Compilers

Coursebook

Mike Spivey
## Contents

### Preface

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 1 Syntax

<table>
<thead>
<tr>
<th>Lecture</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Lecture 1: Introduction</td>
</tr>
<tr>
<td>1.2</td>
<td>Lecture 2: Lexical analysis</td>
</tr>
<tr>
<td>1.3</td>
<td>Lecture 3: Syntax analysis</td>
</tr>
</tbody>
</table>

### 2 Expressions and statements

<table>
<thead>
<tr>
<th>Lecture</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Lecture 4: Code for expressions</td>
</tr>
<tr>
<td>2.2</td>
<td>Lecture 5: Control structures</td>
</tr>
</tbody>
</table>

### 3 Lab one: Control structures

<table>
<thead>
<tr>
<th>Lecture</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Setting up the labs</td>
</tr>
<tr>
<td>3.2</td>
<td>Compiler structure</td>
</tr>
<tr>
<td>3.3</td>
<td>Building and running the compiler</td>
</tr>
<tr>
<td>3.4</td>
<td>Implementing repeat statements</td>
</tr>
<tr>
<td>3.5</td>
<td>Implementing loop statements</td>
</tr>
<tr>
<td>3.6</td>
<td>Implementing case statements (optional)</td>
</tr>
</tbody>
</table>

### 4 Types and data structures

<table>
<thead>
<tr>
<th>Lecture</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Lecture 6: Semantic analysis</td>
</tr>
<tr>
<td>4.2</td>
<td>Lecture 7: Data structure access</td>
</tr>
</tbody>
</table>

### 5 Lab two: Array access

<table>
<thead>
<tr>
<th>Lecture</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Compiler structure</td>
</tr>
<tr>
<td>5.2</td>
<td>Functions on types</td>
</tr>
<tr>
<td>5.3</td>
<td>Semantic analysis</td>
</tr>
<tr>
<td>5.4</td>
<td>Code generation</td>
</tr>
<tr>
<td>5.5</td>
<td>Testing the compiler</td>
</tr>
<tr>
<td>5.6</td>
<td>Run-time checks and optimisation (optional)</td>
</tr>
</tbody>
</table>

### 6 Subroutines

<table>
<thead>
<tr>
<th>Lecture</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Lecture 8: Subroutines</td>
</tr>
<tr>
<td>6.2</td>
<td>Lecture 9: Parameters and nesting</td>
</tr>
<tr>
<td>6.3</td>
<td>Lecture 10: Higher-order functions</td>
</tr>
<tr>
<td>6.4</td>
<td>Lecture 11: A complete compiler</td>
</tr>
</tbody>
</table>
7 Lab three: Subroutines
7.1 Using the lab kit
7.2 Frame layout
7.3 Procedure calls
7.4 Parameters and local variables
7.5 Nested procedures
7.6 Functional parameters (optional)

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

9 Lab four: Machine code
9.1 Building and testing the compiler
9.2 More addressing modes
9.3 Shifter operands
9.4 Multiplying by a constant
9.5 Going the other way

10 Revision problems

Appendices
A OCaml basics
B The Keiko machine
C A rough guide to ARM
D A machine grammar for ARM
E Code listings
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, completely optional, 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 16
    LDLW 20
    JEQ L2

L2: RETURN
```

```
L3: LDLW 16
    LDLW 20
    JLEQ L3

L3: LDLW 20
    LDLW 16
    SUB
    STLW 16
    JUMP L4
```

```
L4: JUMP L1
L2: RETURN
```

Figure 1.1: Postfix code for GCD
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, op-
erators, 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:

\[
\begin{align*}
\epsilon & \text{ empty string} \\
\alpha & \text{ literal character} \\
E_1 E_2 & \text{ concatenation} \\
E_1 \mid E_2 & \text{ union} \\
E_1^* & \text{ closure } = \epsilon \mid E_1 \mid E_1 E_1 \mid E_1 E_1 E_1 \mid \ldots \\
[a\cdots z] & \text{ any character except } a, b, c, z \\
[a\cdots z0\cdots 9] & \text{ any character at all}
\end{align*}
\]

and also

\[
\begin{align*}
[a\cdots z] & = a \mid b \mid c \\
[a\cdots z0\cdots 9] & = a \mid b \mid \ldots \mid z \\
E_1^? & = \epsilon \mid E_1 \\
E_1^+ & = E_1 E_1^* \\
[^{\wedge}abc] & \text{ any character except } a, b, c \\
\cdot & \text{ any character at all}
\end{align*}
\]

Examples:
• \([A\cdots z][A\cdots z0\cdots 9]^*\) - identifiers in Pascal.
• \(-?[0\cdots 9]+[0\cdots 9a\cdots f]^+\) - integer constants in C.
• \("[^{\wedge}]{1,}^*\")\(^*\) - 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:
4 Syntax

![Diagram of NFA for integer constants]

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 $s_{\emptyset}$, to which all otherwise undefined transitions are taken to lead.

![Diagram of DFA for integer constants]

Figure 1.4: DFA for integer constants

What Lex does:

- The input script is a list of rules containing regexps $E_1, E_2, \ldots, E_n$.
- Make a DFA for the united regexp $E_1 | E_2 | \ldots | E_n$, 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:

