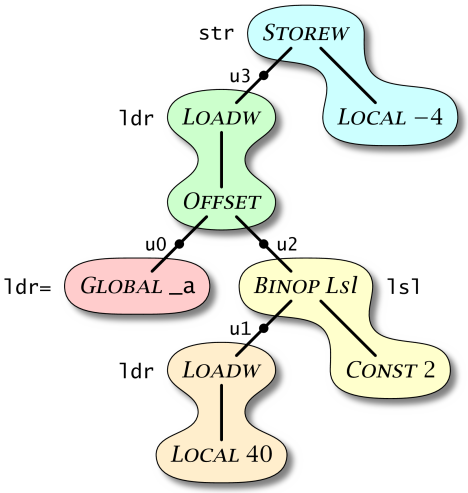
Coursebook

Compilers

Mike Spivey



Draft of 28.xi.2023

Contents

Preface v

1 Syntax 1
- 1.1 Lecture 1: Introduction 1
- 1.2 Lecture 2: Lexical analysis 3
- 1.3 Lecture 3: Syntax analysis 6

2 Expressions and statements 14
- 2.1 Lecture 4: Code for expressions 14
- 2.2 Lecture 5: Control structures 16

3 Lab one: Control structures 24
- 3.1 Setting up the labs 24
- 3.2 Compiler structure 26
- 3.3 Building and running the compiler 27
- 3.4 Implementing `repeat` statements 30
- 3.5 Implementing `loop` statements 31
- 3.6 Implementing `case` statements (optional) 32

4 Types and data structures 36
- 4.1 Lecture 6: Semantic analysis 36
- 4.2 Lecture 7: Data structure access 38

5 Lab two: Array access 44
- 5.1 Compiler structure 44
- 5.2 Functions on types 45
- 5.3 Semantic analysis 45
- 5.4 Code generation 47
- 5.5 Testing the compiler 47
- 5.6 Run-time checks and optimisation (optional) 47
- 5.7 Array assignment (optional) 48

6 Subroutines 49
- 6.1 Lecture 8: Subroutines 49
- 6.2 Lecture 9: Parameters and nesting 52
- 6.3 Lecture 10: Higher-order functions 56

6.4 Lecture 11: A complete compiler 59
6.5 Lecture 12: Objects 62

7 Lab three: Subroutines 69
7.1 Compiler structure 69
7.2 Frame layout 70
7.3 Procedure calls 71
7.4 Parameters and local variables 71
7.5 Nested procedures 72
7.6 Functional parameters (optional) 72

8 Machine code 74
8.1 Lecture 13: The back end 74
8.2 Lecture 14: Instruction selection 81
8.3 Lecture 15: Common sub-expressions 90
8.4 Lecture 16: Register allocation 96

9 Lab four: Machine code 102
9.1 Building and testing the compiler 104
9.2 More addressing modes 106
9.3 Shifter operands 108
9.4 Multiplying by a constant (optional) 109
9.5 Going the other way (optional) 110

10 Revision problems 111

Appendices
A OCaml basics 115
B The Keiko machine 133
C A rough guide to ARM 140
D A machine grammar for ARM 146
E Code listings 150

Preface

This course will show you **one way** to build a compiler for an ordinary programming language (like Pascal or C) that generates reasonably good code for a modern machine (the ARM).

To make the task manageable, we will write the compiler in a functional style in quite a high-level language (OCaml), and we will use power tools (Lex and Yacc) to automate the easy part – analysing the syntax of the input language. The designs we create can also be used to implement hand-crafted compilers without the benefit of these power tools.

Our compiler will be organised for clarity, being structured as the functional composition of many small passes, each carrying out one part of the compiling task. This organisation allows us to focus on the representations of the program being compiled that are handed from one pass to the next, understanding each pass in terms of the transformation it must achieve.

The course doesn't aim to be a survey of all the ways the task might be done, though occasionally we will pause to mention other choices we might have made. Because we will be building a big-ish program, we will naturally want to deploy some of the technical means to deal with large-scale software development: a version control system to keep track of the changes we make, automated building (using Make), and automated testing against a stored corpus of test cases.

The course will bring us into contact with some of the major theoretical and practical ideas of Computer Science. On the theoretical side, we will be using regular expressions and context free grammars to describe the structure of source programs (though relying on automated tools to construct recognisers for us), and we will find a use for many algorithms and data structures. Practically speaking, we will need to deal with the target machine at the level of assembly language, using registers, addressing modes and calling conventions to make a correct and efficient translation of source language constructs.

Lab exercises

This book contains instructions for four practical exercises associated with the course.

- The first exercise (Chapter 3) asks you to add two kinds of loop statement and case statements to a compiler for a simple flowchart language.
- The second exercise (Chapter 5) asks you to implement access to arrays. Programs in the source language still consist of a single routine, but now the language has grown complex enough that a separate type checking or semantic analysis phase is needed in the compiler.
- The third exercise (Chapter 7) asks you to add procedure calls to another variant of the language. This involves generating code for procedures with parameters, nesting, and optionally higher-order functions.
- The fourth exercise (Chapter 9) concerns a compiler that implements a complete, Pascal-like language and contains a code generator that translates operator trees into assembly language for the ARM. You are asked to add rules to the instruction selector that exploit additional addressing modes of the ARM.

The compilers used in Labs 1–3 generate code for the same virtual machine as the Oxford Oberon–2 compiler. Included in the lab kit is an implementation of this virtual machine, based on an interpreter for the Keiko bytecode. Appendix B contains a summary of the Keiko machine at the assembly language level. In Lab 4 we will generate code for the ARM using a set of rules that constitute a tree grammar. A basic guide to the ARM instruction set can be found in Appendix C, and a list of tree grammar rules that describes all the relevant features of the instruction set appears in Appendix D.

Chapter 1

Syntax

1.1 Lecture 1: Introduction

This course covers **one** approach to building a simple compiler. Our compilers will be written in OCaml, exploiting algebraic types, pattern matching and recursion.

- In the first part of the course, we build up to translating a language that is approximately Pascal into *postfix code* for the Keiko machine (like the JVM, but a bit lower level). See Figure 1.1.

- Towards the end of the course, we go further by adding a *back end*, translating Keiko code into code for ARM, the processor in the Raspberry Pi and (very likely) your mobile phone.

```
while x <> y do
  (* Inv: gcd(x, y) = gcd(x0, y0) *)
  if x > y then
    x := x - y
  else
    y := y - x
  end
end
```

```
L1:   LDLW -4
      LDLW -8
      JEQ L2
      LDLW -4
      LDLW -8
      JLEQ L3
      LDLW -4
      LDLW -8
      SUB
      STLW -4
      JUMP L4
L3:   LDLW -8
      LDLW -4
      SUB
      STLW -8
L4:   JUMP L1
L2:   RETURN
```

Figure 1.1: *Postfix code for GCD*

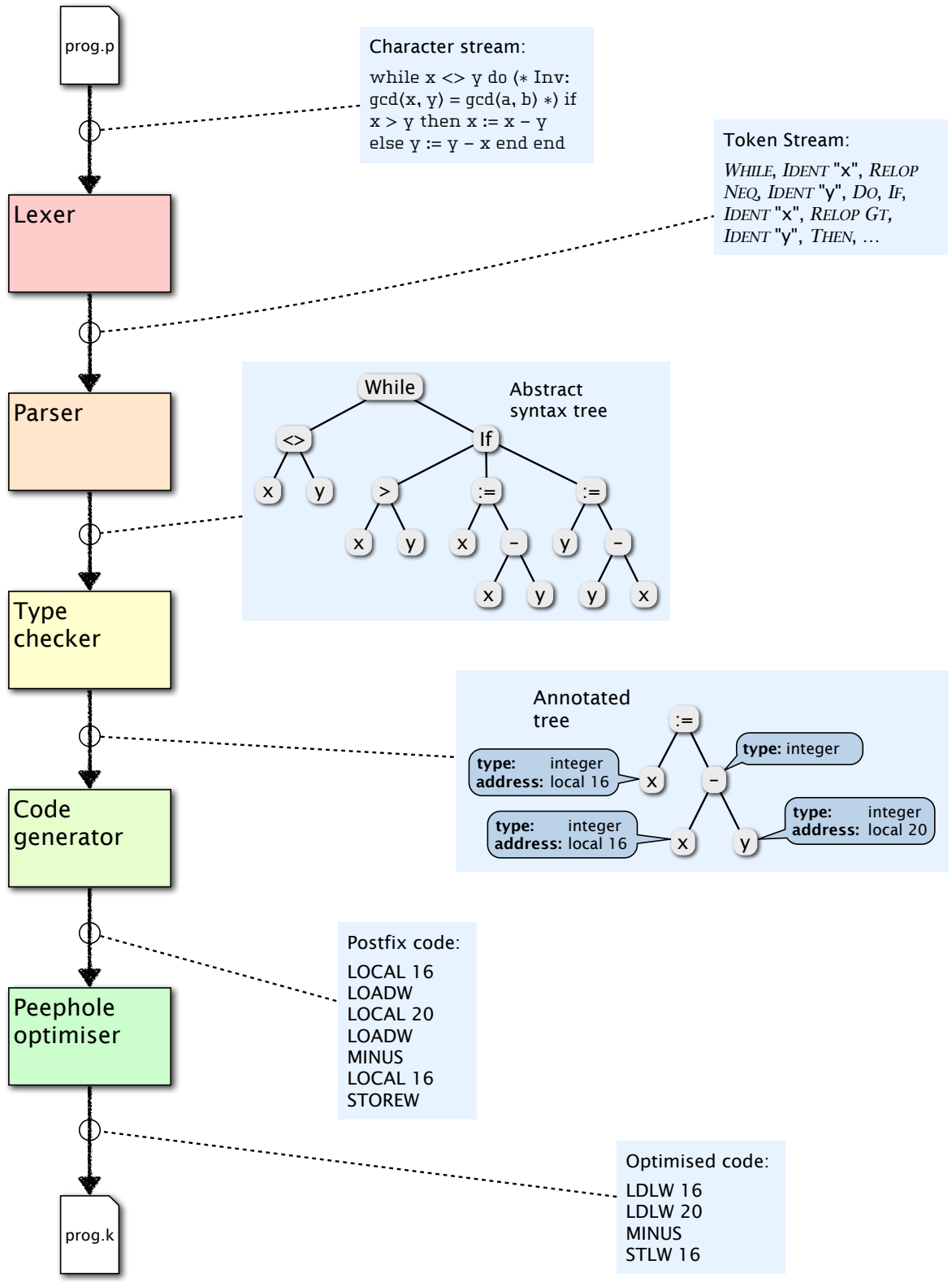


Figure 1.2: *Road map (front end)*

In the Keiko code shown in Figure 1.1, we are supposing that the two variables `x` and `y` are stored at offsets −4 and −8 within the stack frame of a subroutine. The first three instructions implement the condition at the top of the `while` loop, fetching the values of `x` (with `LDLW -4`) and `y` (with `LDLW -8`), then comparing them and jumping to label `L2` (with `JEQ L2`) if they are equal. Since `L2` is placed at the end of this fragment of code, this effectively exits the loop if `x = y`.

The nested boxes show how the translation of the whole program is guided by its structure: the outer box shows the translation of the `if` construct that forms the body of the `while` loop, and the two inner boxes show the translation of the two assignment statements that form the `then` and `else` branches of the `if` statement. The code shown here is not the best translation of the source program because (for example) at label `L4` there is a jump directly to label `L1`, and any jump to `L4` could be improved by redirecting it to go striaght to `L1`. Such improvements can be achieved most simply by subjecting this initial draft of the code – correct but suboptimal – to an optimisation process that repeatedly rewrites the code to make an improvement, until no further improvements are possible. It is easier to achieve compact and efficient code in this way than by considering all possible improvements when the code is first generated.

As the 'road map' in Figure 1.2 shows, we will build *multi-pass* compilers, exploiting the power of functional composition. Single-pass compilers can be super-fast, and usually occupy less storage space with representations of the program being compiled. But these days, machines are fast and storage is cheap, and we will benefit from the added clarity we get by decomposing the compiler into successive transformations.

We will not do a very good job of reporting errors in the source programs that are submitted to our compilers. Briefly, the errors that can be reported are as follows.

- *1: The program is syntactically invalid, so that the parser is unable to construct an abstract syntax tree.
- *2: The program contains undeclared variables or mismatched types that are detected during semantic analysis.
- *3: The program is successfully translated into object code, but at run-time an error is detected, such as a subscript being outside the bounds of the array.

The course is accompanied by lab exercises that call for various tools and techniques:

- Lex and Yacc – generators for lexical and syntax analysers.
- Mercurial – version control.
- Make.
- Home-made regression testing tools.

1.2 Lecture 2: Lexical analysis

Lexical analysis divides the input text into *tokens*: identifiers, keywords, operators, punctuation, so as to

- make parsing easier
- deal with trivial matters by discarding layout, comments, etc.
- read the input quickly.

It isn't hard to write a lexer by hand, but it's made even easier if we use an automated tool.

In modern languages, it tends to be the case that each kind of token can be described by a regular expression. A reminder:

ϵ	empty string
a	literal character
E_1E_2	concatenation
$E_1 \mid E_2$	union
E_1^*	closure = $\epsilon \mid E_1 \mid E_1E_1 \mid E_1E_1E_1 \mid \ldots$

and also

$[abc]$	$= a \mid b \mid c$
$[a\text{–}z]$	$= a \mid b \mid \ldots \mid z$
$E_1?$	$= \epsilon \mid E_1$
E_1+	$= E_1E_1^*$
$[\hat{}abc]$	any character except a, b, c
.	any character at all

Examples:

- [A–Za–z][A–Za–z0–9]∗ – identifiers in Pascal.
- –?[0–9]+|0x[0–9a–f]+ – integer constants in C.
- "([^"]|"")∗" – string constants in Oberon.

Recall that for any regexp, we can find a non-deterministic finite automaton (NFA) that describes the same set of strings (Thompson's construction). For example:

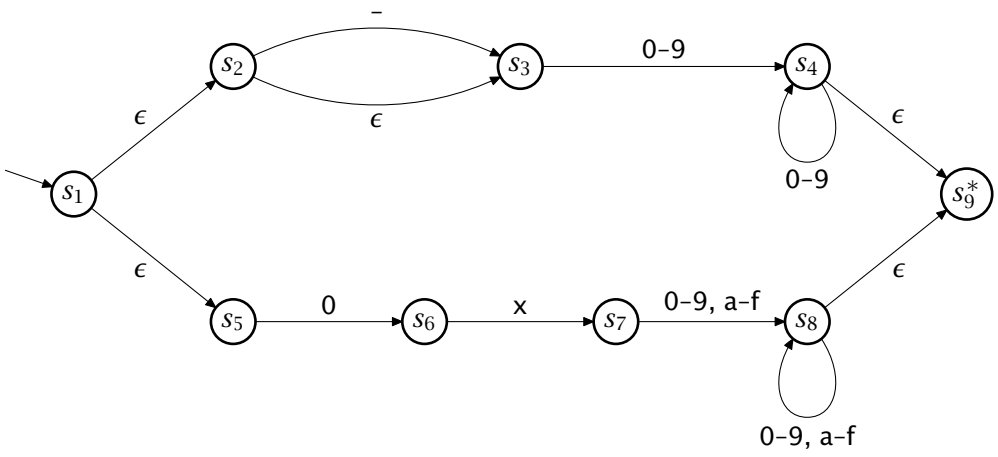


Figure 1.3: *NFA for integer constants*

An NFA accepts a string if there is a path from the *start state* to any *final state* that is labelled with the string.

Recall too that any NFA has an equivalent deterministic finite automaton (DFA) in which the transition relation is a (total) function. Subset construction – each state of the DFA is labelled with a *set* of states of the NFA. The transition function is made total by the existence of a dustbin state $s_\emptyset$, the target of all transitions that do not lead to a valid state.

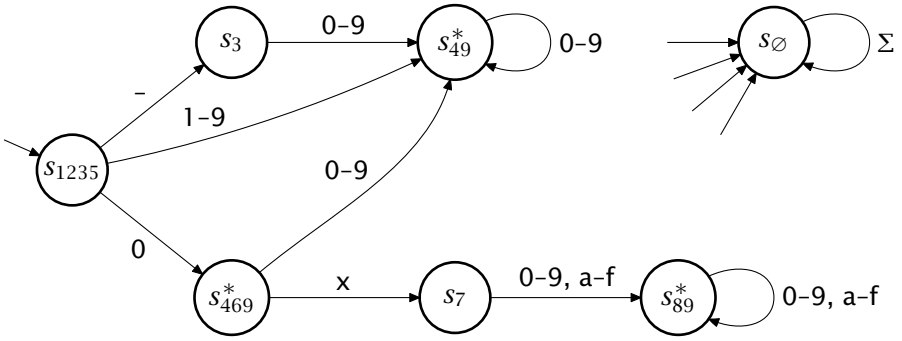


Figure 1.4: *DFA for integer constants*

What Lex does:

- The input script is a list of rules containing regexps E_1, E_2, ..., E_n.
- Make a DFA for the united regexp $E_1 \mid E_2 \mid \ldots \mid E_n$, labelling each final state with the rule that matches.
- Look for the longest match, and break ties by choosing the rule written first in the script.

At runtime, the lexer follows an algorithm that feeds successive input characters to the DFA, looking up each state transition in the *delta* array, and keeping track of the most recent time that the automaton was in an accepting state.

```
state = 1; len = 0; match = -1; i = 0;
while (state != 0) {
      state = delta[state][input[i]]; i = i+1;
      if (accept[state] > 0) {
           len = i; match = accept[state];
      }
}
```

The reading of the input continues until the DFA reaches its 'dustbin' state, numbered zero here. Once that happens, we can be sure that the DFA will never reach an accepting state again, and so that the last time it was in an accepting state corresponds to the longest token that appears at the start of the remaining input. The value of *len* tells us how long the token is, and the value of **match** tells us which rule matched. Before the lexer is invoked again, the *len* characters of the token are removed from the start of the input, and the lexer starts in state 1 again. The *delta* array is sparse, in that for most states there are only relatively few input characters that can be valid and do not lead immediately to the dustbin state. So to save space the array

is packed into a special data structure, still with constant-time access. For efficiency, the lexer reads the input file in big chunks, avoiding too many system calls, and keeps track of where it has reached in the *input* array.

Example: lab1/lexer.mll (see page 150) contains a lexer for a small language. Describe comments by a regexp or (in Ocamllex) by using tail recursion to make a loop.

Data type of tokens:

```
type Token =
    IDENT of string
  | NUMBER of int
  | IF | THEN | ELSE | ...
  | COMMA | ASSIGN | ...
  | ADDOP of op | MULOP of op | ...
  | BADTOK | EOF
```

Note: we could internalise the strings that represent identifiers. We should be more careful with target values.

1.3 Lecture 3: Syntax analysis

If we want to capture the way programs have a nested structure, it's natural to consider recursive rules that say, for example, that a statement can have the form `if–then–else–end`, with an expression and two other (sequences of) statements inside; or the form `while–do–end`, with an embedded expression and a sequence of statements.

stmt → *IF expr THEN stmts ELSE stmts END*

stmt → *WHILE expr DO stmts END*

The idea *stmts* can also be described recursively, as a single statement, or else a statement and a smaller sequence joined with a semicolon.

stmts → *stmt*

stmts → *stmt SEMI stmts*

In these grammar rules, we're describing the form of programs in terms of sequences of tokens as returned by the lexer, and that's a natural thing to do in a compiler.

In most programming languages, the structure of programs can be captured by a context free grammar (CFG), consisting of

- An alphabet Σ of *tokens* or *terminal symbols*.
- An alphabet N of *variables* or *non-terminals*,
- A starting symbol $S \in N$, and
- A set of productions $A \to \alpha$, where $A \in N$ and α is a string over $N \cup \Sigma$.

These grammars are called "context free" because a production $A \to \alpha$ is independent of the context in which A occurs.

Let's explore some examples using a more compact set of conventions where the variables are written in upper case (like E), and we use symbols

like +, ∗ and **id** (an identifier) as the tokens. For example, let G_1 be a grammar with tokens $\{+, *, \mathbf{id}\}$ and one variable E, and the productions $\{E \to \mathbf{id}, E \to E + E, E \to E * E\}$.

Derivation trees: have nodes labelled with grammar symbols. The root is labelled with the starting symbol S, and any node that is labelled with a variable A has children labelled with the symbols from the right-hand side α of a production $A \to \alpha$. E.g. for G_1,

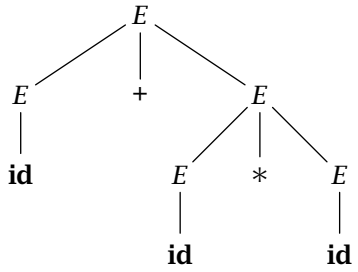


Figure 1.5: *A derivation tree*

The frontier of a derivation tree is a *sentential form* or (if it contains only tokens) a *sentence*. The *language* of G is $L(G) \subseteq \Sigma^*$, the set of all sentences.

Alternatively, a *derivation* is a sequence of strings over $N \cup \Sigma$, with the single symbol S as the first string, and each string obtained from the one before by replacing a single variable A with the RHS α of some production $A \to \alpha$. Sentences usually have more than one derivation for trivial reasons, but in an ambiguous grammar, sentences may have more than one *leftmost* or *rightmost* derivation.

Grammar G_1 is *ambiguous* because some sentences, such as **id** + **id** ∗ **id**, have more than one derivation tree.

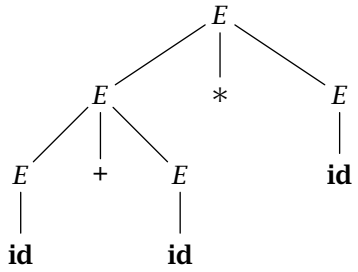


Figure 1.6: *Another derivation tree*

An unambiguous grammar G_2 for the same language:

$E \to T \mid E + T$
$T \to \mathbf{id} \mid T * \mathbf{id}$

(I've simplified this grammar quite a lot from the one used in Lab 1 so as to make the examples shorter. The semantic actions in the *ocamlyacc* script below are a bit messy because of this.)

Stack	Input	Action
ϵ	**id** + **id** $*$ **id**	shift
id	+ **id** $*$ **id**	reduce $T \to$ **id**
T	+ **id** $*$ **id**	reduce $E \to T$
E	+ **id** $*$ **id**	shift
E +	**id** $*$ **id**	shift
E + **id**	$*$ **id**	reduce $T \to$ **id**
$E + T$	$*$ **id**	shift
$E + T *$	**id**	shift
$E + T *$ **id**	ϵ	reduce $T \to T *$ **id**
$E + T$	ϵ	reduce $E \to E + T$
E	ϵ	accept

Figure 1.7: *Moves of a parsing machine*

Parsing problem: for a grammar G. Given $s \in \Sigma^*$, determine whether $s \in L(G)$, and produce a derivation (tree) $S \Rightarrow^* s$. One way of solving the problem is to use a (non-deterministic) bottom-up parsing machine (see Figure 1.7). Each derivation tree corresponds to a unique *rightmost* derivation; and the NBUPM plays out a rightmost derivation in reverse:

$$E \Rightarrow E + \underline{T} \Rightarrow E + \underline{T} * \mathbf{id} \Rightarrow \underline{E} + \mathbf{id} * \mathbf{id}$$
$$\Rightarrow \underline{T} + \mathbf{id} * \mathbf{id} \Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}$$

So $s \in L(G)$ iff there exists a sequence of moves of the NBUPM that accepts the string s. Yacc's solution to the parsing problem is to provide criteria for choosing the next move at each stage, making the machine deterministic.

Abstract syntax trees: the parser communicates with the rest of the compiler by building an abstract syntax tree.

```
type expr = Variable of string
          | Binop of op * expr * expr
```

A yacc script has an expression for each production showing how to build the tree.

```
expr : expr PLUS term          { Binop (Plus, $1, $3) }
  | term                       { $1 } ;

term : term TIMES IDENT        { Binop (Times, $1, Variable $3) }
  | IDENT                      { Variable $1 } ;
```

The key idea is that in an AST, nodes are labelled with *productions*, whereas in a derivation trees they are labelled with grammar symbols. Usually the 'AST grammar' is simpler that the concrete grammar used by the parser.

The parser keeps a stack of trees next to the stack of the BUPM. For example, midway through analysing an expression, there might be symbols $E + T *$ **id** on the parser stack, and next to the E, the T, and the **id** appear trees that have been built for the parts of the expression already analysed.

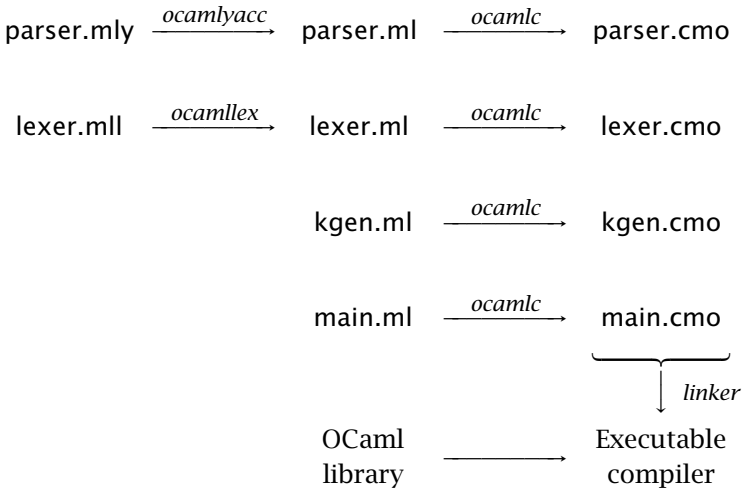
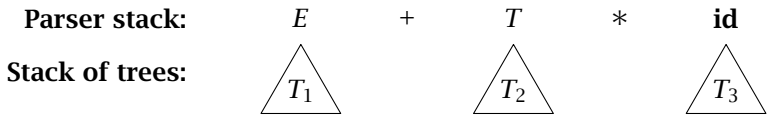
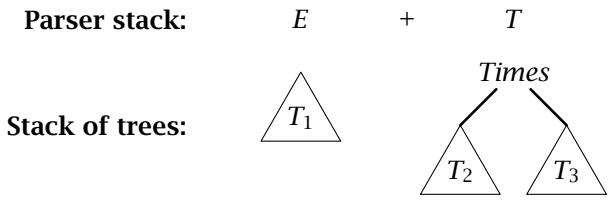


Figure 1.8: *Building a compiler*



When a reduction happens using the production $T \to T * \textbf{id}$, the action associated with that production runs. It refers to the trees for T and **id** as `$1` and `$3` respectively, and from them constructs a bigger tree for the whole term, which is placed next to the LHS symbol on the stack.



In this way, a tree for the whole program can be built using the actions associated with each production.

A shift action of the BUPM pushes a terminal symbol on the stack, and can push a corresponding value on the stack of trees, to become a leaf in the AST (though symbols like *PLUS* and *TIMES* in the example above have no corresponding value).

A reduce action of the BUPM pops the right-hand side of some production from the stack and pushes the non-terminal on the left-hand side of the production. The semantic action of the production gives a rule for computing the corresponding AST, embedding the AST fragments for the symbols on the right-hand side of the production.

Using lex and yacc to build a compiler: The tools *ocamllex* and *ocamlyacc* transform the lexer and parser into OCaml source code that embeds the transition tables for the DFA and the BUPM. This OCaml source is compiled, together with the hand-written OCaml source for the rest of the compiler, then linked with the OCaml library to give an executable compiler (see Figure 1.8).

In the file parser.ml, the automaton that was generated for driving the bottom-up parsing machine is represented by a table of state transitions, and the OCaml library contains interpreters for these tables. Various things can go wrong in the process of building the compiler:

- Errors in the grammar (such as non-terminals that are used but never defined), and grammars that are too hard for the parsing process to handle, result in error messages from *ocamlyacc*.

- Errors in the ML code written for semantic actions in the compiler results in error messages from the OCaml compiler *ocamlc* when it compiles the file parser.ml. The tools do their best to relate the error message back to the offending line in parser.mly, but sometimes the correspondence is not perfect.

Error detection and recovery: an LR parser never shifts a token unless that token could come next in a valid input. So we can *detect* the first error quite well. Error recovery means patching up the input so that we can find more errors after the first. This is still important, but less so than in the days of punched cards and batch processing. Two approaches: minimum cost error repair algorithms, or skipping to the next semicolon.

Exercises

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

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

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

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

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

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

Figure 1.9: *Abbreviated syntax for chains of* `else if`*'s*

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

1.3 Lex has the conventions that the longest match wins, and that earlier rules have higher priority than later ones in the script. These conventions are exploited in the lexer shown in Exercise 1.2 that recognises both keywords and identifiers. Would it be possible to describe the set of identifiers that are not keywords by a regular expression, without relying on these rules? If so, would this be a practical way of building a lexical analyser?

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

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

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

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

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

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

1.7 In the language of Lab 1, **if** statements have an explicit terminator **end** that removes the ambiguity discussed in the preceding exercise. However, this makes it cumbersome to write a chain of **if** tests, since the **end** keyword must be repeated once for each **if**. Show how to change the parser from Lab 1 to allow the syntax shown on the left in Figure 1.9 as an abbreviation for the syntax on the right. An arbitrarily long chain of tests written with the keyword **elsif** can have a single **end**; the **else** part remains optional. Arrange for the parser to build the same abstract syntax tree for the abbreviated program as it would for its equivalent written without **elsif**.

1.8 One grammar for lists of identifiers contains the productions,

idlist → **id**
| *idlist* "," **id**

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

idlist → **id**
| **id** "," *idlist*

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

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

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

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

`* + x y - x y`.

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

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

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

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

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

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

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

%%

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

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

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

Figure 1.10: *Ocamlyacc grammar for Oscars data*

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

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

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

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

Chapter 2

Expressions and statements

2.1 Lecture 4: Code for expressions

The simplest approach to compiling a high-level language is to use a machine with an evaluation stack, and generate postfix code where operations appear after their operands, expecting to find the operand values on the stack and replacing them with the result of the operation.

The Keiko machine: a low-level virtual machine designed for efficient implementation by a byte-code interpreter. (Lower level than JVM because, e.g., in Keiko data structure access is compiled down to address arithmetic.) Instructions communicate via an expression stack.

```
type code =
    CONST of int              (* CONST n *)
  | LDGW of symbol            (* LDGW x *)
  | STGW of symbol            (* STGW x *)
  | MONOP of op               (* UMINUS *)
  | BINOP of op               (* PLUS, MINUS, TIMES, ... *)
  | JUMP of codelab           (* JUMP lab *)
  | JUMPC of op * codelab     (* JLT lab, JEQ lab, ... *)
  | LABEL of codelab          (* LABEL lab *)
  | ...
```

(see Appendix B for a complete list of Keiko instructions used in this book.)

Downstream, an assembler/linker converts the textual form of Keiko programs into a binary form (with typically 1 byte/instruction). At runtime, the bytecode interpreter uses a *big switch* as shown in Figure 2.1, with one case for each kind of instruction.

For compactness and speed, Keiko has many instructions that stand for combinations of others. E.g., LDGW $x \equiv$ GLOBAL x; LOADW, where GLOBAL x pushes the address x onto the stack, and LOADW pops an address and pushes its contents. Using abbreviations like this certainly makes the program shorter, but it makes it faster too, because the time taken to fetch and decode any instruction is many times greater than the time taken to perform the actual work, and fewer instructions mean fewer cycles of fetching and

```
while (!halted) {
    switch (*pc++) {
        case CONST:
            sp++; stack[sp] = *pc++; break;

        case PLUS:
            stack[sp-1] = stack[sp-1] + stack[sp];
            sp--; break;

        ...
    }
}
```

Figure 2.1: *The big switch*

decoding. These abbreviations can be introduced by a peephole optimiser pass in the compiler (or used directly).

A tiny compiler: for a little language with only integer variables; no subroutines; no data structures. No need for semantic analysis. Code sequences are represented using additional *code* constructors:

```
type code =
    ...
  | SEQ of code list
  | NOP
  | LINE of int
```

A value of type *code* is either a single Keiko instruction (e.g., *CONST* 3) or it is a sequence of smaller code fragments, e.g.,

SEQ [*LDGW* "_x"; *CONST* 1; *BINOP Plus*]

The *SEQ* construction can be nested, and that makes it convenient to use in a recursive function that puts together code for a complete program from code that has (recursively) been generated for its parts.

Expressions are compiled into postfix form:

```
let rec gen_expr =
  function
      Variable x → LDGW x
    | Constant n → CONST n
    | Binop (w, e1, e2) →
        SEQ [gen_expr e1; gen_expr e2; BINOP w]
```

This defines a **rec**-ursive function by pattern matching on the argument, an abstract syntax tree of type *expr* (see tree.mli). It returns code that leaves the value of the expression on the stack. The instruction represented by *BINOP w* will appear in the compiler output as some specific Keiko instruction such as PLUS or TIMES, depending on the value of *w*. For example, the code generated for the expression **x+1** is as shown in the figure.

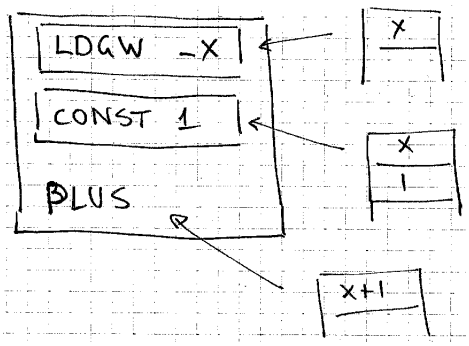


Figure 2.2: *Compiling* `x+1`

In the object code, the global variable `x` is denoted by the symbol `_x`. The initial underscore protects us from confusion between the variable and any part of the runtime system that happens to have the same name. For example, the body of the program is labelled `MAIN` in the object code, and that label does not become confused with the symbol `_MAIN` that corresponds to a global variable named `MAIN`.

2.2 Lecture 5: Control structures

Abstract syntax: can be simpler than the concrete syntax.

type *stmt* =
  Skip
| *Seq* **of** *stmt list*
| *Assign* **of** *name* ∗ *expr*
| *IfStmt* **of** *expr* ∗ *stmt* ∗ *stmt*
| *WhileStmt* **of** *expr* ∗ *stmt*
| ...

Example: The statement `if y > max then max := y end` is represented by the abstract syntax tree

IfStmt (*Binop* (*Gt*, *Variable* "y", *Variable* "max"),
  Assign ("max", *Variable* "y"), *Skip*).

The concrete syntax for an `if` statement might be

`if` *expr* `then` *stmts* `else` *stmts* `end`

where *stmts* denotes a sequence of statements; and (as in Exercise 1.7) chains of `else if`'s might be supported by an abbreviated syntax – but the meaning of all this can be represented in terms of the abstract syntax shown above, by introducing *Seq* costructors wherever there is a sequence of statements that should be treated as a unit.

Code generation: with jumping code for boolean expressions. Assume

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

such that *gen_cond e tlab flab* generates code that jumps to *tlab* if the value of *e* is true and to *flab* if it is false. (Represent true by 1, false by 0). Then

```
let rec gen_stmt =
  function ...
    | IfStmt (test, thenpt, elsept) →
        let lab1 = label () and lab2 = label () and lab3 = label () in
        SEQ [gen_cond test lab1 lab2;
          LABEL lab1; gen_stmt thenpt; JUMP lab3;
          LABEL lab2; gen_stmt elsept; LABEL lab3]
```

So `if y > max then max := y end` becomes

```
LDGW _y
LDGW _max
JGT L1
JUMP L2
LABEL L1
LDGW _y
STGW _max
JUMP L3
LABEL L2
LABEL L3
```

This is correct but messy, and we can plan to use a *peephole optimiser* to tidy it up. This form of optimiser concentrates on a small window of adjacent instructions, looking for groups of instructions that can be made more efficient.

In almost every use we make of the function *gen_cond*, we can be sure that either *tlab* or *flab* labels the very next statement, so that only one branch rather than two is typically needed. The extra branch is easily removed by the peephole optimiser, and the recursive structure of our two-label form of *gen_cond* is a bit simpler than a one-label form that can be used to jump if the condition is true or if it is false.

Rules for the peephole optimiser might include the following:

- LABEL a; LABEL b → LABEL a and make $a \equiv b$.
- JUMP a; LABEL b → LABEL b if $a \equiv b$.
- JGT a; JUMP b; LABEL a → JLEQ b; LABEL a (etc.)
- LABEL a → [] if a is unused.

In our case, we will augment the matching process with a data structure that represents the set of labels in the program, keeping track of which labels are equivalent to each other, and how many times each group of equivalent labels is used in the program. This allows us to make sense of the side conditions "$a \equiv b$" and "a is unused" that appear in the rules, as well as the side effect "make $a \equiv b$".

In the example above, labels L2 and L3 are made equivalent, and then the instruction JUMP L3 becomes redundant; also the JGT can be transformed into a JLEQ, merging it with the following JUMP. Here is the result:

```
LDGW _y
LDGW _max
JLEQ L3
LDGW _y
```

```
STGW _max
LABEL L3
```

In compiling a `while` statement, the best code puts the test at the end, so that the repeating part of the loop contains only one jump.

```
let rec gen_stmt =
  function ...
    | WhileStmt (test, body, elsept) →
        let lab1 = label ( ) and lab2 = label ( ) and lab3 = label ( ) in
        SEQ [JUMP lab2; LABEL lab1; gen_stmt body;
             LABEL lab2; gen_cond lab1 lab3 test; LABEL lab3]
```

So the loop `while r > y do r := r - y end` becomes (after optimisation)

```
JUMP L2
LABEL L1
LDGW _r
LDGW _y
MINUS
STGW _r
LABEL L2
LDGW _r
LDGW _y
JGT L1
```

Short-circuit evaluation: Consider `if (i < n) & (a[i] = x) then ...`. In many languages, this is safe, even if `a[n]` does not exist: the `&` is evaluated in a short-circuit way, with the right-hand operand not evaluated if the left-hand operand determines the answer. To implement this, we want to compile the condition into code that jumps to a label `L1` if it is true and `L2` if it is false.

```
LDGW _i
LDGW _n
JGEQ L2
⟨ code to fetch a[i]⟩
LDGW _x
JNEQ L1
JUMP L2
```

We can achieve this with a suitable definition of the function *gen_cond*.

```
let rec gen_cond e tlab flab =
  match e with
      Binop (And, e1, e2) →
        let lab = label ( ) in
        SEQ [gen_cond e1 lab flab;
             LABEL lab; gen_cond e2 tlab flab]
    | ...
```

(The example code results from tidying up the results a little, and depending on the context it might be tidied up some more.) The analogous treatment of `or` follows from de Morgan's laws, and `not` just swaps the true and false labels:

```
    | Monop (Not, e1) → gen_cond e1 flab tlab
```

For simple comparisons, we can generate a conditional jump for when the comparison is true, followed by an unconditional jump that is taken when it is false.

```
| Binop ((Eq | Neq | Gt | Lt | Leq | Geq) as w, e1, e2) →
    SEQ [gen_expr e1; gen_expr e2; JUMPC (w, tlab); JUMP flab]
```

(Our compiler will wrongly generate non-short-circuit code for assignments like `b := (i < n) & (a[i] = x)`.)

