ambiguity, when enacting either derivation a shift–reduce parser will reach a state where

`if` *expr* `then` `if` *expr* `then` *stmt*

is on the stack, and the input contains `else` *b*. At this point, the parser can either shift the `else`, and after shifting the token *b* and reducing, it reaches

`if` *expr* `then` `if` *expr* `then` *stmt* `else` *stmt*.

Two more reductions complete the parse. On the other hand, at the same point it can reduce by the production *stmt* → `if` *expr* `then` *stmt* to reach

`if` *expr* `then` *stmt*.

[i] It's important to distinguish between the property of a *sentence* that holds if the sentence has more than one (essentially different) derivation and the property of a *grammar* that holds if any sentence has that property. This second property is the one we refer to by the term 'ambiguous'. If asked for an unambiguous grammar, it's not enough to say that the ambiguity can be avoided, e.g., by adding extra parentheses or, in this case, extra `begin` . . . `end` brackets.

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

Figure 3: *Abbreviated syntax for chains of* `else if`*'s*

Shifting `else` *b* on top of this allows the parse to be completed in another way.

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 3 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`.

Answer: Replace the rules

```
stmt :
    IF expr THEN stmts END                { IfStmt ($2, $4, Skip) }
  | IF expr THEN stmts ELSE stmts END     { IfStmt ($2, $4, $6) }
```

with the following:

```
stmt :
    IF expr THEN stmts iftail END         { IfStmt ($2, $4, $5) }

iftail :
    /* empty */                           { Skip }
  | ELSE stmts                            { $2 }
  | ELSIF expr THEN stmts iftail          { IfStmt ($2, $4, $5) } ;
```

1.8 One grammar for lists of identifiers contains the productions,

$$\begin{aligned} \textit{idlist} &\rightarrow \textbf{id} \\ &\mid\ \textit{idlist}\ \text{","}\ \textbf{id} \end{aligned}$$

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

$$\begin{aligned} \textit{idlist} &\rightarrow \textbf{id} \\ &\mid\ \textbf{id}\ \text{","}\ \textit{idlist} \end{aligned}$$

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*?

Answer: For the left-recursive grammar, the set of viable prefixes is finite:

> ϵ
> **id**
> L
> L ,
> L , **id**

It follows that the parser stack never contains more than three items: after the initial reduction of $L \to$ **id**, the stack always contains an L at the bottom. For each subsequent item in the list, the parser shifts a comma and an identifier, then immediately reduces by $L \to L$, **id**.

For the right-recursive grammar, a typical parser state is **id** , **id** , **id** , **id** during the first part of the parse, when *all* the tokens from the input are shifted onto the stack until the end of the file; and **id** , **id** , **id** , L during the second part, when a reduction by $L \to$ **id** is followed by many reductions by $L \to L$, **id**.

The left-recursive grammar leads to a parser that can read an arbitrarily long list in constant stack space; the right-recursive grammar is less well suited to *LR* parsing, and requires a stack space of 199 to parse a list of 100 identifiers. On the other hand, if we want to build a (cons-based) list as the abstract syntax tree, then the right-recursive grammar can include the rule,

> *idlist* : *IDENT COMMA idlist* { $1 :: $3 }

but the left-recursive grammar needs

> *idlist* : *idlist COMMA ident* { $1 @ [$3] }

and building the abstract syntax tree for a list of length N needs $O(N^2)$ time. These considerations are not very important in practice – until a compiler needs to deal with input texts that have been generated not by human editing but by another program, when all finite limits are likely to be strained.

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

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

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.

Answer: (a) Using the same token for both kinds of minus:

```
expr -> + expr expr | - expr expr | * expr expr | / expr expr
  | - expr | NUMBER | IDENT
```

This is ambiguous because the string – – 3 2 has two leftmost derivations:

```
expr ==> - expr ==> - - expr expr ==> - - 3 expr ==> - - 3 2

expr ==> - expr expr ==> - - expr expr ==> - - 3 expr ==> - - 3 2
```

The same ambiguity would be present if – were replaced by any symbol that could play both roles.

(b) We should lump together all unary operators and all binary operators:

```
type token =
    NUMBER of int
  | IDENT of string
  | BINOP of op
  | MONOP of op
```

(c) Using the abstract syntax trees

```
type expr =
    Number of int
  | Variable of string
  | Monop of op * expr
  | Binop of op * expr * expr
```

we can write the following rules:

```
expr :
    NUMBER                  { Number $1 }
  | IDENT                   { Variable $1 }
  | MONOP expr              { Monop ($1, $2) }
  | BINOP expr expr         { Binop ($1, $2, $3) } ;
```

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

Figure 4 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.

```
%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 4: *Ocamlyacc grammar for Oscars data*

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

Answer: (a) After shifting a *YEAR* and on seeing a comma, the parser cannot tell whether to reduce by the rule *years* → *YEAR* or to shift the comma as part of the RHS of *years* → *YEAR COMMA years*.

(b) Among several possible solutions, one is to remove the right recursion for *years* in favour of left recursion:

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

It then becomes possible for the parser to shift a comma and look at the next token before deciding whether the comma ends the list of years.

Other possible answers are to remove both instances of right recursion by writing also

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

In that case, the set of viable prefixes becomes finite.

Also, one can simply adjust the placement of the comma token in the grammar, like this:

```
record : years ACTOR                { ($1, $2) } ;

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

The crucial decision to reduce by the rule *years* → *YEAR COMMA* is now taken when *ACTOR* is seen as the look-ahead token.

(c) A file is a sequence of records, *file* = *record**, and a record is a non-empty sequence of years followed by an actor, with commas in the right places:

$$record = \textit{YEAR}\ (\textit{COMMA YEAR})^*\ \textit{COMMA ACTOR}.$$

So the whole thing can be described by a regular expression.

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.

Answer: Here is my implementation of binary search:

```
begin
  x := 200000000;
  a := 0;
  b := 20000;
  (* Inv: a↑2 <= x < b↑2 *)
  while a+1 < b do
    m := (a+b) div 2;
    if m*m <= x then
      a := m
    else
      b := m
    end
  end;
  print a; newline
end.
```

Compiling it with the compiler from Lab 1 and turning on the peephole optimiser gives the code shown in Figure 5. I have combined multiple lines into one to save space, but I have not deleted any of the boilerplate that surrounds the generated code.

The first three lines (1–3) are a header, giving the name Main of this module, and informing the Keiko assembler/linker that it needs to be combined with another module Lib that contains I/O subroutines.

The remainder of the file is a definition of a single subroutine MAIN that contains all the program code, followed by (lines 33–36) the declarations of the four variables, all global, that are used in the program. Each of these variables is an integer occupying 4 bytes; their names have been systematically prefixed with an underscore in the compiler output to avoid the situation where a variable named MAIN might clash with the name of the main program. The heading for the subroutine (line 5) gives its name, and specifies that it needs no space for local variables.

The code for the procedure body begins with assignments of constants to the global variables `y`, `a` and `b`. Each assignment has a CONST instruction that pushes a constant onto the stack and an STGW instruction that pops a value and STores in a Global variable that occupies a Word of storage.

Next comes the implementation of the `while` loop; the form of code shown here puts the test at the top (line 14) and consequently needs an unconditional jump to get back to the test after executing the loop body. In this program, the peephole optimiser has duplicated this jump at the ends of both arms of the `if` statement, and it has become the JUMP instructions on lines 21 and 25.

The test of the `while` loop is on line 14. The expression `a+1` in evaluated by pushing the value of `a` and the constant 1 and adding them together. Then `b` is pushed on top, and the JGEQ instruction branches to the next statement after the loop if the test is false.

The `if` test on line 18 is noteworthy because `m*m` is computed by fetching the value of `m` twice and multiplying the two copies of the value together. This code is certainly correct, and though an alternative exists that fetches `m` just once –

```
LDGW _m; DUP; TIMES
```

```
MODULE Main 0 0
IMPORT Lib 0
ENDHDR

FUNC MAIN 0
!   x := 200000000;
CONST 200000000; STGW _x
!   a := 0;
CONST 0; STGW _a
!   b := 20000;
CONST 20000; STGW _b
LABEL 1
!   while a+1 < b do
LDGW _a; CONST 1; PLUS; LDGW _b; JGEQ 2
!     m := (a+b) div 2;
LDGW _a; LDGW _b; PLUS; CONST 2; DIV; STGW _m
!     if m*m <= x then
LDGW _m; LDGW _m; TIMES; LDGW _x; JGT 3
!       a := m
LDGW _m; STGW _a
JUMP 1
LABEL 3
!       b := m
LDGW _m; STGW _b
JUMP 1
LABEL 2
!   print a; newline
LDGW _a; CONST 0; GLOBAL Lib.Print; PCALL 1
CONST 0; GLOBAL Lib.Newline; PCALL 0
RETURN
END

GLOVAR _m 4
GLOVAR _x 4
GLOVAR _b 4
GLOVAR _a 4
```

Figure 5: *Keiko code for a binary search*

– it is not clear that this code offers any significant advantage in a bytecode interpreter. When we come to generate code for a register machine, however, we will take care to recognise that `m` is a common sub-expression and evaluate it just once.

The statements `print a` and `newline` are translated as if they were calls to subroutines `print(a)` and `newline()`, in a way that we will examine closely in coming lectures. It is only the lack of subroutine calls in the syntax of our language that stops us from writing them that way in the source code.

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 evaluated in depth d_2, then $\mathit{Binop}\,(w, e_1, e_2)$ can be evaluated in depth

$$\mathit{min}\,(\mathit{max}\; d_1\; (d_2 + 1))\,(\mathit{max}\,(d_1 + 1)\; d_2).$$

Write a function $\mathit{cost} : \mathit{expr} \to \mathit{int}$ 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 $\mathit{cost}\; e \le N$.

(c) Write an expression compiler $\mathit{gen\_expr} : \mathit{expr} \to \mathit{code}$ that generates the code that evaluates an expression e within stack depth $\mathit{cost}\; e$. [Hint: use cost in your definition.]

Answer: (a) The best way to evaluate `1/(1+x)` is to tackle the denominator first:

```
CONST 1; LDGW _x; PLUS
CONST 1; SWAP; DIV
```

(b) More generally, we have a choice of which operand of $\mathit{Binop}\,(w, e_1, e_2)$ to evaluate first. The sequence

```
⟨Code for $e_1$⟩
⟨Code for $e_2$⟩
BINOP $w$
```

first evaluates e_1 (in stack depth d_1), then keeps this value on the stack while evaluating e_2 (in depth $d_2 + 1$), finally combining the two values to give the value of the whole expression. The maximum stack depth is $\mathit{max}\; d_1\; (d_2 + 1)$.

Similarly, the sequence

```
⟨Code for $e_2$⟩
⟨Code for $e_1$⟩
SWAP
BINOP $w$
```

keeps the value of e_2 on the stack while evaluating e_1, giving a maximum stack depth of $\mathit{max}\,(d_1 + 1)\; d_2$. We can choose whichever of these is better, giving an overall maximum of $\mathit{min}\,(\mathit{max}\; d_1\; (d_2 + 1))\,(\mathit{max}\,(d_1 + 1)\; d_2)$.

The optimal cost (in terms of stack depth) is computed by the following function:

```
let rec cost =
  function
      Number n → 1
    | Variable x → 1
    | Monop (w, e1) → cost e1
    | Binop (w, e1, e2) →
        let d1 = cost e1 and d2 = cost e2 in
        min (max d1 (d2 + 1)) (max (d1 + 1) d2)
```

To show that $\mathit{cost}\; e \le N$ if e has fewer than 2^N operands, we use structural induction on e, in a form where the value of N can vary in applications of the induction hypotheses, so that logically the result to be proved by induction is

$$P(e) \equiv (\forall N.\; \mathit{size}\, e < 2^N \Rightarrow \mathit{cost}\; e \le N).$$

If e is a constant or variable, it can be evaluated in stack depth 1; and if it is a unary expression *Monop* (w, e_1), then it can be evaluated in the same depth as e_1, which also has the same number of operands.