```c
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 \textit{delta} array is sparse, so to save space it is packed into a special data structure, still with constant-time access. For efficiency, read the input file in big chunks to avoid too many system calls.

\textit{Example:} lab1/lexer.mll (see page 143) contains a lexer for a small language. Describe comments by a regexp or (in Ocamllex) by using tail recursion to make a loop.

\textbf{Data type of tokens:}

\begin{verbatim}
  type Token =
    IDENT of string |
    NUMBER of int |
    IF | THEN | ELSE | ...
    COMMA | ASSIGN | ...
    ADDOP of op | MULOP of op | ...
    BADTOK | EOF
\end{verbatim}

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

\subsection{1.3 Lecture 3: Syntax analysis}

In most programming languages, the larger-scale structure of programs can be captured by a context free grammar (CFG). A CFG has an alphabet $\Sigma$ of terminal symbols or tokens, variables $N$ (also called non-terminals), a starting symbol $S \in N$ and a set of productions $A \rightarrow \alpha$, where $A \in N$ and $\alpha$ is a string over $N \cup \Sigma$.

For example, $G_1$ is a grammar with alphabet \{+, *, id\} and one variable $E$, and the productions \{$E \rightarrow \text{id}$, $E \rightarrow E + E$, $E \rightarrow E \ast E$\}.

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

\textit{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 $\alpha$ of a production $A \rightarrow \alpha$. E.g. for $G_1$,

\begin{center}
\begin{tikzpicture}
  \node (E) {$E$}
  child {node (id) {$\text{id}$}
    edge from parent node [left] {}}
  child {node (E) {$E$}
    child {node (id) {$\text{id}$}
      edge from parent node [left] {}}
    child {node (E) {$E$}
      child {node (id) {$\text{id}$}
        edge from parent node [left] {}}
      child {node (id) {$\text{id}$}
        edge from parent node [left] {}}
      edge from parent node [left] {}}
    edge from parent node [left] {}}
  child {node (E) {$E$}
    child {node (id) {$\text{id}$}
      edge from parent node [left] {}}
    child {node (id) {$\text{id}$}
      edge from parent node [left] {}}
    edge from parent node [left] {}}
  edge from parent node [left] {}}
\end{tikzpicture}
\end{center}

\textbf{Figure 1.5: A derivation tree}

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

Grammar $G_1$ is \textit{ambiguous} because some sentences, such as $\text{id} + \text{id} \ast \text{id}$, have more than one derivation tree.
An unambiguous grammar $G_2$ for the same language:

$$
E \rightarrow T \mid E + T \\
T \rightarrow F \mid T * F \\
F \rightarrow id
$$

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 $\alpha$ of some production $A \rightarrow \alpha$.

**Parsing problem:** for a grammar $G$. Given $s \in \Sigma^*$, determine whether $s \in L(G)$, and produce a derivation (tree) $S \Rightarrow^* s$.

(Non-deterministic) bottom-up parsing machine:

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\epsilon$</td>
<td>id + id * id</td>
<td>shift</td>
</tr>
<tr>
<td>id</td>
<td>+ id * id</td>
<td>reduce $F \rightarrow id$</td>
</tr>
<tr>
<td>$F$</td>
<td>+ id * id</td>
<td>reduce $T \rightarrow F$</td>
</tr>
<tr>
<td>$T$</td>
<td>+ id * id</td>
<td>reduce $E \rightarrow T$</td>
</tr>
<tr>
<td>$E$</td>
<td>+ id * id</td>
<td>shift</td>
</tr>
<tr>
<td>$E +$</td>
<td>id * id</td>
<td>shift</td>
</tr>
<tr>
<td>$E + id$</td>
<td>* id</td>
<td>reduce $F \rightarrow id$</td>
</tr>
<tr>
<td>$E + F$</td>
<td>* id</td>
<td>reduce $T \rightarrow F$</td>
</tr>
<tr>
<td>$E + T$</td>
<td>* id</td>
<td>shift</td>
</tr>
<tr>
<td>$E + T *$</td>
<td>id</td>
<td>shift</td>
</tr>
<tr>
<td>$E + T * id$</td>
<td>$\epsilon$</td>
<td>reduce $F \rightarrow id$</td>
</tr>
<tr>
<td>$E + T * F$</td>
<td>$\epsilon$</td>
<td>reduce $T \rightarrow T * F$</td>
</tr>
<tr>
<td>$E + T$</td>
<td>$\epsilon$</td>
<td>reduce $E \rightarrow E + T$</td>
</tr>
<tr>
<td>$E$</td>
<td>$\epsilon$</td>
<td>accept</td>
</tr>
</tbody>
</table>