Exercises

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

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

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

(b) Prove that if expression e_1 (containing variables, constants and unary and binary operators) can be evaluated in depth d_1, and e_2 can be evaluated in depth d_2, then *Binop* (w, e_1, e_2) can be evaluated in depth

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

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

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

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

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

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

Assuming $N \ge 2$, define a new version of *gen_expr* that exploits these new instructions, and places no limit on the size of expressions. The code

generated should use as few GET and PUT instructions as possible, but you may ignore the possibility that the source expression contains repeated sub-expressions. There's no need to re-use temps, so you can use a different temp whenever you need to save the value of a sub-expression.

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

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

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.]

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

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

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

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

2.5 [2013/2] The *scalar product machine* uses an evaluation stack, but replaces the usual floating point addition and multiplication instructions with a single ADDMUL instruction that, given three numbers x, y and z on the stack, pops all three and replaces them with the quantity $x + y * z$. Thus the expression $b * b + 4 * a * c$ could be computed on this machine with the sequence

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

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

(a) Suggest a suitable representation for expressions involving addition, multiplication, constants and (global) variables, and describe in detail a

translation process that produces code like that shown in the example, using the smallest possible number of instructions.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Control structures are translated using a function

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

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

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

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

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

expanding the abbreviation in creating the abstract syntax tree.

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

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

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

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

Chapter 3

Lab one: Control structures

In this lab, you are provided with the source code of a compiler for a Pascal-like source language that has only simple integer variables and has no procedures, but includes a number of control structures. Figure 3.1 shows a syntax summary of the language. To avoid confusion, `true` and `false` have been added as keywords, respectively equivalent to the constants `1` and `0`. Notice also that `print` and `newline`, which might be pre-defined procedures in a bigger language, are provided as distinct kinds of statement here; the code generator will translate them as if they were procedure calls.

The compiler generates bytecode for the same virtual machine, Keiko, as the Oxford Oberon–2 compiler, and an interpreter for the bytecode is included with the lab materials. Your task is to augment the compiler with `repeat` and `loop` statements, and optionally `case` statements.

In adding the new forms of statement, you will have to make changes to two principal parts of the compiler:

- to the parser and lexical analyser to make them recognise the new statements,
- to the virtual machine code generator (module *Kgen*) to generate appropriate code for the new statements.

Listings of the files lexer.mll, parser.mly, tree.mli and kgen.ml appear in Appendix E.

3.1 Setting up the labs

Materials for the labs are held in a Mercurial repository on the lecturer's college machine, with read-only anonymous access. In order to check out a copy of the lab materials, you should give the command,

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

This will create a directory called compilers and populate it with the lab materials, with separate sub-directories compilers/lab[1234] for each lab, and an additional sub-directory compilers/keiko, containing the runtime system for the Keiko virtual machine.

Before attempting the lab exercises, you'll need to build the Keiko software, which is used both to translate the textual output of your compiler into

```
program  →  begin stmts end "."

stmts    →  stmt { ";" stmt }

stmt     →  empty
         |  ident ":=" expr
         |  print expr
         |  if expr then stmts [ else stmts ] end
         |  while expr do stmts end

expr     →  ident
         |  number
         |  true | false
         |  monop expr
         |  expr binop expr
         |  "(" expr ")"

monop    →  "-" | not

binop    →  "*" | "/"
         |  "+" | "-"
         |  "<" | "<=" | "=" | "<>" | ">" | ">="
```

Figure 3.1: *Syntax summary of picoPascal*

binary form, and to run the resulting object code. To do this, just change to the keiko sub-directory and type the command,

```
$ (cd compilers/keiko; make)
```

Everything needed will then be built automatically, and the result will be two executable programs called *pplink* and *ppx*. The *pplink* program is the assembler/linker for Keiko bytecode, and *ppx* is an interpreter that executes the bytecode one instruction at a time. If you like, you can explore the source code for these programs, all of which is provided. Some of the C source was in fact generated (using programs written in the interpreted language TCL) from scripts that are also provided.

You will also need to build a small library of OCaml functions that are common to all the lab compilers. To do this, use the command,

```
$ (cd compilers/lib; make)
```

The library contains these modules:

Module	**Description**
Source	Access to source file indexed by lines
Print	Formatted output
Growvect	Extensible arrays

The parts of the compiler you will be modifying in this and future labs make only occasional references to these modules, so you will not need to learn much about their interfaces, still less their implementations.

The module *Source* is particularly useful in compiler experiments, because it makes it simple to include lines from the source program as comments in the compiler output, easing understanding of where the object code came from.

3.2 Compiler structure

The compiler for this lab – like the one in the course – is structured into more than one pass, with an abstract syntax tree as the main interface between each pass and the next. The first pass consists of a parser and lexer built with *ocamlyacc* and *ocamllex*. In this lab, there is no semantic analysis pass, because all variables have the same size and are represented by name in the object code. The second pass generates code for the virtual machine directly. The files can be found in sub-directory lab1 of the lab materials. Here is a list of modules in the compiler:

Module	Description
Tree	Abstract syntax trees.
Lexer	The lexical analyser, generated with *ocamllex*.
Parser	The parser, generated with *ocamlyacc*.
Keiko	Abstract machine code
Peepopt	Peephole optimiser
Kgen	The abstract machine code generator.
Main	The main program.

Each module apart from the main program consists of an interface file called module.mli and an implementation file called module.ml. Often the interface file is little more than a list of functions exported from the module. In other cases (e.g., the *Tree* module) most of the substance is in the interface file, which defines types that are used throughout the compiler. In such cases, the implementation file contains only a few functions that are useful for manipulating the types in question.

In brief, the changes you will need to make in implementing the new control structures are as follows:

- In tree.mli and tree.ml, you will need to add constructors to the type *stmt* of statements to represent the new constructs. (Sadly, because of the rules of OCaml, the type definitions that appear in tree.mli must be duplicated in tree.ml.)
- You will need to change lexer.mll to add new kinds of tokens, and parser.mly to add the syntax of the new statements.
- Finally, you will need to enhance the function *gen_stmt* in kgen.ml to deal with each new statement type.

```
#!/bin/sh

KEIKO=‘cd ../keiko; pwd‘

set -x

./ppc1 $1 >a.k \
    && ../keiko/pplink -custom -nostdlib -i $KEIKO/ppx \
                    ../keiko/lib.k a.k -o a.out >/dev/null \
    && chmod +x a.out
```

Figure 3.2: *The shell script* compile

3.3 Building and running the compiler

You should begin the lab work by building the compiler and trying it out on some example programs. You can build (or re-build) the compiler by changing to the subdirectory compilers/lab1 and giving the command

```
$ make
```

As usual, the *make* program analyses dependencies and file time-stamps, and executes the right sequence of commands to bring the compiler up to date.

The ML source files for the *Parser* and *Lexer* modules are generated by the *ocamlyacc* and *ocamllex* programs, but they contain fragments of code that are copied from the descriptions you write, and these fragments of code may contain errors. There's a certain temptation to edit the files parser.ml and lexer.ml directly to fix any errors that the ML compiler reports, but this is a very bad idea, because your changes would be lost next time these files are re-generated by *ocamlyacc* or *ocamllex*. Most of the contents of these files are impenetrable tables of numbers, however, so the temptation to edit by mistake is reduced.

To help you compile and link picoPascal programs with your compiler, there's a shell script named compile, shown in Figure 3.2. You can invike this script with a command such as

```
$ ./compile gcd.p
```

The translation process happens in three stages:

- First, the picoPascal compiler translates the input program (represented by $1 in the script) to obtain a file a.k containing bytecode in textual form.
- Next, the bytecode assembler *pplink* is used to combine the bytecode with some library functions to obtain a binary bytecode file a.out.
- Finally, the file permissions on a.out are changed so that UNIX treats it as a program that can be run.

The file a.out that is produced exploits a UNIX convention concerning files that begin with "#!". The first line of such a file is taken as the file path of an interpreter – in this case, the program ppx that interprets Keiko bytecode

```
(* gcd.p *)

begin
  x := 3 * 37; y := 5 * 37;
  while x <> y do
    if x > y then
      x := x - y
    else
      y := y - x
    end
  end;
  print x
end.
```

Figure 3.3: *File* gcd.p

– and the rest of the file is passed to it as a program to be executed.[1] After this process, you can run your program by typing ./a.out at the shell prompt, just as if it were a program in native machine code.

For a first experiment with running the compiler, you might like to try running the program in the file gcd.p (see Figure 3.3): it computes the greatest common divisor of two numbers using Euclid's algorithm. This will give you a chance to see the virtual machine code that the compiler produces and trace its execution. To compile this program, simply give the command

```
$ ./compile gcd.p
```

The three individual steps in compiling and linking the program will be shown as they are executed, and the result will be two new files: a.k and a.out. You can run the program a.out from the shell:

```
$ ./a.out
 37
```

If you try to open the file a.out with an editor, you will find that it is in a binary format and makes no apparent sense. On the other hand, the file a.k that is directly output by our compiler is a text file, and its contents for the input gcd.p are shown in Figure 3.4. The object code mostly consists of a single subroutine MAIN that corresponds to the body of the source program. The compiler embeds lines from the source file as comments (starting with !) to show how they correspond to the object code. Details of the instruction set of the Keiko machine are given in Appendix B.

The code shown in Figure 3.4 is a bit messy: it contains jumps that lead to other jumps, and conditional jumps that skip over unconditional jumps to elsewhere. The compiler contains a peephole optimiser that can tidy up these flaws; it is not run by default in order to make debugging easier, but it

[1] A white lie is being told here, because the #! convention works badly when the path to the interpreter may contain spaces, as it might if you store your lab work under a directory with an name like My Lab Work. The actual compile script supplied with the lab materials uses a program /usr/bin/env to work around this problem.

```
MODULE Main 0 0
IMPORT Lib 0
ENDHDR

FUNC MAIN 0
!   x := 3 * 37; y := 5 * 37;
CONST 3
CONST 37
TIMES
STGW _x
CONST 5
CONST 37
TIMES
STGW _y
JUMP L2
LABEL L1
!     if x > y then
LDGW _x
LDGW _y
JGT L4
JUMP L5
LABEL L4
!       x := x - y
LDGW _x
LDGW _y
MINUS
STGW _x
JUMP L6
LABEL L5
!       y := y - x
LDGW _y
LDGW _x
MINUS
STGW _y
LABEL L6
LABEL L2
!   while x <> y do
LDGW _x
LDGW _y
JNEQ L1
JUMP L3
LABEL L3
!   print x; newline
LDGW _x
CONST 0
GLOBAL Lib.Print
PCALL 1
CONST 0
GLOBAL Lib.Newline
PCALL 0
RETURN
END

GLOVAR _x 4
GLOVAR _y 4
```

Figure 3.4: *Code for the gcd program*

can be activated by using the command

```
$ compile -0 gcd.p
```

in place of compile gcd.p.

To help with debugging, it's possible to get the virtual machine to print each instruction as it is executed. Use the shell command,

```
$ ../keiko/ppx -d -d ./a.out
```

The virtual machine prints first a listing of the binary program, then a trace of its execution. Be prepared for a huge amount of output! (With only one -d, the virtual machine prints the listing of the program but not the subsequent trace.)

The makefile also automates the process of testing the compiler by running the four test programs gcd.p, repeat.p, loop.p, and case.p and comparing the output with what was expected:

```
$ make test
*** Test gcd.p
./ppc1 gcd.p >a.k
```

```
../keiko/pplink -nostdlib ../keiko/lib.k a.k -o a.x >/dev/null
../keiko/ppx ./a.x >a.test
sed -n -e '1,/^(\*<</d' -e '/^>>\*)/q' -e p gcd.p | diff - a.test
*** Passed

*** Test repeat.p
./ppc1 repeat.p >a.k
"repeat.p", line 6: syntax error at token 'i'
make[1]: *** [Makefile:28: test-repeat] Error 1
$
```

In this test sequence, the program gcd.p was successfully compiled and run (by invoking the ppx program directly, rather than via the #! convention); then the attempt to compile repeat.p failed because the unmodified compiler does not recognise the syntax of **repeat** statements. The test sequence stops after the first test that fails. After each modification to the compiler, you can re-run the entire battery of tests by giving the command make test again. In this way, you can be sure that later modifications have not spoiled things that were working earlier.

3.4 Implementing repeat statements

Your first task is to implement **repeat** loops, like those provided in Pascal. These differ from **while** loops in that they have the test at the end, like this:

```
repeat
  x := x + 1;
  y := y + x
until x >= 10
```

The loop body is always executed at least once, even if the test is true when the loop begins.

The following suggestions for adding **repeat** statements are roughly in the order of data flow through the compiler. Most of the code can be designed by copying the code for **while** loops with appropriate changes.

(1) In tree.mli, add to the type *stmt* of statements a new constructor *RepeatStmt* with appropriate arguments, and copy it into tree.ml.

(2) In parser.mly, add new tokens *REPEAT* and *UNTIL* to the list near the beginning of the file. Also add a production for the non-terminal *stmt* that gives the syntax of **repeat** statements and constructs the new kind of node in the abstract syntax tree.

(3) In lexer.mll, add entries for "**repeat**" and "**until**" to the list that is used to initialise the keyword table. Associate them with the token values *REPEAT* and *UNTIL* that you introduced in step 2.

(4) In the function *gen_stmt*, add a rule for *RepeatStmt*, modelling it on the existing rule for *WhileStmt*.

In all, these modifications require about 10 lines of code to be added or changed. When the changes have been made, it should be possible to rebuild the compiler without *any* error or warning messages from the ML compiler. I find it helpful to adopt a style in which even warnings about **function**

and **match** expressions that do not cover all cases are avoided, in favour of adding where necessary a catch-all rule at the bottom that raises a recognisable exceptions – but that's partly a matter of taste.

You will want to devise your own programs for testing the new implementation of `repeat` loops, but one program `repeat.p` is provided as part of the lab kit. Your tests should be more thorough: do things work properly, for example, if the body of the loop is empty? Is it permissible to add an extra semicolon at the end of the loop body? You can add your tests to the list that is given in the makefile, and if you include in the test program a comment that gives the expected output, then automated testing will continue to work. Just follow the layout of the existing tests.

3.5 Implementing `loop` statements

A more general kind of loop is provided by the `loop` statement, similar to the one provided in Modula and Oberon. In principle, this is an infinite loop, but its body may contain one or more `exit` statements. When control reaches an `exit` statement, the effect is to exit from the (closest enclosing) `loop` statement immediately. Here is an example of a `loop` statement with an `exit` statement inside it:

```
loop
  x := x + 1;
  if x >= 10 then exit end;
  y := y + x
end
```

Unlike the `repeat` loop given as an example in Section 3.4, this loop exits as soon as `x` reaches ten, without modifying `y` in that iteration. As the example shows, an `exit` statement may appear anywhere in the text of a loop body, even nested inside other constructs like `if`, `while` or `repeat`. Also, we will allow `exit` statements to appear outside a `loop` statement, and in this case they terminate the whole program, even if they are contained within other constructs.

To translate a `loop` statement, we need two labels, one at the top of the loop, and another just after it. The label at the top is used as the target of a jump that follows the loop body, and any `exit` statements in the loop body can be translated into jumps to the label at the bottom. For example, here is a plan for the code generated from the loop above:

LABEL L_2
⟨code for `x := x + 1`⟩
⟨if `x >= 10` then jump to L_3⟩
⟨code for `y := y + x`⟩
JUMP L_2
LABEL L_3

It will also be necessary to place a label at the end of entire program to act as a target for `exit` statements that are outside any loop.

Here is a suggested plan of work for adding `loop` and `exit` statements:

(1) Augment the abstract syntax with new constructors for the type *stmt*.

(2) Add the new concrete syntax to the parser, and the new keywords to the lexer.

(3) Extend the intermediate code generator. To do this, it works nicely to give *gen_stmt* an extra parameter, the label that an `exit` statement should jump to:

```
let rec gen_stmt s exit_lab =
  match s with...
```

Again, about a dozen lines of code need to be added or changed in all. You will want to use your own test cases, but an example program loop.p is provided to get you started.

3.6 Implementing `case` statements (optional)

A `case` statement looks like this:

```
case i of
     1, 3, 5:
       i := i + 3
  | 2, 6:
       i := i - 1
  | 8:
       i := i + 2
else
  i := i + 1
end;
```

There are several cases, each labelled with a list of the values for the control expression `i` that lead to that case being chosen. There may also be an `else` part that is executed if none of the labels match. The `case` statement is more difficult to implement than the `repeat` statement, because it has a significantly more complex syntax, and needs a new instruction in the virtual machine. The implementation is made easier if you use the high-level pre-defined functions of ML instead of programming everything by explicit recursion over lists. My solution required about 50 lines of code to be added to the compiler.

3.6.1 Extending the syntax

Here are some more explicit instructions for adding the `case` statement. The syntax looks like this:

```
case expr of
      num, ..., num: stmts;
  |  ...
  |  num, ..., num: stmts
else
  stmts
end
```

The list of cases following `of` may be empty, and the `else` keyword and the sequence of statements that follows it are optional. We first need to extend the parser to accept this syntax.

(1) In tree.mli and tree.ml, add a new constructor to the type *stmt*:

CaseStmt **of** *expr* $*$ (*int list* $*$ *stmt*) *list* $*$ *stmt*.

A node *CaseStmt* (*switch*, *cases*, *default*) will represent a case statement with the control expression *switch*, a list *cases* that forms the body, and the `else` part *default*. Each element of the list of cases consists of a list of values, each an integer, and a statement to be executed if the value of the control expression matches one of the values. In the concrete syntax, a sequence of statements can appear in each arm of the `case` statement, represented in the abstract syntax using the *Seq* constructor. If there is no `else` part, then *default* can be set to the empty statement *Skip*.

(2) Add token types *CASE* and *OF* to the list of keywords near the beginning of parser.mly, and add the "`case`" and "`of`" keywords to the list in lexer.mll. Also add a token type *VBAR* (for the vertical bar that separates cases) to the list of punctuation marks on parser.mly. Add a rule for it in lexer.mll, being careful to add the rule *before* the catch-all rule beginning "_".

(3) In parser.mly, add a new production for *stmt* that describes `case` statements, adding some other non-terminals to help you describe the syntax. Write the semantic action to build a *CaseStmt* node in the abstract syntax tree.

You can test your work so far by re-building the compiler, even before extending the code generator. You will get warning messages that indicate the new statement type is not handled by pattern-matching in the code generator, but the parser of the resulting compiler should work correctly. If you have made a clean job of extending the grammar, then *ocamlyacc* will report no 'conflicts' when it processes parser.mly.

Submitting a program without `case` statements to your half-modified compiler should give the same results as before. Submitting a program with correctly-formed `case` statements should result in no syntax error messages, but your compiler will fall over with a pattern match failure in kgen.ml.

3.6.2 Translating `case` statements

A `case` statement can be translated into code like that shown in Figure 3.5. The code begins by evaluating the control expression; then comes a multi-way branch instruction *CASEJUMP N* that compares the value on the evaluation stack with N values in a table, represented as a list of instructions *CASEARM* (x_{ij}, L_i). Each value x_{ij} is associated with a label L_i attached to one of the arms of the case statement. Multiple values may be associated with each label, if multiple cases are handled by the same arm of the case statement. Note that the number N is the total number of following *CASEARM* instructions, and this may be more than the number n of arms. The virtual machine already supports this form of multi-way jump.

If any of the values x_{ij} matches the value from the stack, the *CASEJUMP* instruction jumps to the corresponding label; if none match, then execution continues with the instruction after the table, a branch to the default case. Code for the n arms and the default case follows, and each piece of code ends with a jump to a label at the bottom.

⟨Code for the control expression⟩
CASEJUMP N
CASEARM (x_{11}, L_1)
CASEARM (x_{12}, L_1)
...
CASEARM $(x_{n\,m(n)}, L_n)$
JUMP def_lab
LABEL L_1
⟨Code for first case⟩
JUMP exit_lab
LABEL L_2
⟨Code for second case⟩
JUMP exit_lab
...
LABEL L_n
⟨Code for *n*'th case⟩
JUMP exit_lab
LABEL def_lab
⟨Code for default part⟩
LABEL exit_lab

Figure 3.5: *Code for* `case` *statements*

It's possible that the same value appears among the labels of several arms in the `case` statement, something that reveals an error in the source program. There's no need for you to detect this situation or treat it specially; later, when we introduce a semantic analysis phase into the compiler, it would be possible to check for duplicate cases and generate an error message that ties the error to a location in the source program.

To generate this code, you will need to write a rule for the pattern

CaseStmt (*switch*, *cases*, *default*)

in the function *gen_stmt*. I suggest the following steps:

(1) Invent a list $[L_1; L_2; \ldots; L_n]$ of n labels, where n is the number of cases. This can be done by mapping the list of cases through an appropriate function. Invent also two single labels *def_lab* and *exit_lab*.

(2) Use *List.combine*, *List.map* and *List.concat* to generate the table

$$[(x_{11}, L_1); (x_{12}, L_1); \ldots; (x_{n\,m(n)}, L_n)]$$

that should appear in the multi-way branch.

(3) Build the code for the statement by wrapping this and all the following steps in *SEQ* [...]. Generate code for the control expression using the function *gen_expr*.

(4) Generate the *CASEJUMP N* instruction and the N following *CASEARM* instructions making up the multi-way branch, followed by a jump to the default label *def_lab*. Note that the parameter N here is the number of *CASEARM* instructions and *not* the number of cases, each of which may handle several values.

(5) Use *List.combine* to join the list of labels from part (1) with the list of cases and *List.map* to generate the label, code for the body, and jump to *exit_lab* for each of them. Use *gen_stmt* to generate the code for each case arm.

(6) Place *def_lab* and generate code (using *gen_stmt*) for the default part.

(7) Finally, place *exit_lab*.

When these steps are complete, so is your new compiler.

3.6.3 Testing `case` statements

You can test the new control structure by running the program `case.p` from the lab kit, and other test programs of your own. Here are a couple of things that you can check: that omitting the `else` part is equivalent to including an empty `else` part; and that the empty statement is allowed as the body of a case.

The *Keiko* module of the compiler is responsible for outputting the generated code in textual form, and it also performs a sanity check on the code, checking for example that the evaluation stack is used in a consistent way, and also that each instruction *CASEJUMP N* is followed by exactly *N* occurrences of *CASEARM*, and gives an error message in case of discrepancy. These sanity checks help to insulate you from the horrors of debugging object code at the binary level – about which the best thing can be said is that it is more productive to spend time reading the compiler output than running it and decoding the resulting mess.

Chapter 4

Types and data structures

4.1 Lecture 6: Semantic analysis

In all but the simplest programming languages, variables can have different types. The compiler must reserve the right amount of space for each variable, and either check that each variable that is used is also declared, or (if variables need not be declared) collect a list of variables that are used. It also needs to check that variables are used in a way that is consistent with their types. All this is the job of *semantic analysis*.

In the language implemented so far, there have been only global integer variables.[1] Now we'll add variables with different types, and require them to be declared at the top of the program. To do this, we add a *semantic analysis* pass:

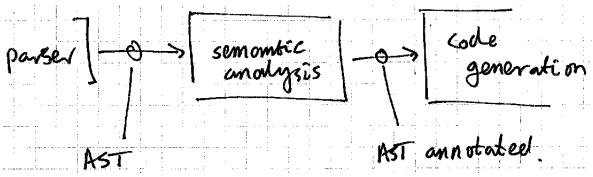


Figure 4.1: *Interface for semantic analysis*

In some languages (particularly ones where identifiers must strictly be declared *before* they are used) it's possible to do parsing and semantic analysis simultaneously. But it's easier to understand the process if we separate the two, making the input to semantic analysis be the abstract syntax tree (AST) produced by the parser, and the output be an annotated AST where each *applied occurrence*, where a name is used, is annotated with the relevant definition, derived from the declaration of the name.

We'll annotate each expression with its type and each variable with its runtime address. For this, we'll use OCaml's record types with mutable fields:

[1] In Lab 1, we cheated by making the lexer collect all names that are mentioned in the program, and reserving for each of them a storage location big enough to hold an integer.

```
type expr =
  { e_guts : expr_guts;
    mutable e_type : ptype }

and expr_guts =
    Constant of int * ptype
  | Variable of name
  | Subscript of expr * expr
  | Binop of op * expr * expr
  | ...

and name =
  { x_name : ident;              (* Name of the reference *)
    x_line : int;                (* Line number *)
    mutable x_def : def option } (* Definition in scope *)
```

The type *ptype*[2] represents types in the language we are compiling. At first, we might allow integers and booleans, and also arrays $Array(n, t)$, where n is a fixed bound and t is some other type.

```
type ptype =
    Integer
  | Boolean
  | Array of int * ptype
  | Void
```

The *ptype* value *Void* is a dummy value, used for type annotations that have not yet been filled in, and also later for things like the return type of a subroutine that returns no result. Each variable has a definition giving its name and type, and also a compiler-generated label for the place the variable can be found at run time.

```
type def =
  { d_tag : ident;               (* Name that is being defined *)
    d_type : ptype;              (* Type of the variable *)
    d_lab : string }             (* Runtime label *)
```

Semantic analysis will create and use *symbol tables* to move information around the tree. The goal is to verify that the input program obeys the rules, and to annotate it so that code generation becomes a purely local process, no longer dependent on context.

Semantic analysis is another recursive process. Much confusion surrounds accounts in books about compilers of the abstract data type of *symbol tables* or *environments*, mostly because of a historical concern for efficiency. If storage space is restricted, it becomes attractive to have the compiler keep just one symbol table, and to update it by adding the variables local to each program part before analysing that part, removing them afterwards before adding the variables the pertain to another part of the program. All this adds unnecessary complication.

The whole thing is simple, really, if we accept the idea of using an immutable, applicative representation, and forming a fresh environment for each part of the program, discarding it afterwards. The crucial decision is to

[2] The silent *p* in *ptype* is needed because **type** is a reserved word in OCaml.

make the *define* operation applicative, taking an environment as input and producing another, independent, environment as output.

val *empty* : *environment*

val *define* : *def* → *environment* → *environment*

val *lookup* : *ident* → *environment* → *def*

val *defs* : *environment* → *def list*

(we'll add a couple more operations later, but keeping it applicative, not imperative.)

The semantic analyser's job is to annotate each expression with its type and each variable with its definition.

```
let rec check_expr e env =
  let t = expr_type e env in
  e.e_type ← t; t

and expr_type e env =
  match e.e_guts with
      Variable x →
        let d = lookup x.x_name env in
        x.x_def ← Some d; d.d_type
    | Binop (Plus, e1, e2) →
        let t1 = check_expr e1 env
        and t2 = check_expr e2 env in
        if t1 = Integer && t2 = Integer then
          Integer
        else
          error "type error"
    | ...
```

Most languages allow other things to be named than just variables, and have a more complicated system of types and rules for type equivalence. They usually have operator overloading, so that the same symbol `+` may be used for integer and floating point addition. There may be type conversions, so that you can write `x+i` and have the `i` implicitly converted to a floating point number before the addition. All these things are the business of the semantic analyser.

The intermediate code generator's job is to express data structure access in terms of address arithmetic. In a native-code compiler, later passes will implement this arithmetic using the machine instructions and addressing modes of the target machine.

4.2 Lecture 7: Data structure access

Unless the target language (as with the JVM) has primitive operations for accessing an element of an array or a field of a record, the compiler needs to reduce data structure access to arithmetic on addresses.

Dealing with declarations: The abstract syntax of a program includes both a list of declarations and a list of statements to be executed.

type *program* = *Program* **of** *decl list* ∗ *stmt*

In outline, semantic analysis builds up an environment by analysing the declarations, then uses this environment to check and annotate the statements.

```
let annotate (Program (ds, ss)) =
  let env = accum check_decl ds empty in
  check_stmt env ss
```

The function *accum* is a version of *foldl* with the arguments swapped appropriately: see Appendix A, page 127. The code generation phase produces executable code for the statements, and processes the declarations again to reserve space for the variables.

```
let translate (Program (ds, ss)) =
  printf "FUNC MAIN 0\n" [ ];
  Keiko.output (gen_stmt ss);
  printf "RETURN\nEND\n" [ ];
  List.iter gen_decl ds
```

For example, the declaration `var a: array 10 of int` generates the directive

```
GLOVAR _a 40
```

which tells the Keiko assembler to reserve 40 bytes of memory and denote it by the label `_a`. That label can be mentioned in the executable code for the program, so that the instruction `GLOBAL _a` pushes the address of the reserved storage onto the evaluation stack.

Possible errors:

- same variable declared more than once.
- variable used without being declared.
- variable used in a way that is not consistent with its type.

All these are detected during semantic analysis; code generation should not produce error messages.

Implementation: Semantic errors can be detected by having functions in the environment module (*Dict*) raise exceptions for undeclared or multiply declared identifiers, then catching and handling these in the semantic analyser by printing error messages. It's easy to make a compiler that detects one error and stops with a message. Allowing the compiler to continue is helped in practice by inserting dummy declarations for undeclared identifiers, and introducing a universal error type that can replace the types of incorrect expressions; we won't bother with that, and will content ourselves with making a compiler that gives up after detecting the first error. Carrying on and finding more errors used to be vital in the punched card era, when most programming was done via batch processing.

Generating code: Let's begin by considering how to generate code from the annotated tree. Consider as an example the assignment `a[i] := a[j]`. We'll use an AST where both sides are represented by expressions:

```
Assign (Subscript (Variable "a", Variable "i"),
  Subscript (Variable "a", Variable "j")).
```

(We'll rely on the parser to ensure that the left hand side has the form needed for a variable, and isn't something like an integer constant.) To carry out the

assignment, the two sides must be treated differently, so that the RHS yields a value and the LHS yields an address to store it in. So we use two functions in the code generator: *gen_addr* finds the address of a varaible, and *gen_expr* as before finds the value of an expression. Because an expression like `a[i]` in the example that denotes an address can contain a sub-expression `i` that denotes a value, it's natural to make the two functions mutually recursive.

```
let rec gen_addr v =
  match v.e_guts with
      Variable x →
        let d = get_def x in GLOBAL d.d_lab
    | Subscript (e1, e2) →
        SEQ [gen_addr e1;          (* address of array *)
          gen_expr e2;             (* value of subscript *)
          CONST (size_of v.e_type); BINOP Times; OFFSET]
    | _ → failwith "gen_addr"

and gen_expr e =
  match e.e_guts with
      Variable _ | Subscript (_, _) →
        SEQ [gen_addr e; LOADW]
    | Binop (w, e1, e2) → ...

let rec gen_stmt =
  function ...
    | Assign (v, e) →
        SEQ [LINE (line_number v); gen_expr e; gen_addr v; STOREW]
```

Code for data structure access: For arrays,

$$addr(a[i]) = addr(a) + i * \text{size of element}.$$

For example, `x := a[i]` (on line 23 of the source program) gives

```
GLOBAL _a
GLOBAL _i; LOADW
( CONST 10; BOUND 23 )
CONST 4; TIMES; OFFSET; LOADW
GLOBAL _x; STOREW
```

After the first two lines, the *base address* of `a` is on the stack, and on top of it is the *value* of `i`. The optional third line checks that `i` is in the range 0 up to 9. Then the fourth line combines the base address and the index to form the address of `a[i]`, and loads from that address. The last line stores the resulting value in `x`. On Keiko, this sequence is equivalent to

```
GLOBAL _a; LDGW _i; CONST 10; BOUND 23; LDIW; STGW _x
```

with the LDIW instruction performing the index calculation and the load in one operation.

This scheme also works for multi-dimensional arrays laid out row-by-row, such as

```
var b: array 10 or array 10 of integer;
```

$$addr(b[i][j])) = addr(b) + i * 40 + j * 4.$$

As the syntax suggests, we can simply treat the multi-dimensional array as an array of arrays.

Record types are often used to implement linked lists, as in the type

```
type dogptr = pointer to dogrec;
  dogrec = record
      name: array 10 of char;
      age: integer;
      next: dogptr;
    end;
```

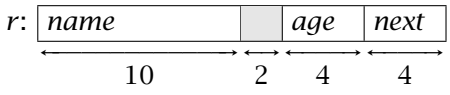


Figure 4.2: *Layout of record type* `dogrec`

A two-byte filler is needed after the *name* field to ensure that the following fields are properly aligned: more about this will be said later. We can find the address of any field x in a record r by the equation.

$$addr(r.x) = addr(r) + \text{offset of } x.$$

For example, with the declaration `var r: dogrec`, the assignment `r.age := 29` gives the code

```
CONST 29
GLOBAL _r
CONST 12; OFFSET; STOREW
```

When records are accessed via pointers, Pascal-like languages use the syntax $p \uparrow .x$, with $p \uparrow$ denoting the record to which p points. We can implement this construction by exploiting the equation

$$addr(p\uparrow) = value(p).$$

For example, with `var p: dogptr` the assignment `p := p↑.next` (appearing, say, on line 25) becomes

```
GLOBAL _p; LOADW
( NCHECK 25 )
CONST 16; OFFSET; LOADW
GLOBAL _p; STOREW
```

The optional instruction NCHECK 25 on the second line checks that the pointer is not null before following it. The third line can be abbreviated to the single instruction LDNW 16.

In some languages, the use of pointers is implicit: e.g., all Java variables of class type are pointers.

Equivalence rules for types: After

```
type t1 = pointer to array 10 of integer;
  t2 = pointer to array 10 of integer;
```

is the assignment `p1 := p2` allowed? According to *structural equivalence*, the answer is Yes; but if the typing rule is *name equivalence* then the two

types `p1` and `p2` are different, and the answer is No. With name equivalence, each occurrence of a type constructor such as `pointer` or `array` generates a distinct type.

Apparently, structural equivalence is preferable, but testing structural equivalence of recursive types is a complicated business; and sometimes name equivalence is what is wanted: for example, a point and a rectangle are different kinds of thing, even if they are both records with fields `x` and `y` – so we might want a subroutine that operates on points *not* to accept a rectangle as argument, even though the two types are structurally equivalent.

Exercises

3.1 Assume the following declarations.

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

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

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

```
s := s + q↑.age;
q := q↑.next
```

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

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

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

For example, here is a program that prints 53:

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

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

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

```
loop outer:
```

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

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

(a) Suggest an abstract syntax for this construct.

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

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

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

3.4 In some programming languages, it is a mistake to use the value of a variable if it has not first been initialised by assigning to it. Write a function that, for the language of Lab 1, tries to identify uses of variables that may be subject to this mistake. Discuss whether it is possible to do a perfect job, and if not, what sort of approximation to the truth it is best to make.

Chapter 5

Lab two: Array access

This lab is based on a compiler for a little language very similar to the one we used in Lab 1, except that variables are typed and must be declared. The compiler for this lab can be found in directory `lab2` of the lab kit. It has a semantic analysis pass that annotates each expression with its type and each occurrence of a variable with its definition, including the information needed to find its address.

As things stand, the compiler supports only simple integer and Boolean variables: the file `gcd.p` contains an example program. Your task is to add arrays, thereby changing the language from a toy into one that could be used to write small but non-trivial programs. Part of the work has been done for you, because the parser and abstract syntax of the compiler already contain array types and subscript expressions `a[i]`. Your part of the work is to extend the semantic analyser and intermediate code generator to handle these parts of the language. Listings of the files `check.ml` and `kgen.ml` appear in Appendix E.

5.1 Compiler structure

The structure of the compiler is much the same as the compiler of Lab 1, with the addition of two modules: *Check* is the semantic analyser, and *Dict* implements the environments that the semantic analyser uses to keep track of which variables have been declared. Your changes in this lab will affect *Check* and the intermediate code generator *Kgen*.

The type for elements of an array may itself be an array type, so we can have both simple arrays, declared like this:

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

(this array has ten elements that can be accessed as `a[0]`, . . . , `a[9]`) and multi-dimensional arrays, declared like this:

```
var M: array 5 of array 10 of boolean;
```

with 50 elements `M[0][0]` to `M[4][9]`. The definition for `a` would contain the *ptype* value *Array* (10, *Integer*), and the definition of `M` would contain the value *Array* (5, *Array* (10, *Bool*)).

As usual in this course, the interfaces between each pass and the next are abstract syntax trees with annotations. Specifically, each place that an identifier is used in the program is represented in the tree by a record of type *name*, as defined on page 36, and each such record *x* contains a mutable reference *x.x_def* to a *def* record, defined as on page 37 giving the corresponding definition. The syntax analyser initialises all these fields to a dummy value, and part of the job of the semantic analyser is to fill in the fields with the proper definition, so that the intermediate code generator can output the correct code to address the variable.

The syntax summary in Figure 5.1 repeats the one given for Lab 1 in Figure 3.1, with the changes that are needed to add arrays; those changes are highlighted with a shaded background A new syntactic category *variable* has been introduced: it contains simple identifiers `x`, in addition to array references like `a[i]` or `M[i][j]`. Any of these kinds of variables can appear in expressions and on the left-hand side of assignments.

In the abstract syntax, variables and expressions are amalgamated into one type *expr*, and the left-hand side of an assignment is an *expr*:

type *stmt* = ...
  | *Assign* **of** *expr* ∗ *expr*
  | ...

This makes it look as if the left-hand side could be any expression; in fact, it can only be a variable, and we can rely on the parser to build only trees where this is so.

5.2 Functions on types

The first task is to extend the semantic analyser to cope with arrays. It is best to begin by adding a few auxiliary functions to the dictionary module implemented by dict.mli and dict.ml. For each of the following functions, you should add the declaration to dict.mli and add your definition to dict.ml:

(1) Define a function *type_size* : *ptype* → *int* that computes the size occupied by a value of a given type. This is 4 for integers and 1 for Booleans, and for arrays it is obtained by multiplying the bound and the element size.

(2) Define a function *is_array* : *ptype* → *bool* that returns **false** for the types *Integer* and *Boolean*, and **true** for array types.

(3) Define a function *base_type* : *ptype* → *ptype* such that if *t* is an array type, then *base_type t* is the underlying type for elements of the array. For example, the base type of the type `array 5 of array 10 of boolean` is `array 10 of boolean`, and the base type of this type is `boolean`. If *t* is not an array type, then *base_type* should raise an exception.

These functions will be useful in extending the semantic analyser.

5.3 Semantic analysis

The semantic analyser *Check* needs to be extended in a couple of areas:

```
program  →  { decl ";" } begin stmts end "."

decl     →  var ident { "," ident } ":" type

type     →  integer
         |  boolean
         |  array number of type

stmts    →  stmt { ";" stmt }

stmt     →  empty
         |  variable ":=" expr
         |  print expr
         |  if expr then stmts [ else stmts ] end
         |  while expr do stmts end

expr     →  variable
         |  number
         |  true | false
         |  monop expr
         |  expr binop expr
         |  "(" expr ")"

variable →  ident
         |  variable "[" expr "]"

monop    →  "-" | not

binop    →  "*" | "/"
         |  "+" | "-"
         |  "<" | "<=" | "=" | "<>" | ">" | ">="
```

Figure 5.1: *Syntax additions for declarations and arrays*

(1) The analyser does not recognise subscripted variables $v[e]$, which are represented in the abstract syntax tree by *Sub* (v, e), so you will have to add a clause to the function *check_expr* to handle them. This function has type *environment* → *expr* → *ptype*; it takes an environment and the abstract syntax tree for an expression and returns the type of the expression as a *ptype* value. You should implement the following checks:

   (a) the sub-expressions v and e are themselves correct expressions.