So now suppose that e has the form *Binop* (w, e_1, e_2), and has has fewer than 2^N operands. In that case, one of e_1, e_2 must have fewer than 2^{N-1} operands; for if both had 2^{N-1} or more, then the total would not be less than 2^N. Suppose e_1 has less than 2^{N-1} operands and e_2 has less than 2^N. We may then deduce from the induction hypothesis that *cost* $e_1 \leq N-1$ and *cost* $e_2 \leq N$, so *cost* $e \leq N$ also. (Proof checked in outline with the Boyer-Moore theorem prover.)

(c) We use *cost* in *gen_expr* to decide which strategy to choose:

```
let rec gen_expr =
  function
      Number n → CONST n
    | Variable x → LDGW x.x_name
    | Monop (w, e1) →
        SEQ [gen_expr e1; MONOP w]
    | Binop (w, e1, e2) →
        if cost e1 ≥ cost e2 then
          SEQ [gen_expr e1; gen_expr e2; BINOP w]
        else
          SEQ [gen_expr e2; gen_expr e1; SWAP; BINOP w]
```

The only slight worry with this is that the cost is being computed afresh at each node, making the whole process quadratic in the size of the expression. If this is important, then we could annotate the tree with costs before generating code, or compute code and cost together as in the next exercise. For commutative operators, we could avoid the *SWAP*: another job for a peephole optimiser.

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 $N \geq 2$, 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

$$gen : expr \rightarrow code * int$$

that returns code to evaluate a given expression, together with the number n of stack slots used by the code, with $n \leq N$. 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.]

Answer: The function *gen* shown below always returns a depth that is at most N; so in compiling *Binop* (w, e_1, e_2), the only case where a temp is needed is if the costs for both e_1 and e_2 are equal to N.

```
let temp = ref 0

let alloc_temp () = incr temp; !temp

let rec gen =
  function
      Number n → (CONST n, 1)
    | Variable x → (LDGW x.x_name, 1)
    | Monop (w, e1) →
        let (code1, c1) = gen e1 in
        (SEQ [code1; MONOP w], c1)
    | Binop (w, e1, e2) →
        let (code1, c1) = gen e1
        and (code2, c2) = gen e2 in
        if c1 ≥ c2 && c2 < N then
          (SEQ [code1; code2; BINOP w], max c1 (c2 + 1))
        else if c1 ≤ c2 && c1 < N then
          (SEQ [code2; code1; SWAP; BINOP w], max c2 (c1 + 1))
        else
          (* c1 = c2 = N *)
          let y = alloc_temp () in
          (SEQ [code2; PUT y; code1; GET y; BINOP w], N)

let gen_expr e = fst (gen e)
```

Given the observation that the value c returned by *gen* represents the stack depth used by the code, it seems clear that the function generates code that is feasible, in that it never occupies more than N cells on the stack.

[The following is not really required, but I include it for completeness.] To see that the code is optimal, in that it contains the smallest possible number of *PUT/GET* pairs, note that the code corresponds to an expression tree in which certain nodes have been marked so that they will be saved in a temp. If a node has two children with the same cost c, then marking one of them allows the node itself to be computed with cost c instead of $c + 1$. The code generated above exploits this fact to keep the cost down to N in cases where it would otherwise rise to $N + 1$. It is clear that none of these marks could be eliminated without spoiling the feasibility.

But now consider a feasible marking in which some node a has a marked child b, but a is a *light* node with at least one child of cost strictly less than N. We will argue that the mark can be removed from b, and either the marking will remain feasible, or it can be made feasible again by adding a mark closer to the root of the tree next to a node that is not light. Thus we can take any marking with one or more marks adjacent to light nodes and move or remove these marks one at a time to obtain a tree with at worst the same number of *PUT/GET* pairs where no light node has a marked child.

Unmarking the child b can raise the cost of node a by at most one, and this can increase in turn the costs of a's parent and its more distant ancestors. There are several ways in which this path of increased costs can terminate. It may reach a node whose cost is not increased when one of its children increases in cost (because the increased child was the cheaper one, and has not been made equal in cost to the other). It may reach the root of the tree, raising the cost of the root by one but leaving it no bigger than N. It may reach a node that already has a marked child, and for that reason terminate by failing to raise the cost of the node. The worst that can happen is that the path reaches a node whose children, both unmarked, had costs $N - 1$ and N, and raises the costs to N and N; in that case, we can mark one of the two children and leave the cost of the node itself as *N*. Whichever case applies, we have removed the mark on a child of a light node, at the possible cost of adding a mark on a child of a node that is not light.

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.

Answer: (a) A little experimenting with the materials for Lab 1 helps here. This source program containing an `elsif` chain:

```
begin
  if x = 0 then y := 1
  else if x = 1 then y := 2
  else if x = 2 then y := 3
  end end end
end.
```

compiles into the following code (reformatted to save space):

```
!  if x = 0 then y := 1
LDGW _x; CONST 0; JEQ 1
JUMP 2
LABEL 1
CONST 1; STGW _y
JUMP 3
LABEL 2
!  else if x = 1 then y := 2
LDGW _x; CONST 1; JEQ 4
JUMP 5
LABEL 4
CONST 2; STGW _y
JUMP 6
LABEL 5
!  else if x = 2 then y := 3
LDGW _x; CONST 2; JEQ 7
JUMP 8
LABEL 7
CONST 3; STGW _y
JUMP 9
LABEL 8
LABEL 9
LABEL 6
LABEL 3
```

This code is a bit untidy because each test becomes a jump over a jump, there are multiple labels at the end, and it contains a useless jump (to label 9) at the end, as does the code for any `if` statement lacking an `else` part.

Again, this program, where `if`'s are nested inside the `then` parts of other `if`'s:

```
begin
  if x = 0 then
    if x = 1 then
      if x = 2 then y := 3
      else y := 4 end
    else y := 5 end
  else y := 6 end
end.
```

generates still poorer code:

```
!   if x = 0 then
LDGW _x; CONST 0; JEQ 1
JUMP 2
LABEL 1
!     if x = 1 then
LDGW _x; CONST 1; JEQ 4
JUMP 5
LABEL 4
!       if x = 2 then y := 3
LDGW _x; CONST 2; JEQ 7
JUMP 8
LABEL 7
CONST 3; STGW _y
JUMP 9
LABEL 8
!       else y := 4 end
CONST 4; STGW _y
LABEL 9
JUMP 6
LABEL 5
!     else y := 5 end
CONST 5; STGW _y
LABEL 6
JUMP 3
LABEL 2
!   else y := 6 end
CONST 6; STGW _y
LABEL 3
```

Here there is a long chain of jumps that lead to other jumps.

(b) All these blemishes can be tidied up with a few obvious optimisation rules, suitable for a peephole optimiser that maintains an equivalence relation on labels. First, multiple labels at the same place can be made equivalent:

- LABEL a; LABEL b → LABEL b, making $a \equiv b$.

Next, a jump to an immediately following label can be elided:

- JUMP a; LABEL b → LABEL b, when $a \equiv b$.

Third, a conditional jump that just skips over an unconditional jump becomes simpler if it is reversed:

- JEQ a; JUMP b; LABEL a → JNEQ b; LABEL a.

(It's more than likely that the label will subsequently be deleted.)

These three rules are sufficient to tidy up the first code example. For the second example, we need the rule that a label followed by an unconditional jump can be deleted, and the label made equivalent to the target of the jump.

- LABEL a; JUMP b → JUMP b, provided initially $a \not\equiv b$, making $a \equiv b$ afterwards.

This rule on its own is sufficient to improve the code in the second example, so that all the unconditional jumps lead to a single label at the end.

(c) Fixing up the code with a peephole optimiser is easy; generating the same code directly is less so. First, the code generated by *gen_cond e tlab flab* can be improved by observing that in any application either *tlab* or *flab* labels the next instruction. Better code results in each case if *gen_cond* is replaced by a pair of mutually recursive functions *jump_if_true* and *jump_if_false* that cover these two cases. Here is a version of *jump_if_false*:

```
let jump_if_false e lab =
  match e with
      Number x →
        if x = 0 then JUMP lab else NOP
    | Binop ((Eq | Neq | Lt | Gt | Leq | Geq) as w, e₁, e₂) →
        SEQ [gen_expr e₁; gen_expr e₂; JUMPC (negate w, lab)]
    | Monop (Not, e₁) →
        jump_if_true e₁ lab
    | Binop (And, e₁, e₂) →
        SEQ [jump_if_false e₁ lab; jump_if_false e₂ lab]
    | Binop (Or, e₁, e₂) →
        let lab₁ = label () in
        SEQ [jump_if_true e₁ lab₁; jump_if_false e₂ lab; LABEL lab₁]
    | _ →
        SEQ [gen_expr e; JUMPB (false, lab)]
```

For conditional jumps, this uses a function *negate* defined so that, for example, *negate Leq* = *Gt*.

In order to tidy up the jumps and labels generated from control structures, various ad-hoc approaches yield partial solutions. A more systematic approach follows from the observation that, in generating code for a statement, we need information about any labels attached to the code that follows, so as to avoid generating duplicate labels. We can also allow the context to inform us that, instead of continuing with the next statement, we should instead branch to a specified label; in this way, we can avoid generating branches that lead to other branches.

Let use define a *continuation* to be a value of type *cont*:

```
type cont =
      Next                      (* Fall through to next statement *)
    | CanJump of codelab        (* Fall through to label *)
    | MustJump of codelab       (* Jump to a label *)
```

We will replace the existing *gen_stmt* with a function

```
val gen_stmt : stmt → cont → cont * code,
```

with the idea that if *gen_stmt s* k_1 = (k_2, *code*), then k_1 gives information to *gen_stmt* about the following context of the code it generates, and it returns with the code a continuation k_2 that describes the context for the code that precedes it.