Each derivation tree corresponds to a unique rightmost derivation; and the NBUPM plays out a rightmost derivation in reverse:

$$
E \Rightarrow E + T \Rightarrow E + T * F \Rightarrow E + T * id \Rightarrow E + F * id \Rightarrow F + id * id
$$

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.

\[
\text{type expr} = \text{Variable of string} \\
\quad | \text{Binop of op * expr * expr}
\]

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

\[
\begin{align*}
\text{expr} & : \text{expr PLUS term} \quad \{ \text{Binop (Plus, $1, $3)} \} \\
& \quad | \text{term} \quad \{ $1 \}; \\
\text{term} & : \text{term TIMES factor} \quad \{ \text{Binop (Times, $1, $3)} \} \\
& \quad | \text{factor} \quad \{ $1 \}; \\
\text{factor} & : \text{IDENT} \quad \{ \text{Variable $1} \};
\end{align*}
\]

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:

![Diagram of trees on the parser stack](image)

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). The OCaml library includes an interpreter for the lexer and parser tables.

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 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:

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

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

1.2 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?

1.3 Lex has the conventions that the longest match wins, and that earlier
Figure 1.9: Abbreviated syntax for chains of else if’s

rules have higher priority than later ones in the script. These conventions are
exploited in the lexer shown in Exercise 1.1 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 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.

1.5 The following productions for if statements appeared in the original
definition of Algol-60:

\[
\text{stmt} \rightarrow \text{basic-stmt} \\
| \text{if expr then stmt} \\
| \text{if expr then stmt else stmt}.
\]

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 differ-
ent productions. Find a string with two parse trees according to the gram-
mar, and show the parser state where two actions are possible. Describe the
results of shifting and of reducing in this state.

1.6 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. Arrange for the parser to build
the same abstract syntax tree for the abbreviated program as it would for its
equivalent written without elsif.
10 Syntax

1.7 One grammar for lists of identifiers contains the productions,

\[
\text{idlist} \rightarrow \text{id} \\
| \text{idlist }, \text{id}
\]

(we call this \textit{left} recursive), and another (\textit{right} recursive) contains the productions,

\[
\text{idlist} \rightarrow \text{id} \\
| \text{id}, \text{idlist}
\]

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 \textit{cons}?

1.8 [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. In this form, unary and binary operators are written before their operands, and there are no parentheses. Thus the expression normally written \( b * b - 4 * a * c \) would be written

\[- * b b * * 4 a c,\]

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

\[* + x y - x y.\]

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

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

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

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

1.9 [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:

2011, “Colin Firth”

so called after the Polish logician Jan Łukasiewicz.
The following grammar describes the syntax of such files:

```ocaml
%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 } ;
```

Ocamlyacc reports a shift/reduce conflict for this grammar.

(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 string (* LDGW x *)
| Stgw of string (* 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:*

```
while (!halted) {
  switch (*pc++) {
    case PLUS:
      stack[sp-1] = stack[sp-1] + stack[sp];
      sp--; break;
    ...
  }
}
```
For compactness and speed, Keiko has many instructions that stand for combinations of others. E.g., LDGW x ≡ GLOBAL x; LOADW, where GLOBAL x pushes the address x onto the stack, and LOADW pops an address and pushes its contents. Such 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:

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

Expressions are compiled into postfix form:

```plaintext
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.

For example, the code generated for the expression x+1 is as shown in the figure.

![Figure 2.1: Compiling x+1](image)

### 2.2 Lecture 5: Control structures

Abstract syntax: can be simpler than the concrete syntax.