   (b) v has an array type.

   (c) e has type *Integer*.

   The type of the whole expression is the type of elements of v.

(2) Our code generator will not be able to deal with assignments v := e

where the type of *v* and *e* is not a simple integer or Boolean. Add code to the function *check_stmt* to check that this restriction is not violated.

5.4 Code generation

The intermediate code generator *Kgen* needs to be changed:

(1) The function *gen_addr* generates code to push the address of a variable on the stack. Enhance it to deal with subscripted variables by calculating the address using multiplication and addition. [Hint: the function *type_size* will be useful.] Remember that, as in C and Oberon, our subscripts start at zero!

(2) The function *gen_decl* generates assembler directives to reserve space for global variables. At present, it assumes that each variable occupies 4 bytes of storage, but arrays in fact can occupy more than this. Modify the way the size *s* is calculated to cope with this.

(3) The code generator currently uses the instructions *LOADW* and *STOREW* to load and store 4-byte values. These are inappropriate if the value being stored is a 1-byte boolean; in that case, the instructions *LOADC* and *STOREC* should be used instead.

5.5 Testing the compiler

Four test programs are provided: in addition to the program gcd.p that works initially using no arrays, there are:

- a program array.p that uses an array to calculate the first few Fibonacci numbers,
- a program pascal.p that computes the first few rows of Pascal's triangle in a matrix, and
- a program binary.p that converts an integer into an array of binary digits expressed as boolean, then converts the array of booleans back into an integer.

This last program will not work properly unless you have arranged to generate a *LOADC* instruction (rather than *LOADW*) for loading an element of a boolean array. You will find however – and you should try the experiment – that it will not reveal the bug if you have forgotten to generate the proper *STOREC* instruction for storing into the array. You should write your own test case that reveals that bug and demonstrate that it does so by re-introducing the bug into your compiler if necessary. You may like to try some other programs of your own: are expressions of the form `a[b[i]]` handled properly, for example?

5.6 Run-time checks and optimisation (optional)

Two glaring faults of the present compiler deserve to be corrected, if you have the time:

(1) Out-of-bound subscripts (that are either negative or too big for the array) are not detected at run time, and can result in weird behaviour that is hard to debug. You can fix this problem by using the instruction *BOUND n*, where *n* is a line number in the source program. This instruction expects to find an array index *i* and the bound *b* of the array on top of the stack. It checks that the index satisfies $0 \leq i < b$; if so, it pops the bound *b* and leaves the value *i* on the stack for use in a subsequent calculation. If not, it stops the program with a message, saying that there was an array bound error on line *n*. Modify the code generator so that it uses this instruction to check that each subscript is within the bounds of the appropriate array.

(2) Many subscripted variables result in ludicrously bad code. For example, any reference to an array of booleans will include a superfluous multiplication by 1, and if a subscript is constant, the compiler still generates code to calculate the address at run time, even though the address is fixed at compile time. You can (partially) fix this problem by improving your implementation of *gen_addr* so that it spots common special cases and generates better code for them. This kind of optimisation could also be done later, in the peephole optimiser.

5.7 Array assignment (optional)

Earlier, you were asked to implement a semantic check to ensure that each assignment copied only a single integer or boolean, and not an entire array. If you like, you could reverse that decision, and enhance the code generator to support memory-to-memory copying of arrays, including rows of two-dimensional arrays.

This task is immensely eased by the existence of a Keiko instruction FIXCOPY that copies a block of memory. FIXCOPY pops three values from the evaluation stack; from bottom to top, they are a destination address, a source address and the number of bytes to copy. The source and destination areas must not overlap, so the instruction behaves very much like the C library routine memcpy.

Chapter 6

Subroutines

6.1 Lecture 8: Subroutines

A bit of computer architecture: The operating system and the memory management unit (MMU) work together to make it appear to each program that it is the only one using the memory of the machine. On the simplest machines, each program gets a contiguous area of memory (*segmentation*); or on a microcontroller, perhaps our program is genuinely the only one present. More powerful machines drop the restriction that the memory allocated to a program (strictly, a process) is contiguous, and use a *page table* to map the layout of memory. Either of these schemes can be combined with the idea of swapping to disk some parts of memory not currently being accessed.

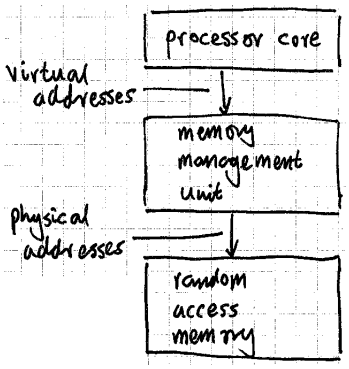


Figure 6.1: *Address translation*

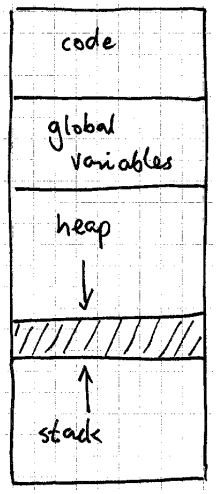
All this concerns us only in so far as its supports the view that the target program can assume a flat, exclusive address space. Traditionally[1] the address space of the program is divided into

- program code: read-only unless using dynamic loading.

[1] Large address space of modern machines makes the memory layout less important, and robustness against virus attacks is increased if some elements of the layout vary randomly from between invocations of the program.

- global variables: at fixed addresses, referred to symbolically by the compiler and fixed by the linker.
- stack: used for subroutines.
- heap: for dynamically allocated storage.

A simple layout that works is to put the code at the lowest addresses, with the statically allocated global variables just above it, then put the stack at the top of memory, growing downwards. Dynamically allocated storage can occupy a region that grows towards the stack from the end of the globals.



Note: higher addresses at bottom.

Figure 6.2: *Address space layout*

Conventions for stack layout and for calling subroutines are fixed partly by hardware, partly by system software. Typically, each subroutine *activation* has a stack frame containing storage for parameters and local variables of the subroutine, and a frame head where register values are saved so that they can be restored when the subroutine returns. Parameters may be passed on the stack or in registers.

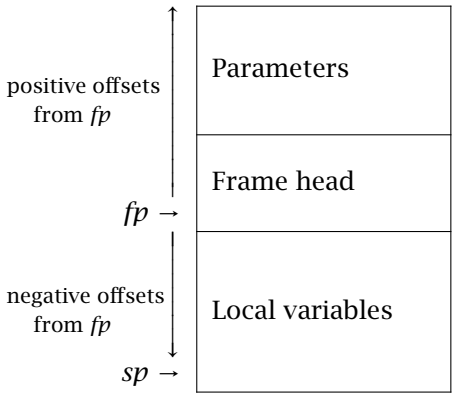


Figure 6.3: *Stack frame layout*

Language features:

- recursion: supported automatically by stack-based approach.
- value parameters: the value of the actual parameter expression becomes the (initial) value of the formal parameter.
- reference parameters: uses of the formal parameter refer to a variable supplied as the actual parameter.

Algol-like languages have both kinds of parameter; Java and C have only value parameters, but they get some of the same effect by letting the address of a variable (or an object in the case of Java) be passed as a value parameter.

On Keiko: parameters are assembled in the evaluation stack, and become frozen as part of the the new stack frame when a procedure is called. The program

```
proc f(x, y);
  var a;
begin
  a := x - y;
  return a * a
end;

... print f(5, 2) ...
```

compiles into

```
FUNC f 4
! a := x - y
LOCAL 16  - push fp+16
LOADW     - fetch contents
LOCAL 20
LOADW
SUB
LOCAL -4
STOREW
! return a * a
LOCAL -4
LOADW
LOCAL -4
LOADW
TIMES
RETURN

! ... f(5, 2) ...
CONST 2  - arg 2
CONST 5  - arg 1
CONST 0  - "static link"
GLOBAL _f  - proc address
PCALLW 2  - call with 2 args and a word-size result
```

The frame head contains:

- the static link.
- the context pointer: this is the address used to call the procedure, and is also held in the CP register while the procedure is active.

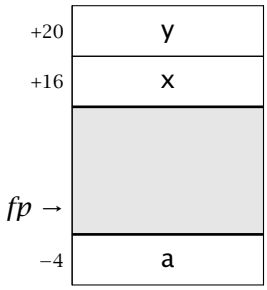


Figure 6.4: *Frame layout for* `f`

- the return address: saved value of the PC from the caller, giving the address of the next instruction after the call.
- the dynamic link: saved value of the FP register.

The official account is that $n + 2$ items are prepared on the evaluation stack, and these are transferred into part of the new stack frame in response to the CALL instruction. The reality in the bytecode interpreter is different and simpler: the interpreter keeps the evaluation stack of a subroutine in the memory region below its fixed stack frame – a fact that cannot be detected by any legal Keiko program. It turns out that the $n + 2$ items prepared before the call are in exactly the right place to be fenced off as part of the new stack frame, without having to move them at all.

Four pieces of code collaborate to create and destroy activation records.[2]

- *Preparation*: evaluate arguments and put in place; save PC and jump to subroutine.
    - *Prelude*: complete creation of stack frame.
    - (subroutine body)
    - *Postlude*: save result; partially destroy stack frame; branch to return address.
- *Tear-down*: fetch result; complete destruction of stack frame.

6.2 Lecture 9: Parameters and nesting

Value and reference parameters: with parameters passed by value, the procedure `inc(x)` is useless.

```
proc inc(x);
begin x := x+1 end;

... inc(y) ...
```

The code makes it clear what happens:

```
FUNC inc 0
LDLW 16
```

[2] Some of these are hidden in Keiko, because the frame for a procedure is created and destroyed implicitly as part of the CALL and RETURN instructions.

```
CONST 1
PLUS
STLW 16
RETURN
...
LDGW _y
CONST 0
GLOBAL _inc
PCALL 1
...
```

The STLW 16 instruction stores a new value for the local variable of the procedure; but this local variable is destroyed immediately afterwards when the procedure returns. On the other hand, the procedure has access only to the *value* of the global `y` (loaded by LDGW _y), and can't in any case do anything to change it.

What is written as PCALL or PCALLW in Keiko assembly language turns into two bytecode instructions: one (JPROC) sets up a stack frame and jumps to the subroutine, leaving the return address pointing to the second instruction. This one (SLIDE or SLIDEW) does the tear-down of the stack frame, popping *n* values off the evaluation stack and (in the case of SLIDEW) preserving the procedure result in place of them.

A more useful procedure makes `x` a parameter passed by reference: a `var` parameter in Pascal terminology. The code generated is slightly different:

```
FUNC inc 0
LDLW 16
LOADW
CONST 1
PLUS
LDLW 16
STOREW
RETURN
...
GLOBAL _y
CONST 0
GLOBAL _inc
PCALL 1
...
```

Here the actual parameter value is the address of a variable, and getting its value requires *two* load operations; LDLW 16; LOADW is equivalent to LOCAL 16; LOADW; LOADW. To complement this, the procedure call loses a load operation, and has GLOBAL _y in place of LDGW _y.

Access to global variables: the local variables of a procedure live in its stack frame (at negative offsets). Global variables have fixed (symbolic) addresses.

```
var z;

proc setz(x);
begin z := x end;

FUNC setz 0
```

```
LDLW 16
STGW _z
END
```

Aggregate parameters: arrays and records. If these are passed by reference, it's easy – just pass the address of the array or record variable, then use this address as the basis of addressing calculations inside the subroutine. For "open array parameters" with no fixed bound, like those in Oberon:

```
proc sum(var a: array of integer): integer;
begin
  for i := 0 to len(a)-1 do ...
end;
```

Compile it as if it were

```
proc sum(var a: array ? of integer; n: integer): integer;
begin
  for i := 0 to n-1 do ...
end;
```

(and modify each call to match this).

Aggregate parameters passed by value are a feature of Pascal and related languages, but are less common in other traditions. The meaning of the language is that the entire array or record should be copied when the procedure is invoked, usually into the stack frame of the procedure. This can be implemented tidily by pretending such parameters are passed by reference, then making the procedure's prelude copy them into the stack frame. That's slow, but unavoidable in general, and goes to explain why few programming languages offer this feature.

Nested procedures: For example, `twopow(x, y, n)` computes $x^n + y^n$:

```
proc twopow(x, y, n: integer): integer;
  proc pow(t: integer): integer;
    var i: integer;
  begin
    while i < n do ...
  end;
begin
  return pow(x) + pow(y)
end;
```

This, and the other examples we shall reach in a minute, look a bit artificial in a Pascal-like language, but nested and higher-order procedures are meat and drink in functional languages:

let *inc_all n xs* = *map* (**fun** $x \rightarrow x + n$) *xs*

In this ML fragment, the expresion (**fun** $x \rightarrow \ldots$) denotes a nameless local procedure nested within *inc_all*, and one that moreover accesses the parameter *n* of the outer procedure.

To implement nested procedures (with access from inner procedures to local variables of outer procedures, as with `n` in these two examples), we get the

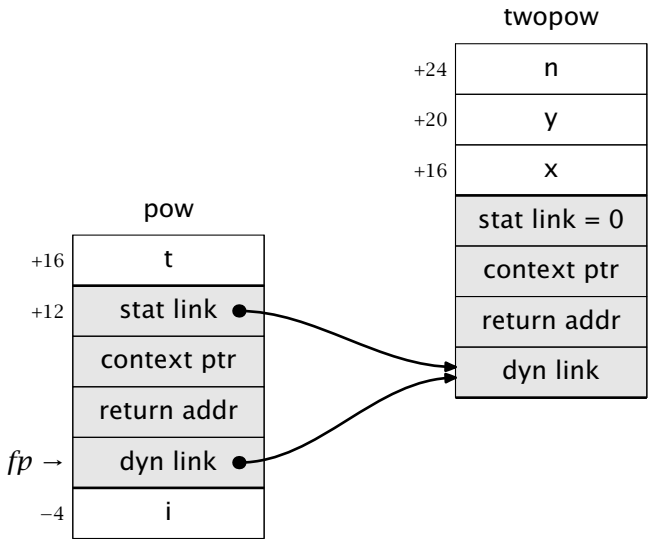


Figure 6.5: *Frame layout for* `pow` *and* `twopow`

semantic analyser to annotate each variable with a 2-D address (*level*, *offset*), like this:

```
proc twopow¹(x^(1,16), y^(1,20), n^(1,24): integer): integer;
  proc pow²(t^(2,16):integer): integer;
    var i^(2,−4): integer;
  begin
    while i^(2,−4) < n^(1,24) do ...
  end;
begin
  return pow²(x^(1,16)) + pow²(y^(1,20))
end;
```

At runtime, we arrange that the static link of each level n procedure activation points to the frame of the enclosing level $n-1$ procedure, where n denotes the level of nesting in the source program, not the depth of nesting of subroutine calls.

Access to locals (level n) and globals (level 0) is easy. For intermediate frames, use the chain of static links to find the correct activation, guided by the 2-D coordinates. For the expression $\mathtt{i}^{(2,-4)} < \mathtt{n}^{(1,24)}$ in `pow`:

```
FUNC pow 4
...
LOCAL −4; LOADW
LOCAL 12; LOADW
CONST 24; OFFSET; LOADW
JGEQ ...
```

Here, LOCAL −4; LOADW fetches the value of the local variable `i`. The sequence LOCAL 12; LOADW; CONST 24; OFFSET; LOADW first fetches the static link, a pointer to the stack frame of the enclosing invocation of `twopow`, and then fetches the word that is at offset 24 in that frame. That is the value

of the parameter `n`. As usual, these common addressing operations can be abbreivated, in this case to LDLW –4; LDLW 12; LDNW 24.

To call a nested procedure, we pass the frame address of the parent as the static link. Here is the code for the call `pow(x)` in `twopow`:

```
FUNC twopow 4
...
LOCAL 16; LOADW
LOCAL 0
GLOBAL _pow
PCALLW 1
```

Here, the instruction LOCAL 0 pushes the value of the FP register, and that is the appropriate static link to pass to a procedure that is nested inside the current one. Note the absence of a LOADW instruction: it is the address of the stack frame we want to pass, not the contents of a word within it.

Generally, a procedure P at level n may directly call another procedure Q at level m for $1 \le m \le n+1$. The level m is 1 if Q is global; n if Q is P itself, or a sibling of P at the same level of nesting, and $n+1$ if Q is a procedure nested inside P. To find the appropriate static link, start at fp and follow static links $n - m + 1$ times (or just use 0 if $m = 1$). Sometimes the static link and the dynamic link for a procedure are both non-trivial but different, as we'll see later.

We have chosen to make even global procedures expect a static link, even though we know this will always be the dummy value zero, because that makes all procedures have a uniform interface. A better implementation would avoid passing static links that are known not to be needed. Other languages implemented on Keiko use a different mechanism to make this possible, and our later machine-code compilers will pass the static link in a register that need not be set if the static link is a dummy.

6.3 Lecture 10: Higher-order functions

The next level of sophistication is to allow procedures that accept other procedures as parameters:

```
proc sum(a, b: integer; proc f(i: integer): integer): integer;
```

which might compute $\sum_{a \le i \le b} f(i)$. We could use `sum` like this, to compute $\sum_{a \le i \le b} i^n$:

```
proc sumpow(a, b, n: integer): integer;
  proc pow(x: integer): integer;
    var i, p: integer;
  begin
    i := 0; p := 1;
    while i < n do p := p*x; i := i+1 end;
    return p
  end
begin
  return sum(a, b, pow)
end;
```

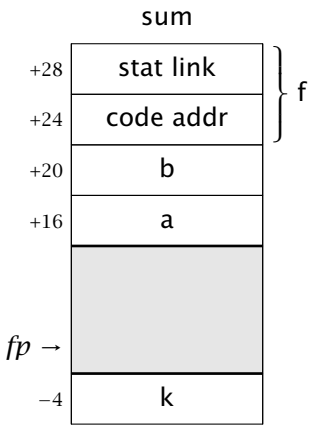


Figure 6.6: *Frame layout for* `sum`

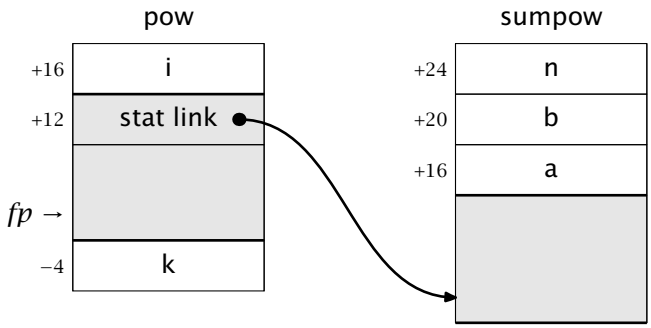


Figure 6.7: *Frame layout for* `pow` *and* `sumpow`

(A form like `sum(a, b, lambda (x) pow(x, n))` is easily converted to this form by systematically transforming lambdas into local functions with compiler-generated names.)

To call nested functions, we know it's necessary to pass a static link – the base address of the enclosing frame. To implement procedural parameters, what's needed is for the caller to supply, together with the address of the procedure to call, also the static link that should be used to call it.

In the example, the frame is laid out as shown in Figure 6.6, and a call `f(k)` from `sum` would be translated like this:

```
LDLW -4
LDLW 28
LDLW 24
PCALLW 1
```

The frames for `pow` and `sumpow` have the layout shown in Figure 6.7. The call `sum(a, b, pow)` in `sumpow` looks like this:

```
LOCAL 0
GLOBAL _pow
LDLW 20
LDLW 16
CONST 0
```

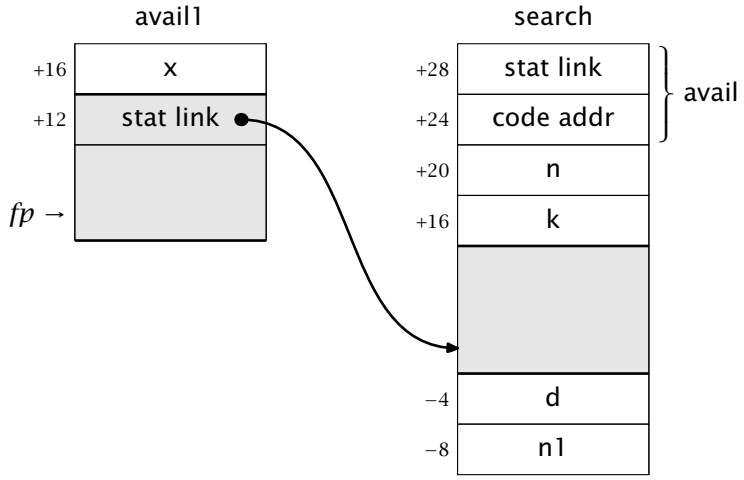


Figure 6.8: *Frame layout for* `avail1` *and* `search`.

```
GLOBAL _sum
PCALLW 4
```

A more elaborate example: Here's an example that combines nesting and procedural parameters with recursion. The problem is to find a permutation $d_1 d_2 \ldots d_9$ of the digits 1 to 9 such that $(d_1 d_2 \ldots d_9)_{10}$ is divisible by 9 and $(d_1 d_2 \ldots d_8)_{10}$ is divisible by 8, and so on. The solution is a recursive procedure `search(k, n, avail)` that expects `n` to be a `k`-digit number already chosen, and `avail` to be the set of digits still unchosen – represented as a Boolean function `proc avail(x: integer): boolean`.

```
proc search(k, n: integer; proc avail(x: integer): boolean);
  var d, n1: integer;

  proc avail1(x: integer): boolean;
  begin
    if (x <> d) then
      return avail(x)
    else
      return false
    end
  end;

begin
  ...
  for d := 1 to 9 do
    if avail(d) then ...
      search(k+1, n1, avail1)
    ...
  end
end;

proc avail0(x: integer): boolean;
begin return true end;

... search(0, 0, avail0) ...
```

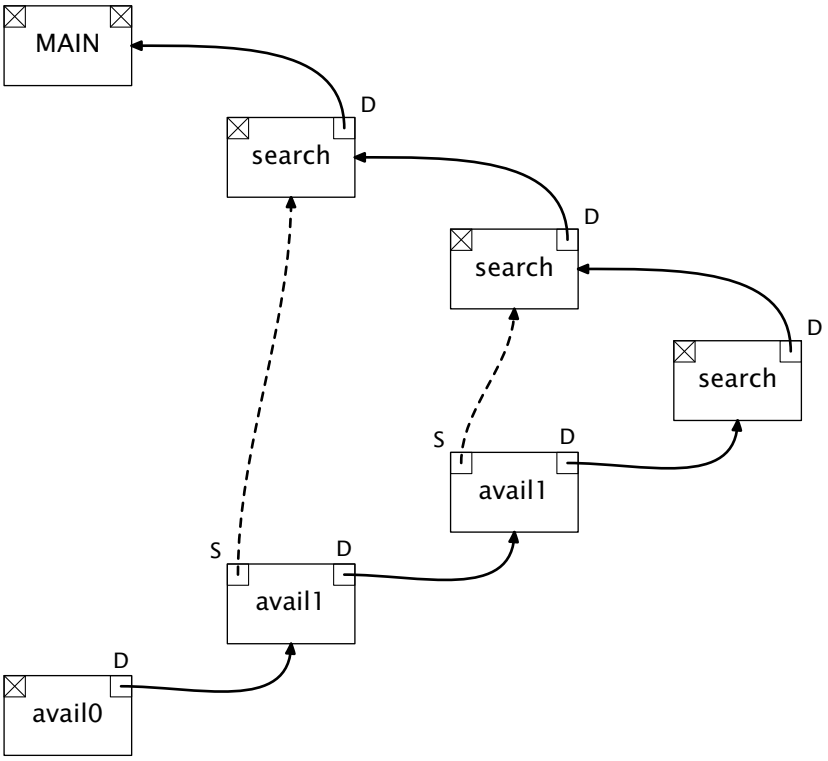


Figure 6.9: *Stack layout for* `digits`

The frames are laid out as shown in Figure 6.8, and the body of **`avail1`** becomes

```
LDLW 16 – x
LDLW 12; LDNW –4 – d
JEQ L1

LDLW 16 – x
LDLW 12; LDNW 28 – static link
LDLW 12; LDNW 24 – code address
PCALLW 1
JUMP L2

LABEL L1
CONST 0

LABEL L2
RETURN
```

Each invocation of **`search`** has its own copy of **`avail1`**, and the network of static and dynamic look like Figure 6.9.

6.4 Lecture 11: A complete compiler

With the techniques introduced so far, we can put together a compiler from a Pascal-like source language to code for the Keiko machine. The sub-directory

`ppc4` of the lab materials contains the source for this compiler: about 2500 lines of OCaml.

The lexer and parser are done with lex and yacc in the familiar way.

Semantic analysis annotates each applied occurrence with a definition:

```
type def =
  { d_tag : ident;                    (* The identifier *)
    d_kind : def_kind;                (* Kind of object – see below *)
    d_type : ptype;                   (* Type *)
    d_level : int;                    (* Nesting level *)
    mutable d_addr :
      Local of int | Global of symbol }     (* Address *)
```

The pair (*d_level*, *d_addr*) is a 2-D address. Definitions come in various kinds:

```
type def_kind =
    ConstDef of value
  | VarDef | CParamDef | VParamDef
  | ProcDef | TypeDef | ...
```

where for us *value* = *int*, but more generally could incorporate other target values like floating point.

Declarations are processed by

```
check_decl : decl → environment → (def → unit) → environment
```

where the *def* → *unit* argument is one of *loc_alloc* (allocate space for a local variable, downwards in the stack frame), *param_alloc* (for a parameter, upwards), *field_alloc* (for record fields), *global_alloc* (for global variables).

Use an abstract data type of environments, with operations

```
val empty : environment

val lookup : ident → environment → def

val define : def → environment → environment

val new_block : environment → environment

val top_block : environment → def list
```

This *functional* interface supports nested blocks, with only one declaration of an identifier allowed at each level of nesting.

```
let check_heading env (Heading (x, fparams, result)) =
  let pcount = ref 0 in
  let env' =
    check_decls fparams (new_block env) (param_alloc pcount) in
  ...
  mk_type (ProcType { p_fparams = top_block env';
                      p_pcount = !pcount; ... })
```

Each type has a (machine-dependant) representation that includes both size and alignment:

```
type metrics = { r_size : int; r_align : int }
```

This allows us to lay out correctly a type like `record c: char; n: integer end`

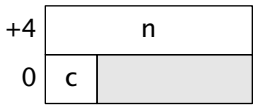


Figure 6.10: *Record layout with padding*

The *intermediate code generator* is much as we have seen, but includes new stuff in *gen_addr* and *address* to use the static chain:

```
(* address - generate code for address of a definition *)
let address d =
  match d.d_addr with
      Global g → GLOBAL g
    | Local off →
        if d.d_level = !level then
          LOCAL off
        else
          SEQ [schain (!level − d.d_level); CONST off; OFFSET]
        end
```

Peephole optimiser: code is generated into a buffer, and rules are applied to simplify the code before it is output. Two data structures:

- A sequence of instructions with pattern matching and replacement; implemented as two stacks.
- An equivalence relation ("disjoint sets") on labels; implemented with trees. No need for path compression, union-by-rank, $\log^*$, etc.! Each label has a reference count.

The peephole optimiser scans the buffer repeatedly until no more rules apply.

```
let ruleset replace =
  function
      LOCAL n :: LOAD s :: _ →
        replace 2 [LDL (n, s)]
    | ...
```

Rules: (i) replace common sequences of basic operations with special instructions, more compact and faster as bytecode.

- LOCAL n; STOREW → STLW n
- GLOBAL x; LOADW → LDGW x
- GLOBAL x; STOREW → STGW x
- CONST n; OFFSET; LOADW → LDNW n

(ii) Simplify, especially where constants appear as operands:

- CONST a; PLUS; CONST b; PLUS → CONST $(a + b)$; PLUS
- LOCAL a; CONST b; OFFSET → LOCAL $(a + b)$

(iii) Tidy up jumps and labels:

- LABEL a; LABEL b → LABEL a; equate a and b
- LABEL a → [ ], if *refcount* $a = 0$

6.5 Lecture 12: Objects

Object-oriented programming rests on three ideas:

- *Encapsulation*: the implementation of a class should be hidden from its users.
- *Object identity*: each instance of a class should have a distinct identity, so that multiple instances can co-exist.
- *Polymorphism*: if several classes have the same interface, instances of them can be used interchangeably.

Note that inheritance is not essential to this, except if polymorphism is achieved by inheritance from a common (abstract) superclass. Oberon–2 is an interesting language because the implementation of these ideas is separate and partially visible.

Let's start with *identity*: each object can be stored as a heap-allocated record, and we can then use the address of the record as its identity.

```
type Car = pointer to CarRec;
  CarRec = record reg, miles: integer end;

var c: Car;

new(c)
```

In many languages, the type `Car` would be *implicitly* a pointer type, and there are no variables that have records as their values directly.

Each object has fields for its instance variables, and also knows what class it belongs to. Methods can refer to these variables; in Oberon–2, these references are explicit:

```
procedure (self: Car) drive(dist: integer);
begin
  self.miles := self.miles + dist
end;
```

Many languages make the name `self` or `this` implicit, and allow the assignment to be written `mile := miles + dist`. In either case, we can produce object code that looks like this:[3]

```
LDLW 12
LDNW 8
LDLW 16
PLUS
LDLW 12
STNW 8
```

(If methods are not nested, there is no need for static links.)

Method invocation: to allow polymorphism, we must arrange that the method that is activated by any call is determined by the class of the receiving object, not the type of the variable storing its identity. If `Vehicle` is the superclass of `Car` then in

```
var v: Vehicle;
```

[3] There are white lies in all these code samples because the layout of records and descriptors is not exactly as shown in the diagram.

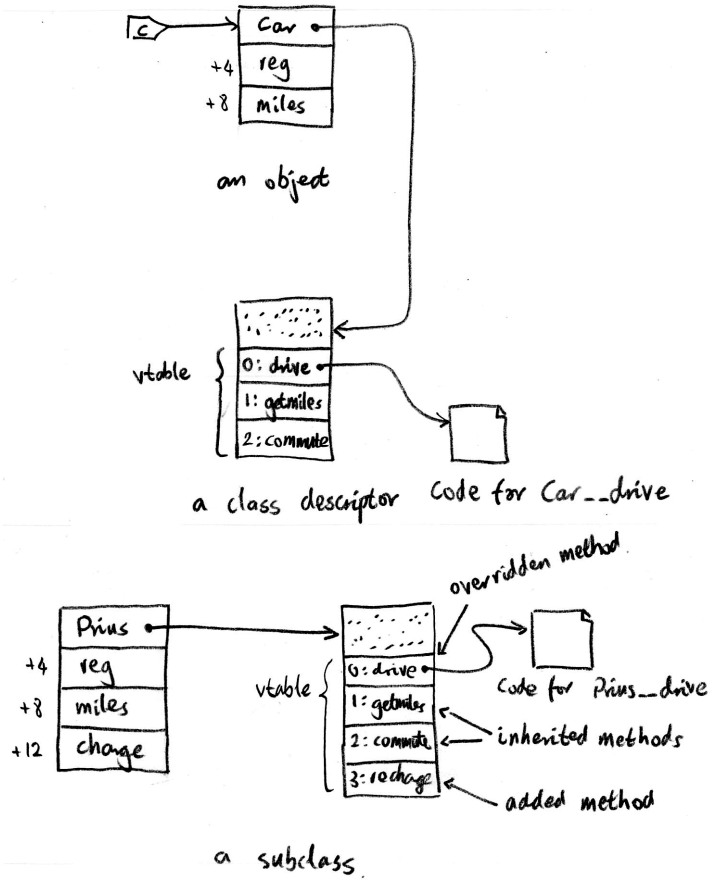


Figure 6.11: *Classes and objects*

```
v := c;
v.drive(100)
```

it is the `drive` method of `Car` that is invoked, not a method provided by the `Vehicle` class.

To this end, each class has a *descriptor* containing a *virtual method table*, and each object `v` has a pointer `v.class` to the class descriptor. Now `v.drive(100)` is shorthand for

```
v.class.vtable[0](v, 100)
```

where the `0` is the index for the `drive` method in the vtable. This immediately gives us *polymorphism*: provided we arrange for all vehicles to use a consistent index `0` for the method, the same code will call the `drive` method of whichever class the vehicle `v` belongs to.

On Keiko, we can use the following code for the call.

```
CONST 100
LDGW Cars.v
DUP
LOADW
LDNW 12
CALL 2
```

Note that the value of v is used both to look up the method and to provide the first parameter; that's the reason for the DUP instruction. The instruction LOADW fetches the descriptor address from the object, and the instruction LDNW 12 reflects the fact that the vtable is at offset 12 in the class descriptor.

The third leg of the stool, *encapsulation*, can be enforced as a facet of semantic analysis: each class is represented in the compiler by a little environment in which instance variables and methods can be marked as public or private. It's an error to refer to a private instance variable or method except from the text of the same class. In Java, additional checks are made at the time the program is loaded into the JVM, but generally there is no protection mechanism at the level of machine code.

(Single) *inheritance* means that a class can be defined by extending an existing class, adding instance variables and methods and redefining existing methods.

The Car class has two instance variables, reg and miles, and three methods drive, getmiles and commute.

The Prius class adds an instance variable called charge, redefines the drive method, inherits the getmiles and commute methods, and adds a method called recharge.

Several things need explaining: (i) After

```
var c: Car; p: Prius;
new(p); c := p
```

the call c.drive(10) will invoke the Prius version of the drive method. This happens naturally as a result of the vtable mechanism. The Car part of the vtables share a common layout, so the drive method can be found in the same way in both of them.

(ii) However it is invoked, the shared getmiles method correctly returns the value of the miles instance variable, even if the receiver is actually an instance of Prius. This works because the Car part of a Prius's instance variables is laid out the same way as an ordinary car.

(iii) If Car.commute() is defined as

```
for i := 1 to 5 do self.drive(50) end
```

then invoking commute with a Prius as the receiver will use the Prius version of the drive method: this is 'late binding'.

(iv) Writing Prius.drive(dist) as

```
super.drive(d); self.charge := self.charge - dist div 10
```

involves a 'super call' (written self.drive↑(dist) in Oberon–2) that can be translated as a static call to Car.drive.

This implementation of inheritance suffers from the 'fragile binary interface problem', in that changes to Car that don't affect its public interface (e.g., adding a new private method) will require all subclasses like Prius to be recompiled, because the vtable layout will have changed. For Java, this was viewed as unacceptable, because code could be spread all over the web. Consequently, the JVM delays laying out the vtables until the classes are loaded, rather than doing it at compile time.

The last detail in a typical object-oriented language is type tests. We can write v is Car as a boolean expression that will yield true if v is a Car or Prius but false if it is an ordinary Vehicle or another subclass of Vehicle

that is not also a subclass of `Car`. To implement this in a fixed number of instructions, we can label each class descriptor with its level k, which is 0 for `Vehicle` (or for `Object` if that exists as a universal superclass), 1 for `Car` and 2 for `Prius`. Each descriptor also contains an array of $k + 1$ pointers to ancestor descriptors: for `Car` this will contain [`Vehicle`, `Car`] and for `Prius` it will contain [`Vehicle`, `Car`, `Prius`]. Then the test `v is Car` is implemented by

```
(v.class.level >= 1) & (v.class.ancestor[1] = Car),
```

giving the correct answer in each case without having to search the superclass chain.