Given a suitable library of continuation-handling functions, we can define *gen_stmt* as follows:

```
let rec (gen_stmt : stmt → cont → cont ∗ code) = function s →
  match s with
      Skip → skip
    | Seq stmts →
        sequence (List.map gen_stmt stmts)
    | Assign (v, e) →
        quote (SEQ [LINE v.x_line; gen_expr e; STGW v.x_lab])
    | IfStmt (test, thenpt, elsept) →
        with_exit_lab (fun lab₂ →
          sequence [
            with_exit_lab (fun lab₁ →
              sequence [quote (jump_if_false test lab₁);
                gen_stmt thenpt;
                jump lab₂]);
            gen_stmt elsept])
    | WhileStmt (test, body) →
        let lab₁ = label () in
        with_exit_lab (fun lab₂ →
          sequence [put_label lab₁;
            quote (jump_if_false test lab₂);
            gen_stmt body;
            jump lab₁])
```

This definition resembles the existing definition of *gen_stmt*. The only unfamiliar looking form is

```
with_exit_lab (fun lab → ...),
```

which introduces a label *lab* that is attached to the code immediately following, either reusing an existing label or creating a new one for the purpose. To go through the details:

- *skip* generates no code, and sets the continuation for the code that precedes it to be the same as the code that follows.

  ```
  let skip k = (k, NOP)
  ```

- *seq* s_1 s_2 joins together the code from s_1 and s_2, propagating continuations along the chain.

  ```
  let seq s₁ s₂ k =
    let (k₂, code₂) = s₂ k in
    let (k₁, code₁) = s₁ k₂ in
    (k₁, SEQ [code₁; code₂])
  ```

- *sequence ss* does the same thing, but with a sequence of code fragments instead.

  ```
  let sequence ss = List.fold_right seq ss skip
  ```

- if *code* is a piece of code that is independent of its context, then *quote code* takes the context into account, inserting a *JUMP* instruction when it is needed.

  ```
  let quote code₁ k =
  ```

```
    match k with
        (Next | CanJump _) → (Next, code₁)
      | MustJump lab → (Next, Seq [code₁; Jump lab])
```

- To place a label, we use *put_label lab*, which also informs preceding code that a label is present.

```
let put_label lab _ = (CanJump lab, Label lab)
```

- To jump to a label, we use *jump lab*. This does not generate a *Jump* instruction directly, but informs preceding code that it must jump to the label instead of continuing to the next statement. If the label is in fact attached to the next statement following, then no jump is needed.

```
let jump lab k =
  match k with
      CanJump lab₁ when lab₁ = lab → (CanJump lab, Nop)
    | _ → (MustJump lab, Nop)
```

- Lastly, *with_exit* attaches a fresh label to the immediately following code, unless a label is there already, or the code wants us to branch to a different label instead. The argument function f receives both the label and a continuation that allows it to terminate by jumping to the label.

```
let with_exit_lab f k =
  match k with
      Next →
        let lab = label ( ) in
        let (k₁, code) = f lab (CanJump lab) in
        (k₁, Seq [code; Label lab])
    | CanJump lab →
        f lab (CanJump lab)
    | MustJump lab →
        f lab (MustJump lab)
```

The code generated by this version of *gen_stmt* is very close to optimal, but it is still possible to create a duplicate label in cases where the code from *gen_cond* ends with a label, but there is also a label on the immediately following statement. This could be prevented by making *gen_cond* also accept a continuation. The name *continuation* is an allusion to a much more general way of organising the compiling process, described at length in Andrew Appel's book *Compiling with continuations* (Cambridge, 1992).

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.

Answer: (a) We use an abstract syntax tree to represent expressions:

```
type expr =
    Const of value | Var of ident
  | Plus of expr * expr | Times of expr * expr
```

To generate code, we use a recursive function that uses pattern matching to prioritise fruitful uses of ADDMUL over less fruitful uses.

```
let rec gen_expr =
  function
      Const v → CONST v
    | Var x → LOAD x
    | Plus (e1, Times(e2, e3)) →
        SEQ [gen_expr e1; gen_expr e2; gen_expr e2; ADDMUL]
    | Plus (Times (e1, e2), e3) →
        SEQ [gen_expr e3; gen_expr e1; gen_expr e2; ADDMUL]
    | Plus (e1, e2) →
        SEQ [gen_expr e1; gen_expr e2; CONST 1; ADDMUL]
    | Times (e1, e2) →
        SEQ [CONST 0; gen_expr e1; gen_expr e2; ADDMUL]
```

As in the course, this function represents the code as a cat-tree, using *SEQ* [. . .] for the internal nodes.

(b) This sequence operates in depth 3:

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

(c) The expression $e_3 + e_4 * e_5$ could be evaluated in depth 5 by evaluating the three sub-expressions in the order e_5, e_4, e_3 to obtain values v_5, v_4, v_3 on the stack, then permuting the values into the order v_3, v_4, v_5 and applying an ADDMUL instruction. This permutation cannot be achieved using the SWAP instruction, because e_5 must be evaluated last and its value must appear lowest on the stack.

(d) A suitable instruction SWAP2 swaps the top item on the stack with the item next but one below it. Together, SWAP and SWAP2 allow any permutation of the top three values to be achieved; though actually there's no need to swap the top two items before invoking ADDMUL, and an acceptable permutation can be achieved with at most one extra instruction.

The plan for evaluating an expression $x + y * z$ is to evaluate x, y and z in decreasing order of cost, permute the values to put x at the bottom, and finish with an ADDMUL instruction. If the sub-expressions are evaluated in the order u, v, w with costs $c_u \geq c_v \geq c_w$, then the cost of the whole expression is $max(c_u, c_v + 1, c_w + 2)$. If the evaluation order is y, x, z, for example, then the code will be

Evaluate y, evaluate x, SWAP, evaluate z, ADDMUL

For an expression $x * y + z * w$, either of the multiplications can be combined with the addition, and the best approach can be determined from the costs of the four sub-expressions. The whole process can be accomplished in linear time by first annotating each sub-expression with its cost, then computing the code that achieves this cost in a second pass.

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.

Answer: (a) For the abstract syntax, we can use a list of (*guard*, *body*) pairs for the `while` statement. The `loop` statement and the embedded `exit` statements are syntactically independent from each other.

```
type stmt = ...
  | WhileStmt of (expr * stmt) list
  | LoopStmt of stmt
  | ExitStmt
```

The concrete syntax is specified as follows:

```
stmt : ...
  | WHILE clauses END                { WhileStmt $2 }
  | LOOP stmts END                   { LoopStmt $2 }
  | EXIT                             { ExitStmt } ;
```

```
clauses :
    expr DO stmts                      { [($1, $3)] }
  | expr DO stmts ELSIF clauses        { ($1, $3) :: $5 } ;
```

(b) We need a function *gen_stmt* that takes an exit label as an argument:

```
let rec gen_stmt exitlab =
  function ...
    | WhileStmt clauses →
        let top_lab = label( ) in
        let gen_clause (test, body) =
          let $lab_1$ = label ( ) and $lab_2$ = label ( ) in
          SEQ [gen_cond test $lab_1$ $lab_2$;
            LABEL $lab_1$; gen_stmt exitlab body; JUMP top_lab;
            LABEL $lab_2$] in
        SEQ [LABEL top_lab; SEQ (List.iter gen_clause clauses)]
    | LoopStmt body →
        let top_lab = label ( ) and bot_lab = label ( ) in
        SEQ [LABEL top_lab;
          gen_stmt bot_lab body; gen (JUMP top_lab);
          gen (LABEL bot_lab)]
    | ExitStmt →
        JUMP exitlab
```

(c) For the `while` statement, writing `x` and `y` for the offsets −4 and −8:

```
LABEL 1
LDLW x; LDLW y; JLEQ 2
LDLW x; LDLW y; MINUS; STLW x
JUMP 1
LABEL 2
LDLW x; LDLW y; JGEQ 3
LDLW y; LDLW x; MINUS; STLW y
JUMP 1
LABEL 3
```

For the equivalent `loop` statement, with labels 1 and 5 pertaining to the loop and labels 2, 3, and 4 to the embedded `if` statement:

```
LABEL 1
LDLW x; LDLW y; JLEQ 2
LDLW x; LDLW y; MINUS; STLW x
JUMP 4
LABEL 2
LDLW x; LDLW y; JGEQ 3
LDLW y; LDLW x; MINUS; STLW y
JUMP 4
LABEL 3
JUMP 5
LABEL 4
JUMP 1
LABEL 5
```

(d) The code for the `loop` statement contains jumps that lead to other, unconditional jumps. These can be eliminated by a peephole optimiser that keeps a table of equivalents, using the rule

- LABEL a; JUMP b → JUMP b, provided $a \not\equiv b$, with $a \equiv b$ added to the table afterwards. Applying this rule gives the program,

  ```
  LABEL 1
  LDLW x; LDLW y; JLEQ 2
  LDLW x; LDLW y; MINUS; STLW x
  JUMP 1
  LABEL 2
  LDLW x; LDLW y; JGEQ 5
  LDLW y; LDLW x; MINUS; STLW y
  JUMP 1
  JUMP 5
  JUMP 1
  LABEL 5
  ```

  The remaining useless JUMP instructions can be removed by a rule for deleting unreachable code:

- JUMP a; xxx → JUMP a provided xxx is not a label.

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 programming 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

$$gen\_expr : expr \to code.$$

Control structures are translated using a function

$$gen\_cond : expr \to codelab \to codelab \to code,$$

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.

Answer: (a) Abstract syntax:

```
type expr =
    . . .
  | IfExpr of expr ∗ expr ∗ expr
```

The concrete syntax can be incorporated into a parser at the top level of the hierarchy for expressions: rename the existing *expr* nonterminal to $expr_0$, then add

```
expr :
    expr_0                          { $1 }
  | IF expr THEN expr ELSE expr     { IfExpr ($2, $4, $6) } ;
```

This does not create ambiguity because (among other things) each `if` in an expression must have a matching `else`.

(b) We must make *gen_expr* and *gen_cond* mutually recursive, and exploit the freedom on a stack machine to leave values on the evaluation stack across jumps.

```
let rec gen_expr =
  function
      . . .
    | IfExpr (e_1, e_2, e_3) →
        let lab_1 = label () and lab_2 = label () and lab_3 = label () in
        SEQ [gen_cond e_1 lab_1 lab_2;
          LABEL lab_1; gen_expr e_2; JUMP lab_3;
          LABEL lab_2; gen_expr e_3; LABEL lab_3]
```

Continuing the mutually recursive definition begun above,

```
and gen_cond e tlab flab =
  match e with
      . . .
    | IfExpr (e_1, e_2, e_3) →
        let lab_1 = label () and lab_2 = label () in
        SEQ [gen_cond e_1 lab_1 lab_2;
          LABEL lab_1; gen_cond e_2 tlab flab;
          LABEL lab_2; gen_cond e_3 tlab flab]
    | . . .
```

(c) The given statement is treated as equivalent to

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

The generated code is as follows.

```
LDGW _i; CONST 0; JLT L1
<Code for a[i]>; LDGW _x; JLEQ L3
JUMP L2
LABEL L1; JUMP L3
LABEL L2
<Code for i := i+1>
LABEL L3
```

where <Code for a[i]> is

```
GLOBAL _a; LDGW _i; CONST 1; MINUS
CONST 4; TIMES; OFFSET; LOADW
```

and <Code for i := i+1> is

```
LDGW _i; CONST 1; PLUS; STGW _i
```

Translating the short-circuit conditional directly (and as above, using specific code for the `if` statement with no `else`) gives

```
LDGW _i; CONST 0; JLT L3
<Code for a[i]>; LDGW _x; JLEQ L3
<Code for i := i+1>
LABEL L3
```

We can fix up the previous code by using a peephole optimiser that tracks equivalences between labels, using the rules

- LABEL a; JUMP b → JUMP b, where $a \not\equiv b$, making $a \equiv b$
- JUMP a; JUMP b → JUMP a
- JUMP a; LABEL a → LABEL a
- LABEL a → [] if a is never used

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.

Answer: In the record type `dogrec`, the `age` field is at offset 12 and the `next` field at offset 16. So in the following, the first LDNW instruction replaces the address of a record by the contents of its `age` field, and second one fetches the `next` field.

```
LDGW _s
LDGW _q
LDNW 12
PLUS
STGW _s
LDGW _q
LDNW 16
STGW _q
```

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.

Answer: (I'll give many details here that aren't necessarily expected in a student's answer. Those who have an appetite for more details still can look at the file prob3.diff in the wiki, giving changes needed to the code of Lab 2 to implement the local construct and also the loop construct in Exercise 3.3.)

Beginning with the abstract syntax, we add a new kind of statement:

```
type stmt = ...
  | LocalStmt of decl list * stmt
```

Then it's necessary to enhance the lexer with keywords local and in; and the parser with the same keywords as tokens, and with the syntax for the new kind of statement:

```
stmt : ...
  | LOCAL decls IN stmts END                { LocalStmt ($2, $4) } ;
```

In the semantic analyser, we must add a case to *check_stmt*:

```
let rec check_stmt s env =
  match s with...
    | LocalStmt (decls, body) →
        let env' = check_decls decls (new_block env) in
        check_stmt body env'
```

For this to work, we'll need to enhance the *Dict* module to support nested blocks; something like the version of *Dict* that was supplied with Lab 2 will fit the bill. This is necessary so that (as in the example) inner blocks can declare names that are also declared in the outer blocks that surround them.