```plaintext
type stmt =
    Skip
    | Seq of stmt list
    | Assign of name * expr
    | IfStmt of expr * stmt * stmt
```
14  Expressions and statements

| WhileStmt of expr * stmt
| . . .

Example: if y > max then max := y end is

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 1.6) 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.

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. We can augment this with a data structure that keeps track of an equivalence relation on labels. Rules for the peephole optimiser might include the following:

- LABEL a; LABEL b → LABEL a and make a = b.
- JUMP a; LABEL b → LABEL b if a = b.
- JGT a; JUMP b; LABEL a → JLEQ b; LABEL a (etc.)
- LABEL a → [] if a is unused.
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 lab3; gen_cond lab1 lab3 test; LABEL lab3]
```

So while \( r > y \) do \( r := r - y \) end becomes

```
JUMP L2
LABEL L1
LDGW _r
LDGW _y
MINUS
STGW _r
LABEL L2
LDGW _r
LDGW _y
JGT L1
JUMP L3
LABEL L3
```

**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
(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) ->
```
Expressions and statements

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.) 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 $\sqrt{200\,000\,000}$ using binary search. Compile it 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 $e₁$ (containing variables, constants and unary and binary operators) can be evaluated in depth $d₁$, and $e₂$ can be evaluated in depth $d₂$, then $Binop (w, e₁, e₂)$ can be evaluated in depth

$$\min (\max (d₁ (d₂ + 1)), (\max (d₁ + 1) d₂)).$$

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 $cost e \leq N$.

(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:

\[
\text{type code} = \ldots
\]

\[
\begin{align*}
| \text{PUT of int} & \quad (\ast \text{Save temp (address) \ast}) \\
| \text{GET of int} & \quad (\ast \text{Fetch temp (address) \ast})
\end{align*}
\]

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

Assuming } N \geq 2, \text{ define a new version of } \text{gen, expr} \text{ that exploits these new instructions, and places no limit on the size of expressions. The code generated should use as few } \text{GET} \text{ and } \text{PUT} \text{ 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.}

\[\text{Hint: optimal code for an expression can be generated by a function} \]

\[\text{gen: expr } \to \text{ code } \ast \text{ int} \]

that returns code to evaluate a given expression, together with the number } n \text{ of stack slots used by the code, with } n \leq N. \text{ If both } e_1 \text{ and } e_2 \text{ require } N \text{ slots then evaluation of } \text{Binop} (w, e_1, e_2) \text{ will need to use a temp.}

2.4 Programs commonly contain nested if statements, so that either the \text{then} part or (more commonly) the \text{else} part of an if statement is another if statement. (The latter possibility can be abbreviated using the \text{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 \text{scalar product machine} uses an evaluation stack, but replaces the usual floating point addition and multiplication instructions with a single \text{ADDMUL} instruction that, given three numbers } x, y \text{ and } z \text{ on the stack, pops all three and replaces them with the quantity } x + y \ast z. \text{ Thus the expression } b \ast b + 4 \ast a \ast c \text{ could be computed on this machine with the sequence}

\[
\begin{align*}
\text{CONST 0} & \\
\text{LOAD b} & \\
\text{LOAD b} & \\
\text{ADDMUL} & \\
\text{CONST 0} & \\
\text{CONST 4} & \\
\text{LOAD a} & \\
\end{align*}
\]
18 Expressions and statements

ADDMUL
LOAD c
ADDMUL

The first ADDMUL instruction computes \( t_1 = 0 + b \times b \), the second computes \( t_2 = 0 + 4 \times a \), and the third computes the answer as \( t_1 + t_2 \times c \). 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 \( b \times b + 4 \times a \times c \) 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 [2011/2] This question is about the problem of generating optimal code for simple arithmetic expressions that contain only variables and the binary operators \(+\), \(-\), \(*\) and \(/\). All variables are held in memory, and there is an inexhaustible supply of locations \( t_1, t_2, \ldots \) that can be used to hold temporary values. For simplicity we will treat all operations as if they were neither commutative nor associative, and assume that the expression to be evaluated contains no repeated sub-expressions.

The Hextium MMXI is a machine with two registers \( A \) and \( B \), and instructions like these, each with unit cost:

- add A, B Add the contents of register \( B \) to register \( A \).
- sub B, y Subtract the contents of memory location \( y \) from register \( B \).
- load A, x Load register \( A \) with the contents of memory location \( x \).
- store A, t1 Store the contents of register \( B \) into temporary \( t_1 \).

In each instruction, the first operand must be a register, but the second may be a register or a memory location.

(a) Find an optimal code sequence that evaluates the expression \((x - y) \times (u - v/w)\) into a register.