Exercises

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

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

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

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

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

4.2 Here is a procedure that combines nesting and recursion:

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

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

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

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

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

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

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

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

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

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

4.4 [2013/3; see ppc4/test/sumarray.p] Figure 6.12 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`.

```
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 6.12: *Program for exercise 4.4*

(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?

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.

Chapter 7

Lab three: Subroutines

This lab asks you to add recursive subroutines with nesting, and (optionally) procedural parameters to a compiler that translates a simple, untyped programming language into postfix code. The language has integers as the only data type, but otherwise has a Pascal-like syntax. A syntax summary appears in Figure 7.1. The lab kit contains a working compiler that can already handle the parts of the language that are not connected with procedures.

7.1 Compiler structure

The files needed for this lab can be found in the directory `lab3` of the lab kit. Here is a list of the modules that make up the compiler:

Module	Description
Tree	Abstract syntax
Lexer	Lexical analyser
Parser	Syntax analyser
Check	Semantic analysis
Dict	Symbol tables
Keiko	Abstract machine code
Peepopt	Peephole optimiser
Kgen	Abstract machine code generator

The first step is to build the parts of the compiler that are provided. To do this, just type

```
$ make
```

A number of commands will be executed: these generate the lexical and syntax analysers by running *ocamllex* and *ocamlyacc* on their respective script files, and compile them and other modules from the kit. The end product is an executable program `ppc3`. The compiler as supplied cannot deal with programs that contain procedure calls, but it *can* run the usual GCD program, supplied as the file `gcd.p`.

Your task during the lab will be to add to the code generator *Kgen* the ability to generate code for procedure calls, the ability to access parameters and

```
program  →  block "."

block    →  [ var idents ";" ]
            { proc ident "(" [ idents ] ")" ";" block ";" }
            begin stmts end

stmts    →  stmt { ";" stmt }

stmt     →  empty
         |  ident ":=" expr
         |  if expr then stmts [ else stmts ] end
         |  while expr do stmts end
         |  print expr { "," expr }
         |  return expr

expr     →  number
         |  true | false
         |  ident
         |  ident "(" [ expr { "," expr } ] ")"
         |  monop expr
         |  expr binop expr
         |  "(" expr ")"

binop    →  "*" | "/"
         |  "+" | "-"
         |  "<" | "<=" | "=" | "<>" | ">" | ">="

idents   →  ident { "," ident }
```

Figure 7.1: *Syntax summary*

local variables in nested procedures, and (optionally) the ability to pass to one procedure as an argument to another. For the optional part of the practical, you will need to make some small additions to the semantic analyser *Check* also. Listings of file check.ml and kgen.ml appear in Appendix E.

7.2 Frame layout

Like the implementation of procedures that is described in the course notes, the compiler in this lab uses a stack for activation records that grows downwards in memory. The frame layout is identical to that described in the lectures. Before a procedure call, the expression stack should contain (in order from bottom to top) the following:

- The n parameters of the call, starting with the n'th and ending with the first.
- The static link.

- The procedure address.

The instruction PCALLW *n* saves the current context and jumps to the start of the procedure. PCALLW is the instruction for calling a procedure that returns a one-word result. As part of the process of calling the procedure, space is reserved in the new stack frame for the local variables of the procedure, and the frame pointer is set to point to the frame head.

Execution of a procedure body ends with a RETURN instruction. This expects the procedure's result (if any) on the expression stack, and leaves it there for use by the caller. It removes the stack frame from the subroutine stack, restoring the old values of the stack pointer and frame pointer, then jumps to the return address.

7.3 Procedure calls

The code generator already has the ability to compile the definitions of procedures, but it cannot compile procedure calls. Your first step should be to fill in this part of the function *gen_expr*, replacing the "*failwith*" that is there as a place-holder. Until Section 7.5, you can pass zero as the static link for every procedure.

For now, your compiler will not be able to translate procedure bodies that access parameters or local variables, but to prepare for the next step you should allow for parameters to be passed on the stack. If you like, you can combine this step with the next and implement procedures with parameters all at once.

Two suggestions for testing your new compiler: either modify gcd.p by making parts of the main program into a subroutine, or try the program fac0.p, which uses recursion (but no parameters) to compute 10 factorial.

7.4 Parameters and local variables

At present, the code generator is only able to handle references to global variables. Your next step should be to add the ability to handle local variables and parameters, both of which are accessed by an offset from the base of a stack frame.

You should enhance the code generator to handle local variables and parameters, both of which have a definition whose *d_level* field is non-zero, but with a *d_offset* field that is positive for parameters and negative for locals. You'll find that you have to modify the compiler function *gen_addr* so that it uses a LOCAL instruction to handle variables that are local to the current procedure. For a test program, try fac.p, the usual recursive factorial program. Try writing further tests of your own, including a test that gives different answers depending on whether parameters are evaluated in left-to-right order or in right-to-left order.

7.5 Nested procedures

Next, you should implement nested procedures by passing the proper static link in place of the dummy value of 0 you used before. You will also need to enhance *gen_addr* further to deal with parameters and other variables that are neither local to the current procedure nor global to the whole program. For these, it is necessary to follow one or more links of the static chain.

- In the body of a procedure at level m, we might access a local variable x at offset o in the procedures ancestor at level n, where $0 < n \leq m$. To compute the address of x, we need a code sequence made up of LOCAL 0, followed by $m - n$ repetitions of CONST 12; OFFSET; LOADW, followed by CONST o; OFFSET.[1]

- From a procedure at level m, we can call another procedure that is nested inside one of its ancestors. If the procedure being called is at level n, where $1 < n \leq m + 1$, then we need to pass as its static link the base address of its parent at level $n - 1$, computed as above.

To test your implementation, you can use the program sumpow.p, which computes the sum $1^k + 2^k + \cdots + n^k$, using a nested procedure to calculate the powers j^k. If it runs correctly, then the answer 979 is printed.

7.6 Functional parameters (optional)

Functional parameters are typically represented by closures: that is, pairs that contain a code address and a frame pointer. Usually, such a closure will occupy two words and an ordinary parameter will occupy only one word, and the code generator needs help from the semantic analyser in order to keep things straight. In this lab, we want to avoid the complication of including type checking in the semantic analysis, and that means that all runtime values must have the same representation – a single 32-bit word.

On our simulated machine, the size of the stack is quite small and the memory words are quite big, so we can fudge things by packing both values into a single word when a closure is created, and unpacking them again each time it is called. To help with this, I've provided two instructions, PACK and UNPACK. The PACK instruction pops two values from the expression stack, packs them into a single value, and pushes that value back onto the stack. The UNPACK instruction does the exact reverse.[2]

To implement functional parameters, you should follow these steps:

(1) Enhance the semantic analyser *check* to allow procedure names to be used as identifiers in expressions, and to allow variables to be used as the procedure names in calls. The places to make your modifications

[1] This sequence can be optimised by replacing LOCAL 0; CONST k; OFFSET with LOCAL k, replacing LOCAL k; LOADW with LDLW k, and replacing CONST k; OFFSET; LOADW by LDNW k.

[2] These rather limited instructions are not used by the Oberon compiler from which the Keiko machine is adapted, which instead has a semantic analyser that can allocate two words to a procedural parameter. The implementation of PACK and UNPACK supports up to 256 different code addresses, and static links that point anywhere within a subroutine stack of up to 16MB.

are where error messages about these usages are issued in the existing analyser. When a call uses a variable as the function to be called, you won't be able to check that it is called with the right number of parameters, because that needs a semantic analyser that associates a type with each variable, so that when a parameter has a procedure type, we know how many arguments it expects, and presumably the types of those arguments.

(2) Enhance the code generator to handle the same forms of expression. When a procedure name is used in an expression, the code should push the two words of its closure onto the stack just as if it was about to be called, then use a PACK instruction to pack the two words into one. When a procedure call contains a variable, the code should load its value, then use an UNPACK instruction to recover the two words of the closure.

You can test your implementation on two sample programs that are included in the lab kit: sumpow2.p solves the same problem as sumpow.p, but using a higher-order function to do the summation; and digits.p is our favourite nine digits program.

To think about: if procedure names can be used as expressions, does this allow procedure-valued variables and not just parameters? Under what circumstances does it make sense for a procedure to return a procedure value as its result?

Chapter 8

Machine code

8.1 Lecture 13: The back end

The story so far is that we know how to implement a high level programming language in terms of the low level operations of the Keiko machine. But what if we want to translate the high level language into the machine code native to a particular real machine, instead of the invented code of Keiko? A good solution is to make a separate translator (the *back end*) that takes the Keiko code for a program (or something like it) produced by the *front end* of the compiler, and translates it into machine code. There are many advantages to this approach, because it separates concerns about the meaning of the high level language from concerns about the instruction set of the target machine. For example, we could make compilers for the same programming language on several different target machines by combining a common front end with multiple, interchangeable back ends.

We shall take the intermediate representation to be not sequences of Keiko instructions but (broadly speaking) the Keiko instructions arranged as an *operator tree*, showing the flow of values between instructions explicitly instead of leaving them implicit in the stack-oriented nature of the machine. As an example, the statement `x := a[i]`, where `x` and `i` are locals and `a` is a global array, we might have this Keiko code:

```
GLOBAL _a
LOCAL 40; LOADW
CONST 2; LSL; OFFSET
LOADW
LOCAL -4; STOREW
```

We will arrange this into the tree shown in Figure 8.1, showing (e.g.) that the inputs to the OFFSET are the global address of _a and the result of the LSL. We'll write that tree also as

```
⟨STOREW,
  ⟨LOADW,
    ⟨OFFSET,
      ⟨GLOBAL _a⟩,
```

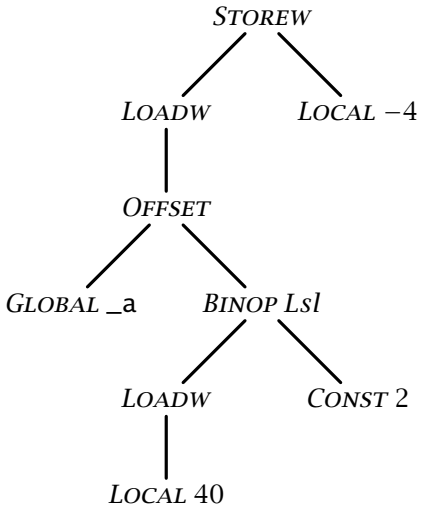


Figure 8.1: *An operator tree*

$\langle$*BINOP Lsl*, $\langle$*LOADW*, $\langle$*LOCAL* 40$\rangle\rangle$, $\langle$*CONST* 2$\rangle\rangle\rangle$
$\langle$*LOCAL* $(-4)\rangle\rangle$

using an (I hope) obvious notation. At this point, there is no advantage in using compressed instructions like *LDLW* 40 in place of the combination of *LOCAL* 40 and *LOADW*. Previously, such abbreviations were introduced by the peephole optimiser, but now a similar job will be performed by the pattern matching process that chooses machine instructions, combining multiple operations into one instruction when that is advantageous. The rules for choosing machine instructions will be simpler if we stick to a small variety of operations in the tree.

It's not too hard to modify the front end (in particular the code generator *Kgen*) to generate these trees instead of sequences of Keiko instructions, especially is we can assume that the evaluation stack is empty at each jump or label. For example, the translation of assignment statements was expressed like this:

```
let rec gen_stmt =
  function ...
    | Assign (v, e) →
        SEQ [gen_expr e; gen_addr v; STOREW]
```

We can recast this to produce an operator tree instead (from the module *Tgen* in Lab 4):

```
let rec gen_stmt =
  function ...
    | Assign (v, e) →
        ⟨STOREW, gen_expr e, gen_addr v⟩
```

Other code generator subroutines like *gen_expr* and *gen_addr* also change to generate operator trees instead of plain lists of Keiko instructions. These operator trees form a type *optree* that is defined in the *Optree* module:

```
type optree = Node of inst * optree list
```

We'll continue to use a special notation for elements of this type, writing

$$\langle \textsc{Offset}, t_1, \langle \textsc{Const}\ n \rangle \rangle$$

for what ML would render as

$$\mathit{Node}\,(\textsc{Offset}, [t_1; \mathit{Node}\,(\textsc{Const}\ n, [\,])]),$$

In the lab materials, there's a tool called *nodexp* that takes an ML source file containing the compact notation and expands it to proper ML before it is submitted to the ML compiler.

We choose this uniform representation for optrees because it makes it simpler to write functions that treat all optrees alike: for example, the compiler includes a pass that counts the number of *Temp* nodes in a tree, and the function that does this treats *Temp* nodes specially without having a separate case for each other operation. The disadvantage is that the tree is less strongly typed: while, for example, every *Offset* node should have two children, this is not enforced by the OCaml type system, but only by convention as the compiler is developed.

The translation of a procedure body becomes a *sequence* of operator trees. The roots of the trees are labelled with instructions that don't produce a result, such as *Storew* and conditional branches, or with labels for the branches to target. Their children are labelled with operations like *Offset* and *Loadw* and *Local n* that do produce a result; each such operation corresponds to a Keiko instruction that pops a number of values from the stack and pushes a single result, and a node labelled with that operation will have as many children as there are values consumed by the instruction. Although the front end now generates trees rather than sequences of Keiko instructions, it still translates control structures into networks of labels and conditional branches, and still renders data structure access using arithmetic on addresses.

The task of the back end is not to replicate the work of the front end, but to decide how to realise these conditional branches and addressing calculations using features of the target machine, translating the trees into assembly language. On the ARM, reasonable assembly language for the statement `x := a[i]` is as follows:[1]

```
ldr r0, =_a
ldr r1, [fp, #40]
lsl r1, r1, #2
ldr r0, [r0, r1]
str r0, [fp, #-4]
```

Appendix C gives a guide to assembly language programming for the ARM, but here is a rough interpretation of this snippet of code.

- The first instruction, `ldr r0, =_a`, puts the global address `_a` into register `r0`: it is a specific type of load instruction that uses PC-relative addressing to fetch a constant stored in the code segment near the procedure.

[1] This 'reasonable' translation fails to exploit the addressing mode where the value in one register is shifted left before being added to another. We will come to that presently!

- The second instruction, `ldr r1, [fp, #40]`, loads the value of `i` into register `r1` by adding together the value of the frame pointer `fp` and the offset 40 and loading from that address.
- The third instruction, `lsl r1, r1, #2`, multiplies this by 4, shifting it two bits to the left, putting the result back in `r1`.
- The fourth instruction, `ldr r0, [r0, r1]`, adds `r0` and `r1` together to form an address, loads from this address, and puts the value in `r0`.
- The last instruction, `str r0, [fp, #-4]` stores this value into **`x`**, again using addressing relative to the frame pointer.

Compiler phases: The work of the compiler can be split into tasks for the front end and the back end.

- Front end tasks are: replacing control structures with labels and conditional branches; replacing data structures with address arithmetic; replacing variable scopes with access code.
- Back end tasks are: selecting machine instructions for each operation; deciding which values should live in registers; assigning a machine register to each value.

In a portable compiler, the front end is almost completely machine-independent, so only the back end needs to be rewritten for each target machine. The output of the back end is either assembly language or binary machine code for the target machine. Binary output removes the need for a separate assembly phase by incorporating the functions of the assembler (formatting instructions, fixing up branches and labels) into the compiler, but assembly language output insulates the compiler from issues of binary encoding, and makes the compiler easier to debug.

Stages in a simple back end, based on operator trees: A complete back end can be structured as a series of stages that communicate via lists of optrees (see Figure 8.2). The first couple of stages take over the role that used to be played by the peephole optimiser.

- A simplifier (enabled with the `-O` flag on the command line) that works on each tree separately, removing trivial operations like adding 0, propagating constants, and replacing expensive operations like multiplication by a power of 2 with cheaper ones like shifting.
- A jump optimiser (also enabled by `-O`) that tidies up jumps and labels, so that there are no branches that lead to another (unconditional) branch, or branches that skip over an unconditional branch.
- A simple form of *common sub-expression elimination* (enabled by the `-O2` flag), that identifies parts of a tree that are computed twice, and arranges for them to be computed just once, holding the value in a temporary register as long as it is needed.

Actually, these three stages can be almost machine-independent, so we might think of them as a 'middle end' in their own right: both the input and the output of this middle end is a sequence of optrees for each subroutine, and each stage can be omitted if we don't care so much about the quality of

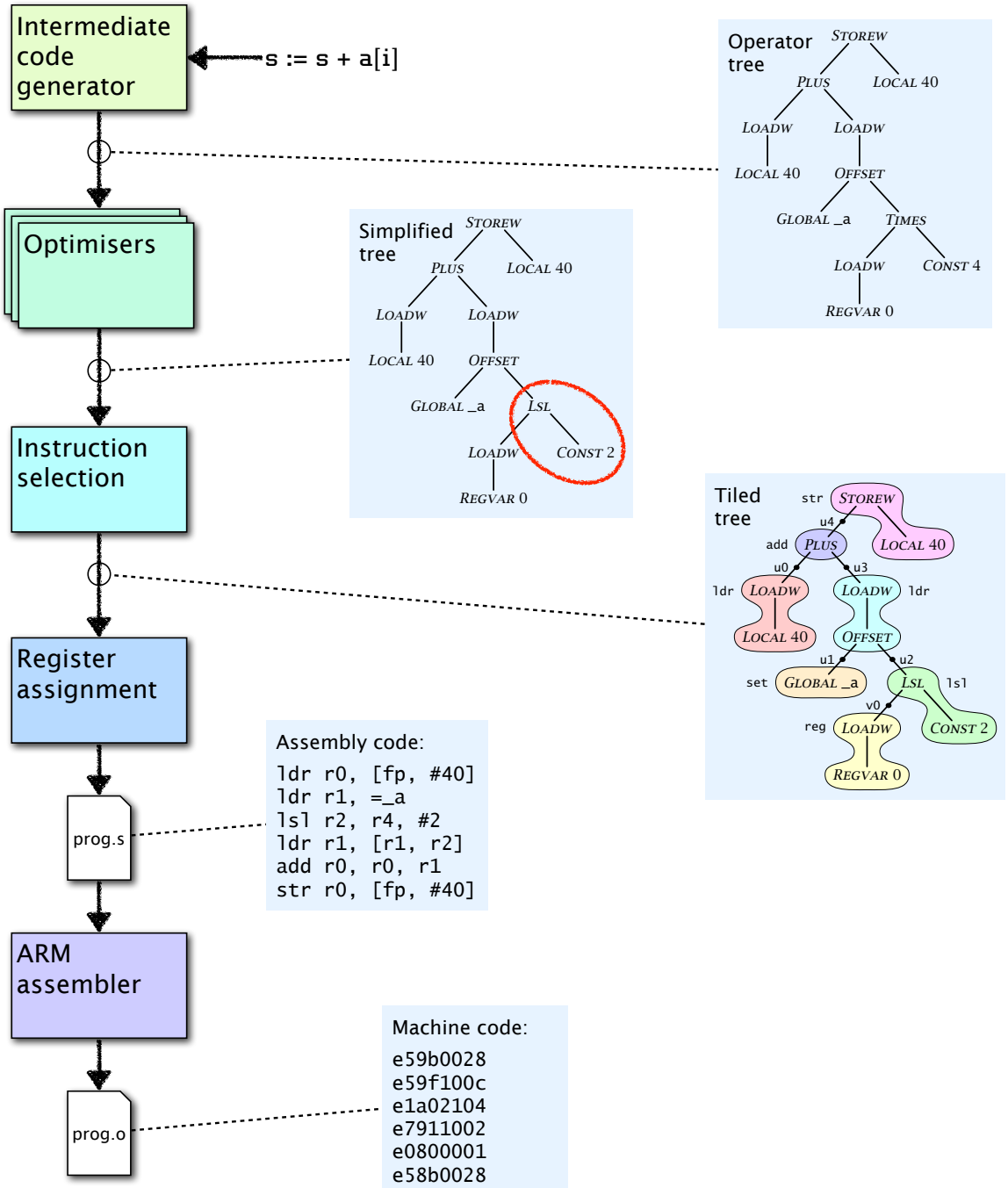


Figure 8.2: *Road map (back end)*

the object code. The back end proper also consists of several stages (see Figure 8.2):

- Instruction selection by tiling the tree – Section 8.2.
- Register assignment, using the finite register set to hold values between instructions – Section 8.4.

In reality, the different problems interact: for example, register assignment interacts with all previous stages because of the finite register set. The back end will work one procedure at a time, using a fixed interface between procedures (a slight extension of the ARM calling convention or ABI) to ensure that no special information about the procedure being called is needed to translate the procedure containing a call. The back end does a reasonable job of translating typical code, but it does not consider sophisticated optimisations that use the control and data flow of the program to move calculations out of loops, and keep values in registers over bigger regions.

Let's look at the translation of a whole procedure containing the statement `x := a[i]` that we studied earlier.

```
var a: array 10 of integer;

proc f(i: integer): integer;
  var x: integer;
begin
  x := a[i];
  return 3 * x
end;
```

The compiler places the parameter `i` at offset 40 in the stack frame for `f`, and the local `x` at offset −4. It generates the following sequence of two trees for the procedure body. (The first tree is indentical with the one shown earlier.)

⟨*STOREW*,
  ⟨*LOADW*,
    ⟨*OFFSET*,
      ⟨*GLOBAL* _*a*⟩,
      ⟨*BINOP Lsl*, ⟨*LOADW*, ⟨*LOCAL* 40⟩⟩, ⟨*CONST* 2⟩⟩⟩⟩,
  ⟨*LOCAL* −4⟩⟩

⟨*RESULTW*, ⟨*BINOP Times*, ⟨*LOADW*, ⟨*LOCAL* −4⟩⟩, ⟨*CONST* 3⟩⟩⟩

The back end generates code for this procedure in three parts. First, there is a prelude that saves the parameter (which according to the ARM calling convention arrives in a register) into the stack frame, and also saves those ARM registers that must be preserved by the procedure, before allocating space for local variables: see Figure 8.3. For simplicity the prelude saves all of `r4--r10`, even though not all of them are used in the body of the procedure; this would be one of the first things to improve in a compiler that generated better code.

As you can see, paramter `i` has an offset of 40 instead of the offset of 16 that it would have on Keiko, because of the greater size of the frame head; this difference is easily accommodated by adjusting the constants used by the semantic analyser when calculating the layout of the frame. Spare space is allocated in the frame so as to ensure that `fp` and `sp` are both multiples of

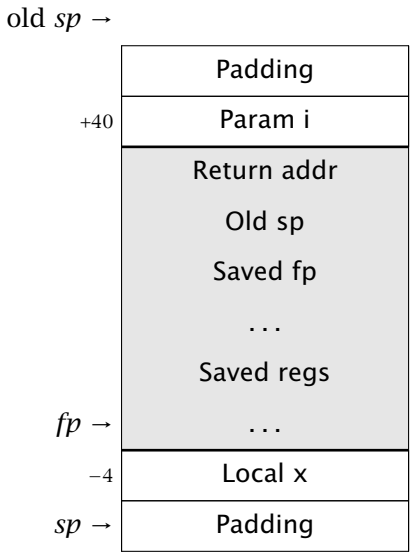


Figure 8.3: *Frame layout for* `f`

8, as the ABI requires. This is achieved by saving the values of both r0 and r1, though r1's value is never used, and decrementing the stack pointer by 8, though only 4 bytes of local variable space is needed.

```
_f:
      mov ip, sp
      stmfd sp!, {r0-r1}
      stmfd sp!, {r4-r10, fp, ip, lr}
      mov fp, sp
      sub sp, sp, #8
```

Then comes code for the procedure body, obtained by translating the optrees that were generated by the front end.

```
@   x := a[i];
      ldr r0, =_a
      ldr r1, [fp, #40]
      lsl r1, r1, #2
      ldr r0, [r0, r1]
      str r0, [fp, #-4]
@   return 3 * x
      ldr r0, [fp, #-4]
      mov r1, #3
      mul r0, r0, r1
```

Note also that this code contains a store followed by a load from the same address that could be optimised away.

Finally, there is a postlude that restores the registers that were saved earlier, in the process destroying the stack frame and returning to the caller.

```
      ldmfd fp, {r4-r10, fp, sp, pc}
      .pool
```

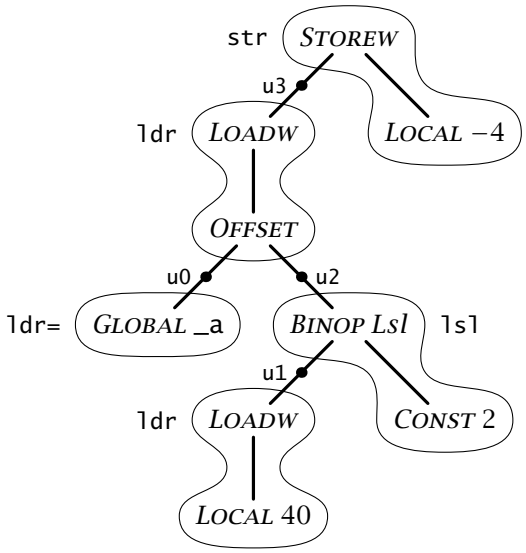


Figure 8.4: *Tiling an operator tree*

8.2 Lecture 14: Instruction selection

Let's start with the back end proper, and consider the problem of translating trees (such as the one shown in Figure 8.1) into sequences of instructions, ignoring for the moment the problem of choosing which registers to use. We can represent the results this process as a sequence of instructions where, instead of naming a specific register, each value is given a unique name u*k*, like this:

```
ldr u0, =_a
ldr u1, [fp, #40]
lsl u2, u1, #2
ldr u3, [u0, u2]
str u3, [fp, #-8]
```

It's then a separate problem to determine that `u0` and `u3` can live in the real register `r0`, and `u1` and `u2` can live in `r1`, without interference between them.

Each of the instructions corresponds to a 'tile' that covers one or more nodes of the operator tree, in a way that is seen in Figure 8.4. For example, the last `ldr` instruction covers the top *LOADW* node and the *OFFSET* node below it. The tiles are linked together by edges in the tree that I've labelled with virtual registers `u0`, ..., `u3`.

Many tilings of a tree are usually possible. For example, another tiling of the tree for **`x := a[i]`** is shown in Figure 8.5. This tiling fails to exploit the addressing hardware to add together the address and offset for the array access, and also pointlessly computes the address of **`x`** into a register – it's *feasible* but not optimal. Here is the code that results:

```
ldr u0, =_a
ldr u1, [fp, #40]
lsl u2, u1, #2
add u3, u0, u2
```

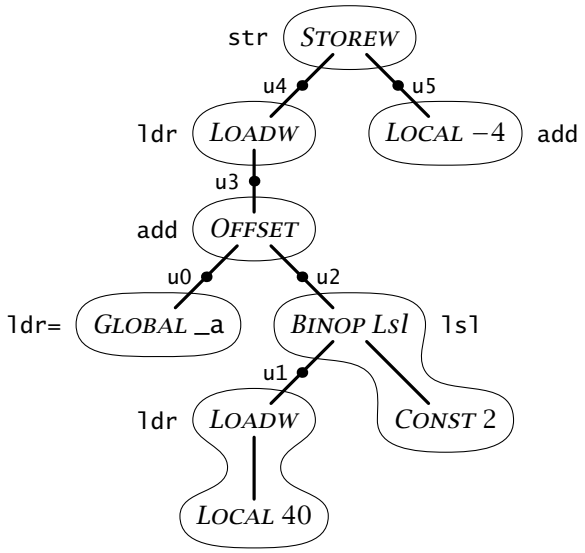


Figure 8.5: *Alternative tiling of an operator tree*

```
ldr u4, [u3]
add u5, fp, #-4
str u4, [u5]
```

In general, we want to find the tiling that gives the smallest number of instructions, or (if different instructions take different times) the fastest execution time.

To implement instruction selection, we need to solve two problems: first, to define what is meant (for a particular machine) by a feasible tiling of an operator tree; and second, to develop an algorithm for finding a near-optimal tiling for any tree. Surprisingly, the first problem is solved well by giving a context-free grammar (actually a *regular tree grammar*) for the set of optrees that can be translated into machine code. Of course, we hope that this set will include all optrees that can be generated by the front end of the compiler. We will use a non-terminal *reg* in the grammar to denote the set of optrees whose values we can compute into a register, and we might include a production such as

$$reg \rightarrow \langle \textsc{Binop}\ Plus, reg, reg \rangle$$

in the grammar to encode the fact that if we know how to compute the values of optrees t_1 and t_2 into registers, then we can also compute the value of $\langle \textsc{Binop}\ Plus, t_1, t_2 \rangle$ by first computing t_1 and t_2 and then executing an `add` instruction.

Because there is another form of `add` instruction that adds a register and a constant, we will also want to include a production

$$reg \rightarrow \langle \textsc{Binop}\ Plus, reg, \langle \textsc{Const}\ n \rangle \rangle$$

in the grammar. And then, since the second operand of many other instructions can also be either a register or a constant, it makes sense to introduce another non-terminal *rand* with the productions

$$rand \rightarrow reg$$

$rand \rightarrow \langle \textsc{Const}\ n \rangle$

and factor the productions that describe these instructions:

$reg \rightarrow \langle \textsc{Binop}\ Plus, reg, rand \rangle$
$reg \rightarrow \langle \textsc{Binop}\ Minus, reg, rand \rangle$
$reg \rightarrow \langle \textsc{Binop}\ BitAnd, reg, rand \rangle$

and so on.

As we cover more ARM instructions, we will also want a non-terminal *rand* for the addresses that appear in load and store instructions, so our list of nonterminals is

- *stmt* – the root of a tree;
- *reg* – a value in a register;
- *addr* – an address for `ldr` or `str`;
- *rand* – a register or constant.

For example, we could cover the *Storew* by either of the two productions:

$stmt \rightarrow \langle \textsc{Storew}, reg, \langle \textsc{Local}\ n \rangle \rangle$,

corresponding to the instruction `str` *reg*`, [fp, #`*n*`]`, or (using subscripts to distinguish multiple occurrences of the non-terminal *reg*)

$stmt \rightarrow \langle \textsc{Storew}, reg_1, reg_2 \rangle$,

corresponding to the instruction `str` reg_1`, [`reg_2`]`. It's more compact to write one production $stmt \rightarrow \langle \textsc{Storew}, reg, addr \rangle$, and provide productions for *addr* that are shared by *Loadw* and *Storew*.

$stmt \rightarrow \langle \textsc{Storew}, reg, addr \rangle$	{ `str` *reg*`,` *addr* }
$addr \rightarrow \langle \textsc{Local}\ n \rangle$	{ `[fp, #`*n*`]` }
$addr \rightarrow reg$	{ `[`*reg*`]` }

Figure 8.6 shows a set of productions relevant to the tree we have been using as an example, and a full set of rules is given in Appendix D.

To ensure that a tiling always exists, it's best to include rules like

$reg \rightarrow \langle \textsc{Offset}, reg_1, rand \rangle$	{ `add` *reg*`,` reg_1`,` *rand* }

that generate each operation on its own and can have inputs and output in registers, though this will not always give the best code, because we hope that the *Offset* node can be computed for nothing as part of the addressing mode in a load or store instruction. Local variables in a large stack frame are a case in point. If the frame is small, then loading from a local variable, $\langle \textsc{Loadw}, \langle \textsc{Local}\ n \rangle \rangle$, is covered by the rules

$reg \rightarrow \langle \textsc{Loadw}, addr \rangle$	{ `ldr` *reg*`,` *addr* }
$addr \rightarrow \langle \textsc{Local}\ n \rangle$	{ `[fp, #`*n*`]` }

But the second of these has a side-condition that *n* should be small enough to fit in the offset field of a load instruction. If that is not the case, the compiler falls back on a combination of other rules:

$addr \rightarrow reg$	{ `[`*reg*`]` }
$reg \rightarrow \langle \textsc{Local}\ n \rangle$	{ `ldr ip, =`*n*`; add` *reg*`, fp, ip` }

$reg \to \langle \textsc{Local}\ n \rangle$	{ `add` *reg*`,` `fp,` *#n* }
$addr \to \langle \textsc{Local}\ n \rangle$	{ `[fp,` *#n*`]` }
$reg \to \langle \textsc{LoadW}, addr \rangle$	{ `ldr` *reg*`,` *addr* }
$reg \to \langle \textsc{Const}\ n \rangle$	{ `mov` *reg*`,` *#n* }
$rand \to \langle \textsc{Const}\ n \rangle$	{ *#n* }
$reg \to \langle \textsc{Binop}\ Lsl, reg_1, rand \rangle$	{ `lsl` *reg*`,` reg_1`,` *rand* }
$reg \to \langle \textsc{Offset}, reg_1, rand \rangle$	{ `add` *reg*`,` reg_1`,` *rand* }
$addr \to \langle \textsc{Offset}, reg_1, reg_2 \rangle$	{ `[`reg_1`,` reg_2`]` }
$addr \to \langle \textsc{Offset}, reg, \langle \textsc{Const}\ n \rangle \rangle$	{ `[`*reg*`,` *#n*`]` }
$stmt \to \langle \textsc{StoreW}, reg, addr \rangle$	{ `str` *reg*`,` *addr* }
$reg \to \langle \textsc{Global}\ x \rangle$	{ `ldr` *reg*`,` *=x* }
$rand \to reg$	{ *reg* }
$addr \to reg$	{ `[`*reg*`]` }

Figure 8.6: *Productions for a tree grammar*

(The last of these applies even if n is very large.) The resulting code uses the scratch register `ip`, a synonym for `r12`:

```
ldr ip, =n
add u1, fp, ip
ldr u2, [u1]
```

If access to locals at large offsets is rare, then this code is good enough.

The tree grammar is highly ambiguous, so that a typical tree can be derived in many ways. We are looking for the derivation that uses the smallest number of tiles; or alternatively, we might annotate each production with a cost, and look for the derivation with the least total cost. For a RISC machine like the ARM, it's generally good enough to make a recursive matcher that implements a greedy algorithm, working from the root of the tree towards the leaves, and always biting off the biggest piece it can. For more complex instruction sets, there is a dynamic programming algorithm that always finds the best tiling, and tools exist that take a tree grammar and generate a matcher that uses the algorithm.

Actually, the tiling shown in Figure 8.4 is not optimal for the ARM, because the machine has an addressing mode where one register is shifted left, multiplying it by a power of two, before adding it to another. We could exploit this addressing mode by adding a new rule to the grammar:

$$addr \to \langle \textsc{Offset}, reg_1, \langle \textsc{Binop}\ Lsl, reg_2, \langle \textsc{Const}\ n \rangle \rangle \rangle$$
{ `[`reg_1`,` reg_2`, LSL #`*n*`]` }

This then gives an optimal four-instruction sequence for `x := a[i]`.

For RISC machines like the ARM, where a greedy selection algorithm is good enough, we can implement instruction selection using a family of mutually recursive functions, one corresponding to each non-terminal in the tree grammar. We will define functions

eval_reg : *optree* → *register* → *fragment*

eval_rand : *optree* → *fragment*

eval_addr : *optree* → *fragment*

exec_stmt : *optree* → *unit*

exec_call : *optree* → *unit*,

corresponding to the non-terminals *reg*, *rand*, *addr*, *stmt* and *call*. Note that each of these functions takes an *optree* as a parameter, and three of them return a *fragment* as a result, representing part of an assembly language instruction; we will define that type a bit later. The function *eval_reg* takes an additional parameter of type *register* that specifies which register should hold the result. Usually this value is the dummy register *R_any* that stands for any register, but sometimes (e.g., when evaluating the parameters to pass to a subroutine) we will want to specify a particular register. The function *exec_call* is used only for procedure calls, and we'll return to it later. The compiler used in Lab 4 has an additional function *eval_shift* that implements a non-terminal *shift* in the grammar, corresponding to the shift amount in instructions like `lsl` or logical-shift-left. The reasons for this are explained in Appendix D.

Analysing the input: Initially, let's ignore the fragments that are returned by these functions, and just concentrate on the way they take apart their *optree* argument, because that corresponds to choosing a tiling using a greedy algorithm. Figure 8.7 shows the parts of the functions that apply the productions shown in Figure 8.6.

The translation from rules to code generator functions is quite straightforward. Take, for example, the rule

stmt → ⟨STOREW, *reg*, *addr*⟩

This rule is reflected in the case for *exec_stmt* where we match a tree with the pattern ⟨STOREW, t_1, t_2⟩, then make recursive calls *eval_reg* t_1 *R_any* and *eval_addr* t_2. The other rules are reflected in the other functions in a similar way; note that *eval_rand* and *eval_addr* each have a catch-all case at the end that defers to *eval_reg*, corresponding to the productions *rand* → *reg* and *addr* → *reg*. Note also the side-conditions *fits_immed k* and *fits_offset n* that express the restriction (omitted in Figure 8.6) that *n* and *k* must be small integers in order to fit in an instruction.

The functions defined by this process are not very interesting: they take a tree and check that it can be covered by rules in the grammar, failing with a pattern match exception in *exec_stmt* or *eval_reg* if it cannot. To make them useful, we will have to enhance them so that, in addition to checking the tree can be covered, they also output the code for it.

Producing the output: Several of the proposed functions return a value of type *fragment*, intended to represent a fragment of code that doesn't amount to a complete instruction. We will treat these fragments as an abstract data type, with primitive fragments corresponding to numbers, symbols, register names and labels, and a 'constructor'

val *frag* : *string* → *fragment list* → *fragment*

```
let rec exec_stmt t =
  match t with
    | ⟨STOREW, t1, t2⟩ →
        eval_reg t1 R_any; eval_addr t2;
    | ...
and eval_reg t r =
  match t with
      ⟨CONST n⟩ when fits_move n → ()
    | ⟨LOCAL n⟩ when fits_add n → ()
    | ⟨GLOBAL x⟩ → ()
    | ⟨LOADW, t1⟩ →
        eval_addr t1
    | ⟨OFFSET, t1, t2⟩ →
        eval_reg t1 R_any; eval_rand t2
    | ⟨BINOP Lsl, t1, t2⟩ →
        eval_reg t1 R_any; eval_rand t2
    | ...
and eval_rand t =
  match t with
      ⟨CONST k⟩ when fits_immed k → ()
    | _ → eval_reg t R_any
and eval_addr t =
  match t with
      ⟨LOCAL n⟩ when fits_offset n → ()
    | ⟨OFFSET, t1, ⟨CONST n⟩⟩ when fits_ n →
        eval_reg t1 R_any
    | ⟨OFFSET, t1, t2⟩ →
        eval_reg t1 R_any; eval_reg t2 R_any
    | _ →
        eval_reg t R_any
```

Figure 8.7: *Code generator skeleton*

that joins together several existing fragments to make a bigger one. For example, if v_1 and v_2 are the two fragments *register* (*R* 0) and *number* 8 that represent the ARM register name `r0` and the constant `8`, then

frag "[$1, #$2]" [v_1; v_2]

is a fragment representing the address `[r0, #8]`. The two markers `$1` and `$2` in the template string refer to the two sub-fragments v_1 and v_2 supplied as a list. This scheme is closely related to the system of formats used with *printf*, but in addition keeps track of what registers are used in each fragment, so that the registers can be allocated and freed appropriately as instructions are generated.

Using this type, we can start to rewrite the code-generating functions so that they piece together the instructions that correspond to the tiling, and output the instructions as they go. Let's take as an example the rule

addr → ⟨*LOCAL n*⟩ { `[fp, #n]` }

which produces a piece of assembler syntax `[fp, #n]`. This *fragment* is what *eval_addr* should return if the rule applies, so that an appropriate case in that function is

```
let rec eval_addr t =
  match t with ...
    | ⟨LOCAL n⟩ when fits_offset n →
        frag "[fp, #$1]" [number n]
```

This value is returned to the caller of *eval_addr*, which will make the operand part of a larger instruction. To see this, let's look at a call to *eval_addr* from the function *exec_stmt*:

```
let rec exec_stmt t =
  match t with ...
    | ⟨STOREW, t1, t2⟩ →
        let v1 = eval_reg t1 R_any in
        let v2 = eval_addr t2 in
        gen "str $1, $2" [v1; v2]
```

This calls *eval_reg* and *eval_addr* to form two operand fragments, one a register name and the other an address, and then uses a subroutine *gen* to emit an instruction containing the two operands. The fragments in question could be `r1` and `[fp, #-4]`, and in that case the call *gen* "str $1, $2" [$v_1$; v_2] would output the instruction

```
str r1, [fp, #-4]
```

All the instructions output by the code generator are produced by means of the two functions,

gen : *string* → *fragment list* → *unit*

gen_reg : *string* → *reg* → *fragment list* → *fragment*

The difference between them is that *gen_reg* generates an instruction that puts its result in a register denoted by $0 in the template string, and it returns as a fragment the name of the register it has been chosen for the result.

With these conventions in place, we can start to rewrite the code generation functions so that they output the code as they go. Here is part of the enhanced definition of *eval_reg*:

```
let rec eval_reg t r =
  match t with ...
      ⟨CONST n⟩ when fits_move n →
        gen_reg "mov $0, #$1" r [number n]
    | ⟨CONST n⟩ →
        gen_reg "ldr $0, =$1" r [number n]
    | ⟨LOCAL n⟩ when fits_add n →
        gen_reg "add $0, fp, #$1" r [number n]
    | ⟨GLOBAL x⟩ →
        gen_reg "ldr $0, =$1" r [symbol x]
    | ⟨LOADW, t1⟩ →
        let v1 = eval_addr t1 in
```

```
        gen_reg "ldr $0, $1" r [v1]
    | ⟨OFFSET, t1, t2⟩ →
        let v1 = eval_reg t1 R_any in
        let v2 = eval_rand t2 in
        gen_reg "add $0, $1, $2" r [v1; v2]
    | ⟨BINOP Lsl, t1, t2⟩ →
        let v1 = eval_reg t1 R_any in
        let v2 = eval_rand t2 in
        gen_reg "lsl $0, $1, $2" r [v1; v2]
    | ...
```

The interaction between *gen_reg* and the special operand value *R_any* needs a bit of explanation. Typically, the argument *r* passed to *eval_reg* will be *R_any*, and when it is passed as the first operand in *gen_reg*, it indicates that *gen_reg* should choose some convenient register in which to compute the result, and should return that register as the value of *gen_reg*; this then becomes the result returned by *eval_reg*. For example, consider the tree

$$t = \langle \textsc{Loadw}, \langle \textsc{Local}\ 40 \rangle \rangle.$$

The call *eval_reg t R_any* matches the *LOADW* case in *eval_reg*, and results in a call of *eval_addr* ⟨*LOCAL* 40⟩. This, as we have seen before, returns a fragment v_1 representing `[fp, #40]`. So now *eval_reg* makes the call

gen_reg "ldr $0, $1" *R_any* [$v_1$]

It's the job of the register allocator (which has *gen_reg* as part of its interface) to choose a register that's not currently being used – say `r1`. The *gen_reg* call then outputs the instruction

```
ldr r1, [fp, #40]
```

and returns `r1` as the fragment *register* (*R* 1), which also becomes the result of *eval_reg*.

Working along the same lines, we can fill in more of the function *eval_addr*, incorporating the case we covered before.