Two additional things need to be take care of in semantic analysis. Since all storage in the language is statically allocated, we will need to collect together a list of all the declarations made in the whole program, so that the code generator can produce assembler directives to reserve storage for each of them. To this end, we can add a global variable that is updated by *check_decl* to keep a list of all definitions that it creates, and save this list as an annotation at the root of the AST, changing the type to

```
type program = Program of decl list * stmt * def list ref
```

Also, because two different variables can have the same name, it's necessary to adjust *check_decl* so that it gives a different label to each definition it creates. It's easiest to do this by appending the value of a counter to each label. The complete new version of *check_decl* is as follows:[ii]

```
let gensym = ref 0
let globals = ref [ ]

(* check_decl – check declaration and return extended environment *)
let check_decl (Decl (vs, t)) env0 =
  let declare x env =
    incr gensym;
    let lab = sprintf "_$_$" [fStr x.x_name; fNum !gensym] in
    let d = make_def x.x_name t lab in
    globals := d :: !globals;
    x.x_def ← Some d; add_def d env in
  List.fold_left (fun env1 x → declare x env1) env0 vs
```

In the code generator, dealing with the new construct is easy: the semantic analyser has already dealt with the declaration, so all we need to do is produce code for the statement that forms the body:

```
let rec gen_stmt =
  function . . .
    | LocalStmt (decls, body) →
        gen_stmt body
```

Instead of traversing the global declarations to reserve storage, the code generator can use the list of definitions saved by the semantic analyser:

```
let gen_decl d =
  let s = 4 in
  printf "GLOVAR $ $\n" [fStr d.d_lab; fNum s]

(* translate – generate code for the whole program *)
let translate (Program (ds, ss, glodefs)) =
  . . .
  List.iter gen_decl !glodefs
```

(The constant 4 here should be replaced by a calculation of the size occupied by *d.d_type* in a solution that also implements the changes required in Lab 3.)

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
```

[ii] The use of *fold_left* here is a bit clumsy because the arguments of *declare* come in the wrong order. In some of the lab compilers, I use instead a function *Util.accum* with type $(\alpha \to \beta \to \beta) \to \alpha\ list \to \beta \to \beta$ and the obvious definition that fits in better with the syntax-first style.

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.

Answer: (a)

```
type stmt = ...
  | LoopStmt of name * stmt list
  | ExitStmt of name
  | NextStmt of name
  | ...
```

(b) During analysis of the loop body, the symbol table should contain a definition of the loop name that contains two labels, one for use in the `next` statement and another for the `exit` statement.

(c) The semantic analyser should check that statements `exit L` and `next L` occur only inside a loop labelled `L`. Each occurrence of the label `L`, whether at the top of a loop or in a `exit` or `next` statement, should be annotated with a definition containing the two labels.

(d)

```
let rec gen_stmt =
  function ...
    | LoopStmt (x, body) →
        let d = get_def x in
        SEQ [LABEL d.d_nextlab; gen_stmts body;
          JUMP d.d_nextlab; LABEL d.d_exitlab]
    | NextStmt x →
        let d = get_def x in JUMP d.d_nextlab
    | ExitStmt x →
        let d = get_def x in JUMP d.d_exitlab
    | ...
```

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.

Answer: This question has a nice answer if we choose to use an applicative, persistent, functional data structure to represent sets of variables, but a less nice answer if we allow ourselves to get distracted into representing sets by an imperative data structure such as a hash table. In OCaml, the library provides a module for sets represented as binary trees which we can import with the (slightly cryptic) code,

```
module IdSet = Set.Make(struct
  type t = ident
  let compare = compare
end)
```

The function *check_expr e ini* checks that all variables used in the expression *e* are in the set *ini*, giving an error message if not:

```
let rec check_expr e ini =
  match e with
      Number n → ()
    | Variable x →
          if not (IdSet.mem x.x_name ini) then
            err_message x.x_line
              "Variable $ may not be initialised" [fStr x.x_name]
    | Monop (w, e1) → check_expr e1 ini
    | Binop (w, e1, e2) → check_expr e1 ini; check_expr e2 ini
```

To check a statement, we must do two things: check that the statement does not use any variables that are not initialised, and compute the set of variables that are initialised after the statement has been executed. The function *check_stmt* returns this latter set.[iii]

```
let rec check_stmt s ini =
  match s with
      Skip → ini
    | Seq ss →
        List.fold_left (fun ini1 s → check_stmt s ini1) ini ss
    | Assign (x, e) → check_expr e ini; IdSet.add x.x_name ini
    | Print e → check_expr e ini; ini
    | Newline → ini
    | IfStmt (test, thenpt, elsept) →
        check_expr test ini;
        let s1 = check_stmt thenpt ini
        and s2 = check_stmt elsept ini in
        IdSet.inter s1 s2
    | WhileStmt (test, body) →
        check_expr test ini;
        ignore (check_stmt body ini);
        ini
```

Note that we are taking a conservative approximation by not allowing for variables that are initialised in the body of a `while` loop, since the loop body may not be executed at all. That means we will report that variables may be used without being initialised when in fact no execution of the program would do that. In practice, this seems a much more useful approximation than the other one; it is easy for the programmer to insert a dummy initialisation if the warning appears wrongly.

For the `if` statement, checking the two arms *thenpt* and *elsept* results in two sets of variables s_1 and s_2, each of which may extend the given set *ini*. We take the

[iii] The OCaml library function *ignore*, defined by

```
let ignore x = ()
```

allows an expression of any type to be used as a 'statement' of type *unit*.

intersection of the two sets in order to say that a variable is initialised after the if under the circumstances that it was in *ini* originally, or it is initialised by both arms of the conditional.

The last detail is that the whole program is checked beginning with the empty set of variables:

```
let check_prog (Program ss) =
  ignore (check_stmt ss IdSet.empty)
```

In a usable implementation, it would be wise to report each variable as uninitialised only once, and that would require keeping track of the set of variables that have been mentioned in a warning so far.

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.
```

Answer: This code comes from the *ppc4* compiler in the lab kit. I turned on the peephole optimiser to make the code more compact.

```
MODULE Main 0 0
IMPORT Lib 0
ENDHDR

FUNC _double 0
!   return x + x
LDLW 16         -- push x
LDLW 16         -- push x again
PLUS            -- compute x + x
RETURN          -- return
END

FUNCC _apply3 0
!   return f(3)
CONST 3         -- push argument 3
LDLW 20         -- fetch static link of f
LDLW 16         -- fetch code address of f
PCALLW 1        -- call f with 1 argument
RETURN          -- result from f is returned by apply3
END

FUNC MAIN 0
!   print_num(apply3(double));
CONST 0         -- static link for double
GLOBAL _double  -- code address for double
CONST 0         -- static link for apply 3
GLOBAL _apply3  -- address of apply3
PCALLW 2        -- call apply3 with 2 words of args
CONST 0
GLOBAL _print_num
PCALL 1         -- call library routine
!   newline()
CONST 0
GLOBAL _newline
PCALL 0         -- call library routine
RETURN
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.

Answer:

```
proc flip$^1$(x: integer): integer;
  proc flop$^2$(y: integer): integer;
  begin
    if y$^{(2,16)}$ = 0 then return 1
    else return flip$^1$(y$^{(2,16)}$-1) + x$^{(1,16)}$ end
  end;
begin
  if x$^{(1,16)}$ = 0 then return 1 else return 2 * flop$^2$(x$^{(1,16)}$-1) end
end;
```

(For the stack layout, see Figure 6.)

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

```
proc fac(n: integer;
```

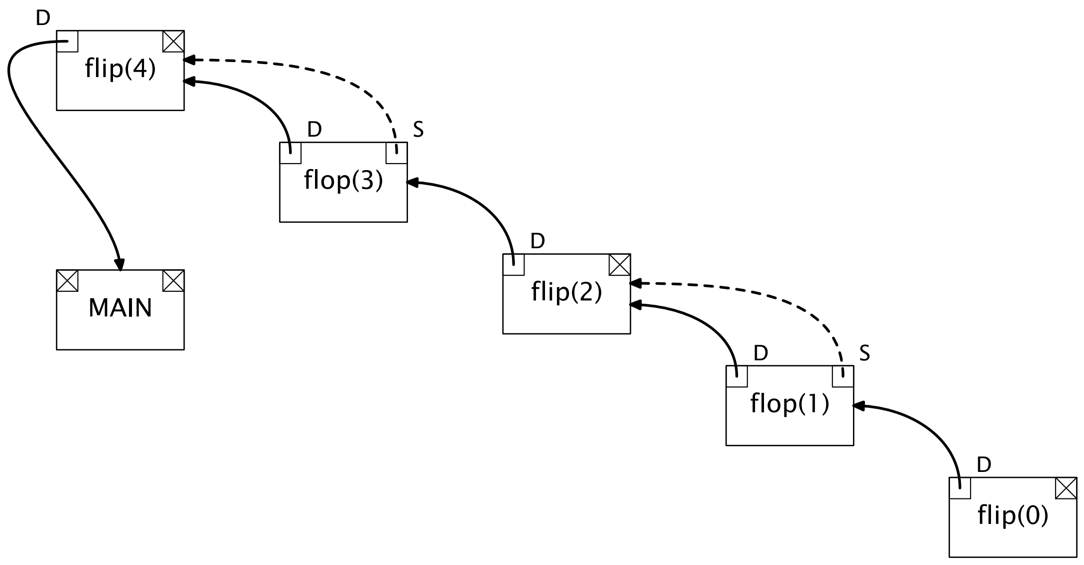


Figure 6: *Stack layout for exercise 4.2*

```
            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).

Answer: (a) The sequence of calls is as follows:

$$\text{fac}(3, \text{id}) \longrightarrow \text{fac}(2, \text{k1}^3) \longrightarrow \text{fac}(1, \text{k1}^2) \longrightarrow \text{fac}(0, \text{k1}^1)$$
$$\longrightarrow \text{k1}^1 1 \longrightarrow \text{k1}^2(1) \longrightarrow \text{k1}^3(2) \longrightarrow \text{id}(6),$$

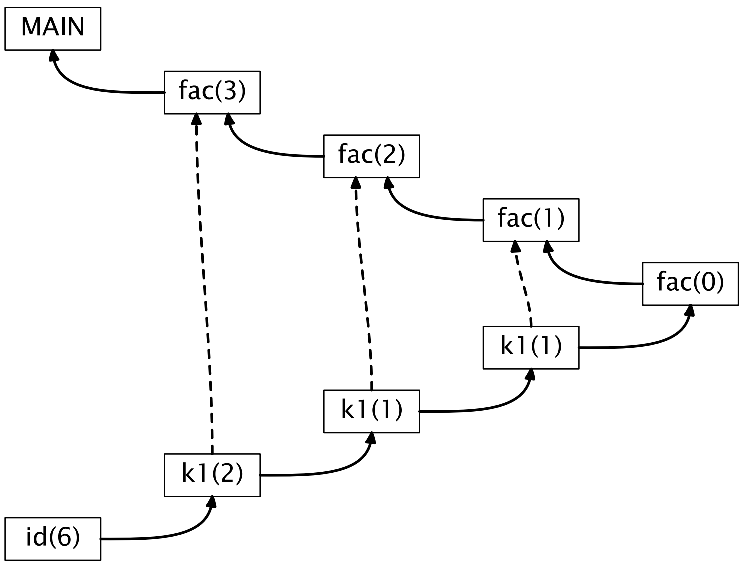


Figure 7: *Stack layout for exercise 4.3*

where $\mathtt{k1}^n$ denotes the instance of k1 that is nested inside the invocation of fac with the specified value of n. Because k1 is the only procedure that is nested inside another, only it has non-null static links. Each invocation of k1 uses its static link to fetch n and k from the enclosing invocation of fac. The stack layout is as shown in Figure 7. Typically, the procedural argument k that is invoked by k1 is itself the instance of k1 in the next invocation of fac going down the stack.