(b) What is the simplest expression that requires a temporary value to be saved in memory? Show the expression and its code.

Optimal code for evaluating an expression \( e \) can be determined by means of a dynamic programming algorithm that computes a triple of costs \((c_1, c_2, c_m)\), where

- \( c_1 \) is the cost of computing \( e \) into a register if only one register is available, or \( \infty \) if this is impossible.
- \( c_2 \) is the cost if both registers are available, and temporary locations can be used if needed, and
- \( c_m \) is the cost of making the value of \( e \) accessible in a memory location, assuming both registers are available.

We may take \( c_m = 0 \) if \( e \) is a variable, because its value is already accessible in memory.

For the expression \( a - b/(c + d) \), we have \( c_1 = \infty \), because the value cannot be found without either using both registers or using a temp in memory. We have \( c_2 = 6 \) because using both registers we can compute the value with the six instructions,

\[
\text{load A, c; add A, d; load B, b; div A, B; load B, a; sub B, A.}
\]

(c) Verify that the costs for the expression \((x - y) \times (u - v/w)\) are \((\infty, 7, 8)\), and compute the costs for the expression you wrote in answer to part (c).

(d) Define recursively a function,

\[
cost : expr \rightarrow int \times int \times int,
\]

that computes the triple of costs for an expression.

(e) Given the function \( cost \), explain how it is possible to generate optimal code for an expression; you need not give all the details, but you should describe what functions are needed and how decisions between alternative code sequences are made.

2.7 [2012/2] The programming language Oberon07 contains a new form of loop construct, illustrated by the following example:

\[
\text{while } x > y \text{ do}
\]

\[
x := x - y
\]

\[
\text{elsif } x < y \text{ do}
\]

\[
y := y - x
\]

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.

2.8 [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 0 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.

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 \textit{gen\_cond \textit{e} \textit{tlab} \textit{flab}} generates code that jumps to label \textit{tlab} if expression \textit{e} has boolean value \textit{true}, and the label \textit{flab} if it has value \textit{false}.

(b) Show how to enhance \textit{gen\_expr} and \textit{gen\_cond} to deal appropriately with conditional expressions.

It is suggested that short-circuit boolean \texttt{and} could be translated by treating \textit{e}_1 \texttt{and} \textit{e}_2 as an abbreviation for the conditional expression

\[
\text{if } e_1 \text{ then } e_2 \text{ else false,}
\]

expanding the abbreviation in creating the abstract syntax tree.

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

\[
\text{if } (i >= 0) \texttt{and} (a[i] > x) \text{ then } i := i+1 \text{ end}
\]

according to your translation, assuming both \textit{i} and \textit{x} are global integer variables, and \textit{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 \texttt{and} operator directly, suggest rules for post-processing the code so that it is equally good.
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 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 and a JIT translator for the bytecode are 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
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:

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>Access to source file indexed by lines</td>
</tr>
<tr>
<td>Print</td>
<td>Formatted output</td>
</tr>
<tr>
<td>Growvect</td>
<td>Extensible arrays</td>
</tr>
<tr>
<td>Bytes</td>
<td>Mutable bytestrings</td>
</tr>
</tbody>
</table>

\[1\] The Bytes module is included only for versions of the OCaml compiler that do not have it.
Most of these modules are rarely referenced from the parts of the compilers you will be modifying.

### 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:

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tree</td>
<td>Abstract syntax trees.</td>
</tr>
<tr>
<td>Lexer</td>
<td>The lexical analyser, generated with <code>ocamllex</code>.</td>
</tr>
<tr>
<td>Parser</td>
<td>The parser, generated with <code>ocamlyacc</code>.</td>
</tr>
<tr>
<td>Keiko</td>
<td>Abstract machine code</td>
</tr>
<tr>
<td>Peepopt</td>
<td>Peephole optimiser</td>
</tr>
<tr>
<td>Kgen</td>
<td>The abstract machine code generator.</td>
</tr>
<tr>
<td>Print</td>
<td>Formatted output.</td>
</tr>
<tr>
<td>Source</td>
<td>Access to source code.</td>
</tr>
<tr>
<td>Main</td>
<td>The main program.</td>
</tr>
</tbody>
</table>

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.
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. As this script reveals, compiling a program, for example with the command

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

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.

After this process, you can run your program by typing `./a.out` at the shell prompt. This invokes the bytecode interpreter and runs it on the binary code contained in the file.