```
[let rec] eval_addr t =
  match t with
      ⟨LOCAL n⟩ when fits_offset n →
        frag "[fp, #$]" [number n]
    | ⟨OFFSET, t1, ⟨CONST n⟩⟩ when fits_offset n →
        let v1 = eval_reg t1 R_any in
        frag "[$, #$]" [v1; number n]
    | ⟨OFFSET, t1, t2⟩ →
        let v1 = eval_reg t1 R_any in
        let v2 = eval_reg t2 R_any in
        frag "[$, $]" [v1; v2]
    | _ →
        let v1 = eval_reg t R_any in
        frag "[$]" [v1]
```

The aim here is to use the addressing modes to get an addition for free, whether it comes from a *OFFSET* operation or a *LOCAL* node. Trees of the form ⟨*OFFSET*, t_1, ⟨*CONST n*⟩⟩ use the addressing mode `[`*reg*`, #`*n*`]` when *n* is

sufficiently small. If n is too large for this, then the general rule for *OFFSET* that follows will put it in a register, and then use the [reg_1, reg_2] addressing mode. Likewise, a tree ⟨*LOCAL n*⟩ will be evaluated into a register if the value of n is too large to fit in the relevant instruction field.

There's also a function *eval_rand* that prepares the second operand of arithmetic instructions; this can be a register or a constant.

```
[let rec] eval_rand =
  function
      ⟨CONST k⟩ when fits_immed k → frag "#$" [number k]
    | t → eval_reg t R_any
```

The condition **when** *fits_immed k* ensures that the operand form #*k* is used only when the constant k is small enough to fit in the immediate field of an instruction. If a tree ⟨*CONST k*⟩ does not satisfy the condition, then the catch-all case at the bottom is used, and the constant is developed into a register. In this way, the expression `x+3` produces the `add` instruction,

```
add u1, u0, #3
```

while the expression `x+31416` produces a `ldr=` and an `add`:

```
ldr u2, =31416
add u1, u0, u2
```

Similar checks *fits_offset* and *fits_move* are included in some rules above, and the details are included in the code for Lab 4. Multiple different tests are needed because different ARM instructions have different rules for encoding constant operands and offsets.

For the root of an operator tree, we use a function *exec_stmt*. Again, a few example rules:

```
let exec_stmt =
  function
      ⟨STOREW, t1, t2⟩ →
        let v1 = eval_reg t1 R_any in
        let v2 = eval_addr t2 in
        gen "str $1, $2" [v1; v2]
    | ⟨LABEL lab⟩ → emit_lab lab
    | ⟨JUMP lab⟩ → gen "b $1" [codelab lab]
    | ⟨JUMPC (Eq, lab), t1, t2⟩ →
        let v1 = eval_reg t1 R_any in
        let v2 = eval_rand t2 in
        gen "cmp $1, $2 / beq $3" [v1; v2; codelab lab];
    | ...
```

The rule for *JUMPC* illustrates an additional convention by which a call to *gen* or *gen_reg* can generate multiple instuctions written with / as a separator.

Working together, these functions will take an optree and output assembly language code for it. They rely on *gen* and *gen_reg* to keep track of which registers are in use, so that *gen_reg* never tries to put a value in a register that is already occupied by another value that is still wanted. This can be achieved with a simple scheme where we keep track of the set of registers that are free. Then *gen_reg* can choose a free register and remove it from the

set, and both *gen* and *gen_reg* can look for registers among the inputs of the instruction and return them to the free set.

8.3 Lecture 15: Common sub-expressions

As with postfix Keiko code, it's helpful to supplement a simple intermediate code generator with an optimiser that finds and simplifies trivial operations, and also tidies up jumps and labels, removing jumps-to-jumps, unused labels, and so on. These jobs were done by the peephole optimiser in our earlier compilers, but in a back end based on optrees, they can be done by one stage (*Simp*) that makes a bottom-up pass over individual trees, and another (*Jumpopt*) that deals with branches and labels by looking only at the roots of a sequence of trees. The details are not hard to work out, and implementations can be found in the code supplied for Lab 4.

In addition to these, it is helpful in even a simple compiler to include a stage that eliminates common sub-expressions from the code. This is not so much because the programmer using our compiler might write the same expression twice, but more because repeated calculations are inevitably introduced as part of the translation process, and on a register machine eliminating them can result in significantly better code.

For example, a natural translation turns the statement **x := x * x**, where **x** is a global variable, into the tree

⟨*STOREW*,
  ⟨*BINOP Times*,
    ⟨*LOADW*, ⟨*GLOBAL _x*⟩⟩,
    ⟨*LOADW*, ⟨*GLOBAL _x*⟩⟩⟩,
  ⟨*GLOBAL _x*⟩⟩

If translated directly into machine code, this would load the value of **x** twice, and would actually put the address of **x** into a register three times, once of each of the loads and again for the store. Common sub-expression elimination turns the single tree into a sequence of three tress that communicate via two temporaries:

⟨*DEFTEMP* 1, ⟨*GLOBAL _x*⟩⟩
⟨*DEFTEMP* 2, ⟨*LOADW*, ⟨*TEMP* 1⟩⟩
⟨*STORE*, ⟨*BINOP Times*, ⟨*TEMP* 2⟩, ⟨*TEMP* 2⟩⟩, ⟨*TEMP* 1⟩⟩

We can keep the temps in registers (call them `t1` and `t2`) and translate the trees into these instructions:

```
ldr t1, =_x
ldr t2, [t1]
mul u1, t2, t2
str u1, [t1]
```

The common sub-expressions here are small: the address of the global variable **x** and its value, but the nature of RISC machines means eliminating them saves computing the address of **x** into a variable three times and loading it twice.

Our approach will be to transform an input tree into a directed acyclic graph (DAG) in which repeated subtrees are shared, then identify nodes in the

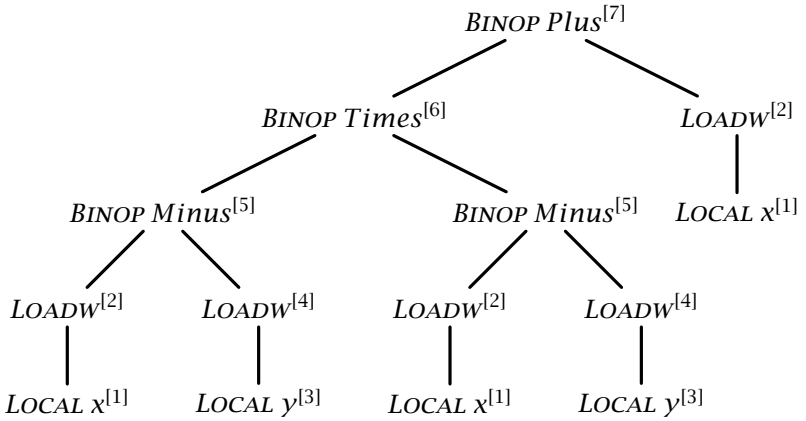


Figure 8.8: *Expression tree with value numbers*

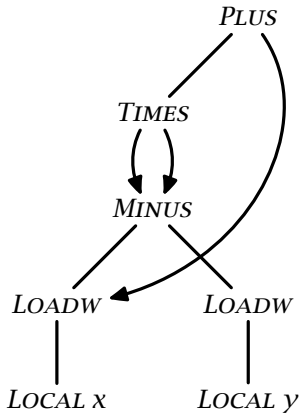


Figure 8.9: *Directed acyclic graph*

DAG that have more than one parent and compute their values into temps. For example, take a tree for `(x-y) * (x-y) + x` (see Figure 8.8). Turn it into a DAG by sharing subtrees (Figure 8.9), then use temps to turn it back into trees:

⟨*DEFTEMP* 1, ⟨*LOADW*, ⟨*LOCAL x*⟩⟩⟩

⟨*DEFTEMP* 2, ⟨*BINOP Minus*, ⟨*TEMP* 1⟩, ⟨*LOADW*, ⟨*LOCAL y*⟩⟩⟩⟩

⟨*BINOP Plus*, ⟨*BINOP Times*, ⟨*TEMP* 2⟩, ⟨*TEMP* 2⟩⟩, ⟨*TEMP* 1⟩⟩

Naive code for the original tree:

```
ldr u0, [fp, #x]
ldr u1, [fp, #y]
sub u2, u0, u1
ldr u3, [fp, #x]
ldr u4, [fp, #y]
sub u5, u3, u4
mul u6, u2, u5
ldr u7, [fp, #x]
```

```
add u8, u6, u7
```

After common sub-expression elimination:

```
ldr t1, [fp, #x]
ldr u0, [fp, #y]
sub t2, t1, u0
mul u1, t2, t2
add u2, u1, t1
```

(Note that the forms ⟨*DEFTEMP n*, _⟩ and ⟨*TEMP*, *n*⟩ generate no code if the temp lives in a register.)

The conversion from tree to DAG is done by an algorithm called *value numbering*: give each DAG node a serial number as it is created, and keep a table (use a hash table) showing the operator and operands and the value number for the result.

Value	Operation	Ref. count
[1]	⟨*LOCAL x*⟩	1
[2]	⟨*LOADW*, [1]⟩	2
[3]	⟨*LOCAL y*⟩	1
[4]	⟨*LOADW*, [3]⟩	1
[5]	⟨*BINOP Minus*, [2], [3]⟩	2
[6]	⟨*BINOP Times*, [5], [5]⟩	1
[7]	⟨*BINOP Plus*, [6], [2]⟩	0

The algorithm works by recursively copying the tree into the DAG, using the table to share nodes when possible. It's useful also to keep track of the *reference count* of each DAG node, so that nodes with a reference count > 1 can generate temps.

It's possible to extend common sub-expression elimination from single trees to *basic blocks*, straight-line fragments of code with a single entry at the top. Doing so requires us to recognise when a store operation might change the value of a subsequent load operation from the same address. As we process the basic block, we can just let the table grow, and deal with stores by killing the corresponding loads in the value table. For example, start with `y := x - y; z := x - y`:

⟨*STORE*,
  ⟨*BINOP Minus*, ⟨*LOADW*, ⟨*LOCAL x*⟩,
    ⟨*LOADW*, ⟨*LOCAL y*⟩⟩⟩,
  ⟨*LOCAL y*⟩⟩

⟨*STORE*,
  ⟨*BINOP Minus*, ⟨*LOADW*, ⟨*LOCAL x*⟩,
    ⟨*LOADW*, ⟨*LOCAL y*⟩⟩⟩,
  ⟨*LOCAL z*⟩⟩

The value table after processing the first tree looks like this:

Value	Operation	Ref. count
[1]	⟨*LOCAL x*⟩	1
[2]	⟨*LOADW*, [1]⟩	2

[3]	⟨*LOCAL y*⟩	2
[4]	~~⟨*LOADW*, [3]⟩~~	1
[5]	⟨*BINOP Minus*, [2], [3]⟩	2
[6]	⟨*STOREW*, [5], [3]⟩	0

At this point, the variable `y` changes, so we must be sure that the reference to `y` in the second statement does not use a copy of the value `y` had in the first statement. We can achieve this by *killing* the *LOADW* node [4] that references `y`, that is, removing it from the hash table while leaving it part of the DAG. I've suggested this by striking through the entry shown above.

What we need to do at each *STOREW* node is to search the table for *LOADW* nodes that *might* reference the same location. We need (a conservative approximation to) *alias analysis* to determine what to kill. In this example, we kill one *LOADW* but not the other, because we may assume that *LOCAL x* and *LOCAL y* are different addresses. Procedure calls kill all variables that could be modified by the procedure (conservative approximation: kill everything in memory).

We can also play a trick and insert a fresh node [7] for ⟨*LOADW*, [3]⟩ to see if it is used in the future: if so, then we can avoid reloading the value that was stored in the *STOREW*. Processing of the second tree goes as follows.

Value	**Operation**	**Ref. count**
[7]	⟨*LOADW*, [3]⟩	1
[8]	⟨*BINOP Minus*, [2], [7]⟩	1
[9]	⟨*LOCAL z*⟩	1
[10]	⟨*STOREW*, [8], [3]⟩	0

Because the artificial *LOADW* has been used, we allocate a temp for the value stored and reuse it later. Converting back to trees and selecting instructions, we get this code:

⟨*DEFTEMP* 1, ⟨*LOADW*, ⟨*LOCAL x*⟩⟩⟩
```
    ldr t1, [fp, #x]
```
⟨*DEFTEMP* 2, ⟨*BINOP Minus*, ⟨*TEMP* 1⟩, ⟨*LOADW*, ⟨*LOCAL y*⟩⟩⟩⟩
```
    ldr u1, [fp, #y]
    sub t2, t1, u1
```
⟨*STOREW*, ⟨*TEMP* 2⟩, ⟨*LOCAL y*⟩⟩
```
    stw t2, [fp, #y]
```
⟨*STOREW*, ⟨*BINOP Minus*, ⟨*TEMP* 1⟩, ⟨*TEMP* 2⟩⟩, ⟨*LOCAL z*⟩⟩
```
    sub u2, t1, t2
    stw u2, [fp, #z]
```

An example: Consider the procedure

```
proc swap(i, j: integer);
  var t: reg integer;
begin
  t := a[i];
  a[i] := a[j];
  a[j] := t
end;
```

where `t` lives in a register and `a` is global. Initial code is the three trees,

⟨*STOREW*,
  ⟨*LOADW*,
    ⟨*OFFSET*, ⟨*GLOBAL* _*a*⟩,
      ⟨*BINOP Lsl*, ⟨*LOADW*, ⟨*LOCAL* 40⟩⟩, ⟨*CONST* 2⟩⟩⟩⟩,
  ⟨*REGVAR* 0⟩⟩

⟨*STOREW*,
  ⟨*LOADW*,
    ⟨*OFFSET*, ⟨*GLOBAL* _*a*⟩,
      ⟨*BINOP Lsl*, ⟨*LOADW*, ⟨*LOCAL* 44⟩⟩, ⟨*CONST* 2⟩⟩⟩⟩,
  ⟨*OFFSET*, ⟨*GLOBAL* _*a*⟩,
    ⟨*BINOP Lsl*, ⟨*LOADW*, ⟨*LOCAL* 40⟩⟩, ⟨*CONST* 2⟩⟩⟩⟩

⟨*STOREW*,
  ⟨*LOADW*, ⟨*REGVAR* 0⟩⟩,
  ⟨*OFFSET*, ⟨*GLOBAL* _*a*⟩,
    ⟨*BINOP Lsl*, ⟨*LOADW*, ⟨*LOCAL* 44⟩⟩, ⟨*CONST* 2⟩⟩⟩⟩

Clearly the global address `_a` should be shared. Also, the values of `i` and `j` can be shared, provided alias analysis tells us that the assignments `t := a[i]` and `a[i] := a[j]` do not affect their values. We get the trees,

⟨*DEFTEMP* 1, ⟨*GLOBAL* _*a*⟩⟩

⟨*DEFTEMP* 2,
  ⟨*OFFSET*, ⟨*TEMP* 1⟩, ⟨*BINOP Lsl*, ⟨*LOADW*, ⟨*LOCAL* 40⟩⟩, ⟨*CONST* 2⟩⟩⟩⟩

⟨*STOREW*, ⟨*LOADW*, ⟨*TEMP* 2⟩⟩, ⟨*REGVAR* 0⟩⟩

⟨*DEFTEMP* 3,
  ⟨*OFFSET*, ⟨*TEMP* 1⟩, ⟨*BINOP Lsl*, ⟨*LOADW*, ⟨*LOCAL* 44⟩⟩, ⟨*CONST* 2⟩⟩⟩⟩

⟨*STOREW*, ⟨*LOADW*, ⟨*TEMP* 3⟩⟩, ⟨*TEMP* 2⟩⟩

⟨*STOREW*, ⟨*LOADW*, ⟨*REGVAR* 0⟩⟩, ⟨*TEMP* 3⟩⟩

from which we can generate the code,

⟨*DEFTEMP* 1, ⟨*GLOBAL* _*a*⟩⟩
```
    ldr t1, =_a
```
⟨*DEFTEMP* 2,
  ⟨*OFFSET*, ⟨*TEMP* 1⟩, ⟨*BINOP Lsl*, ⟨*LOADW*, ⟨*LOCAL* 40⟩⟩, ⟨*CONST* 2⟩⟩⟩⟩
```
    ldr u1, [fp, #40]
    lsl u2, u1, #2
    add t2, t1, u2
```
⟨*STOREW*, ⟨*LOADW*, ⟨*TEMP* 2⟩⟩, ⟨*REGVAR* 0⟩⟩
```
    ldr v0, [t2]
```
⟨*DEFTEMP* 3,
  ⟨*OFFSET*, ⟨*TEMP* 1⟩, ⟨*BINOP Lsl*, ⟨*LOADW*, ⟨*LOCAL* 44⟩⟩, ⟨*CONST* 2⟩⟩⟩⟩
```
    ldr u3, [fp, #44]
    lsl u4, u3, #2
    add t3, t1, u4
```
⟨*STOREW*, ⟨*LOADW*, ⟨*TEMP* 3⟩⟩, ⟨*TEMP* 2⟩⟩
```
    ldr u5, [t3]
```

```
    str u5, [t2]
⟨STOREW, ⟨LOADW, ⟨REGVAR 0⟩⟩, ⟨TEMP 3⟩⟩
    str v0, [t3]
```

Sharing has improved this procedure, but the code is not quite optimal, because the `add` instructions could have been folded into the loads and stores. The compiler used in Lab 4 avoids this problem by introducing an extra phase in the CSE algorithm: after identifying maximal common sub-expressions, it inspects each shared node to see if in its context its evaluation can be repeated with no cost. If so, the compiler reverses the sharing before converting the DAG back into trees.

Procedure calls: It's convenient to allocate each procedure call to a temp, treating it as a common sub-expression even though its value is used only once. In an imperative language, even identical procedure calls must not be shared, because each call may have its own side effects. If we make this transformation, then procedure calls will no longer be nested inside other operations, and that makes it easier to generate code that puts the arguments in the right registers. This requires some care if side effects are present; e.g., `y + nasty(x)` may not be equivalent to

```
t := nasty(x); y + t
```

if `nasty` has a side effect on `x`, and we should pay attention to what the language manual says about cases like this.

Our treatment, allowing the transformation, is to generate the following initial code for the statement `z := y + nasty(x)` (see lab4/test/nasty.p).

```
⟨STOREW,
  ⟨BINOP Plus,
    ⟨LOADW, ⟨GLOBAL _y⟩⟩,
    ⟨CALL 1,
      ⟨GLOBAL _nasty⟩,
      ⟨STATLINK, ⟨CONST 0⟩⟩,
      ⟨ARG 0, ⟨LOADW, ⟨GLOBAL _x⟩⟩⟩⟩⟩,
  ⟨GLOBAL _z⟩⟩
```

The compiler next transforms the code so that the result of the call is assigned to a temp.

```
⟨DEFTEMP 1,
  ⟨CALL 1,
    ⟨GLOBAL _nasty⟩,
    ⟨STATLINK, ⟨CONST 0⟩⟩,
    ⟨ARG 0, ⟨LOADW, ⟨GLOBAL _x⟩⟩⟩⟩⟩
⟨STOREW, ⟨BINOP Plus, ⟨LOADW, ⟨GLOBAL _y⟩⟩, ⟨TEMP 1⟩⟩, ⟨GLOBAL _z⟩⟩
```

Next the code is flattened so that arguments appear as separate trees before the call.

```
⟨ARG 0, ⟨LOADW, ⟨GLOBAL _x⟩⟩⟩
⟨STATLINK, ⟨CONST 0⟩⟩,
⟨DEFTEMP 1, ⟨CALL 1, ⟨GLOBAL _nasty⟩⟩
⟨STOREW, ⟨BINOP Plus, ⟨LOADW, ⟨GLOBAL _y⟩⟩, ⟨TEMP 1⟩⟩, ⟨GLOBAL _z⟩⟩
```

Translate each tree.

⟨*ARG* 0, ⟨*LOADW*, ⟨*GLOBAL* _*x*⟩⟩⟩

```
    ldr r0, =_x
    ldr r0, [r0]
```

⟨*STATLINK*, ⟨*CONST* 0⟩⟩

⟨*DEFTEMP* 1, ⟨*CALL* 1, ⟨*GLOBAL* _*nasty*⟩⟩⟩

```
    bl _nasty
```

⟨*STOREW*,
⟨*BINOP Plus*, ⟨*LOADW*, ⟨*GLOBAL* _*y*⟩⟩, ⟨*TEMP* 1⟩⟩,
⟨*GLOBAL* _*z*⟩⟩

```
  ldr r1, =_y
  ldr r1, [r1]
  add r0, r1, r0
  ldr r1, =_z
  str r0, [r1]
```

8.4 Lecture 16: Register allocation

The remaining component needed for a working compiler is some way of assigning real registers `r0`, `r1`, ... to the symbolic registers u*i* and t*j* that we have shown in the output of instruction selection. The absolute requirement is that we do not try to use the same register to hold two different values at once; but we must also try to re-use registers where possible, or we will run out. In the simplest compiler, running out of registers is fatal – but for production use, a compiler should be able to recover by identifying the values that can be *spilled* to memory and inserting appropriate instructions to store and reload them. Doing so with least cost is the job of more sophisticated register allocation schemes than we have time to consider in this course.

Here is a simple scheme for managing registers that works fairly well on RISC machines, and can be adapted to other, more complex, machines. Instructions are selected by a function *eval_reg* that takes an *optree*, possibly outputs some instructions, and returns a *fragment* that denotes the register containing the result. We'll add an additional argument of type *register*:

eval_reg : *optree* → *register* → *fragment*

This added argument specifies which register should be chosen for the result, and the *fragment* that is returned indicates which register was actually used. We allow the special value *R_any* as the argument if we don't care.

For example, the `mul` instruction on the ARM requires its two operands in registers, so we make the rule,

let rec *eval_reg t r* =
  match *t* **with**...
    | ⟨*BINOP Times*, t_1, t_2⟩ →
      let v_1 = *eval_reg* t_1 *R_any* **in**
      let v_2 = *eval_reg* t_2 *R_any* **in**
      gen_reg "mul $0, $1, $2" *r* [$v_1$; v_2]

We use *R_any* twice in the recursive calls to denote the fact that we don't care what registers are used for the two operands of the `mul` instruction; we do

care that they are different, but we will deal with that in a moment. The final call to *gen_reg* has as operands the two values v_1 and v_2 to be multiplied, and also the result register r that came as an argument to *eval_reg*.

This subroutine *gen_reg* both emits an instruction and looks after allocating a register for the result.

```
let gen_reg fmt r rands =
  List.iter release rands;
  let r' = get_reg r in
  emit_inst fmt r' rands;
  register r'
```

Here, *release* marks any registers occupied by the operands as available for re-use, and *fix_reg* chooses a register r' to contain the result and marks it as in use. (If r names a specific register then r and r' will be the same; if r is *R_any* then r' will be the first actual register that is free.) The call to *emit_inst* actually outputs the instruction to the file of assembly language, and the function returns r' so it can be used in future instructions. Note that the operand registers are released before allocating a register to hold the result; it is this that lets us generate instructions that use the same register as both input and output, but it depends on a property of assembly language instructions that they read all their inputs before writing their output register. This property must hold of all fragments of code generated by the compiler, even if occasionally those fragments consist of multiple instructions.

In addition to *eval_reg*, there are other mutually recursive subroutines *eval_rand*, *eval_addr* and *exec_stmt*. They deal with registers in a similar way via *gen_reg* and (for *exec_stmt*) a similar function *gen* that does not have a result register.

Temps: To implement temps, we need a rule in *exec_stmt* and another in *eval_reg*. In *exec_stmt*, we compile ⟨DEFTEMP n, t_1⟩ by evaluating t_1 into any register (this time preferring a callee-save register by specifying *R_temp* in place of *R_any*) and remembering that as the location of the temp:

```
let rec exec_stmt =
  function ...
    | ⟨DEFTEMP n, t1⟩ →
        let v1 = eval_reg t1 R_temp in
        Regs.def_temp n (reg_of v1)
```

In *eval_reg* we compile ⟨TEMP n⟩ into an optional move instruction:

```
let rec eval_reg t r =
  match t with...
    | ⟨TEMP n⟩ →
        gen_move r (Regs.use_temp n)
```

If we are not fussy about r, then there is no need for a move instruction, and *gen_move* makes r the same register as the temp. We need to keep a reference count for temps (rather than just a Boolean flag) so that the register can be freed when there are no more uses to compile.

Procedure calls: So far, we have not needed to use the ability to specify where the value of an expression should be put. That changes when we look at procedure calls, because like most RISC architectures, the ARM needs us to

pass the first few arguments of a procedure in specific registers.[2] We can rely on the CSE pass to flatten nested procedure calls and transform each procedure call into a sequence of *ARG* trees followed by a *CALL* node, so that `f(x, 3)` becomes the sequence,

⟨*ARG* 1, ⟨*CONST* 3⟩⟩
⟨*ARG* 0, ⟨*LOADW*, ⟨*LOCAL x*⟩⟩⟩
⟨*CALL* ⟨*GLOBAL f*⟩⟩.

Then for $i < 4$ we implement ⟨*ARG* i, t⟩ like this:

let rec *exec_stmt* =
  function . . .
    | ⟨*ARG* i, t_1⟩ **when** $i < 4$ →
        spill_temps [*R i*];
        ignore (*eval_reg* t_1 (*R i*))

so that the tree is evaluated into the specific register. Then the *CALL* operation is implemented by

    | ⟨*CALL n*, ⟨*GLOBAL f*⟩⟩ →
        spill_temps volatile;
        gen "bl $1" [*symbol f*];
        release_args ( )

In both parts, *spill_temps* moves temps out of the caller-save registers into callee-save registers that will be preserved across the call: first the specific register r*i*, and later all remaining "volatile" registers `r0`. . . `r3`.

An example: The expression `g(a)+g(b)` is the simplest that causes a spill, generating the instruction `mov r4, r0` shown below.

⟨*ARG* 0, ⟨*LOADW*, ⟨*LOCAL* 40⟩⟩⟩
```
    ldr r0, [fp, #40]
```
⟨*DEFTEMP* 1, ⟨*CALL* 1, ⟨*GLOBAL* _*g*⟩⟩⟩
```
    bl _g
```
⟨*ARG* 0, ⟨*LOADW*, ⟨*LOCAL* 44⟩⟩⟩
```
    mov r4, r0
    ldr r0, [fp, #44]
```
⟨*DEFTEMP* 2, ⟨*CALL* 1, ⟨*GLOBAL g*⟩⟩⟩⟩
```
    bl _g
```
⟨*STOREW*, ⟨*BINOP Plus*, ⟨*TEMP* 1⟩, ⟨*TEMP* 2⟩⟩, ⟨*LOCAL* −4⟩⟩
```
    add r0, r4, r0
    str r0, [fp, #-4]
```

Looking wider: We can allocate registers to values

- in a single expression.
- across a basic block, using common sub-expression elimination.
- across an entire procedure by allowing register variables – better identified by compiler heuristic than by hand.

[2] We could exploit this in leaf routines by leaving the arguments in those registers and saving memory traffic, but we don't do so in this course.

A more sophisticated compiler can allocate values to registers over a loop, intermediate in size between a basic block and a whole procedure. By analysing the control and data flow of the program, they can keep values in registers from one iteration of a loop to the next, avoid redundant stores, eliminate repeated calculations over bigger regions, and move invariant computations out of loops. All that is beyond the scope of this course, however.

We have allocated registers greedily, assuming that we will not run out. That is good enough for a simple compiler, but optimising compilers can generate a much larger demand for registers, and that makes it vital for them to be able to spill registers to memory when they run out.

Instruction scheduling: this means re-ordering independent instructions for speed. It matters because on modern machines, (a) loads have a latency and can stall the pipeline; (b) on some machines, well-matched groups of instructions can be executed in parallel; (c) branches also have complex interactions with the pipeline. Example for (a):

```
ldr r0, [fp, #x]
ldr r1, [fp, #y]
(nop)
add r0, r0, r1
ldr r2, [fp, #z]
(nop)
mul r0, r0, r2
```

The stalls can be avoided by re-ordering the instructions:

```
ldr r0, [fp, #x]
ldr r1, [fp, #y]
ldr r2, [fp, #z]
add r0, r0, r1
mul r0, r0, r2
```

Exercises

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

5.2 The ARM has a multiply instruction `mul r1, r2, r3` that, unlike other arithmetic instructions, demands that both operands be in registers, and does not allow an immediate operand. How is this restriction reflected in the code generator?

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

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

where *rand* is the same non-terminal that describes the second operand of arithmetic instructions line `add`. Identify a source program that would be

wrongly translated by a compiler incorporating this rule. What goes wrong, how does the grammar in Appendix D avoid the problem?

5.4 Consider the following data type and procedure:

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

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

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

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

```
s := s + q↑.age;
q := q↑.next
```

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

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

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

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

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

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

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

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

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

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

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

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

5.10 [part of 2012/3, edited]

(a) Show the trees that represent the statement

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

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

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

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

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

Chapter 9

Lab four: Machine code

This lab is based on a compiler for a Pascal subset that generates code for the ARM. Following the suggested design from lectures, the compiler translates source programs into operator trees, then uses pattern matching to select instructions for the trees. The lab concentrates mostly on the process of instruction selection, treating the entire front end of the compiler as a black box that produces operator trees.

The compiler has about a dozen modules, listed here in the rough order of passes:

Module	Description
Lexer	Lexical analyser
Parser	Syntax analyser
Tree	Abstract syntax trees
Check	Semantic analyser
Dict	Dictionaries and definitions
Mach	Parameters of target machine
Tgen	Intermediate code generation
Optree	Operator trees
Simp	Simplifier
Jumpopt	Jump optimiser
Share	Common sub-expression elimination
Tran	Instruction selection
Target	Code output
Regs	Register allocation
Main	Main program

The existing instruction selector can handle all the test programs provided with the compiler, but it does not exploit some of the addressing modes supported on the ARM. Specifically, it cannot use the addressing mode that adds two registers, nor the variant of that mode that multiplies one of the

registers by a power of two. For example, the following code puts the value of a[i] in register r0, assuming a is a global array and i is a local at offset 40 in the frame:

```
ldr r1, =_a
ldr r2, [fp, #40]
lsl r2, r2, #2
add r1, r1, r2
ldr r0, [r1]
```

Here the base address of a is put in r1 and the quantity 4*i is computed in r2. These are added with an explicit instruction before fetching the array element. A better code sequence defers the addition to the final ldr instruction:

```
ldr r1, =_a
ldr r2, [fp, #40]
lsl r2, r2, #2
ldr r0, [r1, r2]
```

There is a better code sequence still that performs the scaling also as part of the final instruction:

```
ldr r1, =_a
ldr r2, [fp, #40]
ldr r0, [r1, r2, LSL #2]
```

Your task will be to enhance the instruction selector so that it generates this improved code.

The ARM also allows a shifted register to be used as an operand in other kinds of instruction. Thus, if we want to compute the address of a[i] into a register (as we might if it is used as a reference parameter), then the best code to do it is

```
ldr r1, =_a
ldr r2, [fp, #40]
add r1, r1, r2, LSL #2
```

The last instruction here takes the value in r2, shifts it left two places, adds the result to the value in r1, and puts the sum back in r1. To make the instruction selector generate this code, you will need to add a new kind of operand that includes r2, LSL #2 as an instance.

An optional part of the lab invites you to consider short sequences of instructions that multiply a variable quantity by a known constant more conveniently than using a multiply instruction. For example, we can multiply by 9 in a single instruction, by shifting left three places and adding the original value, and that beats using the mul instruction, which requires 9 to be put in a register first.

For each of these enhancements, we will measure its effect on a set of test programs, some of them small but others larger and more realistic. In each case, the size of the generated code gives a reasonable measure of the effectiveness of instruction selection. As well as improving the code generator by adding new rules and operand types, we can also (in a final optional exercise) go the other way and try deleting rules and addressing modes until we reach a minimal set of instruction selection rules, such that deleting any one of

them breaks the compiler. Doing this will let us assess the effectiveness of standard rules in reducing the size of the generated code. Listings of the files target.mli and tran.ml appear in Appendix E.

9.1 Building and testing the compiler

As usual, you can build the compiler by changing to the lab4 directory and typing make. You will see that several different modules of OCaml code are compiled and linked together; but something new also happens. Initially, the build process changes to a sub-directory tools, and there builds a program called *nodexp*; this program is then used as an accessory to the usual OCaml compiler to expand the special notation we use for operator trees. You will see that each .ml file is compiled with a command such as

```
ocamlc -I ../lib -c -g -pp ../tools/nodexp tran.ml
```

that invokes the compiler *ocamlc*, but tells it to use *nodexp* as a pre-processor. Looking more closely, the file tran.ml contains patterns such as

$$\langle \textsc{Offset}, t_1, \langle \textsc{Const}\ n \rangle \rangle,$$

and these are expanded by *nodexp* into the form

$$\mathit{Node}\,(\textsc{Offset}, [t_1; \mathit{Node}\,(\textsc{Const}\ n, [\,])]),$$

which is expressed directly in terms of the type of optrees,

$$\textbf{type}\ \mathit{optree} = \mathit{Node}\ \textbf{of}\ \mathit{inst} * \mathit{optree}\ \mathit{list},$$

defined in the file optree.mli. As you can see from the example, the compact notation becomes increasingly attractive as the patterns get bigger. The *nodexp* tool is built, naturally enough, using the same techniques as the other compilers in the course, including a lexer and parser built with *ocamllex* and *ocamlyacc*. Invisibly, a wrapper program *ocamlwrap* also takes part in the build process: it helps with interpreting error messages from the OCaml compiler in terms of the original source file.

To test the compiler, there is a collection of test cases in the test directory. Each test case is self-contained and embeds both the expected output of the test and also the expected assembly code, as shown in Figure 9.1. There are three ways of using these test cases:

- By typing `make test0-print` or `make test0-digits` or whatever, you can compile the test case and compare the assembly code with what was expected, without trying to run it. You can say `make -k test0` to do this for all the test cases. Without the -k flag, the process stops after the first test case where the code differs.
- By typing `make test1-print` or `make test1`, you can compile one or all of the test cases and also assemble them and run them on an emulated ARM machine using the program *qemu-arm*.
- By typing `make test2-print` or `make test2`, you can run one or all of the test cases on a genuine Raspberry Pi located somewhere on the internet. Try it once or twice, but if it is too slow, then use the emulator instead, and be grateful that you do not live in an age when something

```
(* print.p *)

begin
  print_num(2); newline()
end.

(*<<
2
>>*)

(*[[
@ picoPascal compiler output
        .global pmain

        .text
pmain:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@   print_num(2); newline()
        mov r0, #2
        bl print_num
        bl newline
        ldmfd fp, {r4-r10, fp, sp, pc}
        .pool

        .section .note.GNU-stack
@ End
]]*)
```

Figure 9.1: *Compiler test case*

no more powerful than a Raspberry Pi would provide the computing service for an entire university.

- If you are not in the Software Lab, then by typing `make test3-print` or `make test3`, you can run test cases by copying the code to a lab machine and assembling and running it there using `qemu-arm`. This is convenient if you are using your own machine but haven't managed to install QEMU yourself, especially if you create an identity key to allow password-less logins.

Here are the commands executed by make test1-print:

```
$ make test1-print
arm-linux-gnueabihf-gcc -marm -march=armv6+fp -c pas0.c -o pas0.o
*** Test print.p
./ppc-arm -O2 test/print.p >b.s
3 instructions
arm-linux-gnueabihf-gcc -marm -march=armv6+fp b.s pas0.o -static -o b.out
qemu-arm ./b.out >b.test
sed -n -e '1,/^(\*<</d' -e '/^>>\*)/q' -e p test/print.p | diff - b.test
*** Passed
```

Stage by stage:

- First, the file pas0.c is compiled with the ARM C compiler [line 2].
- Next, our compiler *ppc-arm* is used to compile the test case, producing a file b.s of ARM assembly language [line 4]. As you can see [line 5], the compiler reports that it generated three instructions for the program; these are the three instructions that implement the statements

  ```
  print_num(2); newline()
  ```

  The boilerplate code for procedure entry and exit is not counted in this total.
- The file b.s is next assembled and linked with the library pas0.o to create an ARM executable b.out [line 6]; it looks as if the ARM C compiler is being used again here, but in reality the command *arm-linux-gnueabihf-gcc* just provides a convenient way to invoke the ARM assembler and linker.
- Next, the ARM emulator *qemu-arm* is used to run the program [line 7].
- Finally, the expected output is extracted from the test case and compared with the actual output [line 8]. If there are no differences, then compiler passes the test.

In this test, the output of the test program was captured and compared with the expected output, but it is not displayed. You can see the output itself by running qemu-arm ./b.out immediately afterwards.[1]

Most of the test cases are very small, intended to test one language feature or one situation in the code generator. There are one or two larger ones, however: specifically, sudoku.p is a program that solves a Sudoku puzzle using Don Knuth's 'method of dancing links', and pprolog.p is an interpreter for the logic programming language Prolog, running a Prolog program that solves Exercise 5.1. These larger test cases are useful for getting an idea how useful each rule in the code generator actually is, by showing how the effect on the instruction count. Before you start work on improving the code generator, you should run these two tests and note the instruction counts.

9.2 More addressing modes

The addresses that appear in load and store instructions are compiled by the function *eval_addr* shown as an outline in Figure 9.2. As it stands, this function implements three rules. The first,

addr → ⟨*LOCAL n*⟩ { [fp, #*n*] },

matches the addresses of local variables and uses the indexed addressing mode that can add a constant to the contents of a register. The second rule,

addr → ⟨*OFFSET*, *reg*, ⟨*CONST n*⟩⟩ { [*reg*, #*n*] },

[1] Depending on how your Linux system is set up, it may be possible to get the same results by running ./b.out directly, instead of qemu-arm ./b.out, giving the appearance that the processor is running the ARM machine code directly. Don't be fooled: what is happening is that the Linux kernel recognises that b.out contains foreign instructions, and behind the scenes invokes *qemu-arm* to interpret it.