(b) The call k(n * r) results in the code

```
LDLW 12
LDNW 16
LDLW 16
TIMES
LDLW 12
LDNW 24
LDLW 12
LDNW 20
PCALLW 1
```

The call fac(n–1, k1) results in the code

```
LOCAL 0
GLOBAL _fac.k1
LDLW 16
CONST 1
MINUS
CONST 0
GLOBAL _fac
PCALLW 3
```

4.4 [2013/3; see ppc4/test/sumarray.p] Figure 8 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?

Answer: (a) See Figure 9.

(b) Using only basic operations for clarity:

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

```
LOCAL 24; LOADW          -- fetch address of v
LOCAL -4; LOADW          -- fetch value of i
CONST 4; TIMES; OFFSET   -- add offset for v[i]
LOADW                    -- fetch value of v[i]
LOCAL 20; LOADW          -- static link for f
LOCAL 16; LOADW          -- code address for f
PCALL 1                  -- call f
```

This can be abbreviated to LDLW 24; LDLW -4; LDIW; LDLW 20; LDLW 16; PCALL 1.

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

```
LOCAL 12; LOADW          -- fetch static link
CONST -4; OFFSET; LOADW  -- fetch value of s
LOCAL 16; LOADW          -- fetch x
PLUS
LOCAL 12; LOADW          -- fetch static link
CONST -4; OFFSET; STOREW -- store into s
```

Abbreviated, this becomes LDLW 12; LDNW -4; LDLW 16; PLUS; LDLW 12; STNW -4.

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

```
LOCAL 16; LOADW          -- fetch address of v
LOCAL 0                  -- static link for add
GLOBAL _add              -- code address for add
CONST 0                  -- static link for dovec
GLOBAL _dovec            -- code address for dovec
```

```
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 8: *Program for exercise 4.4*

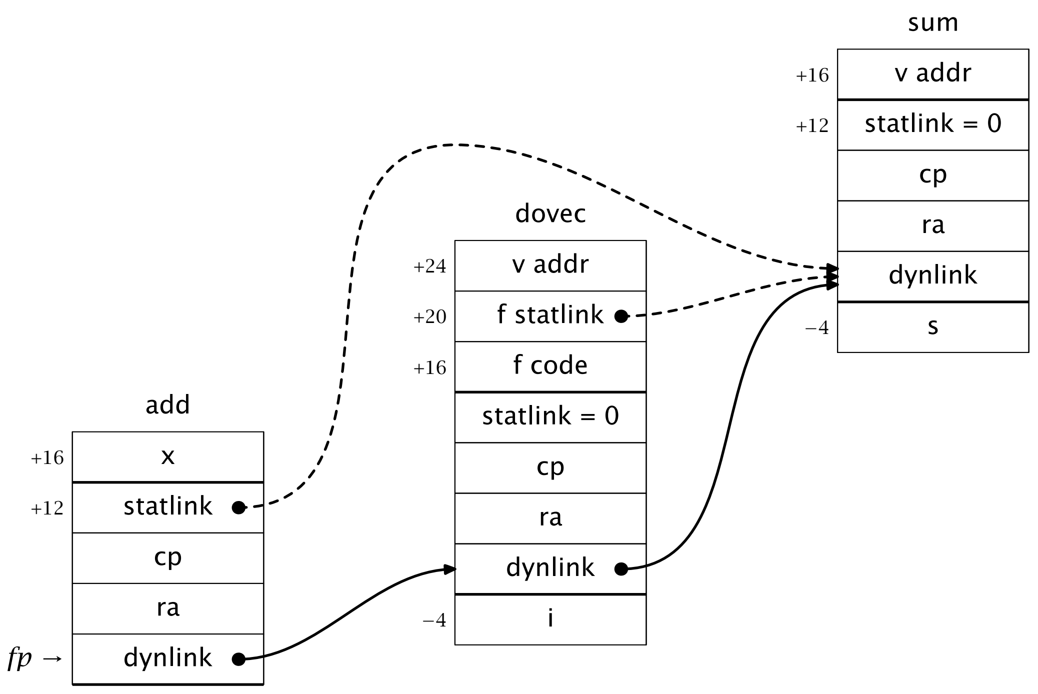


Figure 9: *Stack layout for exercise 4.4*

```
PCALL 3                    -- call dovec
```

The only abbreviation possible here is to start with LDLW 16.

(c) The parameter array would have to be copied at the time of each call to *sum* or *dovec*. Space for the copied parameter would be allocated in the stack frame for each of these procedures below the local variables. The parameter itself would still be passed as the address of the argument array, but code at the beginning of the procedure body would copy it into the space allocated for it; in Keiko, there is a special FIXCOPY instruction for this purpose. References to the array in the body of the procedure would refer directly to the local copy, rather than indirectly to the parameter array, so that the sequence shown as LOCAL 24; LOADW in the code above could be replaced by LOCAL v_copy, saving a load on each access to the array.

Passing an array by value would be faster if the procedure body makes frequent references to elements of the array, so that the time saved in not having to access the elements indirectly through the parameter reference would more than pay for the time needed to do the copying. This advantage could be removed if the compiler were able to detect that the parameter reference was constant over the procedure body and move it to a register, loading the register just once.

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.

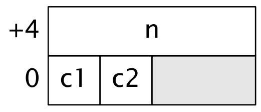
In a different programming language, values of record type are pointers to dynamically allocated storage for a record and these pointers are passed by value, rather like values of class type in Java. Dereferencing of the pointer is implicit in the expression `r.n`.

(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.

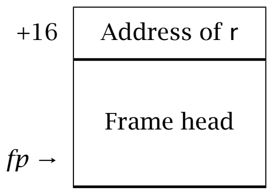
(e) For the Java-like language, give an example of a program demonstrating that parameters are passed by value and not by reference, and state what results are expected from the program in each case.

Answer: (a) Most architectures do not support loads and stores of, e.g., four-byte integers at addresses that are not a multiple of four. It is thus necessary to ensure not only that an integer is allocated four bytes of space, but also that its address is aligned on a four-byte boundary, even if this means inserting some padding. The type `array 4 of char` will also have size 4 but its alignment can be taken as 1.

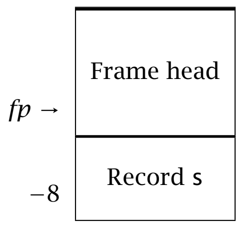
(b) The type `rec` will have one-byte fields for `c1` and `c2` at offsets 0 and 1, two bytes of padding, and a field for `n` at offset 4. (Higher addresses at the top of the picture.)



(c) Using a 16-byte frame head as in the course, `f` will have a frame laid out like this, with one parameter word and no locals.



The procedure g will have this frame layout, with 8 bytes reserved for the local s.



The assignment r.n := r.n + 1 refers to the var parameter r, requiring an extra load instruction in each reference.

```
LDLW 16  ! Load address of r
LDNW 4   ! Load value of r.n
CONST 1; BINOP Plus
LDLW 16  ! Load address of r
STNW 4   ! Store into r.n
```

Instructions used here are those from the lectures, with abbreviations

```
LDLW n = LOCAL n; LOADW
LDNW n = CONST n; OFFSET; LOADW
STNW n = CONST n; OFFSET; STOREW
```

The call f(s) needs us to pass the address of s. Following the conventions of the course, which include a static link of zero for global procedures,

```
LOCAL -8
CONST 0
GLOBAL _f
PCALL 1
```

(d) The parameter r is still represented as the address of a record, and the code for the assignment is unchanged. Now, however, the stack frame of g no longer directly contains a record, but contains the address of a record that is allocated dynamically. So code for the call becomes

```
LDLW -4
CONST 0
GLOBAL _f
PCALL 1
```

(e) We should assign directly to the parameter within the procedure.

```
proc f(r, s: rec);
begin
  r := s
end;

var p, q: rec;

begin (* Main program *)
  p := new(rec); q := new(rec);
  p.n := 3; q.n := 4;
```

```
    f(p, q);
    print p.n
  end.
```

With parameters passed by value, this prints 3; with parameters passed by name, the procedure sets the global p to share the same record as q, and the result is 4.

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.)

Answer: In the specified tree,[iv]

$\langle$*Storew*,
  $\langle$*Loadw*,
    $\langle$*Offset*, $\langle$*Global* _a$\rangle$,
      $\langle$*Lsl*, $\langle$*Loadw*, $\langle$*Local* 40$\rangle\rangle$, $\langle$*Const* 2$\rangle\rangle\rangle\rangle$,
  $\langle$*Local* $-4\rangle\rangle$,

some nodes must compute their values into registers; they are the *Global* and the two *Loadw*'s. Let us agree that they will be computed into registers u_0, u_2 and u_6, in keeping with the longest code sequence shown at the end of this solution.

We can implement each of $\langle$*Storew*, u_6, $\langle$*Local* $-4\rangle\rangle$ and $\langle$*Loadw*, $\langle$*Local* 40$\rangle\rangle$ in either one instruction or two, according to whether we evaluate the local address into a register; this multiplies the number of tilings by a factor of 2 in each case.

With these considerations out of the way, we are left with the trunk of the tree:

$\langle$*Loadw*, $\langle$*Offset*, u_0, $\langle$*Lsl*, u_2, $\langle$*Const* 2$\rangle\rangle\rangle\rangle$

This can be achieved in one instruction, derived using the rule

addr → $\langle$*Offset*, *reg*$_1$, $\langle$*Lsl*, *reg*$_2$, $\langle$*Const n*$\rangle\rangle\rangle$ { [*reg*$_1$, *reg*$_2$, LSL #*n*] }.

But there are six other ways of doing the same job, each corresponding to computing into registers some combination of the three internal nodes *Offset*, *Lsl* and *Const* 2. Using just the rules shown in Figure 8.5, we must compute the *Lsl* into a register, and can obtain four possibilities by choosing to compute the *Offset*, the *Const* 2, or both, needlessly into registers. Additionally, the ARM allows the operand of an arithmetic instruction to contain a shift by a register or a constant, as described by the following rules.

rand → $\langle$*Lsl*, *reg*, *shift*$\rangle$	{ *reg*, LSL *shift* }
shift → $\langle$*Const k*$\rangle$	{ #*k* }
shift → $\langle$*reg*$\rangle$	{ *reg* }

For example, we can combine the *Offset* and *Lsl* into one instruction and obtain the sequence

```
mov u3, #2
add u5, u0, u2, LSL u3
```

[iv] Warning: a perfect answer to this question requires more knowledge of the ARM than perhaps is conducive to continuing mental health.