```bash
#!/bin/sh
KEIKO='cd ../keiko; pwd'
set -x
ppc $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
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 three new files: a.k, a.x, and a.out. You can run the program `a.out` from the shell:

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

If you try to open the file `a.x` 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 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 can be activated by using the command `compile -O 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`,...
3.3 Building and running the compiler

```plaintext
MODULE Main 0 0
IMPORT Lib 0
ENDHDR
PROC MAIN 0 0 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

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:

```bash
$ make test
./compile gcd.p
+ KEIKO=./keiko
+ ppc gcd.p
+ ../keiko/pplink -custom -nostdlib
   ../keiko/lib.k a.k -o a.x
+ cat ../keiko/ppx a.x
+ chmod +x a.out
./a.out >a.test
sed -n -e '1,/ˆ(<</d' -e '/ˆ>>*/q' -e p gcd.p
   | diff - a.test
./compile repeat.p
+ KEIKO=./keiko
```
In this test sequence, the program gcd.p was successfully compiled and run; then the attempt to compile repeat.p failed because the unmodified compiler does not recognise the syntax of repeat statements. 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. These differ from while loops in that they have the test at the end, like this:

```plaintext
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.

Code for statements is generated by the function

```plaintext
gen_stmt : Tree.stmt → Keiko.code
```

in the module kgen.ml. This function takes the abstract syntax tree for a statement and returns a code object, representing a list of instructions and labels. To avoid having to compute the code list by concatenating lots of smaller lists, which would be slow (and to remove the temptation to introduce accumulating parameters everywhere), the code list is not represented as an ordinary OCaml list. Instead, the algebraic type `Keiko.code` has constructors for each Keiko instruction, plus three more constructors `Seq`, `Nop` and `Line`:

```plaintext
type code =
  Const of int (* Push constant (value) *)
| Global of string (* Push symbol (name) *)
| ...
| Seq of code list (* Sequence of code fragments *)
| Nop (* Empty sequence *)
| Line of int (* Line number *)
```

You can see the `Seq` constructor at work in the `gen_stmt` rule for while loops:

```plaintext
let rec gen_stmt =
  function ...
  | 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 lab1 lab3 test; Label lab3]
```
3.5 Implementing loop statements

After inventing three labels, this rule uses a recursive call of `gen_stmt` to generate code for the body of the loop and also generates jumping code for the test and then puts these parts together with some surrounding jumps and labels into a `Seq` structure, representing all the parts joined into a sequence. Later, other components of the compiler can flatten the tree of `Seq` nodes into a single list (using an accumulating parameter for efficiency), or recursively output the sequence with little cost. The `Nop` constructor is equivalent to `Seq []`, and represents the empty sequence of instructions. The code `LINE n` indicates that the instructions that follow it were generated from line `n` of the source program. It’s used to produce the comments in the object code that show the source line for each fragment of code.

The following instructions 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 case for `RepeatStmt`, modelling it on the existing case for `WhileStmt`.

In all, these modifications require about 10 lines of code to be added or changed.

When these changes have been made, it should be possible to re-build 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, 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:

```plaintext
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 10, without modifying \(y\). 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 L2
  ⟨code for x := x + 1⟩
  ⟨if x >= 10 then jump to L3⟩
  ⟨code for y := y + x⟩
JUMP L2
LABEL L3
```

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:

   ```plaintext
   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:

```plaintext
case i of
```
3.6 Implementing case statements

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:

```plaintext
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`:

```plaintext
CaseStmt of expr * (int list * stmt) list * stmt.
```

A node `CaseStmt` with control expression `switch`, the list of cases `cases` and 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.

(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 syn-
tax. 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 failed” message 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 with a multi-way branch instruction CASEJUMP N that compares a 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, potentially with multiple values 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.

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

\[
\text{CaseStmt (switch, cases, default)}
\]

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

\[
[(x_{11}, L_1); (x_{12}, L_1); \ldots; (x_{nm}, L_n)]
\]

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.
3.6 Implementing case statements

Figure 3.5: Code for case statements

\[
\begin{align*}
\text{Code for the control expression) } \\
\text{CASEJUMP N} \\
\text{CASEARM } (x_{11}, L_1) \\
\text{CASEARM } (x_{12}, L_1) \\
\ldots \\
\text{CASEARM } (x_{nm(n)}, L_n) \\
\text{JUMP def.lab} \\
\text{LABEL } L_1 \\
\langle \text{Code for first case} \rangle \\
\text{JUMP exit.lab} \\
\text{LABEL } L_2 \\
\langle \text{Code for second case} \rangle \\
\text{JUMP exit.lab} \\
\ldots \\
\text{LABEL } L_n \\
\langle \text{Code for n’th case} \rangle \\
\text{JUMP def.lab} \\
\text{LABEL exit.lab} \\
\end{align*}
\]

(5) Use List.combined 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 with test cases of your own: 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.
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 collect a list of variables that are declared implicitly. 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. 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:

41

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 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:

```ocaml
type expr =
  { e_guts : expr_guts; }
```

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.
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;
    x_line : int;
    mutable x_def : def option }

The type ptype 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 been filled in. 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.