```
(* eval_addr – evaluate to form an address for ldr or str *)
let eval_addr =
  function
      ⟨LOCAL n⟩ when fits_offset n →
        frag "[fp, #$]" [number n]
    | ⟨OFFSET, t1, ⟨CONST n⟩⟩ when fits_offset n →
        let v1 = eval_reg t1 anyreg in
        frag "[$, #$]" [v1; number n]
    | t →
        let v1 = eval_reg t anyreg in
        frag "[$]" [v1]
```

Figure 9.2: *Outline of the function eval_addr*

matches in other places where an address includes a constant offset, using the same addressing mode. The address that is written [*reg*, #*n*] in assembly language is represented internally in the compiler by the 'fragment'

frag "[$1, #$2]" [v_1; *number n*]

where v_1 is a fragment corresponding to the register name. As you will see, both these rules are qualified in the implemented code generator with the condition *fits_offset n* to ensure that the offset *n* will fit in the 12-bit offset field of an ldr or str instruction. A third rule,

addr → *reg* { [*reg*] },

matches all other addresses, arranging for the address to be computed into a register, then using the indirect address [*reg*].

We can do better by adding additional rules. The machine provides another addressing mode, written [reg_1, reg_2] in the assembler, that takes the sum of two registers. The rule,

addr → ⟨OFFSET, reg_1, reg_2⟩ { [reg_1, reg_2] },

uses this mode for addresses that are the sum of two others. Better still, we can also add the rule,

addr → ⟨OFFSET, reg_1, ⟨BINOP Lsl, reg_2, ⟨CONST *n*⟩⟩⟩
{ [reg_1, reg_2, LSL #*n*] },

that matches sums where the offset is shifted in order to multiply it by a power of two. You should add implementations of these rules, one at a time, to the compiler, and assess their effectiveness by experimenting with examples such pprolog.p, a program that contains plenty of subscript calculations. Compare the instruction counts for the two big test cases with those you recorded earlier.

To help with debugging your rules, you may like to see the operator trees that they are trying to match. By giving the command,

```
$ ./ppc-arm -d 1 -O2 test/digits.p >b.s
```

you can store in file b.s an assembly language program annotated with copious tracing information. For each procedure, you will see first the sequence

of optrees as generated by the intermediate code generator, then the sequence after the simplifier and jump optimiser have worked on it. Next comes the sequence after common sub-expression elimination, and finally the same sequence interspersed with the generated assembly language. Replacing `-d 1` with `-d 2` additionally prints the state of the register allocator after translation of each optree. Since this tracing information appears as comments in the assembler file, it should still be possible to assemble and run the compiler output.

Adding these new rules exposes a weakness in the CSE pass (module *Share*), in that it continues to treat the result of an *OFFSET* calculation as a worthwhile common sub-expression, despite the fact that it can be computed (and re-computed) at no cost as part of the addressing mode. You can see the effect in a statement such as

```
a[i] := a[i]+1
```

where the address of `a[i]` will be computed into a temp. I've written a couple of rules in the function *inspect* in file `share.ml` that reverse this excessive sharing and commented them out. If you uncomment them, you should see an improvement in the code for statements like this one.

9.3 Shifter operands

Another function, *eval_rand*, compiles the operands that appear in arithmetic instructions. At present, it embodies two rules:

rand → ⟨*CONST n*⟩ { #*n* },

that allows a constant to appear as an operand, and the catch-all rule,

rand → *reg* { *reg* },

that evaluates the operand into a register. These rules fail to exploit the quirky operand format of the ARM where a register value is shifted to form the value of the operand. We can write a new rule,

rand → ⟨*BINOP Lsl*, *reg*, ⟨*CONST n*⟩⟩ { *reg*, `LSL #`*n* },

and implement it by extending *eval_rand* so that matches the pattern on the right-hand side of this rule, and returns a fragment

frag "`$, LSL #$`" [v_1; *number n*],

where v_1 is a register and *n* is the constant shift amount. Note that the operand syntax does not contain any square brackets, so that a correct instruction to add `r3` with `4*r4` into `r2` would be

```
add r2, r3, r4, LSL #2
```

You should implement this form of operand, and again measure how much improvement results in the code for examples like `pprolog.p`. It's surprisingly big, because it is very common in that program to compute addresses of array elements in a way that can use this rule.

9.4 Multiplying by a constant (optional)

In addition to making better use of the ARM's addressing modes and operand forms, we can look for special cases of more general operations that can be implemented more efficiently. One source of such operations is multiplication by a constant; the ARM's multiply instruction requires both operands in registers, so multiplying register r0 by a constant n requires two instructions:

```
mov r1, #n
mul r0, r0, r1
```

(Additionally, the mul instruction may, depending on the model of ARM processor, take several cycles to complete.)

As an alternative, it is possible to exploit shifter operands to multiply by some small constants in a single instruction. It's simple enough to multiply by powers of 2 with lsl instructions (which are really mov instructions that use a shifter operand). We can also multiply r0 by 3 with an instruction that adds twice r0 to itself:

```
add r0, r0, r0, LSL #1
```

Again, we can multiply by 15 with a reverse-subtract instruction that takes r0 away from 16 times itself:

```
rsb r0, r0, r0, LSL #4
```

And we can add 10 times r0 to r1 by first computing 5 times r0 in r2, then multiplying by 2 as we add:

```
add r2, r0, r0, LSL #2
add r1, r1, r2, LSL #1
```

A comprehensive catalogue of such tricks would become tedious, and we had better leave it to the ARM experts to map the exact boundary where a simple, honest multiply instruction becomes cheaper. But we can at least try extending the instruction selector with some simple cases. We will be helped in this by the fact that the simplifier *Simp*, which comes earlier in the compiler pipeline, has already rearranged multiplications so that any constant operand appears second (or has performed the multiplication already if both operands are constants). The simplifier also replaces multiplication by powers of 2 with shifts, so we don't have to worry about them.

The test program sudoku.p solves an ordinary Sudoku puzzle, so it naturally contains arrays with index calculations that involve multiplying by constants such as 3, 9, 10, 36 and 324. You can find these by searching for mul instructions in the ARM code that is embedded in the test case. You may like to implement special rules in the code generator for some of these constants, and see how many `mul` instructions you can eliminate.

A small detail: after computing a value into a register v_1, it's tempting to multiply it by 3 (say) by writing

gen_reg "add $0, $1, $2, LSL #1" $r\,[v_1; v_1]$

but this will lead to an inconsistency in the state of the register allocator, because the fragment v_1 denoting the register will be freed twice, with the

result that its reference count will go negative, breaking the register allocator. Instead, you must write

gen_reg "add $0, $1, $1, LSL #1" *r* [$v_1$]

This frees the register only once, but substitutes its name for "$1" in two places in the instruction.

9.5 Going the other way (optional)

As well as adding more rules to the instruction selector, we could try deleting rules one at a time until the remaining set is minimal but still adequate. You can experiment with the effect of deleting rules by commenting them out, surrounding them with (∗ and ∗) in the compiler source.

For example, the rule for *OFFSET* in *eval_addr* can be deleted, at the expense of computing more addresses into registers, and even the rule for *LOCAL n* can go and still leave an adequate set. Are there any other rules in the instruction selector that can be deleted? How big, perversely, can you make the code for pprolog.p and still have it work? Are there any rules in the instruction selector that are not used for any of the supplied test cases? If so, can you write new test cases where these rules do make a difference?

Chapter 10

Revision problems

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* → *code* that returns the code for an assignment statement. Do not attempt any optimisation at this stage.

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

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

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

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

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

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

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

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

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

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

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

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

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

procedure P(i: integer);

    var x: integer;

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

    procedure R();
    begin
        Q()
    end;

begin (* P *)
```

```
    R()
end

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

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

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

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

(c) Similarly, give code for procedure R.

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

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

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

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

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

```
    return sum(n, power)
end;

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

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

Appendix A

OCaml basics

The programs we will work with in this course are written in the functional programming language Objective CAML, which supports functional programming with polymorphic typing and applicative-order evaluation. It has a module system that permits programs to be built from a collection of largely independent components, with the implementation details of each module hidden from the other modules that use it. Objective CAML adds to the purely functional core of the language certain concepts from imperative programming, including assignable variables, control structures and an exception mechanism. These features make it easier to use the ideas of functional programming whilst avoiding the cluttered style it sometimes leads to. For example, if a value is needed in a large part of a program but seldom changes, the imperative features of Objective CAML allow us to assign the value to a global variable, rather than passing it as an argument in every function call.

In this note, I shall introduce just enough of the language to allow understanding of most of the programs in the course. The name "OCaml" – short for Objective Caml – refers to the fact that the language supports a form of object-oriented programming. Although objects can be used fruitfully in writing compilers, we shall not need to use the object-oriented features of the language. Objective CAML also has a sophisticated module system that allows nested modules with parameters, and we shall need to use only the simplest subset, where the text of each module is kept in a separate file and there is no nesting. Objective CAML has many other unusual features, including functions with optional and keyword parameters; I have avoided these features, in the hope that it will make the programs in this course more approachable, and easier to translate into other languages. What you will find in this note is a concise summary of just that part of the language that we shall use in the rest of this course.

This note is intended for readers who already have some familiarity with functional programming, perhaps in some other language such as HASKELL, or another dialect of ML, and need a brief introduction to the syntax of Objective CAML and those of its features that are not part of the purely functional subset. The main way of programming in Objective CAML, as in any language that supports a functional style of programming, is to define functions recursively. This style works well in compilers, because the abstract syntax of a programming language is very naturally modelled by a family of recursive data types, and many tasks that the compiler must carry out, such

as checking that a program obeys the rules of the programming language or translating it into machine code, can naturally be expressed as recursive functions over these data type; in this way, the syntax rules of the programming language guide the construction of the compiler. We begin with some very brief examples of familiar functions defined in Objective CAML.

As well as supporting functional programming, Objective CAML has a number of other features that we shall use: in particular, there is a module system that allows programs to be split cleanly into independent pieces, with the source code of each module kept in a separate file and compiled independently of other modules. The facilities for doing this are explained in Section A.4.

This note ends with a brief explanation of how the printed form of Objective CAML programs that appears in the notes for the course is related to the plain ASCII form that is accepted by the Objective CAML compiler.

A.1 Defining functions

Functions can be defined by pattern matching, as in the following definition of the function *reverse* on lists:

```
let rec reverse =
  function
      [ ] → [ ]
    | x :: xs → reverse xs @ [x]
```

The keywords **let rec** introduce a recursive function definition; *reverse* is defined by two patterns, one matching the empty list [ ], and the other matching a non-empty list $x :: xs$ that has head x and tail xs. The value of *reverse* [ ] is [ ], and the value of *reverse* $(x :: xs)$ is formed by recursively reversing the list xs, forming the singleton list x, and joining the two results with the operator @, denoting concatenation of lists. The Objective CAML programs in this course have been formatted for ease of reading, using different styles of type and special symbols like →; when the programs are entered into a computer, an ASCII form of the language is used, and the arrow symbol is typed as ->, for example. Section A.10 on page 131 explains the correspondence between the two forms.

Objective CAML has a strong, polymorphic type system which does not require types to be specified by the programmer: from the definition of *reverse* given here, the Objective CAML compiler can deduce that *reverse* has the type

$$\alpha\ list \rightarrow \alpha\ list$$

Here, α is a type variable that can stand for any type, and the type constructor *list* is written (as always in Objective CAML) after its argument. So this type expresses the fact that *reverse* can accept any list as argument, and delivers as its result another list of the same type.

Objective CAML has a number of basic types:

- *int*, integers of 31 bits.
- *char*, characters, written between single quotes like this: 'a', 'b', 'c'.
- *string*, strings of characters, written in double quotes like this:

  "This is a string".

Note that in Objective CAML, strings are not the same as lists of characters: they have a more compact representation.

A.2 Type definitions

Objective CAML allows the programmer to define new data types, including recursive types such as trees. We shall use tree-like types a lot, because they provide a way of modelling the syntactic structure of the programs in a compiler: so we choose as an example a type that could be used to model arithmetic expressions.

```
type expr =
    Constant of int
  | Variable of string
  | Binop of op * expr * expr
and op = Plus | Minus | Times | Divide
```

This definition introduces two types *expr* and *op*. The two definitions are joined by the keyword **and**, so that each definition may refer to the other one. The type *op* simply consists of the four values *Plus*, *Minus*, *Times*, and *Divide*, but the type *expr* has a more elaborate recursive structure. An *expr* may be simply a constant *Constant n* for some integer *n*, or a variable *Variable x* named by the string *x*, but it can also be a compound expression *Binop* (*w*, *a*, *b*), where *w* is an operator and *a* and *b* are other expressions. For example, the expression

```
x * (y + 1)
```

would be represented as the following value of type *expr*:

```
Binop (Times, Variable "x",
       Binop (Plus, Variable "y", Constant 1)).
```

Constructors like *Binop* and *Plus* must always begin with an upper-case letter, as must the names of modules (see later), but other names used in an Objective CAML program must begin with a lower-case letter.

We can use recursion to define functions that work on recursive types. Here is a function *eval* that gives the value of an arithmetic expression (at least, if it doesn't contain any variables).

```
let do_binop w a b =
  match w with
      Plus → a + b
    | Minus → a − b
    | Times → a * b
    | Divide → a/b

let rec eval =
  function
      Constant n → n
    | Binop (w, e1, e2) →
        do_binop w (eval e1) (eval e2)
    | Variable x →
        failwith "sorry, I don't do variables"
```

These definitions illustrate several new language features. The function *eval* is defined by pattern-matching on its argument, an expression tree. An expression that contains a binary operator is evaluated by recursively evaluating its two operands, then applying the operator to the results. The curried function *do_binop* takes three arguments: the operator, and the values of its two operands. It is defined by matching the operator against a sequence of patterns in a **match** expression, each pattern matching a particular arithmetic operation. These **match** expressions are rather like the **case** statements of other programming languages.

In more complex examples, we might need to define several recursive functions, each able to call the others. Normally, each function must be defined before it is used, so these *mutually recursive* functions present a problem. The solution provided by Objective CAML is to join the function definitions with the keyword **and**, so that they are treated as a unit by the compiler. To illustrate the syntax, here are two functions f and g that call each other:

let rec $f\ n =$
  if $n = 0$ **then** 1 **else** $g\ n$
and $g\ n =$
  if $n = 0$ **then** 0 **else** $g\ (n-1) + f\ (n-1)$

The function f is defined a little like the Fibonacci function, except that it satisfies the recurrence

$$f\ n = f\ 0 + f\ 1 + \ldots + f\ (n-1) \qquad (n > 0),$$

and the sum on the right-hand side is computed by $g\ n$.[1]

We've seen two sorts of pattern matching: one introduced by the keyword **function**, and the other introduced by the keyword **match**. In fact, these are equivalent, in the sense that a definition

let rec f = **function** ...

can always be replaced by the equivalent form

let rec $f\ x$ = **match** x **with** ...

We'll continue to use both forms, choosing whichever is the more convenient in each instance.

The treatment of expressions of the form *Variable x* illustrates another language feature: exceptions. Evaluating the expression *failwith s*, where *s* is a string, does not result in any value; instead, the outcome is an exception, which can be caught either by a surrounding program or by the Objective CAML system itself. In the latter case, execution of the program is abandoned, and Objective CAML prints an error report. We shall use *failwith* extensively to replace parts of our compilers that we have not yet implemented properly.

A.3 Tuples and records

Objective CAML provides a family of types for n-tuples: for example, (1, 2, "3") is an expression with type *int* $*$ *int* $*$ *string*. We have already seen such types used for the arguments of constructors in user-defined types. Standard

[1] In fact $f\ n = 2^{n-1}$ for $n \geq 1$, as may be proved by induction.

functions *fst* and *snd* are provided for extracting the first component x and the second component y of an ordered pair (x, y): matching.

$$fst : \alpha * \beta \rightarrow \alpha$$
$$snd : \alpha * \beta \rightarrow \beta$$

These functions can be defined by pattern matching like this:

let *fst* $(x, y) = x$
let *snd* $(x, y) = y$

Components of bigger n-tuples can also be extracted by pattern matching; for example, here is a function the extracts the second component of a 3-tuple:

let $second_3\ (x, y, z) = y$

As an alternative to n-tuples, Objective CAML also provides record types. A type definition like

type *def* = { *d_key* : *string*; *d_value* : *int* }

defines a new record type *def* with selectors *d_key* and *d_value*. Values of this record type can be constructed by expressions like

{ *d_key* = "foo"; *d_value* = 3 }

and if d is such a value, its components can be extracted as $d.d\_key$ and $d.d\_value$. For our purposes, these record types provide nothing that cannot be achieved with tuples, but they sometimes serve to make our programs a little clearer.

In addition to type *definitions* that create new record types or recursive algebraic types, Objective CAML also allows type *abbreviations*. For example, the declaration

type *couple* = *string* * *int*

introduces *couple* as an alternative name for the type *string* * *int*.

A.4 Modules

Objective CAML has a sophisticated module system that allows nested modules and modules that take other modules as parameters. However, we shall not need this sophistication in the compilers we build, and can get by instead with a simple subset of Objective CAML's features that supports separate compilation of modules, much as in Modula-2. Each module M consists of an interface file M.mli that advertises certain types and functions, and an implementation file M.ml that contains the implementations of those types and functions. The information in the interface file is shared between the implementation of the module and its users, but the implementation is effectively hidden from the users, thereby enforcing separation of concerns, and allowing the implementation to change without requiring re-compilation of all the user modules.

The interface file contains a declaration giving the type of each function that is provided by the module. For example, a module that contained the

function *reverse* from the preceding section might contain this declaration in its interface:

```
val reverse : α list → α list
```

This declaration is a promise that the implementation of the module will define a function *reverse* with the type shown.

The interface file can also contain the definitions of types that are shared between the implementation and its users, and declarations for types that are implemented by the module but have hidden representations. As an example, a module that implements symbol tables might provide a type *memory*, together with operations for inserting symbols into a table and for looking up a given symbol. This module might have the following declarations in its interface file memory.mli:

```
(* memory.mli *)

type memory

val empty : memory

val insert : string → int → memory → memory

val lookup : string → memory → int
```

These declarations promise that a type *memory* will be implemented, and advertise a constant *empty* and two operations *insert* and *lookup* that will be provided, without revealing the representation that will be chosen in the implementation.

A simple implementation might represent the memory state by a list of records, each with the type *def* we defined earlier. In that case, the implementation file memory.ml would contain the following definitions:

```
(* memory.ml *)

type def = { d_key : string; d_value : int }

type memory = def list

let empty = [ ]

let insert x v t = { d_key = x; d_value = v } :: t

let rec lookup x =
  function
      [ ] → raise Not_found
    | d :: t →
        if d.d_key = x then d.d_value else lookup x t
```

This gives the actual type that is chosen to represent symbol tables, and versions of *insert* and *lookup* that work for this choice of type. The choice is hidden from other modules that use the symbol table, making it possible to replace the data structure by a more efficient one without needing to change or even recompile the other modules. Like the *failwith* function we saw earlier, the expression *raise Not_found* raises an exception,

There are two ways in which expressions in one module can name the exported features of another module. One is to use a *qualified name* such as *Memory.insert* that contains the name of the module and the name of the

exported identifier. The module name appears with an initial capital, and it is this that removes the ambiguity between qualified names $M.x$ and the notation $r.x$ for selecting field x from a record r. The other way is for the client module to contain an "**open**" directive such as

open *Memory*

After such a directive, all the exported identifiers of *Memory* are available directly, without further need for qualification.

If types are defined in the implementation file, they may be made visible to client modules in several ways, with different effects:

- The type may not be mentioned in the interface file at all. In this case, the type is entirely private to the implementation. Functions that are exported in the interface file may not accept arguments or produce results of the type.
- The type may be declared in the interface without any details of its structure, like this:

  type *memory*

  type (α, β) *mapping*

  (As the second example shows, such types can have parameters too.) In this case, functions that are exported by the module may accept parameters and deliver results of that type, but clients may only create and manipulate values of the type by using the exported functions. This style of export is appropriate for modules that implement an abstract data type, because it allows the hidden definition of the type to be changed without the need to change or even recompile client modules.
- The definition from the implementation file may be repeated in full in the interface file. In this case, client modules have full freedom to create values of the type, and to manipulate them by pattern matching. This style is appropriate for modules that define a common data type used throughout a program, such as the abstract syntax trees in our compilers.

A.5 Exceptions

Objective CAML has two groups of features that take it outside the realm of pure functional programming: one is the exception mechanism, and the other is the facility for assignable variables. We have already met the expression

raise Not_found

that raises the exception called *Not_found*, and the function *failwith*, which also raises an exception. We shall use *failwith* only to indicate that part of the compiler is missing or broken, but in general, exceptions provide a useful way of dealing with computations that fail in ways that can be predicted.

For example, the *lookup* function of the preceding section raises the exception *Not_found* if the identifier we are looking for is not present in the symbol table we are searching. Evaluating the expression *raise Not_found* does not

result in a value in the ordinary way, but instead propagates the exceptional value *Not_found*. The beauty of exceptions (and also their weakness) is that these exception values implicitly propagate through the program until they reach a context where an exception handler has been created, or until they reach the outside world and are dealt with by terminating the whole program.

Complementary to *raise* is the Objective CAML facility for handling exceptions raised during the evaluation of an expression: for example, one might use *lookup* in the following expression, which returns the value associated with "Mike" in the table *age*, or 21 if there is no such value:

try *lookup* "Mike" *age* **with** *Not_found* → 21

Though we shall make little use of them, Objective CAML also provides exceptions with arguments; thus *failwith s* is equivalent to *raise* (*Failure s*), where *Failure* is a pre-defined exception constructor that takes a string argument. When this exception is raised and not caught inside a program, the run-time system of Objective CAML is able to catch the exception, extract its string argument, and print it in an error message. This makes *failwith* a convenient way to temporarily plug gaps in a program where it is unfinished; if the gap is ever reached, we can arrange that the Objective CAML system will give a recognisable message before ending execution. Although it is possible to catch the *Failure* exception with a **try** block, we shall never do so.

A.6 References and mutable fields

In addition to the purely functional data types we've seen so far, Objective CAML provides *reference types*, whose values can be changed. These values or *cells* can be used to simulate the variables of a language like PASCAL, allowing an imperative style of programming where that is more convenient than a purely functional style.

If x is any value, then evaluating the expression *ref* x creates a new cell r (different from all the others in use) and fills it with the initial value x. At any time, we can evaluate the expression $!r$ to retrieve the current value stored in cell r, and we can evaluate the expression $r := y$ to update the contents of cell r so that it contains the value y.

We shall use reference types for many purposes in our compilers:

- They allow us to write algorithms in an imperative style when that is convenient. For example, we shall often write programs that simulate the action of an abstract machine, and it is more natural to write these programs imperatively, so that they destructively update the state of the machine in the same way that hardware does.
- They allow us to build modules that have an internal state, with operations for changing and inspecting that state. This can make our compilers simpler than they would be if all the data had to be passed around in a functional style. For example, we shall build a module whose internal state is the sequence of machine instructions that have been generated so far by the compiler, and provide an operation that adds another instruction to the sequence. This is both more efficient and more manageable than a scheme that passes around explicit lists of instructions.

- They allow us to build data structures with modifiable components. This is how we shall allow the semantic analysis phase to annotate the tree with information that is needed by the code generator.

As an example of a simple use of references, here is a simple division algorithm written in an imperative style. The **if** ... **then** .. **else** expressions we have already met can be used as a conditional statement for imperative programming, but Objective CAML also provides sequencing (;), grouping with **begin** and **end**, and **while** loops:

```
let divide a b =
  let q, r = ref 0, ref a in
  begin
    while !r ≥ b do
      r := !r − b; q := !q + 1
    done;
    !q
  end
```

The keywords **begin** and **end** are simply a more familiar alternative to grouping with parentheses (...). Also illustrated here is the notation **let** *lhs* = *rhs* **in** *exp* for introducing a local definition; in this case, it used to declare two local reference cells, but it can also be used to define local variables or functions.

The result of a function written in imperative style is the last expression in its body, here the final value !*q* of the modifiable variable *q*. As you can see, the need to write the dereferencing operator ! explicitly at every variable reference is a powerful force in favour of a more functional style, when that is convenient:

```
let divide a b =
  let rec div1 q r =
    if r < b then q else div1 (q + 1) (r − b) in
  div1 0 a
```

Again, we have used **let**, this time to introduce a local function $divide_1$ that is *tail-recursive*, meaning that the only recursive call is the very last action in the function's body. A decent implementation of Objective CAML will execute such a tail-recursive function with the same efficiency as the imperative version. Nevertheless, the imperative style will still be useful in some contexts, where a purely functional program would be more complicated.

As an example of a module with internal state, here is the definition of a module that maintains a single table of names and values:

```
(* onemem.mli *)

val add : string → int → unit

val find : string → int
```

The idea is that *add* inserts a new string into the table, returning the unit value ( ), that is to say, no value at all. The *find* function looks up a name and returns the corresponding value, or raises the exception *Not_found* if no such value exists.

We could implement this module in terms of the *memory* module we defined before, using a global reference cell to keep hold of the current table:

```
(* onemem.ml *)
```

```
let table = ref Memory.empty

let add x v = (table := Memory.insert x v !table)

let find x = Memory.lookup x !table
```

One way of making data structures that can be modified imperatively is to incorporate references into them: for example, we could make records that have a reference as one of their fields. A more attractive option uses an added feature of Objective CAML records: they can have mutable fields that can be changed destructively. To give an example of how this works, here is a version of the *expr* type we defined earlier, extended so that each expression can be annotated with its value:

```
type expr =
  { e_guts : expr_guts; mutable e_value : int }
and expr_guts =
    Binop of op * expr * expr
  | Constant of int
```

If we have analysed an expression e by looking at its field $e.e\_guts$ and found it to have value v, then this value can be stored for future reference by the assignment $e.e\_value \leftarrow v$. (Note the use of $\leftarrow$ in place of :=). Here, then, is a function that evaluates an expression, *both* returning the value as its result *and* storing the value of each sub-expression as an annotation:

```
let rec eval e =
  let v = match e.e_guts with
          Binop (w, e1, e2) → do_binop w (eval e1) (eval e2)
        | Constant n → n in
  e.e_value ← v; v
```

After calling *eval e*, we have its value as the result that is returned, but can also refer to it as $e.e\_value$; and we can refer to the value of any sub-expression e_1 as $e_1.e\_value$. The only fly in the ointment is that even an *un*evaluated expression has an *e_value* field, but we can usually find a dummy value (perhaps 0 in this case) to use for it.

In fact, we can view references as a special case of mutable fields by imagining that the type α *ref* is defined by

```
type α ref = { mutable contents : α }
```

with $x := t$ standing for $x.contents \leftarrow t$ and $!x$ standing for $x.contents$.

A.7 Library functions

Objective CAML comes with a large library of standard modules that are quite well suited to writing compilers, perhaps because the library was partly developed to support the implementation of the Objective CAML system itself. In this section, I'll give a brief overview of the functions from the standard library that we'll use in this course.

Two important abstract data types used in most compilers are *mappings* and *associative tables*, both representing functions from arbitrary keys to arbitrary values; often the keys will be identifiers in the program being compiled, and the values will be an attribute computed by the compiler, such as

the types of the identifiers or their run-time addresses. The type of mappings provides a purely applicative interface, whilst the type of associative tables provides an imperative interface, where the mapping is built up by destructive modification of a single table. This makes greater efficiency possible, because the imperative interface can be implemented as a hash table with essentially constant-time access, whereas the applicative interface is implemented by a search tree that gives access in $O(\log N)$ time, where N is the number of keys in the mapping.

The last module in our library defines a family of functions – analogous to *printf* in C – that provide formatted output. Again, there is an implementation in the standard library of Objective CAML, but the version I shall present is extensible to handle the printing of new types of data.

A.7.1 Lists

The module *List* provides many of the standard operations on lists. Here are the basic ones:

val *length* : α *list* → *int*
val (@) : α *list* → α *list* → α *list*
val *hd* : α *list* → α
val *tl* : α *list* → α *list*
val *nth* : α *list* → *int* → α
val *rev* : α *list* → α *list*

If xs is the list $[x_0; x_1; \ldots; x_{n-1}]$ and ys is the list $[y_0; y_1; \ldots; y_{m-1}]$, then these functions deliver results as follows:

$length\ xs = n$

$xs\ @\ ys = [x_0; \ldots; x_{n-1}; y_0; \ldots; y_{m-1}]$

$hd\ xs = x_0$

$tl\ xs = [x_1; \ldots; x_{n-1}]$

$nth\ xs\ i = x_i$

$rev\ xs = [x_{n-1}; \ldots; x_1; x_0]$

The function *concat* takes a list of lists and flattens it into a single list:

val *concat* : (α *list*) *list* → α *list*

If xss is the list of lists $[xs_0; xs_1; \ldots; xs_{n-1}]$, then

$$concat\ xss = xs_0\ @\ xs_1\ @\ \cdots\ @\ xs_{n-1}.$$

Much of the power of functional programming comes from the standard higher-order functions that can be defined, reducing the need for recursive definitions in programs that use them. We shall use the following:

val *map* : (α → β) → α *list* → β *list*
val *iter* : (α → *unit*) → α *list* → *unit*
val *fold_left* : (β → α → β) → β → α *list* → β
val *fold_right* : (α → β → β) → α *list* → β → β
val map_2 : (α → β → γ) → α *list* → β *list* → γ *list*

Taking xs to be the list $[x_0; x_1; \ldots; x_{n-1}]$ as before, we find that

$$map\ f\ xs = [f\ x_0; f\ x_1; \ldots; f\ x_{n-1}].$$

Because Objective CAML is not a purely functional language, it sometimes matters in what order the terms $f\ x_0$, $f\ x_1$, ..., $f x_{n-1}$ are evaluated: for example, the function f might have a side-effect that allows one evaluation of f to affect the action of future evaluations. The function *map* is defined so that these terms are evaluated in left-to-right order, so that any side-effects of evaluating $f\ x_0$ happen before those of $f\ x_1$, and so on. Here is a definition of *map* that has this property:

```
let rec map f =
  function
      [ ] → [ ]
    | x :: xs → let y = f x in y :: map f xs
```

It is the **let** expression that ensures that the value of $f\ x$ is determined before the recursive call *map f xs* is evaluated. The left-to-right evaluation is useful for tasks like defining the function *totals* that returns a list of running totals, like this:

```
totals [1; 2; 3; 4; 5] = [1; 3; 6; 10; 15]
```

We can use a reference cell to keep the running total, and map a function over the list that both increases the total and returns its current value:

```
let totals xs =
  let t = ref 0 in
  let f x = (t := !t + x; !t) in
  map f xs
```

In this simple example, it might be easier and clearer to define *totals* directly by recursion:

```
let totals xs =
  let rec tot1 t =
    function
        [ ] → [ ]
      | y :: ys → let t' = t + y in t' :: tot1 t' ys in
  tot1 0 xs
```

In more complicated situations, however, it is good to have the additional flexibility that a left-to-right *map* provides.

The function *iter* is similar to *map*, in that evaluating *iter f xs* entails evaluating $f\ x_0$, $f\ x_1$, ..., $f x_{n-1}$ in left-to-right order, but these evaluations are done purely for the sake of their side-effects, because the results are discarded, and the value returned by *iter* is the *unit* value ( ).

The two functions *fold_left* and *fold_right* combine all the elements of a list using a binary operation:

$$fold\_left\ f\ a\ xs = f\ (\ldots(f\ (f\ a\ x_0)\ x_1)\ \ldots)\ x_{n-1}$$
$$fold\_right\ f\ xs\ a = f\ x_0\ (f\ x_1\ (\ldots(f\ x_{n-1}\ a)\ \ldots))$$

Thus *fold_left* combines the elements of the list starting from the left, and *fold_right* starts from the right.[2] Here are the recursive definitions:

[2] The function *fold_right* takes its arguments in a different order from the function *foldr* of HASKELL, so that *fold_right f xs a = foldr f a xs*.

```
let rec fold_left f a ys =
  match ys with
      [ ] → a
    | x :: xs → fold_left f (f a x) xs

let rec fold_right f ys a =
  match ys with
      [ ] → a
    | x :: xs → f x (fold_right f xs a)
```

If the function f is associative, so that $f\ (f\ x\ y)\ z = f\ x\ (f\ y\ z)$ for all x, y, z, and a is a left and right identity element for f, so that $f\ a\ x = x = f\ x\ a$, then there is no difference between the results returned by *fold_left* and *fold_right*. This is true, for example, if $f\ x\ y = x + y$ and $a = 0$. In this case, both *fold_left f a xs* and *fold_right f xs a* compute the sum of the list of integers *xs*. Using the Objective CAML notation (+) for the addition operator on integers, considered as a function of type $int \to int \to int$, we can therefore define a function $sum : int\ list \to int$ by

```
let sum xs = fold_left (+) 0 xs
```

Where the two give the same result, we prefer *fold_left* over *fold_right* because its tail-recursive definition leads to a program whose space usage is constant, rather than linear in the length of the argument list.

Actually, the programming style adopted in many parts of our compilers doesn't sit well with the order of arguments supported by *fold_left*, and we shall make fairly frequent use of a function *accum* such that

$$accum\ f\ xs\ a = fold\_left\ (\textbf{function}\ b\ x \to f\ x\ b)\ a\ xs.$$

The function *accum* therefore has the type

$$\textbf{val}\ accum : (\alpha \to \beta \to \beta) \to \alpha\ list \to \beta \to \beta$$

with the arguments to both f and $accum\ f$ swapped from their positions in *fold_left*. This is purely a matter of convenience, and it is easy to derive a recursive definition of *accum* that is just as efficient as *fold_left*.

There's also a function map_2, similar to Haskell's *zipwith*, and a function $iter_2$, defined so that

$$map_2\ f\ xs\ ys = [f\ x_0\ y_0; f\ x_1\ y_1; \ldots; f x_{n-1}\ y_{n-1}],$$

$$iter_2\ f\ xs\ ys = f\ x_0\ y_0; f\ x_1\ y_1; \ldots; f x_{n-1}\ y_{n-1}.$$

A number of deal with lists of ordered pairs of type $(\alpha * \beta)\ list$:

$$\textbf{val}\ combine : \alpha\ list \to \beta\ list \to (\alpha * \beta)\ list$$
$$\textbf{val}\ assoc : \alpha \to (\alpha * \beta)\ list \to \beta$$
$$\textbf{val}\ remove\_assoc : \alpha \to (\alpha * \beta)\ list \to (\alpha * \beta)\ list$$

The function *combine* takes two lists (which must be of the same length), and pairs up their elements, returning a list of pairs: $combine\ xs\ ys = [(x_0, y_0); (x_1, y_1); \ldots; (x_{n-1}, y_{n-1})]$. Thus *combine xs ys* is a list of pairs such that $map\ fst\ (combine\ xs\ ys) = xs$ and $map\ snd\ (combine\ xs\ ys) = ys$.

The function *assoc* is useful when a mapping is represented by a list of (argument, value) pairs. The arguments of *assoc* are an argument for the mapping, together with the list of pairs that represents the mapping itself,

and the result is the corresponding value of the mapping; *assoc* raises the exception *Not_found* if no such value exists. If the list *ps* contains more than one pair (u, v) with $u = x$, then *assoc x ps* returns the value of v from the first such pair. Thus *assoc* may be defined as follows:

```
let rec assoc x =
  function
      [ ] → raise Not_found
    | (u, v) :: ps → if u = x then v else assoc x ps
```

There is another function *remove_assoc* that removes the first pair (u, v), if any, with a specified component $u = x$:

```
let rec remove_assoc x =
  function
      [ ] → [ ]
    | (u, v) :: ps →
        if u = x then ps else (u, v) :: remove_assoc x ps
```

If the list is taken to represent a mapping, then this function removes x from its domain.

All the functions listed above are part of the module *List* in the standard library; they are used so often that we will usually open this module so that they can be used without qualification.

A.7.2 Optional values

Lists may contain any number of elements, from zero upwards. Sometimes it's convenient to use a more restrictive type that allows either zero or one element of another type; this is the purpose of the type constructor α *option*, defined as follows:

```
type α option = Some of α | None
```

For example, a language might have a `return` statement that can be followed either by an expression giving the value returned by a subroutine, or by nothing if the subroutine returns no result. In this case, we could represent `return` statements by in the abstract syntax tree like this:

```
type stmt = ...
  | Return of expr option
  | ...
```

Then we could use *return* (*Some e*) for the `return` statement that contains expression *e*, and *return None* for the return statement containing no expression.

A.7.3 Arrays and strings

In addition to lists, which can be of unpredictable length, are purely functional in their behaviour, but require linear time to access an arbitrary element, Objective CAML also provides the alternative data structures of *arrays* and *strings*. An array has a fixed length and allows constant-time access to its elements, which are identified by numeric indices. There is an operation to retrieve an element given its index and an operation to destructively update the array so that a given index is associated with a new value. Arrays thus share with reference cells a non-functional character that depends on

side-effects. Strings in Objective CAML are similar to arrays, but they are immutable, and their elements are always characters; strings are stored in a particularly efficient way so that the memory space occupied by a string is one byte per character, plus a small constant overhead.

An array with elements of type α has the built-in type α *array*. The module *Array* provides the following operations on arrays:

val *create* : *int* → α → α *array*
val *length* : α → *int*

An array of length n is created by *Array.create n x*; all the elements of this array are initialised to x. The i'th element of array a is written $a.(i)$ for $i = 0, 1, \ldots, n-1$, and the i'th element is set to y by the operation $a.(i) \leftarrow y$.[3] The length of a is given by *Array.length a*.

Strings belong to the built-in type *string*, and are created as the values of string constants apearing in Objective CAML programs. Among other functions, the *String* module provides the operations

val *get* : *string* → *int* → *char*
val *length* : *string* → *int*
val (^) : *string* → *string* → *string*
val *sub* : *string* → *int* → *int* → *string*

The i'th character of a string s is written $s.[i]$ for $i = 0, 1, \ldots, n-1$: this notation is an abbreviation for *String.get s i*. The length of s is given by *String.length s*, and two strings s_1 and s_2 of lengths n_1 and n_2 can be concatenated to form a single new string of length $n_1 + n_2$ by writing $s_1 \hat{} s_2$. The sub-string of a string s starting at character $s.[i]$ and continuing to character $s.[i + j - 1]$ (and therefore of length j) is written *String.sub s i j*.

Strings are immutable because there are no operations that can change the contents of a string once it is formed, so they can be safely shared by different parts of a program, and questions about the effect of modifying a string that orginated from a quoted constant do not arise. There is also a mutable variant of the *string* type called *bytes*, particularly useful for building up a sequence of characters incrementally before converting it to an immutable string using the function *String.of_bytes*.

A.8 Mappings and hash tables

The Objective CAML library provides a useful pair of modules for representing finite mappings – one with a functional interface and another with an imperative one based on destructive updates.