```
ldr u6, [u5]
```

The only forbidden tiling is the one where the constant is developed into a register but the shift, add, and load are combined:

```
mov u3, #2
ldr u6, [u0, u2, LSL u3] @ Wrong!
```

This is wrong because the shift amount in an effective address calculation must be a constant, though a register can be used in the shift-and-add instruction shown above.

Putting it all together, possible answers to the question are as follows:

- Using only rules from Figure 8.5, a total of $4 \times 2 \times 2 = 16$ tilings.
- Using also the rule that covers the trunk in a single instruction, a total of $5 \times 2 \times 2 = 20$ tilings.
- Using additionally the two rules that allow shifts in a *rand*, a total of $7 \times 2 \times 2 = 28$ ways of tiling the tree.[v]

The shortest instruction sequence (if scaled addressing is allowed) has four instructions. The longest is the sequence of nine instructions where each node is computed into its own register:[vi]

```
ldr u0, =_a
add u1, fp, #20
ldr u2, [u1]
mov u3, #2
lsl u4, u2, u3
add u5, u0, u4
ldr u6, [u5]
add u7, fp, #16
str u6, [u7]
```

Other valid code sequences exist where the constants 16 and 20 are developed into registers before adding them to *fp* either explicitly or via an addressing calculation. Code sequences like that become necessary if a large frame leads to offsets that do not fit in the immediate field of an instruction. They correspond to a different way of coding the tiles involving ⟨LOCAL *n*⟩, but not to a different way of dividing the tree into tiles. It's worth mentioning too that each tiling can be linearised in multiple ways by permuting independent instructions, so that the number of valid sequences is still greater.

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?

Answer: In *eval_reg*, we include the following case:

```
let rec eval_reg r t =
  match t with ...
    | ⟨BINOP Times, $t_1$, $t_2$⟩ →
        let $v_1$ = eval_reg anyreg $t_1$ in
        let $v_2$ = eval_reg anyreg $t_2$ in
        gen_reg "mul" [r; $v_1$; $v_2$]
```

[v] It's instructive to implement the process of parsing with tree grammars and generate the 28 tilings mechanically. A Haskell program appears at the end of these answers.

[vi] As always, we show each value as occupying a distinct virtual register, leaving it to the register allocator to reuse dead registers.

The use of *eval_reg* in place of *eval_rand* for evaluating t_2 reflects the constraint that both operands of the `mul` instruction be in registers.

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

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

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?

Answer: A program that contains an expression such as (in picoPascal)

```
lsl(x, lsl(y, z))
```

will give rise to a tree such as

$$\langle \textsc{Binop}\ Lsl, x, \langle \textsc{Binop}\ Lsl, y, z\rangle\rangle,$$

where x, y and z are trees such as $\langle \textsc{Loadw}, \langle \textsc{Regvar}\ n\rangle\rangle$ corresponding to the three variables. With the rule shown in the question, this tree can be covered using the productions

$$reg \to \langle \textsc{Binop}\ Lsl, reg_1, rand\rangle$$

$$rand \to \langle \textsc{Binop}\ Lsl, reg, reg\rangle$$

and the resulting code contains an 'instruction'

```
lsl r0, r4, r5, LSL r6
```

This is not legal, because the `lsl` instruction is already an abbreviation for a `mov` with a shifted operand. There isn't space in the instruction to specify two shifts, and there isn't provision in the datapath to use the barrel shifter twice in the same instruction.

What we should do instead is introduce a new non-terminal for shift amounts, allowing each one to be a register or a constant, but forbidding the illegal shift-by-a-shift form.

$$reg \to \langle \textsc{Binop}\ Lsl, reg_1, shift\rangle$$

$$shift \to \langle \textsc{Const}\ k\rangle$$
$$shift \to reg$$

We can put a side condition on the $\langle \textsc{Const}\ k\rangle$ form that requires $0 \le k < 32$. That raises the subtle point that the (very uncommon) shifts by a constant greater than 31 will be compiled into instructions that put the shift amount into a register first. This guarantees that the compiler faithfully reproduces the behaviour of shifts in the underlying machine, without our having to spell out that behaviour in the compiler.[vii] It's a matter of taste whether a compiler warning would be appropriate in this case.

5.4 Consider the following data type and procedure:

```
type dogptr = pointer to dogrec;
```

[vii] That behaviour is probably not what you expect: the machine looks at only the bottom byte b of the shift amount, producing a shifted result if $0 \le b < 32$ and zero if $b \ge 32$. Thus shifting by 255 produces zero, but shifting by 257 produces that same result as shifting by 1. (ARMv6, page A5–10.)

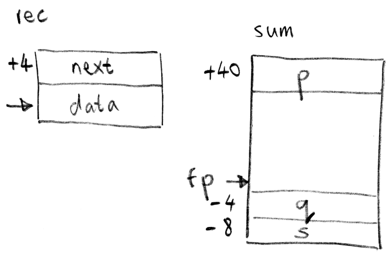


Figure 10: *Layouts for question 1*

dogrec = record name: array 12 of char; age: integer; next: dogptr; 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.

Answer: See Figure 10, which assumes that the frame head occupies 40 bytes. [Note: record layout needs updating.]

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.)

Answer: Two trees:

```
⟨STOREW,
  ⟨BINOP Plus,
    ⟨LOADW, ⟨LOCAL −8⟩⟩,
    ⟨LOADW,
      ⟨OFFSET,
        ⟨LOADW, ⟨LOCAL −4⟩⟩
        ⟨CONST 12⟩⟩⟩⟩,
  ⟨LOCAL −8⟩⟩

⟨STOREW,
  ⟨LOADW,
    ⟨OFFSET,
      ⟨LOADW, ⟨LOCAL −4⟩⟩,
```

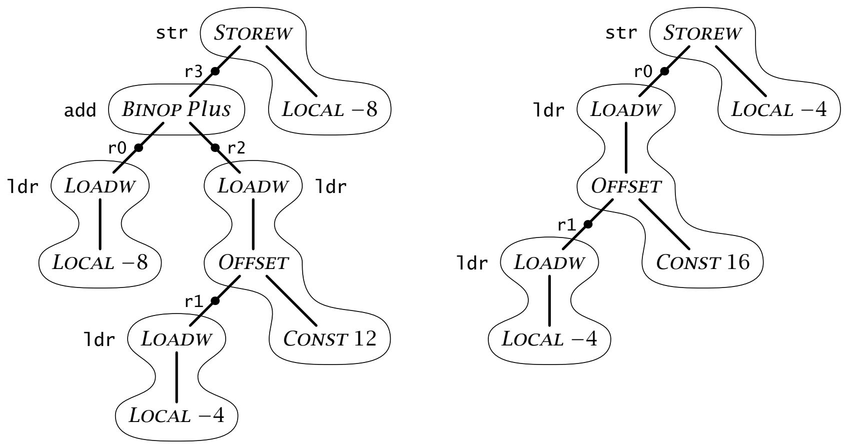


Figure 11: *Trees for Question 5.5*

$\qquad\qquad\langle$*Const* 16$\rangle\rangle\rangle$,
$\qquad\langle$*Local* −4$\rangle\rangle$

(See also Figure 11.)

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

Answer: We can use tiles according to the grammar:

stmt → ⟨*Storew*, *reg*, *addr*⟩	{ str *reg*, *addr* }
reg → ⟨*Binop Plus*, reg_1, reg_2⟩	{ add *reg*, reg_1, reg_2 }
reg → ⟨*Loadw*, *addr*⟩	{ ldr *reg*, *addr* }
addr → ⟨*Local n*⟩	{ [*fp*, #*n*] }
addr → *reg*	{ [*reg*] }
addr → ⟨*Offset*, *reg*, ⟨*Const n*⟩⟩	{ [*reg*, #*n*] }

See Figure 11 once more for the tiling. The object code is as follows:

```
ldr u0, [fp, #-8]
ldr u1, [fp, #-4]
ldr u2, [u1, #12]
add u3, u0, u2
str u3, [fp, #-8]
ldr u4, [fp, #-4]
ldr u5, [u4, #16]
str u5, [fp, #-4]
```

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.

Answer: In the previous code, registers u1 and u4 both contain the value of q, and are computed independently. Within the basic block, we could eliminate the common

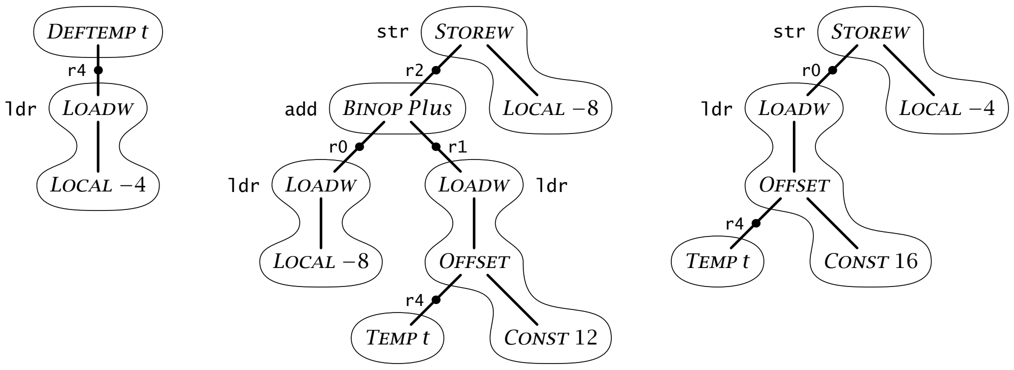


Figure 12: *Trees after CSE*

sub-expression and keep it in a temporary register t. The result can be expressed as a sequence of three trees

```
t := q;
s := s + t↑.data;
q := t↑.next
```

that is to say,

⟨*DEFTEMP t*, ⟨*LOADW*, ⟨*LOCAL* −4⟩⟩⟩

```
⟨STOREW,
  ⟨BINOP Plus,
    ⟨LOADW, ⟨LOCAL −8⟩⟩,
    ⟨LOADW, ⟨OFFSET, ⟨TEMP t⟩, ⟨CONST 12⟩⟩⟩,
  ⟨LOCAL −8⟩⟩
```

```
⟨STOREW,
  ⟨LOADW,
    ⟨OFFSET, ⟨TEMP t⟩, ⟨CONST 16⟩⟩⟩,
  ⟨LOCAL −4⟩⟩
```

(see Figure 12).

Now we need two more tiles:

stmt → ⟨*DEFTEMP t*, *reg*⟩	{ Temp *t* lives in *reg* }
reg → ⟨*TEMP t*⟩	{ Use temp *t* }

The following object code results, using r4 for the temp:

```
ldr r4, [fp, #-4]
ldr r0, [fp, #-8]
ldr r1, [r4, #12]
add r2, r0, r1
str r2, [fp, #-8]
ldr r3, [r4, #16]
str r3, [fp, #-4]
```

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.

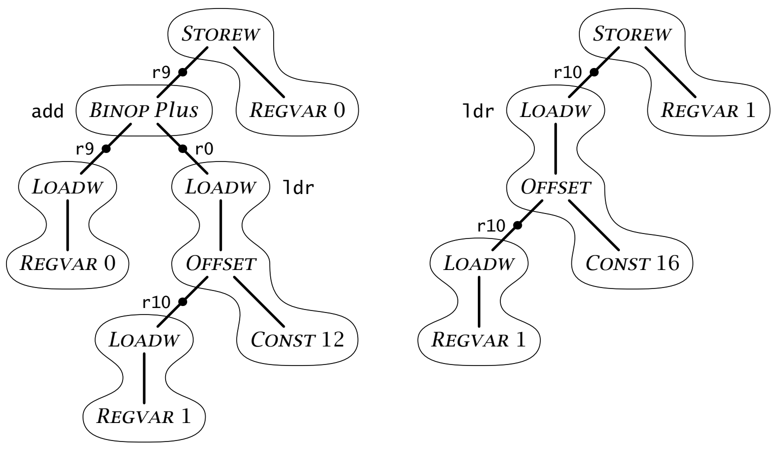


Figure 13: *Trees with register variables*

Answer: Now considering the whole loop, it would be possible to keep both q and s in registers, say r9 and r10. The loop body then becomes

```
ldr r0, [r10, #12]
add r9, r9, r0
ldr r10, [r10, #16]
```

(see Figure 13 for the tiled trees).

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.

Answer: (a) We need to compute the addresses into registers, so we can't use the indexed addressing mode any more. Writing x and y for the frame offsets of the two variables, we get

```
add r0, fp, #x
add r1, fp, #y
movm [r1], [r2]
```

That sequence has cost 3, but a simple load and store have cost 2:

```
ldr r0, [fp, #y]
str r0, [fp, #x]
```

It's fairly reasonable to assign unit cost to each instruction, because the memory access costs of the two sequences are the same. We need to assume also that there is no pipeline stall in either sequence, for example when a load is immediately followed by a store that uses the loaded value – or that both sequences suffer from the stall.

(b) The movm instruction can be used beneficially for the statement x := y, where x and y are global variables, or the statement p↑.x := q↑.x, where p and q are pointer variables and x is a field at offset 0.[viii] For this statement, the addresses p↑ and q↑ must be computed into registers anyway, and the sequence can be completed with either a load and a store or (better) a single movm instruction.