Let's begin by considering how to generate code from the annotated tree. Consider 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 LHS has the form needed for a variable.) But to carry out the assignment, the two sides must be treated differently: the RHS yields a value and the LHS yields an address to store it in. So use two functions in the code generator:

let rec gen_addr v =
  match v.expr guts with
    Variable x ->
      let d = get_def x in GLOBAL d.d_lab
    | Subscript (e1, e2) ->
      SEQ [gen_addr e1;               (* address of array *)}
let rec gen_stmt =
  function . .
  | Assign (v, e) →
      Seq [Line (line_number v); gen_expr e; gen_addr v; Storew]

Semantic analysis is another recursive process. Much confusion surrounds the ADT of symbol tables or environments, mostly because of a historical concern for efficiency. It's simple, really, if we accept the idea of using an immutable, applicative representation:

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 121. 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 "PROC MAIN ...
    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.
Types and data structures

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.

Code for data structure access: For arrays,

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

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
```

On Keiko, this sequence is equivalent to

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

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

\[ \text{var b: array 10 or array 10 of integer;} \]

\[ \text{addr(b[i][j])] = \text{addr(b) + i * 40 + j * 4.} \]

For records,

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

```
r: \begin{array}{c|cc}
\text{name} & \text{age} & \text{next} \\
\hline
12 & 4 & 4 \\
\end{array}
```

Figure 4.2: Layout of record \( r \)

For example, \( r\.age := 29 \) gives

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

For pointers, we use

\[ \text{addr(p↑) = value(p).} \]

For example, \( p := p↑.next \) (on line 25) becomes

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

or the equivalent sequence,

```
LDGW _p; NCHECK 25; LDNW 16; STGW _p
```

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

\[
\begin{align*}
t_1 & = \text{pointer to array 10 of integer}; \\
t_2 & = \text{pointer to array 10 of integer};
\end{align*}
\]

is the assignment \( p_1 := p_2 \) allowed? According to structural equivalence, the answer is Yes; but if the typing rule is name equivalence then the two types \( p_1 \) and \( p_2 \) 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 \( y \).

Exercises

3.1 Consider the following declarations.

\[
\begin{align*}
type \; & \text{ptr} = \text{pointer to rec}; \\
& \text{rec} = \text{record data: integer; next: ptr; end}; \\
\text{var} \; & q: \text{ptr}; s: \text{integer};
\end{align*}
\]

The following two statements form the body of a loop that sums the elements of linked list.

\[
\begin{align*}
s & := s + q↑.\text{data}; \\
q & := q↑.\text{next}
\end{align*}
\]

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:

\[
\text{stmt} \rightarrow \text{local decls in stmts end}
\]

For example, here is a program that prints 53:

\[
\begin{align*}
\text{var} \; & x, y: \text{integer}; \\
\text{begin} & \\
& y := 4; \\
& \text{local} \\
& \quad \text{var} \ y: \text{integer}; \\
& \quad \text{in} \\
& \quad \quad y := 3 + 4; x := y \ast y \\
& \quad \text{end}; \\
& \text{print} \ x + y \\
\text{end}.
\end{align*}
\]

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 \texttt{exit} and \texttt{next} statements. Here is an example program:

\begin{verbatim}
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{verbatim}

Each loop of the form \texttt{loop L: ... end} has a label \texttt{L}; its body may contain statements of the form \texttt{next L} or \texttt{exit L}, which may be nested inside inner loops. A loop is executed by executing its body repeatedly, until a statement \texttt{exit L} is encountered. The statement \texttt{next L} has the effect of beginning the next iteration of the loop labelled \texttt{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.
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 assigns a type to each expression and annotates each 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:

```plaintext
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:

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

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`:

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

The `x_def` field can contain a `def` record that gives the definition of the identifier. 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`:

```plaintext
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:

1. The analyser does not recognise subscripted variables $v[e]$, which are represented in the abstract syntax tree by $\text{Sub}(v, e)$, so you will have to add a clause to the function $\text{check}_\text{expr}$ to handle them. This function has type $\text{environment} \rightarrow \text{expr} \rightarrow \text{ptype}$; it takes an environment and the abstract syntax tree for an expression and returns the type of the expression as a $\text{ptype}$ 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 $\text{Integer}$. 

Figure 5.1: Syntax additions for declarations and arrays
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 $K_{gen}$ 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`. 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; at that stage, one could also do other optimisations like using a fast shift instruction to multiply by powers of two.
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). 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 the address space of the program is divided into

- program code: read-only unless using dynamic loading.
- global variables: at fixed addresses, referred to symbolically by the compiler and fixed by the linker.
- stack: used for subroutines.

Large address space of modern machines makes the memory layout less important.

46
• heap: for dynamically allocated storage.

![Diagram of address space layout](image)

**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.

![Diagram of stack frame layout](image)

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

```keiko
proc f(x, y);
  var a;
begin
  a := x - y;
  return a * a
end;
... print f(5, 2) ...
```

compiles into

```
PROC f 4 0 0
  ! 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
RETURNW – return one-word result
  ! ... 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 procedure address: also held in the CP register while the procedure is active.
• the return address: saved value of the PC from the caller.
• the dynamic link: saved value of the FP register.