The functional interface is provided by the library module *Map*, which uses balanced binary trees internally to represent finite mappings – good enough that we will not need to worry about its efficiency. The module is parametric in the type of keys, and to use it we will need to cast a magic spell: after the declaration

module IdMap = Map.Make(*struct*
  type *t = ident*

[3] The notations $a.(i)$ and $a.(i) \leftarrow y$ are abbreviations for *Array.get a i* and *Array.set a i y*, and use operations provided by the *Array* module.

let *compare* = *compare*
end),

the name *IdMap* refers to a module of mappings with identifiers (*ident*) as keys; it provides a polymorphic type β *IdMap.t* of mappings from identifiers to values of type β, with operations

(∗ *empty* – empty mapping ∗)
val *empty* : β *IdMap.t*

(∗ *find* – look up identifier, or raise *Not_found* ∗)
val *find* : *ident* → β *IdMap.t* → β

(∗ *add* – add a new key and value, returning a new mapping ∗)
val *add* : *ident* → β → β *IdMap.t* → β *IdMap.t*

Other functions are provided, but these are enough to get started. Note that adding a new (*key*, *value*) pair to a mapping results in a new, independent, mapping without modifying the old one; the two mappings will surely share some of the storage used in their internal representation, but that fact cannot be detected externally.

In contrast, the module *Hashtbl* provides mappings with destructive update. It provides a complicated interface of which we will need only a simple part: there is a type (α, β) *Hashtbl.t* of hash tables with keys α and values β, providing operations,

(∗ *create* – make a hash table with *n* buckets ∗)
val *create* : *int* → (α, β) *Hashtbl.t*

(∗ *find* – look up a key. or raise *Not_found* ∗)
val *find* : (α, β) *Hashtbl.t* → α → β

(∗ *add* – add a new key and value ∗)
val *add* : (α, β) *Hashtbl.t* → α → β → *unit*

(∗ *clear* – make a hash table empty again ∗)
val *clear* : (α, β) *Hashtbl.t* → *unit*

This time, adding a new key and value modifies the existing hash table object, rather than creating a new one. When hash tables are first created, we give a rough estimate of the number of entries, and this determines the initial number of buckets. If the hash table become full, however, then it will automatically expand, so the initial estimate is there only to prevent multiple expansion steps in the early part of the hash table's lifetime.

A.9 Formatted output

The module *Print* (not part of the standard library) provides a general facility for formatted output via the following three functions:

(∗ *printf* – print on standard output ∗)
val *printf* : *string* → *arg list* → *unit*

(∗ *fprintf* – print to a file ∗)
val *fprintf* : *out_channel* → *string* → *arg list* → *unit*

```
(* sprintf – print to a string *)
val sprintf : string → arg list → string
```

Calling *printf format args* formats the list of items *args* and inserts the resulting text in the places indicated by dollar signs in the string *format*. For example,

```
printf "$ is $ years old\n" [fStr "Mike"; fNum 34]
```

prints the text "Mike is 34 years old" (followed by a newline) on standard output. Similarly, the function *fprintf* provides formatted output on an arbitrary output channel, and the function *sprintf* formats data in the same way and returns the result as a string. In our compilers, we shall actually use *fprintf* only with the standard error channel *stderr* for printing error messages.

The arguments to *printf* and friends are taken from the type *Print.arg*, which has the following interface:

```
type arg

(* Basic formats *)
val fNum : int → arg              (* Decimal number *)
val fFix : int * int → arg        (* Fixed-width number (val, width) *)
val fFlo : float → arg            (* Floating-point number *)
val fStr : string → arg           (* String *)
val fChr : char → arg             (* Character *)
val fBool : bool → arg            (* Boolean *)

(* fMeta – insert output of recursive call to printf *)
val fMeta : string * arg list → arg

(* fList – format a comma-separated list *)
val fList : (α → arg) → α list → arg

(* fExt – higher-order extension *)
val fExt : ((string → arg list → unit) → unit) → arg
```

The functions listed provide ways of converting various common types into values of type *Print.arg*. Most of the possibilities are self-explanatory, except for the functions *fMeta*, *fList* and *fExt*, which allow for extensions to the range of types that can be printed.

A.10 Computer representation of OCaml programs

The Objective CAML programs that appear in this course have been formatted in a nice way for printing: keywords appear in bold face type, and fancy symbols like → have been used in place of ASCII combinations like ->. I find that these conventions make printed Objective CAML programs much easier to read. The Objective CAML compiler, however, expects programs to be represented in ASCII form, so that the definition that appears in the course as

```
let rec eval =
  function
      Constant n → n
```

Symbol	*ASCII equivalent*
≠	<>
≤	<=
≥	>=
→	->
←	<-
⌢	^

Table A.1: ASCII *equivalents for special symbols*

```
| Binop (w, e1, e2) →
    do_binop w (eval e1) (eval e2)
| Variable x →
    failwith "sorry, I don't do variables"
```

would actually look like this when submitted to the Objective CAML compiler:

```
let rec eval =
  function
      Constant n -> n
    | Binop (w, e1, e2) ->
      do_binop w (eval e1) (eval e2)
    | Variable x ->
      failwith "sorry, I don't do variables"
```

The rules for transcribing programs as they appear in the course into the form expected by the Objective CAML compiler are very simple.[4]

- Replace all **bold-face** keywords and *italic* identifiers by the same keywords and identifiers written in ordinary type. Replace identifiers that appear in *SMALL CAPS* by the same identifiers written in all capitals.
- Replace symbols like ← by equivalents like <- made from several ASCII characters. Table A.1 shows the special symbols that are used in the printed Objective CAML programs in this course, together with their ASCII equivalents.
- Replace subscripted identifiers like env_0 and y_1' by ordinary identifiers like `env0` and `y1'`.
- Replace the Greek letters α, β, etc., used for type variables, by the ASCII forms `'a`, `'b`, etc..

[4] Of course, the transcription really goes the other way, and the author has written a program *ocamlgrind* that converts the ASCII form into input for TEX that produces the printed form.

Appendix B

The Keiko machine

The little compilers you will build as part of the labs 1–3 all output code for the Keiko virtual machine, which I developed as a vehicle for implementing Niklaus Wirth's language Oberon in a portable way. A Keiko program (in so far as concerns us here) has a fixed header and contains a number of procedures, including one with the name MAIN that is invoked as the main program. Each procedure contains a sequence of instructions such as those listed below. The program can also contain, outside any procedure, directives that reserve storage for global variables, define string constants, or create other constant structures. Most of the top-level structure of the compiler output is already implemented in the main program of the compilers we shall work with in the course, so I won't describe it in any detail here. The sections of the chapter follow the flow of the course, listing just the instructions needed to compile the programs we will handle at each stage.

Keiko also has many instructions that we won't use, such as support for arithmetic on 64-bit integers and single and double precision floating point numbers. In addition, there are instructions that combine several operations into one. For example, the instruction *LDLW* 12 is equivalent to the sequence *LOCAL* 12; *LOADW*, and loads in one operation the word that is at offset 12 in the current stack frame. Providing these combined instructions makes the bytecode shorter and also faster, because the virtual machine need go through only one fetch/execute cycle instead of two or more. In a compiler, it's convenient to translate the source program into instructions that each correspond to one operation, then to use a *peephole optimiser* to combine the operations into larger units.

The code has two representations: one as an abstract data type in OCaml that is used internally in our compilers, and another, textual, representation that's accepted by the Keiko assembler. The type that represents Keiko code internally is defined as in Figure B.1. The *code* type also contains the three constructors *SEQ*, *NOP* and *LINE*, which are explained in Section 3.4. In some ways, this internal representation is more general than the textual form, in that (for example) the binary operations of addition and multiplication are represented as *BINOP Plus* and *BINOP Times*, but in the textual form they become PLUS and TIMES. Where the actual instruction mnemonics differ from the OCaml representation, this is noted in the lists of instructions below.

```
type code =
    CONST of int                    (* Push constant (value) *)
  | GLOBAL of string                (* Push symbol (name) *)
  | LOCAL of int                    (* Push local address (offset) *)
  | LOADW                           (* Load word *)
  | STOREW                          (* Store word *)
  | LOADC                           (* Load character *)
  | STOREC                          (* Store character *)
  | LDGW of string                  (* Load global word (name) *)
  | STGW of string                  (* Store global word (name) *)
  | MONOP of op                     (* Perform unary operation (op) *)
  | BINOP of op                     (* Perform binary operation (op) *)
  | LABEL of codelab                (* Code label *)
  | JUMP of codelab                 (* Unconditional branch (dest) *)
  | JUMPC of op * codelab           (* Conditional branch (op, dest) *)
  | PCALL of int                    (* Call procedure (nargs) *)
  | PCALLW of int                   (* Call procedure with result (nargs) *)
  | RETURN                          (* Return from procedure *)
  | BOUND of int                    (* Bounds check (line) *)
  | CASEJUMP of int                 (* Case jump (num cases) *)
  | CASEARM of int * codelab        (* Case value and label *)
  | PACK                            (* Pack two values into one *)
  | UNPACK                          (* Unpack one value into two *)

  | SEQ of code list                (* Sequence of code fragments *)
  | NOP                             (* Empty sequence *)
  | LINE of int                     (* Line number *)
```

Figure B.1: *Keiko instructions*

B.1 Expressions

These instructions are needed to evaluate simple arithmetic expressions and assignments that use global variables. The instructions implicitly act on an evaluation stack.

CONST n
: Push an integer constant onto the evaluation stack.

LDGW x
: Load a 32-bit quantity from a global address. Actually equivalent to the sequence *GLOBAL x*; *LOADW*.

STGW x
: Store a 32-bit quantity into a global address. Actually equivalent to the sequence *GLOBAL x*; *STOREW*.

BINOP op — PLUS, MINUS, TIMES, DIV, MOD, EQ, LT, GT, LEQ, GEQ, NEQ, AND, OR
: These instructions are represented in OCaml programs by *BINOP Plus*, *BINOPMinus*, . . . , *BINOP Or*. Each of them pops two values off the evaluation stack, computes the result of a binary operation, and pushes the

result back on the stack. The comparison operators produce a boolean result represented as 1 (true) or 0 (false); the boolean operators expect inputs that are represented like that and produce a result with the same representation.

MONOP op — UMINUS, NOT
These instructions are represented in OCaml by *MONOP Uminus* and *MONOPNot*. They perform unary operations, popping a value off the stack and pushing the result. The conventions are the same as for the *BINOP* family.

OFFSET
Expect an address and an offset on the stack, and add them together. This instruction performs the same operation as PLUS but is used to mark address calculations for possible later optimisation.

B.2 Control structures

These instructions allow control structures such as if and while to be translated into patterns of conditional jumps.

JUMP lab
Unconditional branch to the label *lab*. The next instruction to be executed will be the one following *LABEL lab* elsewhere in the same procedure.

JUMPC (*op*, *lab*) — JEQ, JNE, JLT, JLEQ, JGT, JGEQ
Pop two integer values from the evaluation stack, compare them, and jump to *lab* if the specified condition holds.

LABEL lab
Not really an executable instruction, but a directive to attach a label to the following instruction.

B.3 Data structures

These instructions complete the set of operations needed to translate access to data structures such as records and arrays into arithmetic on addresses.

GLOBAL x
Push a global address. The symbol *x* should be defined elsewhere by a FUNC directive or a GLOVAR directive. The address of the corresponding procedure or global variable is pushed onto the evaluation stack.

LOCAL n
Push a local address. The offset *n* is added to the frame pointer register *fp*, and the result is pushed onto the evaluation stack.

LOADW
Load a 32-bit word. An address is popped from the evaluation stack, and the contents of the address are pushed in its place.

STOREW
: Store a 32-bit word. An address and then a 32-bit word are popped from the evaluation stack, and the word is stored at the address.

LOADC
: Load an 8-bit character. An address is popped from the evaluation stack, and an 8-bit character is loaded from that address and zero-extended to 32 bits. The resulting value is pushed on the evaluation stack in place of the address.

STOREC
: Store an 8-bit character. An address and than a value are popped from the evaluation stack. All but the lowest-order 8 bits of the value are discarded, and the remaining bits are stored at the address.

BOUND ln
: This instruction pops an array bound b from the evaluation stack, and inspects the value i beneath it without popping the value. If the value i does not lie in the range $[0..b)$, execution halts with an array bound error. The argument *ln* gives the line number that appears in the error message.

NCHECK ln
: This instruction inspects a pointer value on the evaluation stack without popping the value. If the value is a null pointer, execution halts with a null pointer error. The argument *ln* gives the line number that appears in the error message.

B.4 Procedures

These instructions are used to translate procedure calls. The *PCALL* instruction applies to procedures that don't return a result, and the *PCALLW* instruction applies to those that do. The instructions create and destroy activation records on a subroutine stack; the layout of stack frames is described elsewhere.

PCALL n
: Call a procedure. Before this instruction, the evaluation stack should contain $n + 2$ items, consisting of n parameters, a static link, and a procedure address. These are made into a stack frame for the procedure, and control is transferred to the first instruction in the procedure's body. When the procedure returns, the stack frame is destroyed and the $n + 2$ items listed above are removed, leaving untouched any items beneath them on the evaluation stack. Execution continues with the next instruction after the *PCALL* instruction.

PCALLW n
: Call a procedure with a result. This is the same as the call instruction, except that the procedure is expected to return a one-word result; this result remains on the evaluation stack after the procedure returns.

RETURN
: Return from a procedure.

B.5 Specials for Lab 1

These special instructions allow a simple translation of `case` statements into a table that is searched linearly.

CASEJUMP n
: This instruction should be followed by a 'case table' of n entries, made up of *CASEARM* instructions. The instruction pops a value off the evaluation stack, and if it matches any of the values in the case table, then control transfers to the corresponding label. If none of the values in the table matches, then execution continues with the instruction that appears after the jump table.

CASEARM (v, *lab*)
: An entry in the jump table following an *CASEJUMP* instruction. The argument v is an integer value to be matched against the value the *CASEJUMP* instruction finds on the evaluation stack, and *lab* is the corresponding numeric label. The value v should be in the range $-32768 \leq v < 32768$, but there's no need to worry about that in the lab exercise.

Note that `case` statements are implemented in other compilers for Keiko using other instructions, TESTGEQ, JRANGE and JCASE, that can be put together to make a binary search tree with range tests and indexed jump tables at the leaves.

B.6 Specials for Lab 3

These instructions allow functional parameters to be implemented in an otherwise untyped language by enabling a single-word representation of closures.

PACK
: Normally, compilers must allocate two words for the (*code*, *env*) pair that makes up a closure. To simplify matters, the compiler in Lab 3 assumes that these two words can be packed into one using this instruction. This works provided that no more than 256 different procedures are used as the *code* component of closures, and that offsets in the program's stack fit in 24 bits.

UNPACK
: This instruction reverses the effect of a *PACK* instruction, expecting a packed value on the evaluation stack and replacing it with the original (*code*, *env*) pair.

ERROR E_RETURN 0
: The code for Lab 3 inserts this special instruction at the end of each procedure to detect the runtime error of falling off the end of the procedure without executing a `return` statement.

B.7 Common abbreviations

Common sequences of instructions can be replaced by single instructions to make the bytecode more compact and to speed up its execution, because only one cycle of interpreter overhead is needed to get the effect of several operations. Typically these replacements are made by a peephole optimiser. The most important of abbreviations are *LDGW* and *STGW*, the conditional branches *JEQ* etc. (all mentioned above), and the following memory operations:

LDLW n
: Load local word, equivalent to *LOCAL n*; *LOADW*.

STLW n
: Store local word, equivalent to *LOCAL n*; *STOREW*.

LDNW n
: Load indexed word, equivalent to *CONST n*; *OFFSET*; *LOADW*.

STNW n
: Store indexed word, equivalent to *CONST n*; *OFFSET*; *STOREW*.

If the final target is machine code, there's no point in introducing these abbreviations, and it's better to generate the individual operations, then select machine instructions each implement a group of operations.

B.8 Directives

These code items aren't actually Keiko instructions that can be executed, but are used to give a structure to the assembly language output by our compilers. Each program begins with the three lines

```
MODULE Main 0 0
IMPORT Lib 0
ENDHDR
```

These give the program the name Main and record that it uses library routines from a module Lib, which provides the routines Lib.Print and Lib.Newline. Keiko supports programs compiled from several independent parts that are linked together, but we won't use this in the course.

Our compilers translate the main program (enclosed by `begin` and `end` towards the end of the source code) into a procedure called MAIN, and the Keiko bytecode system invokes this procedure when the program starts.

Other directives are used in the program itself. These directives aren't represented in the type *code* of Keiko code sequences, but just printed out when they are needed.

FUNC *P n*
: This begins a procedure named *P* that uses *n* bytes of stack space for local variables. The directive defines the symbol *P* as the address used to call the procedure.

END
: This directive marks the end of a procedure.

GLOVAR *x n*

This directive asks the assembler/linker to reserve *n* bytes of space for a global variable named *x*. The storage will be aligned to that its address is a multiple of 4 bytes.

Appendix C

A rough guide to ARM

This rough guide contains just enough information about the ARM to write a simple code generator.

The conventions for subroutine call and registers usage we describe are *consistent* with those obeyed by GCC and other ARM software, so that it will be possible for code generated by our compiler to be called from code translated by GCC and to call it in turn.

We won't implement floating point arithmetic in our compiler, so the ARM floating point unit isn't described here.[1]

C.1 Registers

In place of the evaluation stack of Keiko, the ARM has a set of 16 registers:[2]

- Registers `r0` – `r3` are caller-save, and are used to pass up to four arguments to a subroutine. They are otherwise available as scratch registers.
- Registers `r4` – `r10` are callee-save. They are suitable for use as register variables, and are otherwise available as scratch registers.
- Register `r11` is also known as `fp`. It points to the base of the stack frame.
- Register `r12` aka `ip` is a scratch register used during the procedure prolog, and also within a few multi-instruction idioms that we shall meet. It is otherwise unused.
- Register `r13` is the stack pointer `sp`. It always points to the outgoing argument area.
- Register `r14` is the link register `lr`, where the return address appears after a subroutine call.
- Register `r15` is the program counter `pc`.

[1] I have nothing against floating point arithmetic! It's just that implementing it would just increase the size of our language and compiler without really illustrating anything new.

[2] Other accounts of the ARM may indicate that it has several *sets* of 16-or-so hardware registers. These multiple sets are helpful in keeping the registers used by the operating system separate from those used by ordinary programs. Each program uses only one set of registers.

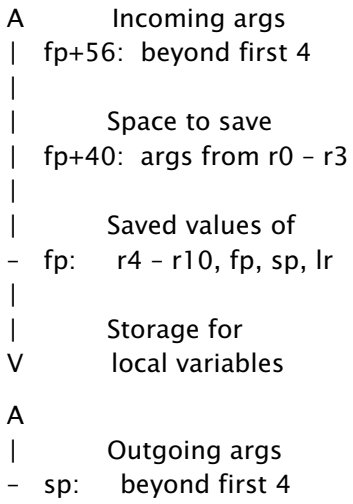


Figure C.1: *Stack frame layout*

Only registers `lr` and `pc` have a special function fixed by the ARM hardware, and `lr` only because there are instructions that simultaneously save the current `pc` value in `lr` and load the `pc` with the address of a subroutine. The remainder of the special functions are assigned to particular registers by software convention.

C.2 Storage layout

Each ARM subroutine has a stack frame, and also has access to statically allocated storage that we shall use to implement global variables. The layout of a stack frame is as shown in Figure C.1. Storage in the upper part of the picture is accessed at positive offsets from the frame pointer `fp`; local variables are at negative offsets from `fp`; and an area for outgoing arguments is addressed at positive offsets from the stack pointer `sp`.

- If the subroutine has more than four words of arguments, then the additional arguments appear at offset `fp+56`. The stack pointer register points at this location when control reaches the subroutine prolog.
- If there are any arguments at all, then there is space at `fp+40` to save them. Our standard prolog will store the incoming arguments here, from the registers `r0` – `r3` where they arrive. This area has size 0, 8 or 16 bytes, padded if necessary so that the value of `fp` is aligned at an 8-byte boundary.[3]
- There is space at `fp+0` to save those machine registers which must be preserved by the subroutine, including `r4` – `r10`, the stack pointer `sp`, the frame pointer `fp` and the link register `lr`, which contains the return address.[4]

[3] In a leaf routine (one that calls no others) there is no need to save the incoming arguments in the stack frame, but we do so for simplicity.

[4] There's no need to save registers that the subroutine doesn't use, but for simplicity we

As an extension to the ARM calling convention, we will pass the static link in `r4` to subroutines that need one. The link will be saved at `fp+0` as part of the prolog, and can be accessed from there in the subroutine body. If a static link if known to be zero, then there is no need to pass it explicitly.

Storage for local variables is addressed at negative offsets from `fp`. If the subroutine calls others that expect more than four words of arguments, then an area big enough for the biggest call is allocated at the end of the stack frame. The value of `sp` is, like `fp`, aligned on a multiple of 8 bytes.

Static storage can be allocated in assembly language with a `.comm` directive. For example,

```
.comm _x, 4, 4
```

allocates 4 bytes of storage aligned on a multiple of 4 bytes and makes `_x` be an assembler symbol equal to its address.

C.3 Subroutine skeleton

Given the frame layout, we can design prolog and epilog sequences for a subroutine that respectively create and destroy the frame. These sequences are generated by the functions in the *Target* module that begin and end translation of each subroutine. The following prolog suits a subroutine with one or two arguments that needs 16 bytes of local variable space.[5]

```
mov ip, sp
stmfd sp!, {r0-r1}
stmfd sp!, {r4-r10, fp, ip, lr}
mov fp, sp
sub sp, sp, #16
```

The two `stmfd!` instructions push different sets of registers on the stack: first, the two argument registers `r0` and `r1`, then the registers `r4` up to `r10`, together with the dynamic link (from `fp`), the stack pointer from the call (from `ip`) and the return address (from `lr`). The `sp!` in these instructions means that the stack pointer is modified as the values are pushed.

At the end of the subroutine body, we can return to the caller with a single instruction:

```
ldmfd fp, {r4-r10, fp, sp, pc}
```

This uses the fact that the frame pointer contains the address of the memory where the registers were saved. The initial values of `r4` to `r10` are restored, the frame pointer and the stack pointer are reset to their values at the call, and the return address is reloaded into the program counter.

---

always do so. If we chose not to save all registers, we could economise on space by allocating a smaller region to save registers; but then the offset of the arguments from `fp` would vary from one subroutine to another, and that would provide an additional administrative complication, especially in the presence of nested subroutines.

[5] The `stmfd` mnemonic stands for STore Multiple words on a stack with pointer addressing a Full word, growing Downwards. There are other instructions that suit other conventions about the direction of stack growth, and whether the stack pointer addresses the last full cell or the first empty one.

C.4 Instructions

Arithmetic instructions specify three registers, or two registers and a small constant. The multiply instruction `mul` must use three registers.

```
add r0, r0, r1        – set r0 to sum of r0 and r1
sub r1, r2, #7        – subtract 7 from r2, result in r1
mul r1, r3, r4        – set r1 to product of r3 and r4
```

Load and store instructions use addressing modes that can add a register and a small constant, or add two registers.

```
ldr r0, [fp, #44]     – load second argument into r0
str r0, [r1, #8]      – store record field at offset 8 from r1
ldrb r1, [r2, r4]     – add r2 and r4 to form an address
                      –    and load one byte from there into r1
```

Large constants (including the addresses of static variables) can be developed into a register with a special instruction:[6]

```
ldr r0, =12345678     – set r0 to an integer constant
```

Getting the value of static variable `x` into `r1` requires two instructions:

```
ldr r0, =_x           – set r0 to address of _x
ldr r1, [r0]          – load from that address into r1
```

The first of these puts the *address* of `_x` into `r0`, and the second loads from that address and puts the result in `r1`.

C.5 Branches

We can write a branch instruction in assembly language using a label. It is compiled into a relative branch that contains the offset between the branch instruction and the target label.

```
    b .L37            – branch to label .L37
    ...
.L37:
```

It's an established convention that compiler-generated labels are given names like `.L37`; such labels are suppressed, for example, in listings of the program's symbols from the assembler or linker.

For conditional branches, we first use a `cmp` instruction to set the *condition codes* `NCVZ` that is are hidden part of the processor status. These codes enable a subsequent conditional branch to determine the result of the comparison.

```
cmp r0, #7            – compare r0 with constant 7
blt .L44              – branch to .L44 if r0 < 7
```

[6] The assembler translates this instruction by compiling a table of constants for each subroutine that is placed in memory after the end of the subroutine's code (where the `.pool` directive appears in the compiler's output). The `ldr=` instruction then becomes `ldr r0, [pc, #`*offset*`]`, where the offset is computed by the assembler.

C.6 More about addressing modes and operands

So far, we've seen two addressing modes: we can form a memory address by adding a register and a small constant, as in `[r0, #8]`, and we can add two registers, as in `[r2, r4]`. As a special case, we can use a single register `[r0]` as an abbreviation for `[r0, #0]`. We shall also want to exploit an ARM addressing mode where we can add two registers but multiply one of them by a power of two by shifting it:

```
ldr r0, [r2, r4, LSL #2]
```

This instruction adds together `r2` and $4 \times$ `r4` (obtained by shifting the value of `r4` left by two bits). It loads from the resulting address and puts the result in `r0`.[7]

These shifts are available not only in instructions that address memory, but also for the operands of arithmetic instructions. For example, this instruction:

```
add r0, r1, r2, LSL #4
```

adds together the value in `r1` and 16 times the value in `r2` and puts the result in `r0`. It's also possible to take the shift amount from a register:

```
add r0, r1, r2, LSL r4
```

does a similar thing, but shifts the value from `r2` by an amount determined by the value of `r4`. This kind of variable shift takes an extra cycle on many implementations of ARM, and it isn't available in load and store instructions, only in arithmetic instructions.

In point of fact, what appear to be shift instructions on the ARM are actually move instructions with a shifted operand, so the following two instructions are identical:

```
lsl r1, r2, #4
mov r1, r2, LSL #4
```

C.7 Conditional execution

It is not only branch instructions that can be made conditional, but in fact almost any instruction at all. For example, this sequence sets `r0` to 0 or 1 depending on whether `r1` is less than `r2`:

```
cmp r1, r2          - compare r1 and r2
mov r0, #0          - set r0 to 0
movlt r0, #1        - reset r0 to 1 if r1 < r2
```

The third instruction is executed only of the result of the comparison indicates that `r1` $<$ `r2`. Note that the state of the condition codes is not affected

[7] The ARM also allows right shifts and rotations in place of the left shift. It can write back the incremented address to the first-named index register, and there is a post-indexed as well as a pre-indexed form of the addressing mode with write-back. All these options apply only to the load-word and load-unsigned-byte instructions (and the corresponding stores); shifts and write-back are not provided for other loads and stores such as load-signed-byte and load-unsigned-halfword. All this we do not use and can ignore.

by the first `mov` instruction, so it is still available to be tested by the conditional move. This sequence of instructions is convenient because it reads its input registers, `r1` and `r2`, before writing the output register `r0`, and would continue to work even if the input and output registers overlapped.

This form of conditional execution can be faster than a branch instruction, but we shall use it only as an idiom for computing the Boolean result of a comparison into a register, and in another tricky idiom for array bounds checks. Here is code for checking that $0 \le$ `r3` < 10 on line 1234:

```
cmp r3, #10          – compare r3 with array bound
ldrhs r0, =1234      – conditionally load line number into r0
blhs check           – conditionally call error routine
```

Both the instruction to load the line number into `r0` and the instruction to call the routine to stop the program with an error message are conditional on the result of the comparison. Trickily, the condition, `hs`, tests whether `r3` ≥ 10 as an *unsigned* comparison, which will evaluate as true if (as a signed number) `r3` is 10 or more, or if it is negative. Similar tricks can be played with the branching code for `case` statements.

Appendix D

A machine grammar for ARM

This list of productions covers all of the ARM instructions and addressing modes that we shall use in the course. Some of the rules (numbers 36, 42 and 43) are missing from the compiler of Lab 4, and part of the task in that lab is to implement them. These rules are not a complete description of the ARM instruction set, because they omit a few integer instructions that we won't use, and they entirely omit instructions that implement floating point arithmetic.

D.1 Legend

There are six non-terminals: *reg* corresponds to expressions computed in registers; *rand* and *addr* to the operands of arithmetic and load/store instructions respectively; *shift* to the operands of shift instructions. Meanwhile, *call* matches procedure calls; and *stmt* matches at the roots of optrees.

Compared with the overview given in Section 8.2, the *shift* nonterminal is new. It allows us to make a distinction between the operands that may appear in most arithmetic instructions, where the form "*reg*, `LSL #`*n*" is allowed, as specified in rule 36, and those that may appear in shift instructions such as `lsl` and `lsr`, where such 'shifter operands' are not allowed, because each instruction can use the shifter only once; the instruction

```
lsl r1, r2, r3, LSL r4
```

looks pretty useless anyway, and seems to ask for `r2` to be shifted by an amount that itself results from shifting `r3` left by the value in `r4`. In truth, a tamer `lsl` instruction such as

```
lsl r1, r2, r3
```

is really an abbreviation for the instruction

```
mov r1, r2, LSL r3
```

that contains one of these shifter operands as the argument of a move. In addition, the constants that may appear as shift amounts are restricted to the range 1 to 31; we choose to develop larger shift amounts into a register first, so we can avoid tangling with the slightly complex rules by which the hardware interprets them.

Some other conventions:

- The instruction template '`?mov` *reg*, *reg*$_1$' denotes a `mov` instruction that may be elided if *reg* and *reg*$_1$ can be assigned the same register.
- The side-condition 'when *fits_xxx n*' corresponds to one of the tests for fitting in a field that are defined in lab4/tran.ml.
- The side-condition 'target *reg* = *reg*$_1$' implies that the sub-tree rooted at *reg*$_1$ should be computed into the same register as the tree at *reg*.
- Some rules (e.g., the rules 57 to 62 for conditional branches) produce more than one assembly language instruction, and use semicolons to separate them.

D.2 Expressions

1. *reg* → ⟨CONST *k*⟩ { `mov` *reg*, `#`*k* } (when *fits_move k*)
2. *reg* → ⟨CONST *k*⟩ { `ldr` *reg*, `=`*k* }

3. *reg* → ⟨LOCAL 0⟩ { `?mov` *reg*, `fp` }
4. *reg* → ⟨LOCAL *n*⟩ { `add` *reg*, `fp`, `#`*n* } (when *fits_add n*)
5. *reg* → ⟨LOCAL *n*⟩ { `ldr ip, =`*n*`; add` *reg*, `fp, ip` }

6. *reg* → ⟨GLOBAL *x*⟩ { `ldr` *reg*, `=`*x* }

7. *reg* → ⟨TEMP *n*⟩ { `?mov` *reg*, *temp*$_n$ }

8. *reg* → ⟨LOADW, ⟨REGVAR *n*⟩⟩ { `?mov` *reg*, *regvar*$_n$ }
9. *reg* → ⟨LOADW, *addr*⟩ { `ldr` *reg*, *addr* }

10. *reg* → ⟨LOADC, ⟨REGVAR *n*⟩⟩ { `?mov` *reg*, *regvar*$_n$ }
11. *reg* → ⟨LOADC, *addr*⟩ { `ldrb` *reg*, *addr* }

12. *reg* → ⟨MONOP *Uminus*, *reg*$_1$⟩ { `neg` *reg*, *reg*$_1$ }
13. *reg* → ⟨MONOP *Not*, *reg*$_1$⟩ { `eor` *reg*, *reg*$_1$, `#1` }
14. *reg* → ⟨MONOP *BitNot*, *reg*$_1$⟩ { `mvn` *reg*, *reg*$_1$ }

15. *reg* → ⟨OFFSET, *reg*$_1$, ⟨CONST *k*⟩⟩ { `add` *reg*, *reg*$_1$, `#`*k* } (when *fits_add k*)
16. *reg* → ⟨OFFSET, *reg*$_1$, *rand*⟩ { `add` *reg*, *reg*$_1$, *rand* }

17. *reg* → ⟨BINOP *Plus*, *reg*$_1$, *rand*⟩ { `add` *reg*, *reg*$_1$, *rand* }
18. *reg* → ⟨BINOP *Minus*, *reg*$_1$, *rand*⟩ { `sub` *reg*, *reg*$_1$, *rand* }
19. *reg* → ⟨BINOP *And*, *reg*$_1$, *rand*⟩ { `and` *reg*, *reg*$_1$, *rand* }
20. *reg* → ⟨BINOP *Or*, *reg*$_1$, *rand*⟩ { `orr` *reg*, *reg*$_1$, *rand* }
21. *reg* → ⟨BINOP *Lsl*, *reg*$_1$, *shift*⟩ { `lsl` *reg*, *reg*$_1$, *shift* }
22. *reg* → ⟨BINOP *Lsr*, *reg*$_1$, *shift*⟩ { `lsr` *reg*, *reg*$_1$, *shift* }
23. *reg* → ⟨BINOP *Asr*, *reg*$_1$, *shift*⟩ { `asr` *reg*, *reg*$_1$, *shift* }
24. *reg* → ⟨BINOP *BitAnd*, *reg*$_1$, *rand*⟩ { `and` *reg*, *reg*$_1$, *rand* }
25. *reg* → ⟨BINOP *BitOr*, *reg*$_1$, *rand*⟩ { `orr` *reg*, *reg*$_1$, *rand* }

26. *reg* → ⟨*BINOP Times*, reg_1, reg_2⟩ { `mul` *reg*, reg_1, reg_2 }

27. *reg* → ⟨*BINOP Eq*, reg_1, *rand*⟩ { `cmp` reg_1, *rand*; `mov` *reg*, `#0; moveq` *reg*, `#1` }

28. *reg* → ⟨*BINOP Neq*, reg_1, *rand*⟩ { `cmp` reg_1, *rand*; `mov` *reg*, `#0; movne` *reg*, `#1` }

29. *reg* → ⟨*BINOP Gt*, reg_1, *rand*⟩ { `cmp` reg_1, *rand*; `mov` *reg*, `#0; movgt` *reg*, `#1` }

30. *reg* → ⟨*BINOP Geq*, reg_1, *rand*⟩ { `cmp` reg_1, *rand*; `mov` *reg*, `#0; movge` *reg*, `#1` }

31. *reg* → ⟨*BINOP Lt*, reg_1, *rand*⟩ { `cmp` reg_1, *rand*; `mov` *reg*, `#0; movlt` *reg*, `#1` }

32. *reg* → ⟨*BINOP Leq*, reg_1, *rand*⟩ { `cmp` reg_1, *rand*; `mov` *reg*, `#0; movle` *reg*, `#1` }

33. *reg* → ⟨*BOUND*, reg_1, *rand*⟩ { `cmp` reg_1, *rand*; `ldrhs r0, =`*line*`; blhs check` }
(target *reg* = reg_1)

34. *reg* → ⟨*NCHECK*, reg_1⟩ { `cmp` reg_1, `#0; ldreq r0, =`*line*`; ble nullcheck` }
(target *reg* = reg_1)

D.3 Operands

35. *rand* → ⟨*CONST k*⟩ { *#k* } (when *fits_immed k*)

36. *rand* → ⟨*BINOP Lsl*, *reg*, *shift*⟩ { *reg*, `LSL` *shift* }

37. *rand* → *reg* { *reg* }

38. *shift* → ⟨*CONST k*⟩ { *#k* } (when $1 \le k < 32$)

39. *shift* → *reg* { *reg* }

D.4 Addresses

40. *addr* → ⟨*LOCAL n*⟩ { `[fp, #`*n*`]` } (when *fits_offset n*)

41. *addr* → ⟨*OFFSET*, *reg*, ⟨*CONST k*⟩⟩ { `[`*reg*`, #`*k*`]` } (when *fits_offset n*)

42. *addr* → ⟨*OFFSET*, reg_1, reg_2⟩ { `[`reg_1`,` reg_2`]` }

43. *addr* → ⟨*OFFSET*, reg_1, ⟨*BINOP Lsl*, reg_2, ⟨*CONST k*⟩⟩⟩
{ `[`reg_1`,` reg_2`, LSL #`*k*`]` } (when $1 \le k < 32$)

44. *addr* → *reg* { `[`*reg*`]` }

D.5 Procedure calls

45. *call* → ⟨*GLOBAL f*⟩ { `bl` *f* }

46. *call* → *reg* { `blx` *reg* }

D.6 Statements

47. *stmt* → ⟨*DEFTEMP n*, *reg*⟩ (set $temp_n$ = *reg*)

48. *stmt* → ⟨*STOREW*, *reg*, ⟨*REGVAR n*⟩⟩ (target *reg* = $regvar_n$)
49. *stmt* → ⟨*STOREW*, *reg*, *addr*⟩ { `str` *reg*, *addr* }

50. *stmt* → ⟨*STOREC*, *reg*, ⟨*REGVAR n*⟩⟩ (target *reg* = $regvar_n$))
51. *stmt* → ⟨*STOREC*, *reg*, *addr*⟩ { `strb` *reg*, *addr* }

52. *stmt* → ⟨*CALL k*, *call*⟩
53. *stmt* → ⟨*DEFTEMP n*, ⟨*CALL k*, *call*⟩⟩ { `?mov` *reg*, `r0` }

54. *stmt* → ⟨*RESULTW*, *reg*⟩ (target *reg* = `r0`)

55. *stmt* → ⟨*LABEL lab*⟩ { *lab*: }
56. *stmt* → ⟨*JUMP lab*⟩ { `b` *lab* }

57. *stmt* → ⟨*JUMPC* (*Eq*, *lab*), *reg*, *rand*⟩ { `cmp` *reg*, *rand*; `beq` *lab* }
58. *stmt* → ⟨*JUMPC* (*Lt*, *lab*), *reg*, *rand*⟩ { `cmp` *reg*, *rand*; `blt` *lab* }
59. *stmt* → ⟨*JUMPC* (*Gt*, *lab*), *reg*, *rand*⟩ { `cmp` *reg*, *rand*; `bgt` *lab* }
60. *stmt* → ⟨*JUMPC* (*Leq*, *lab*), *reg*, *rand*⟩ { `cmp` *reg*, *rand*; `ble` *lab* }
61. *stmt* → ⟨*JUMPC* (*Geq*, *lab*), *reg*, *rand*⟩ { `cmp` *reg*, *rand*; `bge` *lab* }
62. *stmt* → ⟨*JUMPC* (*Neq*, *lab*), *reg*, *rand*⟩ { `cmp` *reg*, *rand*; `bne` *lab* }

63. *stmt* → ⟨*JCASE* (*table*, *deflab*), *reg*⟩ { `cmp` *reg*, `#`*n*;
`ldrlo pc, [pc,` *reg*`, LSL #2]; b` *deflab* }
(where *n* = *length table*, followed by jump table)

64. *stmt* → ⟨*ARG i*, ⟨*TEMP n*⟩⟩ { `?mov r`*i*, $temp_n$ } (when $i < 4$)
65. *stmt* → ⟨*ARG i*, *reg*⟩ (target *reg* = `r`*i*, when $i < 4$)
66. *stmt* → ⟨*ARG i*, *reg*⟩ { `str` *reg*, `[sp,` *o*`]` }
(where $o = 4 * i - 16$, when $i \geq 4$)

67. *stmt* → ⟨*STATLINK*, ⟨*CONST* 0⟩⟩ (no-op)
68. *stmt* → ⟨*STATLINK*, *reg*⟩ (target *reg* = `r4`)