```
ldr r0, [fp, #q]
ldr r1, [fp, #p]
movm [r0], [r1]
```

(c) The single rule *stmt* → ⟨*Storew*, ⟨*Loadw*, reg_1⟩, reg_2⟩ is needed, generating the code movm [reg_2], [reg_1].

(d) In the examples already given, the optimal code for x := y, with tree

⟨*Storew*, ⟨*Loadw*, ⟨*Local y*⟩⟩, ⟨*Local x*⟩⟩,

is obtained by making using the tile ⟨*Storew*, *reg*, ⟨*Local n*⟩⟩ to give a stw instruction. But the pattern ⟨*Storew*, ⟨*Loadw*, *reg*⟩, *reg*⟩ also matches at the root and gives the alternative, sub-optimal code. Neither of these patterns is bigger than the other, and the choice between them must be made depending on whether the two arguments of the *Storew* both need to be computed into registers.[ix]

Further explanation: we are considering here the problem of achieving the effect of the statement *assuming an infinite register set* but using a specific set of instructions, and not the (different) problem of evaluating an expression using the smallest possible number of registers. Each cost vector (*reg*, *addr*, *rand*)

[viii] For MIPS-like machines, the statement a[i] := a[j], where a is a local array, provides an example that benefits from the movm instruction. The limited addressing modes of the MIPS mean that the addresses of both a[i] and a[j] have to computed into registers anyway, and then a single movm instruction could replace a load and a store.

On ARM, however, the better addressing modes allow this statement to be coded as

```
add r0, [fp, #a]
ldr r1, [fp, #j]
ldr r2, [r0, r1, LSL #2]
ldr r3, [fp, #i]
str r2, [r0, r3, LSL #2]
```

and the code using movm is actually longer.

[ix] It isn't actually necessary to have non-nested patterns for the greedy algorithm to fail. For example, the simple, linear tree

⟨*Print*, ⟨*Log*, ⟨*Tan*, ⟨*Load x*⟩⟩⟩⟩

with rules *stmt* → ⟨*Print reg*⟩, *stmt* → ⟨*Print*, ⟨*Log*, *reg*⟩⟩, *reg* → ⟨*Log*, ⟨*Tan*, ⟨*Load x*⟩⟩⟩, *reg* → ⟨*Tan*, *reg*⟩, *reg* → ⟨*Load x*⟩ (all with unit cost) has an optimal tiling with two tiles, but requires three tiles if we choose the largest possible tile to cover the root.

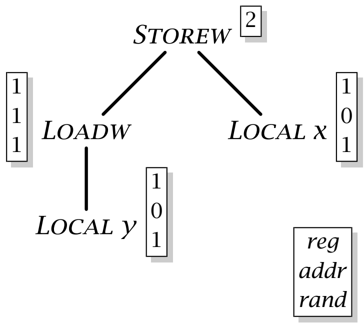


Figure 14: *Cost assignments for* `x := y`

attached to a node denotes the cost (in our case, the number of instructions) for making the value of the node available in a register (*reg*), as an address (*addr*) and as the second operand (*rand*) of an arithmetic instruction. Since there are no constants in the expression, the *rand* cost is in each case equal to the *reg* cost.

We can calculate the costs by starting near the leaves and working upwards: see Figure 14. For the sub-tree ⟨*Loadw*, ⟨*Local y*⟩⟩, we can use the rules *reg* → ⟨*Loadw*, *addr*⟩ [1] and *addr* → ⟨*Local n*⟩ to choose the instruction ldr r1, [fp, #y] with a cost of 1, leaving the result in register r1. This beats the sequence

```
add r0, fp, #y
ldr r1, [r0]
```

that does the addition with an add instruction and not with indexed addressing. The latter sequence corresponds to the rules *reg* → ⟨*Loadw*, *addr*⟩ [1] and *addr* → *reg* and *reg* → ⟨*Local n*⟩ [1].

- The costs for each *Local* node are (1, 0, 1) because it can be calculated into a register with the add instruction shown above, and it can be used as an address with the indexed addressing syntax [fp, #y].
- The costs for the *Loadw* node are (1, 1, 1) because the value can be loaded into a register with the single ldr instruction shown above, and any value in a register can be used as an address or operand with no additional cost (*addr* → *reg* and *rand* → *reg*).
- For the *Storew* node, we need consider only the cost of executing it as a statement.[x] Translating it using *stmt* → ⟨*Storew*, *reg*, *addr*⟩ with a str instruction give a cost of 1 for the instruction, plus 1 for the *Loadw* as a *reg*, and 0 for ⟨*Local x*⟩ as an *addr*, a total of 2. Using *stmt* → ⟨*Storew*, ⟨*Loadw*, *reg*⟩, *reg*⟩ with a movm instruction gives a cost of 1 for the instruction, plus 1 for ⟨*Local y*⟩ as a *reg* and 1 for ⟨*Local x*⟩ as a *reg*, a total of 3.

(e) For `p↑.x := q↑.x`, the tree is

⟨*Storew*,
  ⟨*Loadw*, ⟨*Loadw*, ⟨*Local q*⟩⟩⟩,
  ⟨*Loadw*, ⟨*Local p*⟩⟩⟩

Cost assignments for this tree are shown in Figure 15. Nothing is a constant, so the *rand* costs are equal to the *reg* costs and not shown. Costs for the subtrees

[x] To be fully accurate, each node should have four costs (*reg*, *addr*, *rand*, *stmt*), with the *stmt* cost being irrelevant (and conventionally infinite) everywhere but at the root, and the other costs being infinite at the root.

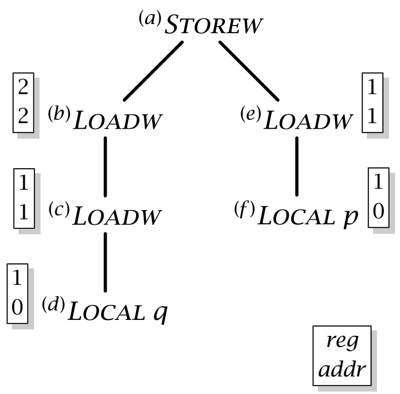


Figure 15: *Cost assignments for* p↑.x := q↑.x

representing ⟨*LOADW*, ⟨*LOCAL*, _⟩⟩ are as before. For the *LOADW* node marked (*b*), the *reg* cost is one greater than the *addr* cost of its child, and its *addr* cost is the same as its *reg* cost. For the *STOREW* node, we can calculate the cost of using the tile ⟨*STOREW*, ⟨*LOADW*, *reg*⟩, *reg*⟩ corresponding to movm as $1 + c_{\text{reg}}(c) + c_{\text{reg}}(e) = 3$. The cost of the tile ⟨*STOREW*, *reg*, *addr*⟩ corresponding to str is $1 + c_{\text{reg}}(b) + c_{\text{addr}}(e) = 4$, so the movm wins.

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.

Answer: [Modified to suit ARM conventions.]

(a) For a[a[i]] := a[i]+i, the initial tree is

```
⟨STOREW,
  ⟨BINOP Plus,
    ⟨LOADW,
      ⟨OFFSET, ⟨GLOBAL _a⟩,
        ⟨LSL, ⟨LOADW, ⟨LOCAL i⟩⟩, ⟨CONST 2⟩⟩⟩⟩,
    ⟨LOADW, ⟨LOCAL i⟩⟩⟩,
  ⟨OFFSET,
    ⟨GLOBAL _a⟩,
    ⟨LSL,
      ⟨LOADW,
```

⟨*Offset*, ⟨*Global _a*⟩,
⟨*Lsl*, ⟨*Loadw*, ⟨*Local i*⟩⟩,
⟨*Const* 2⟩⟩⟩⟩,
⟨*Const* 2⟩⟩⟩⟩

The value table, with reference counts in parentheses:

```
 1  <GLOBAL a> (2)
 2  <LOCAL i> (1)
 3  <LOADW, 2> (2)
 4  <CONST 2> (2)
 5  <LSL, 3, 4> (1)
 6  <OFFSET, 1, 5> (1)
 7  <LOADW, 6> (2)
 8  <BINOP Plus, 7, 3> (1)
 9  <LSL, 7, 4> (1)
10  <OFFSET, 1, 9> (1)
11  <STORE, 8, 10> (0)
```

Of the common sub-expressions, we might treat ⟨*Const* 2⟩ as trivial, leaving ⟨*Global a*⟩ and the two *Loadw*s to be shared. The resulting forest is

⟨*Deftemp* 1, ⟨*Global _a*⟩⟩

⟨*Deftemp* 2, ⟨*Loadw*, ⟨*Local* −4⟩⟩⟩

⟨*Deftemp* 3,
⟨*Loadw*, ⟨*Offset*, ⟨*Temp* 1⟩, ⟨*Lsl*, ⟨*Temp* 2⟩, ⟨*Const* 2⟩⟩⟩⟩⟩

⟨*Storew*,
⟨*Binop Plus*, ⟨*Temp* 3⟩, ⟨*Temp* 2⟩⟩,
⟨*Offset*, ⟨*Temp* 1⟩, ⟨*Lsl*, ⟨*Temp* 3⟩, ⟨*Const* 2⟩⟩⟩⟩

Machine code, with temps labelled `t1` etc., and working registers labelled `u1` etc.:

```
ldr t1, =_a
ldr t2, [fp, #i]
lsl u1, t2, #2
ldr t3, [t1, u1]
add u2, t3, t2
lsl u3, t3, #2
str u2, [t1, u3]
```

This code uses the *reg* + *reg* addressing mode, but doesn't fold the shifts into the addressing mode: other codings are possible. We can put `u1` and `u2` in `r0` and `u3` in `r1`.

(b) For the other example, we begin with

⟨*Storew*,
⟨*Minus*, ⟨*Loadw*, ⟨*Local x*⟩⟩,
⟨*Loadw*, ⟨*Local y*⟩⟩⟩,
⟨*Local x*⟩⟩

⟨*Storew*,
⟨*Minus*, ⟨*Loadw*, ⟨*Local x*⟩⟩,
⟨*Loadw*, ⟨*Local y*⟩⟩⟩,
⟨*Local y*⟩⟩

```
⟨STOREW,
  ⟨MINUS, ⟨LOADW, ⟨LOCAL x⟩⟩,
    ⟨LOADW, ⟨LOCAL y⟩⟩⟩,
  ⟨LOCAL z⟩⟩
```

The value table, exploiting the trick of giving each node ⟨*STOREW*, i, j⟩ an extra child ⟨*LOADW*, j⟩:

```
 1  <LOCAL x> (2)
 2  <LOADW, 1> (1)
 3  <LOCAL y> (3)
 4  <LOADW, 3> (2) – temp 1
 5  <MINUS, 2, 4> (1)
(kill 2)
 6  <LOADW, 1> (2) – temp 2
 7  <STOREW, 5, 1, 6> (0)
 8  <MINUS, 6, 4> (1)
(kill 4)
 9  <LOADW, 3> (2) – temp 3
10  <STOREW, 8, 3, 9> (0)
11  <MINUS, 6, 9> (1)
12  <LOCAL z> (2)
13  <LOADW, 12> (1)
14  <STOREW, 11, 12, 13> (0)
```

So temp 1 contains the original value of y; temp 2 contains the new value of x; temp 3 contains the new value of y. Forest:

⟨*DEFTEMP* 1, ⟨*LOADW*, ⟨*LOCAL* y⟩⟩⟩

⟨*DEFTEMP* 2, ⟨*MINUS*, ⟨*LOADW*, ⟨*LOCAL* x⟩⟩, ⟨*TEMP* 1⟩⟩⟩

⟨*STOREW*, ⟨*TEMP* 2⟩, ⟨*LOCAL* x⟩⟩

⟨*DEFTEMP* 3, ⟨*MINUS*, ⟨*TEMP* 2⟩, ⟨*TEMP* 1⟩⟩⟩

⟨*STOREW*, ⟨*TEMP* 3⟩, ⟨*LOCAL* y⟩⟩

⟨*STOREW*, ⟨*MINUS*, ⟨*TEMP* 2⟩, ⟨*TEMP* 3⟩⟩, ⟨*LOCAL* z⟩⟩

Machine code:

```
ldr t1, [fp, #y] -- t1
ldr u1, [fp, #x] -- t1, u1
sub t2, u1, t1 -- t1, t2
str t2, [fp, #x] -- t1, t2
sub t3, t2, t1 -- t2, t3
str t3, [fp, #y] -- t2, t3
sub u2, t2, t3 -- u2
str u2, [fp, #z] -- none
```

This time, we could get away with two registers, putting t1 and t3 in one register and u1, u2 and t2 in another; our lab compiler prefers to use callee-save registers for the temps and others as scratch registers, so it uses three different registers for this example.

Addendum: As promised, here is a Haskell implementation of parsing with tree grammars. Let's represent the trees as rose trees, extended to allow embedded non-terminals.

```
data RTree a = Node a [RTree a] | NT Class
data Class = Stmt | Reg | Addr | Rand deriving (Show, Eq)

instance Show a => Show (RTree a) where
  show (Node x ts) =
    "<" ++ show x ++ concat [ ", " ++ show t | t <- ts ] ++ ">"
  show (NT c) = show c
```

We need to include as operators only the ones that are needed for our example subject; and for present purposes we can omit the arguments in operators like LOCAL n and GLOBAL _a.

```
data Op =
  STOREW | LOADW | OFFSET | GLOBAL | LOCAL | LSL | CONST
  deriving (Eq, Show)

subject =
  Node STOREW [Node LOADW [Node OFFSET [Node GLOBAL [ ],
        Node LSL [Node LOADW [Node LOCAL [ ]], Node CONST [ ]]]],
    Node LOCAL [ ]]
```

A grammar rule has a name (prefixed with a star if it corresponds to an instruction), a non-terminal class as its LHS and a tree with embedded non-terminals as its RHS.

```
data Rule = Rule String Class (RTree Op)

rules = [
  Rule "∗str"    Stmt (Node STOREW [reg, addr]),
  Rule "∗ldr"    Reg  (Node LOADW [addr]),
  Rule "∗addfp"  Reg  (Node LOCAL []),
  Rule "local"   Addr (Node LOCAL []),
  Rule "∗add"    Reg  (Node OFFSET [reg, rand]),
  Rule "index"   Addr (Node OFFSET [reg, reg]),
  Rule "scale"   Addr (Node OFFSET [reg, Node LSL [reg, Node CONST []]]),
  Rule "∗ldr="   Reg  (Node GLOBAL []),
  Rule "∗lsl"    Reg  (Node LSL [reg, rand]),
  Rule "lshiftc" Rand (Node LSL [reg, Node CONST []]),
  Rule "lshiftr" Rand (Node LSL [reg, reg]),
  Rule "∗mov"    Reg  (Node CONST []),
  Rule "const"   Rand (Node CONST []),
  Rule "reg"     Rand reg,
  Rule "indir"   Addr reg]
  where
    reg = NT Reg; addr = NT Addr; rand = NT Rand
```

We represent the results of parsing as a tree labelled with rule names, a kind of projection of the subject tree. The parsing process is described in a surprisingly compact way.

```
type Parse = RTree String

dorules :: Class -> RTree Op -> [Parse]
dorules x t = concat [ map (Node name) (match rhs t)
                          | Rule name y rhs <- rules, x == y ]

match :: RTree Op -> RTree Op -> [[Parse]]
match (NT x) t = [ [p] | p <- dorules x t ]
match (Node v ps) (Node w ts) =
  if v /= w then [ ] else map concat (cprod (zipWith match ps ts))
```

The function cprod is Strachey's Cartesian product.

```
cprod :: [[a]] -> [[a]]
cprod [ ] = [[ ]]
cprod (xs:xss) = [ y:ys | y <- xs, ys <- cprod xss ]
```

The endgame: answer is a list of 28 parsings of the subject tree.

```
answer = dorules Stmt subject
```

Type `length answer` for the number, or `putStr (unlines (map show answer))` for a neat listing.

An alternative implementation of tiling written in Prolog can be found in the test case `lab4/test/pprolog.p`, together with a mini-Prolog interpreter written in picoPascal.

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 e1 := e2 *)

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

and expr_guts =
      Var of name                    (* Variable (name, address) *)
    | Sub of expr * expr             (* Subscript e1[e2] *)
    | Binop of op * expr * expr      (* Binary operator e1 op e2 *)

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 $gen\_stmt : stmt \rightarrow code$ 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.

Answer: (a) It's best to use mutually recursive functions *gen_addr* and *gen_expr* that produce *l*-values and *r*-values respectively.

```
let rec gen_addr e =
  match e.e_guts with
      Var x → GLOBAL x.x_addr
    | Sub (e₁, e₂) →
        SEQ [gen_addr e₁; gen_expr e₂;
          CONST e.e_size; BINOP Times; BINOP Plus]
    | _ → failwith "gen_addr"

and gen_expr e =
  match e.e_guts with
      (Var _ | Sub (_, _)) →
        SEQ [gen_addr e; LOAD]
    | Binop (w, e₁, e₂) →
        SEQ [gen_expr e₁; gen_expr e₂; BINOP w]
```

Then *gen_stmt* is quite easy:

```
let gen_stmt =
  function
      Assign (e₁, e₂) →
        SEQ [gen_expr e₂; gen_addr e₁; STORE]
```

(b) This requires only systematic use of the definitions. First comes code to find the value of b[i,j], by taking the address _b and adding on 10 times i and 1 times j, then loading from the resulting address.

```
GLOBAL _b
GLOBAL _i
LOAD
CONST 10
TIMES
PLUS
GLOBAL _j
LOAD
CONST 1
TIMES
PLUS
LOAD
```

The next stage is to compute the value of b[1,1] by a similar process.

```
GLOBAL _b
```

```
CONST 1
CONST 10
TIMES
PLUS
CONST 1
CONST 1
TIMES
PLUS
LOAD
```

These two values are then multiplied together; then the address of a[i,j] is calculated and the result is stored there.

```
TIMES
GLOBAL _a
GLOBAL _i
LOAD
CONST 10
TIMES
PLUS
GLOBAL _j
LOAD
CONST 1
TIMES
PLUS
STORE
```

(c) Multiplication by unity could be elided; constant expressions such as `1*1` and `1*10` could be evaluated at compile time, and the resulting expression `_b+10+1` simplified to `_b+11`. The common sub-expressions implicit in the index calculations for a[i,j] and b[i,j] could be shared.

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.

Answer: (a) The procedure receives a copy of the value of a parameter passed by value. Assignments to the formal parameter within the procedure do not affect the subsequent value of a variable passed as an actual parameter, and changes to the actual parameter after the procedure is called do not affect the value of the formal parameter.

For parameters passed by reference, the actual parameter must be a variable or a component of an array or record. Inside the procedure, the formal parameter refers to the location of this variable, so that assignments to the formal parameter have a persistent effect on the value stored in the actual parameter and vice versa.

(b) The following program prints 1 if the parameter of p is passed by value, and 2 if it is passed by reference.

```
proc p(x); begin x := 2 end;

var y;

begin (* main program *)
  y := 1;
  p(y);
  print y
end.
```

(c) We can change the actual parameter from within the procedure, and see if the change is undone when the final value of the formal parameter is written back. This program prints 1 if the parameter of p is passed by value-result, and 2 is it is passed by reference.

```
var y;

proc p(x); begin y := 2 end;

begin (* main program *)
  y := 1;
  p(y);
  print y
end.
```

(d) For a parameter passed by value-result, the procedure can receive the address of the parameter, but also keep a copy among the local variables in its stack frame. The actual parameter is copied into the frame on procedure entry, and copied back when the procedure returns.

For the program above, with the address of x at offset +16 in the frame of p and the local copy at offset -4:

```
FUNC p 4
LDLW 16 ! Copy initial value of x
LOAD
STLW -4
CONST 3 ! Set global y to 2
STGW _y
LDLW -4 ! Copy back final value of x
LDLW 16
STORE
RETURN

FUNC MAIN 0
CONST 1 ! Set y to 1
STGW _y
CONST _y ! Call p, passing address of y
CONST 0
CONST _p
```

```
CALL 2
LDGW _y ! Print y
PRINT
RETURN
```

For consistency with the rest of the course, I've passed a null static link to the procedure p, though this detail is not essential to the question.

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.

Answer: (a) (See handwritten diagram.) The frames for both Q and R have static links that point to the frame for P.

(b) Let's assume that integers have size 4 in the addressing units of the machine. Q must access i and x using its static link, but j is in its frame.

```
FUNC Q 4
LDLW 12
LDNW -4   ! x
CONST _A
LDLW 12
LDNW 16   ! i
```

```
CONST 40
TIMES
OFFSET    ! address of A[i]
LDLW -4   ! j
CONST 4
TIMES
OFFSET    ! address of A[i][j]
STOREW
RETURN
```

(c) The important point here is that R is at the same nesting level as Q, so it passes its own static link as the static link for Q.

```
FUNC R 0
LDLW 12
CONST Q
CALL 2
RETURN
```

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.

Answer: (a) The dynamic chain records the sequence of calls, and is used to reset the base pointer when a procedure returns. The static chain shows how procedures are nested in the program text, and is used for access to non-local variables.

- *base pointer:* a register pointing to a fixed position in the stack frame for the current procedure activation. On a call, it is changed to point to the newly created frame; when a procedure returns, it is restored from the dynamic link.
- *static link:* a pointer, stored in the frame head of a procedure activation, that leads to a stack frame for the textually enclosing procedure. The static link is passed along with the arguments when a procedure is called.
- *dynamic link:* a pointer, stored in the frame head of a procedure activation, that leads to the stack frame of the caller. It is the saved value of the base pointer before the procedure was called, and is used to restore the base pointer when it returns.
- *return address:* the address of the next instruction after the CALL instruction that created a procedure activation. This is saved in the frame head as part of the action of calling a procedure, and is used to restore the program counter when the procedure returns.

(b) A functional parameter is represented by a pair made from the address of the procedure's code and the static link that should be passed when it is called. If a procedure P at nesting level n passes a procedure Q at level m as a functional parameter to another procedure, it must be that $1 \le m \le n + 1$. If $m = 1$ then procedure Q is global and its static link is null. If $m = n + 1$, then Q is local to P, and P passes its own base pointer as the static link for Q. In intermediate cases, Q is local to one of P's ancestors, and it is necessary to follow the static chain for $n - m + 1$ steps in order to find the static link to pass for Q.

(c) (See the handwritten diagram.) At the point in question, there are six activations on the stack, all linked together by their dynamic links: in order from bottom to top, they are `main`, `sumpowers`, 3 activations of `sum`, and `power`. Since the procedures `sumpowers` and `sum` are global, their static links are null; but the `power` frame, and the functional parameters in each of the `sum` frames, all have their static links pointing to the frame for `sumpowers`.