Appendix E

Code listings

E.1 Lab one

E.1.1 lexer.mll

```
(* lab1/lexer.mll *)
(* Copyright (c) 2017 J. M. Spivey *)

{
open Lexing
open Tree
open Keiko
open Parser

(* |kwtable| -- a little table to recognise keywords *)
let kwtable =
  Util.make_hash 64
    [ ("begin", BEGIN); ("do", DO); ("if", IF ); ("else", ELSE);
      ("end", END); ("then", THEN); ("while", WHILE); ("print", PRINT);
      ("newline", NEWLINE); ("and", MULOP And); ("div", MULOP Div);
      ("or", ADDOP Or); ("not", MONOP Not); ("mod", MULOP Mod);
      ("true", NUMBER 1); ("false", NUMBER 0) ]

(* |idtable| -- table of all identifiers seen so far *)
let idtable = Hashtbl.create 64

(* |lookup| -- convert string to keyword or identifier *)
let lookup s =
  try Hashtbl.find kwtable s with
    Not_found ->
      (* Use |Hashtbl.replace| so each ident appears only once *)
      Hashtbl.replace idtable s ();
      IDENT s

(* |get_vars| -- get list of identifiers in the program *)
let get_vars () =
  Hashtbl.fold (fun k () ks -> k::ks) idtable []
```

```
(* |lineno| -- line number for use in error messages *)
let lineno = ref 1
}

rule token =
  parse
      ['A'-'Z''a'-'z']['A'-'Z''a'-'z''0'-'9''_']* as s
                        { lookup s }
    | ['0'-'9']+ as s   { NUMBER (int_of_string s) }
    | ";"               { SEMI }
    | "."               { DOT }
    | ":"               { COLON }
    | "("               { LPAR }
    | ")"               { RPAR }
    | ","               { COMMA }
    | "="               { RELOP Eq }
    | "+"               { ADDOP Plus }
    | "-"               { MINUS }
    | "*"               { MULOP Times }
    | "<"               { RELOP Lt }
    | ">"               { RELOP Gt }
    | "<>"              { RELOP Neq }
    | "<="              { RELOP Leq }
    | ">="              { RELOP Geq }
    | ":="              { ASSIGN }
    | [' ''\t']+        { token lexbuf }
    | "(*"              { comment lexbuf; token lexbuf }
    | "\r"              { token lexbuf }
    | "\n"              { incr lineno; Source.note_line !lineno lexbuf;
                          token lexbuf }
    | _                 { BADTOK }
    | eof               { EOF }

and comment =
  parse
      "*)"              { () }
    | "\n"              { incr lineno; Source.note_line !lineno lexbuf;
                          comment lexbuf }
    | _                 { comment lexbuf }
    | eof               { () }
```

E.1.2 parser.mly

```
/* lab1/parser.mly */
/* Copyright (c) 2017 J. M. Spivey */

%{
open Keiko
open Tree
%}

%token <Tree.ident>     IDENT
%token <Keiko.op>       MONOP MULOP ADDOP RELOP
%token <int>            NUMBER
%token                  SEMI DOT COLON LPAR RPAR COMMA MINUS VBAR
```

```
%token                  ASSIGN EOF BADTOK
%token                  BEGIN DO ELSE END IF THEN WHILE PRINT NEWLINE

%type <Tree.program>    program

%start                  program

%%

program :
    BEGIN stmts END DOT                 { Program $2 } ;

stmts :
    stmt_list                           { seq $1 } ;

stmt_list :
    stmt                                { [$1] }
  | stmt SEMI stmt_list                 { $1 :: $3 } ;

stmt :
    /* empty */                         { Skip }
  | name ASSIGN expr                    { Assign ($1, $3) }
  | PRINT expr                          { Print $2 }
  | NEWLINE                             { Newline }
  | IF expr THEN stmts END              { IfStmt ($2, $4, Skip) }
  | IF expr THEN stmts ELSE stmts END   { IfStmt ($2, $4, $6) }
  | WHILE expr DO stmts END             { WhileStmt ($2, $4) } ;

expr :
    simple                              { $1 }
  | expr RELOP simple                   { Binop ($2, $1, $3) } ;

simple :
    term                                { $1 }
  | simple ADDOP term                   { Binop ($2, $1, $3) }
  | simple MINUS term                   { Binop (Minus, $1, $3) } ;

term :
    factor                              { $1 }
  | term MULOP factor                   { Binop ($2, $1, $3) } ;

factor :
    name                                { Variable $1 }
  | NUMBER                              { Constant $1 }
  | MONOP factor                        { Monop ($1, $2) }
  | MINUS factor                        { Monop (Uminus, $2) }
  | LPAR expr RPAR                      { $2 } ;

name :
    IDENT                               { make_name $1 !Lexer.lineno } ;
```

E.1.3 tree.mli

```
(* lab1/tree.mli *)
(* Copyright (c) 2017 J. M. Spivey *)
```

```
type ident = string

(* |name| -- type for applied occurrences, with annotations *)
type name =
  { x_name: ident;              (* Name of the reference *)
    x_lab: string;              (* Global label *)
    x_line: int }               (* Line number *)

val make_name : ident -> int -> name


(* Abstract syntax *)

type program = Program of stmt

and stmt =
    Skip
  | Seq of stmt list
  | Assign of name * expr
  | Print of expr
  | Newline
  | IfStmt of expr * stmt * stmt
  | WhileStmt of expr * stmt

and expr =
    Constant of int
  | Variable of name
  | Monop of Keiko.op * expr
  | Binop of Keiko.op * expr * expr

(* seq -- neatly join a list of statements into a sequence *)
val seq : stmt list -> stmt

val print_tree : out_channel -> program -> unit
```

E.1.4 kgen.ml

```
(* lab1/kgen.ml *)
(* Copyright (c) 2017 J. M. Spivey *)

open Tree
open Keiko

(* |gen_expr| -- generate code for an expression *)
let rec gen_expr =
  function
      Constant x ->
        CONST x
    | Variable x ->
        SEQ [LINE x.x_line; LDGW x.x_lab]
    | Monop (w, e1) ->
        SEQ [gen_expr e1; MONOP w]
    | Binop (w, e1, e2) ->
        SEQ [gen_expr e1; gen_expr e2; BINOP w]
```

```
(* |gen_cond| -- generate code for short-circuit condition *)
let rec gen_cond e tlab flab =
  (* Jump to |tlab| if |e| is true and |flab| if it is false *)
  match e with
      Constant x ->
        if x <> 0 then JUMP tlab else JUMP flab
    | Binop ((Eq|Neq|Lt|Gt|Leq|Geq) as w, e1, e2) ->
        SEQ [gen_expr e1; gen_expr e2; JUMPC (w, tlab); JUMP flab]
    | Monop (Not, e1) ->
        gen_cond e1 flab tlab
    | Binop (And, e1, e2) ->
        let lab1 = label () in
        SEQ [gen_cond e1 lab1 flab; LABEL lab1; gen_cond e2 tlab flab]
    | Binop (Or, e1, e2) ->
        let lab1 = label () in
        SEQ [gen_cond e1 tlab lab1; LABEL lab1; gen_cond e2 tlab flab]
    | _ ->
        SEQ [gen_expr e; CONST 0; JUMPC (Neq, tlab); JUMP flab]

(* |gen_stmt| -- generate code for a statement *)
let rec gen_stmt s =
  match s with
      Skip -> NOP
    | Seq stmts -> SEQ (List.map gen_stmt stmts)
    | Assign (v, e) ->
        SEQ [LINE v.x_line; gen_expr e; STGW v.x_lab]
    | Print e ->
        SEQ [gen_expr e; CONST 0; GLOBAL "lib.print"; PCALL 1]
    | Newline ->
        SEQ [CONST 0; GLOBAL "lib.newline"; PCALL 0]
    | IfStmt (test, thenpt, elsept) ->
        let lab1 = label () and lab2 = label () and lab3 = label () in
        SEQ [gen_cond test lab1 lab2;
          LABEL lab1; gen_stmt thenpt; JUMP lab3;
          LABEL lab2; gen_stmt elsept; LABEL lab3]
    | WhileStmt (test, body) ->
        let lab1 = label () and lab2 = label () and lab3 = label () in
        SEQ [JUMP lab2; LABEL lab1; gen_stmt body;
          LABEL lab2; gen_cond test lab1 lab3; LABEL lab3]

(* optflag -- flag to control optimisation *)
let optflag = ref false

(* |translate| -- generate code for the whole program *)
let translate (Program ss) =
  let code = gen_stmt ss in
  Keiko.output (if !optflag then Peepopt.optimise code else code)
```

E.2 Lab two

E.2.1 check.ml

```
(* lab2/check.ml *)
(* Copyright (c) 2017 J. M. Spivey *)

open Print
open Keiko
open Tree
open Dict

(* |err_line| -- line number for error messages *)
let err_line = ref 1

(* |Semantic_error| -- exception raised if error detected *)
exception Semantic_error of string * Print.arg list * int

(* |sem_error| -- issue error message by raising exception *)
let sem_error fmt args =
  raise (Semantic_error (fmt, args, !err_line))

(* |lookup_def| -- find definition of a name, give error is none *)
let lookup_def x env =
  err_line := x.x_line;
  try let d = lookup x.x_name env in x.x_def <- Some d; d.d_type with
    Not_found -> sem_error "$ is not declared" [fStr x.x_name]

(* |add_def| -- add definition to env, give error if already declared *)
let add_def d env =
  try define d env with
    Exit -> sem_error "$ is already declared" [fStr d.d_tag]

(* |type_error| -- report a type error.  The message could be better. *)
let type_error () =
  sem_error "type mismatch in expression" []

(* |check_monop| -- check a unary operator and return its type *)
let check_monop w t =
  match w with
      Uminus ->
        if t <> Integer then type_error ();
        Integer
    | Not ->
        if t <> Boolean then type_error ();
        Boolean
    | _ -> failwith "bad monop"

(* |check_binop| -- check a binary operator and return its type *)
let check_binop w ta tb =
  match w with
      Plus | Minus | Times | Div | Mod ->
        if ta <> Integer || tb <> Integer then type_error ();
        Integer
    | Eq | Lt | Gt | Leq | Geq | Neq ->
```

```
          if ta <> tb then type_error ();
          Boolean
      | And | Or ->
          if ta <> Boolean || tb <> Boolean then type_error ();
          Boolean
      | _ -> failwith "bad binop"

  (* |check_expr| -- check and annotate an expression *)
  let rec check_expr e env =
    let t = expr_type e env in
    (e.e_type <- t; t)

  (* |expr_type| -- check an expression and return its type *)
  and expr_type e env =
    match e.e_guts with
        Variable x ->
          lookup_def x env
      | Sub (v, e) ->
          failwith "subscripts not implemented"
      | Constant (n, t) -> t
      | Monop (w, e1) ->
          let t = check_expr e1 env in
          check_monop w t
      | Binop (w, e1, e2) ->
          let ta = check_expr e1 env
          and tb = check_expr e2 env in
          check_binop w ta tb

  (* |check_stmt| -- check and annotate a statement *)
  let rec check_stmt s env =
    match s with
        Skip -> ()
      | Seq ss ->
          List.iter (fun s1 -> check_stmt s1 env) ss
      | Assign (lhs, rhs) ->
          let ta = check_expr lhs env
          and tb = check_expr rhs env in
          if ta <> tb then sem_error "type mismatch in assignment" []
      | Print e ->
          let t = check_expr e env in
          if t <> Integer then sem_error "print needs an integer" []
      | Newline ->
          ()
      | IfStmt (cond, thenpt, elsept) ->
          let t = check_expr cond env in
          if t <> Boolean then
            sem_error "boolean needed in if statement" [];
          check_stmt thenpt env;
          check_stmt elsept env
      | WhileStmt (cond, body) ->
          let t = check_expr cond env in
          if t <> Boolean then
            sem_error "need boolean after while" [];
          check_stmt body env
```

```
(* |make_def| -- construct definition of variable *)
let make_def x t a =
  { d_tag = x; d_type = t; d_lab = a }

(* |check_decl| -- check declaration and return extended environment *)
let check_decl (Decl (vs, t)) env0 =
  let declare x env =
    let lab = sprintf "_$" [fStr x.x_name] in
    let d = make_def x.x_name t lab in
    x.x_def <- Some d; add_def d env in
  Util.accum declare vs env0

(* |check_decls| -- check a sequence of declarations *)
let check_decls ds env0 =
  Util.accum check_decl ds env0

(* |annotate| -- check and annotate a program *)
let annotate (Program (ds, ss)) =
  let env = check_decls ds init_env in
  check_stmt ss env
```

E.2.2 kgen.ml

```
(* lab2/kgen.ml *)
(* Copyright (c) 2017 J. M. Spivey *)

open Dict
open Tree
open Keiko
open Print

let optflag = ref false

(* |line_number| -- find line number of variable reference *)
let rec line_number e =
  match e.e_guts with
      Variable x -> x.x_line
    | Sub (a, e) -> line_number a
    | _ -> 999

(* |gen_expr| -- generate code for an expression *)
let rec gen_expr e =
  match e.e_guts with
      Variable _ | Sub _ ->
        SEQ [gen_addr e; LOADW]
    | Constant (n, t) ->
        CONST n
    | Monop (w, e1) ->
        SEQ [gen_expr e1; MONOP w]
    | Binop (w, e1, e2) ->
        SEQ [gen_expr e1; gen_expr e2; BINOP w]

(* |gen_addr| -- generate code to push address of a variable *)
```

```
and gen_addr v =
  match v.e_guts with
      Variable x ->
        let d = get_def x in
        SEQ [LINE x.x_line; GLOBAL d.d_lab]
    | _ ->
        failwith "gen_addr"

(* |gen_cond| -- generate code for short-circuit condition *)
let rec gen_cond e tlab flab =
  (* Jump to |tlab| if |e| is true and |flab| if it is false *)
  match e.e_guts with
      Constant (x, t) ->
        if x <> 0 then JUMP tlab else JUMP flab
    | Binop ((Eq|Neq|Lt|Gt|Leq|Geq) as w, e1, e2) ->
        SEQ [gen_expr e1; gen_expr e2;
          JUMPC (w, tlab); JUMP flab]
    | Monop (Not, e1) ->
        gen_cond e1 flab tlab
    | Binop (And, e1, e2) ->
        let lab1 = label () in
        SEQ [gen_cond e1 lab1 flab; LABEL lab1; gen_cond e2 tlab flab]
    | Binop (Or, e1, e2) ->
        let lab1 = label () in
        SEQ [gen_cond e1 tlab lab1; LABEL lab1; gen_cond e2 tlab flab]
    | _ ->
        SEQ [gen_expr e; CONST 0; JUMPC (Neq, tlab); JUMP flab]

(* |gen_stmt| -- generate code for a statement *)
let rec gen_stmt =
  function
      Skip -> NOP
    | Seq stmts -> SEQ (List.map gen_stmt stmts)
    | Assign (v, e) ->
        SEQ [LINE (line_number v); gen_expr e; gen_addr v; STOREW]
    | Print e ->
        SEQ [gen_expr e; CONST 0; GLOBAL "lib.print"; PCALL 1]
    | Newline ->
        SEQ [CONST 0; GLOBAL "lib.newline"; PCALL 0]
    | IfStmt (test, thenpt, elsept) ->
        let lab1 = label () and lab2 = label () and lab3 = label () in
        SEQ [gen_cond test lab1 lab2;
          LABEL lab1; gen_stmt thenpt; JUMP lab3;
          LABEL lab2; gen_stmt elsept; LABEL lab3]
    | WhileStmt (test, body) ->
        let lab1 = label () and lab2 = label () and lab3 = label () in
        SEQ [JUMP lab2; LABEL lab1; gen_stmt body;
          LABEL lab2; gen_cond test lab1 lab3; LABEL lab3]

(* |gen_decl| -- reserve space for global variable *)
let gen_decl (Decl (xs, t)) =
  List.iter (fun x ->
      let d = get_def x in
      let s = 4 in
      printf "GLOVAR $ $\n" [fStr d.d_lab; fNum s]) xs
```

```
(* |translate| -- generate code for the whole program *)
let translate (Program (ds, ss)) =
  let code = gen_stmt ss in
  printf "FUNC MAIN 0\n" [];
  Keiko.output (if !optflag then Peepopt.optimise code else code);
  printf "RETURN\n" [];
  printf "END\n\n" [];
  List.iter gen_decl ds
```

E.3 Lab three

E.3.1 check.ml

```
(* lab3/check.ml *)
(* Copyright (c) 2017 J. M. Spivey *)

open Tree
open Dict
open Print

(* |err_line| -- line number for error messages *)
let err_line = ref 1

(* |Semantic_error| -- exception raised if error detected *)
exception Semantic_error of string * Print.arg list * int

(* |sem_error| -- issue error message by raising exception *)
let sem_error fmt args =
  raise (Semantic_error (fmt, args, !err_line))

(* |lookup_def| -- find definition of a name, give error is none *)
let lookup_def x env =
  err_line := x.x_line;
  try let d = lookup x.x_name env in x.x_def <- Some d; d with
    Not_found -> sem_error "$ is not declared" [fStr x.x_name]

(* |add_def| -- add definition to env, give error if already declared *)
let add_def d env =
  try define d env with
    Exit -> sem_error "$ is already declared" [fStr d.d_tag]

(* |check_expr| -- check and annotate an expression *)
let rec check_expr e env =
  match e with
      Constant n -> ()
    | Variable x ->
        let d = lookup_def x env in
        begin
          match d.d_kind with
              VarDef -> ()
            | ProcDef _ ->
                sem_error "$ is not a variable" [fStr x.x_name]
        end
```

```
      | Monop (w, e1) ->
          check_expr e1 env
      | Binop (w, e1, e2) ->
          check_expr e1 env;
          check_expr e2 env
      | Call (p, args) ->
          let d = lookup_def p env in
          begin
            match d.d_kind with
                VarDef ->
                  sem_error "$ is not a procedure" [fStr p.x_name]
              | ProcDef nargs ->
                  if List.length args <> nargs then
                    sem_error "procedure $ needs $ arguments"
                      [fStr p.x_name; fNum nargs];
          end;
          List.iter (fun e1 -> check_expr e1 env) args

  (* |check_stmt| -- check and annotate a statement *)
  let rec check_stmt s inproc env =
    match s with
        Skip -> ()
      | Seq ss ->
          List.iter (fun s1 -> check_stmt s1 inproc env) ss
      | Assign (x, e) ->
          let d = lookup_def x env in
          begin
            match d.d_kind with
                VarDef -> check_expr e env
              | ProcDef _ ->
                  sem_error "$ is not a variable" [fStr x.x_name]
          end
      | Return e ->
          if not inproc then
            sem_error "return statement only allowed in procedure" [];
          check_expr e env
      | IfStmt (test, thenpt, elsept) ->
          check_expr test env;
          check_stmt thenpt inproc env;
          check_stmt elsept inproc env
      | WhileStmt (test, body) ->
          check_expr test env;
          check_stmt body inproc env
      | Print e ->
          check_expr e env
      | Newline ->
          ()

  (* |serialize| -- number a list, starting from 0 *)
  let serialize xs =
    let rec count i =
      function
          [] -> []
        | x :: xs -> (i, x) :: count (i+1) xs in
    count 0 xs
```

```
(*
Frame layout

        arg n
        ...
fp+16:  arg 1
fp+12:  static link
fp+8:   current cp
fp+4:   return addr
fp:     dynamic link
fp-4:   local 1
        ...
        local m
*)

let arg_base = 16
let loc_base = 0

(* |name_stack| -- stack of nested procedure names *)
let name_stack = ref []

(* |proc_symbol| -- label for current procedure *)
let proc_symbol x =
  "_" ^ String.concat "." (List.rev (x::!name_stack))

(* |declare_local| -- declare a formal parameter or local *)
let declare_local x lev off env =
  let d = { d_tag = x; d_kind = VarDef; d_level = lev;
            d_label = ""; d_offset = off } in
  add_def d env

(* |declare_global| -- declare a global variable *)
let declare_global x env =
  let d = { d_tag = x; d_kind = VarDef; d_level = 0;
            d_label = sprintf "_$" [fStr x]; d_offset = 0 } in
  add_def d env

(* |declare_proc| -- declare a procedure *)
let declare_proc (Proc (p, formals, body)) lev env =
  let lab = proc_symbol p.x_name in
  let d = { d_tag = p.x_name; d_kind = ProcDef (List.length formals);
            d_level = lev; d_label = lab; d_offset = 0 } in
  p.x_def <- Some d; add_def d env

(* |check_proc| -- check a procedure body *)
let rec check_proc (Proc (p, formals, Block (vars, procs, body))) lev env =
  err_line := p.x_line;
  name_stack := p.x_name :: !name_stack;
  let env' =
    Util.accum (fun (i, x) -> declare_local x lev (arg_base + 4*i))
      (serialize formals) (new_block env) in
  let env'' =
    Util.accum (fun (i, x) -> declare_local x lev (loc_base - 4*(i+1)))
      (serialize vars) env' in
```

```
  let env''' =
    Util.accum (fun d -> declare_proc d (lev+1)) procs env'' in
  List.iter (fun d -> check_proc d (lev+1) env''') procs;
  check_stmt body true env''';
    name_stack := List.tl !name_stack

(* |annotate| -- check and annotate a program *)
let annotate (Program (Block (vars, procs, body))) =
  let env = Util.accum declare_global vars empty in
  let env' = Util.accum (fun d -> declare_proc d 1) procs env in
  List.iter (fun d -> check_proc d 1 env') procs;
  check_stmt body false env'
```

E.3.2 kgen.ml

```
(* lab3/kgen.ml *)
(* Copyright (c) 2017 J. M. Spivey *)

open Tree
open Dict
open Keiko
open Print

let optflag = ref false

let level = ref 0

let slink = 12

(* |gen_addr| -- generate code to push address of a variable *)
let gen_addr d =
  if d.d_level = 0 then
    GLOBAL d.d_label
  else
    failwith "local variables not implemented yet"

(* |gen_expr| -- generate code for an expression *)
let rec gen_expr =
  function
      Variable x ->
        let d = get_def x in
        begin
          match d.d_kind with
              VarDef ->
                SEQ [LINE x.x_line; gen_addr d; LOADW]
            | ProcDef nargs ->
                failwith "no procedure values"
        end
    | Constant x ->
        CONST x
    | Monop (w, e1) ->
        SEQ [gen_expr e1; MONOP w]
    | Binop (w, e1, e2) ->
        SEQ [gen_expr e1; gen_expr e2; BINOP w]
    | Call (p, args) ->
```

```
          SEQ [LINE p.x_line;
            failwith "no procedure call"]

  (* |gen_cond| -- generate code for short-circuit condition *)
let rec gen_cond e tlab flab =
  (* Jump to |tlab| if |e| is true and |flab| if it is false *)
  match e with
      Constant x ->
        if x <> 0 then JUMP tlab else JUMP flab
    | Binop ((Eq|Neq|Lt|Gt|Leq|Geq) as w, e1, e2) ->
        SEQ [gen_expr e1; gen_expr e2;
          JUMPC (w, tlab); JUMP flab]
    | Monop (Not, e1) ->
        gen_cond e1 flab tlab
    | Binop (And, e1, e2) ->
        let lab1 = label () in
        SEQ [gen_cond e1 lab1 flab; LABEL lab1; gen_cond e2 tlab flab]
    | Binop (Or, e1, e2) ->
        let lab1 = label () in
        SEQ [gen_cond e1 tlab lab1; LABEL lab1; gen_cond e2 tlab flab]
    | _ ->
        SEQ [gen_expr e; CONST 0; JUMPC (Neq, tlab); JUMP flab]

  (* |gen_stmt| -- generate code for a statement *)
let rec gen_stmt =
  function
      Skip -> NOP
    | Seq ss ->
        SEQ (List.map gen_stmt ss)
    | Assign (v, e) ->
        let d = get_def v in
        begin
          match d.d_kind with
              VarDef ->
                SEQ [gen_expr e; gen_addr d; STOREW]
            | _ -> failwith "assign"
        end
    | Print e ->
        SEQ [gen_expr e; CONST 0; GLOBAL "lib.print"; PCALL 1]
    | Newline ->
        SEQ [CONST 0; GLOBAL "lib.newline"; PCALL 0]
    | IfStmt (test, thenpt, elsept) ->
        let lab1 = label () and lab2 = label () and lab3 = label () in
        SEQ [gen_cond test lab1 lab2;
          LABEL lab1; gen_stmt thenpt; JUMP lab3;
          LABEL lab2; gen_stmt elsept; LABEL lab3]
    | WhileStmt (test, body) ->
        let lab1 = label () and lab2 = label () and lab3 = label () in
        SEQ [JUMP lab2; LABEL lab1; gen_stmt body;
          LABEL lab2; gen_cond test lab1 lab3; LABEL lab3]
    | Return e ->
        failwith "no return statement"

  (* |gen_proc| -- generate code for a procedure *)
let rec gen_proc (Proc (p, formals, Block (vars, procs, body))) =
```

```
  let d = get_def p in
  level := d.d_level;
  let code = gen_stmt body in
  printf "FUNC $ $\n" [fStr d.d_label; fNum (4 * List.length vars)];
  Keiko.output (if !optflag then Peepopt.optimise code else code);
  printf "ERROR E_RETURN 0\n" [];
  printf "END\n\n" [];
  List.iter gen_proc procs

(* |translate| -- generate code for the whole program *)
let translate (Program (Block (vars, procs, body))) =
  level := 0;
  printf "FUNC MAIN 0\n" [];
  Keiko.output (gen_stmt body);
  printf "RETURN\n" [];
  printf "END\n\n" [];
  List.iter gen_proc procs;
  List.iter (function x -> printf "GLOVAR _$ 4\n" [fStr x]) vars
```

E.4 Lab four

E.4.1 target.mli

```
(* lab4/target.mli *)
(* Copyright (c) 2017 J. M. Spivey *)

open Optree

(* |reg| -- ARM registers *)
type reg = R of int | R_fp | R_any | R_temp | R_none

(* |fReg| -- format register for printing *)
val fReg : reg -> Print.arg

(* |volatile| -- list of caller-save registers *)
val volatile : reg list

(* |stable| -- list of callee-save registers *)
val stable : reg list



(* FRAGMENTS *)

type fragment

(* Basic fragments *)
val register : reg -> fragment
val number : int -> fragment
val symbol : symbol -> fragment
val codelab : codelab -> fragment
val quote : string -> fragment

(* Join multiple fragments into one *)
```

```
val frag : string -> fragment list -> fragment

val reg_of : fragment -> reg
val get_regs : fragment -> reg list


(* CODE OUTPUT *)

(* |emit_inst| -- emit an assembly language instruction *)
val emit_inst : string -> reg -> fragment list -> unit

(* |emit_move| -- emit a reg-to-reg move *)
val emit_move : reg -> reg -> unit

(* |emit_lab| -- place a label *)
val emit_lab : codelab -> unit

(* |emit_comment| -- insert a comment *)
val emit_comment : string -> unit

(* |emit_tree| -- Print an optree as a comment *)
val emit_tree : optree -> unit

(* |need_stack| -- ensure stack space *)
val need_stack : int -> unit

(* |preamble| -- emit first part of assembly language output *)
val preamble : unit -> unit

(* |postamble| -- emit last part of assembly language output *)
val postamble : unit -> unit

(* |start_proc| -- emit beginning of procedure *)
val start_proc : symbol -> int -> int -> unit

(* |end_proc| -- emit end of procedure *)
val end_proc : unit -> unit

(* |flush_proc| -- dump out code after failure *)
val flush_proc : unit -> unit

(* |emit_string| -- emit assembler code for string constant *)
val emit_string : symbol -> string -> unit

(* |emit_global| -- emit assembler code to define global variable *)
val emit_global : symbol -> int -> unit
```

E.4.2 tran.ml

```
(* lab4/tran.ml *)
(* Copyright (c) 2017 J. M. Spivey *)

open Optree
open Target
open Regs
```

```
open Print

let debug = ref 0

(* |release| -- release all registers used by a fragment *)
let release frag = List.iter release_reg (get_regs frag)

(* |gen| -- emit an instruction with register allocation *)
let gen fmt rands =
  List.iter release rands;
  emit_inst fmt R_none rands

(* |gen_reg| -- emit instruction with result in a register *)
let gen_reg fmt r rands =
  (* Assume the instruction does not write its output register
     before it has finished reading its inputs *)
  (* First release the input registers *)
  List.iter release rands;
  (* Then allocate the output register, so it can overlap an input *)
  let r' = get_reg r in
  emit_inst fmt r' rands;
  register r'

(* |gen_move| -- move value to specific register *)
let gen_move dst src =
  if dst = R_any || dst = R_temp || dst = src then
    register src
  else
    gen_reg "mov" dst [register src]


(* Tests for fitting in various immediate fields *)

(* |fits_offset| -- test for fitting in offset field of address *)
let fits_offset x = (-4096 < x && x < 4096)

(* |fits_immed| -- test for fitting in immediate field *)
let fits_immed x =
  (* A conservative approximation, using shifts instead of rotates *)
  let rec reduce r =
    if r land 3 <> 0 then r else reduce (r lsr 2) in
  x = 0 || x > 0 && reduce x < 256

(* |fits_move| -- test for fitting in immediate move *)
let fits_move x = fits_immed x || fits_immed (lnot x)

(* |fits_add| -- test for fitting in immediate add *)
let fits_add x = fits_immed x || fits_immed (-x)

(* |line| -- current line number *)
let line = ref 0

(* The main part of the code generator consists of a family of functions
   eval_X t, each generating code for a tree t, leaving the value in
   a register, or as an operand for another instruction, etc. *)
```

```
(* |eval_reg| -- evaluate expression with result in specified register *)
let rec eval_reg t r =
  (* Binary operation *)
  let binary op t1 t2 =
    let v1 = eval_reg t1 R_any in
    let v2 = eval_rand t2 in
    gen_reg op r [v1; v2]

  (* Shifts *)
  and shift op t1 t2 =
    let v1 = eval_reg t1 R_any in
    let v2 = eval_shift t2 in
    gen_reg op r [v1; v2]

  (* Unary operation *)
  and unary op t1 =
    let v1 = eval_reg t1 R_any in
    gen_reg op r [v1]

  (* Comparison with boolean result *)
  and compare op t1 t2 =
    let v1 = eval_reg t1 R_any in
    let v2 = eval_rand t2 in
    gen_reg "cmp $1, $2 / mov $0, #0 / mov$3 $0, #1"
      r [v1; v2; quote op] in

  match t with
      <CONST k> when fits_move k ->
        gen_reg "mov $0, #$1" r [number k]
    | <CONST k> ->
        gen_reg "ldr $0, =$1" r [number k]
    | <LOCAL 0> ->
        gen_move r R_fp
    | <LOCAL n> when fits_add n ->
        gen_reg "add $0, fp, #$1" r [number n]
    | <LOCAL n> ->
        gen_reg "ldr ip, =$1 / add $0, fp, ip" r [number n]
    | <GLOBAL x> ->
        gen_reg "ldr $0, =$1" r [symbol x]
    | <TEMP n> ->
        gen_move r (Regs.use_temp n)
    | <(LOADW|LOADC), <REGVAR i>> ->
        let rv = List.nth stable i in
        reserve_reg rv; gen_move r rv
    | <LOADW, t1> ->
        let v1 = eval_addr t1 in
        gen_reg "ldr" r [v1]
    | <LOADC, t1> ->
        let v1 = eval_addr t1 in
        gen_reg "ldrb" r [v1]

    | <MONOP Uminus, t1> -> unary "neg" t1
    | <MONOP Not, t1> ->
        let v1 = eval_reg t1 R_any in
```

```
          gen_reg "eor $0, $1, #1" r [v1]
      | <MONOP BitNot, t1> -> unary "mvn" t1

      | <OFFSET, t1, <CONST k>> when fits_add k ->
          (* Allow add for negative constants *)
          let v1 = eval_reg t1 R_any in
          gen_reg "add $0, $1, #$2" r [v1; number k]
      | <OFFSET, t1, t2> -> binary "add" t1 t2

      | <BINOP Plus, t1, t2> -> binary "add" t1 t2
      | <BINOP Minus, t1, t2> -> binary "sub" t1 t2
      | <BINOP And, t1, t2> -> binary "and" t1 t2
      | <BINOP Or, t1, t2> -> binary "orr" t1 t2
      | <BINOP Lsl, t1, t2> -> shift "lsl" t1 t2
      | <BINOP Lsr, t1, t2> -> shift "lsr" t1 t2
      | <BINOP Asr, t1, t2> -> shift "asr" t1 t2
      | <BINOP BitAnd, t1, <MONOP BitNot, t2>> ->
          binary "bic" t1 t2
      | <BINOP BitAnd, <MONOP BitNot, t1>, t2> ->
          binary "bic" t2 t1
      | <BINOP BitAnd, t1, t2> -> binary "and" t1 t2
      | <BINOP BitOr, t1, t2> -> binary "orr" t1 t2

      | <BINOP Times, t1, t2> ->
          (* The mul instruction needs both operands in registers *)
          let v1 = eval_reg t1 R_any in
          let v2 = eval_reg t2 R_any in
          gen_reg "mul" r [v1; v2]

      | <BINOP Eq, t1, t2> -> compare "eq" t1 t2
      | <BINOP Neq, t1, t2> -> compare "ne" t1 t2
      | <BINOP Gt, t1, t2> -> compare "gt" t1 t2
      | <BINOP Geq, t1, t2> -> compare "ge" t1 t2
      | <BINOP Lt, t1, t2> -> compare "lt" t1 t2
      | <BINOP Leq, t1, t2> -> compare "le" t1 t2

      | <BOUND, t1, t2> ->
          let v1 = eval_reg t1 r in
          let v2 = eval_rand t2 in
          gen_reg "cmp $1, $2 / ldrhs r0, =$3 / blhs check"
            (reg_of v1) [v1; v2; number !line]

      | <NCHECK, t1> ->
          let v1 = eval_reg t1 r in
          gen_reg "cmp $1, #0 / ldreq r0, =$2 / bleq nullcheck"
            (reg_of v1) [v1; number !line]

      | <w, @args> ->
          failwith (sprintf "eval $" [fInst w])

  (* |eval_rand| -- evaluate to form second operand *)
  and eval_rand t =
    match t with
        <CONST k> when fits_immed k -> frag "#$1" [number k]
      | _ -> eval_reg t R_any
```

```
(* |eval_shift| -- evaluate to form shift amount *)
and eval_shift =
  function
      <CONST k> when k >= 1 && k < 32 -> frag "#$1" [number k]
    | t -> eval_reg t R_any

(* |eval_addr| -- evaluate to form an address for ldr or str *)
and eval_addr =
  function
      <LOCAL n> when fits_offset n ->
        frag "[fp, #$1]" [number n]
    | <OFFSET, t1, <CONST k>> when fits_offset k ->
        let v1 = eval_reg t1 R_any in
        frag "[$1, #$2]" [v1; number k]
    | t ->
        let v1 = eval_reg t R_any in
        frag "[$1]" [v1]

(* |exec_call| -- execute procedure call *)
let exec_call =
  function
      <GLOBAL f> ->
        gen "bl" [symbol f]
    | t ->
        let v1 = eval_reg t R_any in
        gen "blx" [v1]

(* |exec_stmt| -- generate code to execute a statement *)
let exec_stmt t =
  (* Conditional jump *)
  let condj op lab t1 t2 =
    let v1 = eval_reg t1 R_any in
    let v2 = eval_rand t2 in
    gen "cmp $1, $2 / b$3 $4" [v1; v2; quote op; codelab lab] in

  (* Procedure call *)
  let call t =
    spill_temps volatile;           (* Spill any remaining temps *)
    exec_call t;              (* Call the function *)
    List.iter (function r ->  (* Release argument registers *)
      if not (is_free r) then release_reg r) volatile in

  match t with
      <CALL k, t1> ->
        call t1

    | <DEFTEMP n, <CALL k, t1>> ->
        call t1;
        reserve_reg (R 0);
        Regs.def_temp n (R 0)

    | <DEFTEMP n, t1> ->
        let v1 = eval_reg t1 R_temp in
        Regs.def_temp n (reg_of v1)
```

```
      | <(STOREW|STOREC), t1, <REGVAR i>> ->
          let rv = List.nth stable i in
          spill_temps [rv];
          release (eval_reg t1 rv)
      | <STOREW, t1, t2> ->
          let v1 = eval_reg t1 R_any in
          let v2 = eval_addr t2 in
          gen "str" [v1; v2]
      | <STOREC, t1, t2> ->
          let v1 = eval_reg t1 R_any in
          let v2 = eval_addr t2 in
          gen "strb" [v1; v2]

      | <RESULTW, t1> ->
          release (eval_reg t1 (R 0))

      | <LABEL lab> -> emit_lab lab

      | <JUMP lab> -> gen "b" [codelab lab]

      | <JUMPC (Eq, lab), t1, t2> -> condj "eq" lab t1 t2
      | <JUMPC (Lt, lab), t1, t2> -> condj "lt" lab t1 t2
      | <JUMPC (Gt, lab), t1, t2> -> condj "gt" lab t1 t2
      | <JUMPC (Leq, lab), t1, t2> -> condj "le" lab t1 t2
      | <JUMPC (Geq, lab), t1, t2> -> condj "ge" lab t1 t2
      | <JUMPC (Neq, lab), t1, t2> -> condj "ne" lab t1 t2

      | <JCASE (table, deflab), t1> ->
          (* This jump table code exploits the fact that on ARM,
             reading the pc gives a value 8 bytes beyond the
             current instruction, so in the ldrlo instruction
             below, pc points to the branch table. *)
          let v1 = eval_reg t1 R_any in
          gen "cmp $1, #$2 / ldrlo pc, [pc, $1, LSL #2] / b $3"
            [v1; number (List.length table); codelab deflab];
          List.iter (fun lab -> gen ".word $1" [codelab lab]) table

      | <ARG i, <TEMP k>> when i < 4 ->
          (* Avoid annoying spill and reload if the value is a temp
             already in the correct register: e.g. in f(g(x)). *)
          let r = R i in
          let r1 = Regs.use_temp k in
          spill_temps [r];
          ignore (gen_move r r1)
      | <ARG i, t1> when i < 4 ->
          spill_temps [R i];
          ignore (eval_reg t1 (R i))
      | <ARG i, t1> when i >= 4 ->
          need_stack (4*i-12);
          let v1 = eval_reg t1 R_any in
          gen "str $1, [sp, #$2]" [v1; number (4*i-16)]

      | <STATLINK, t1> ->
          let r = R 10 in
```

```
            spill_temps [r];
            ignore (eval_reg t1 r)

285       | <w, @ts> ->
            failwith (sprintf "exec_stmt $" [fInst w])

    (* |process| -- generate code for a statement, or note a line number *)
    let process =
290   function
          <LINE n> ->
            if !line <> n then
              emit_comment (String.trim (Source.get_line n));
            line := n
295       | t ->
            if !debug > 0 then emit_tree t;
            exec_stmt t;
            if !debug > 1 then emit_comment (Regs.dump_regs ())

300 (* |translate| -- translate a procedure body *)
    let translate code =
      (try List.iter process code with exc ->
        (* Code generation failed, but let's see how far we got *)
        Target.flush_proc (); raise exc)
```