Four pieces of code collaborate to create and destroy activation records:

- **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:

```
PROC inc 0 0 0
LDLW 16
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.

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

```
PROC inc 0 0 0
```

*Some of these are hidden in Keiko.*
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 LGW _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.

```plaintext
var z;
proc setz(x);
begin z := x end;

PROC setz 0 0 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. For “open array parameters” with no fixed bound, like those in Oberon:

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

Compile it as if it were

```plaintext
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).

For aggregate parameters passed by value, pretend they are passed by reference, then make the procedure's prelude copy them into the stack frame. That's slow, but unavoidable in general; few programming languages offer this feature.
Nested procedures: for example,

\[
\begin{align*}
\text{proc } \text{twopow}(x, y, n) ; \\
\text{proc } \text{pow}(t) ; \\
\text{var } i ; \\
\text{begin} \\
\text{while } i < n \text{ do } ... \\
\text{end} ; \\
\text{begin} \\
\text{return } \text{pow}(x) + \text{pow}(y) \\
\text{end} ;
\end{align*}
\]

This looks a bit artificial in a Pascal-like language, but nested procedures are meat and drink in functional languages:

\[
\text{let inc_all } n \text{ xs }= \text{map (fun } x \rightarrow x + n) \text{ xs}
\]

To implement nested procedures (with access from inner procedures to local variables of outer procedures, as with \(n\) in these two examples), get the semantic analyser to annotate each variable with a 2-D address (level, offset), like this:

\[
\begin{align*}
\text{proc } \text{twopow}^1(x^{(1,16)}, y^{(1,20)}, n^{(1,24)}) ; \\
\text{proc } \text{pow}^2(t^{(2,16)}) ; \\
\text{var } i^{(2,-4)} ; \\
\text{begin} \\
\text{while } i^{(2,-4)} < n^{(1,24)} \text{ do } ... \\
\text{end} ; \\
\text{begin} \\
\text{return } \text{pow}^2(x^{(1,16)}) + \text{pow}^2(y^{(1,20)}) \\
\text{end} ;
\end{align*}
\]

At runtime, arrange that the static link of each level \(n\) procedure activation points to the frame of the enclosing level \(n - 1\) procedure.
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. For the expression \( i < n \) in \texttt{pow}:

\begin{verbatim}
PROC pow 0 0 4
...
LOCAL -4; LOADW
LOCAL 12; LOADW
CONST 24; OFFSET; LOADW
JGEQ ...
\end{verbatim}

Here, \texttt{LOCAL -4; LOADW} (equivalent to \texttt{LDLW -4}) fetches the value of the local variable \( i \). The sequence \texttt{LOCAL 12; LOADW; CONST 24; OFFSET; LOADW} can be abbreviated to the two instructions \texttt{LDLW 12; LDNW 24}; it first fetches the static link, a pointer to the stack frame of the enclosing invocation of \texttt{twopow}, and then fetches the word that is at offset 24 in that frame. That is the value of the parameter \( n \).

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

\begin{verbatim}
PROC twopow 0 0 4
...
LOCAL 16; LOADW
LOCAL 0
GLOBAL _pow
PCALLW 1
\end{verbatim}

Here, the instruction \texttt{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.

Generally, a procedure \( P \) at level \( n \) may directly call another procedure \( Q \) at level \( m \) for \( 1 \leq m \leq n + 1 \). 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 \texttt{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.

### 6.3 Lecture 10: Higher-order functions

The next stage in sophistication is to allow procedures that accept other procedures as parameters:

\begin{verbatim}
proc sum(a, b: integer; proc f(i: integer): integer): integer;
\end{verbatim}

which might compute \( \sum_{a \leq i \leq b} f(i) \). We could use \texttt{sum} like this:

\begin{verbatim}
proc sumpow(a, b, n: integer): integer;
proc pow(x: integer): integer;
begin
(* return x^n *)
end
begin
\end{verbatim}
6.3 Lecture 10: Higher-order functions

return sum(a, b, pow)
end;

(The form \texttt{sum(a, b, \texttt{lambda (x) pow(x, n)})} is easily converted to this form by systematically transforming lambdas into named local functions.)

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.

Figure 6.6: Frame layout for \texttt{sum}

In the example, \texttt{sum} calls \texttt{f(k)} like this:

\begin{verbatim}
LDLW -4
LDLW 28
LDLW 24
PCALLW 1
\end{verbatim}

Figure 6.7: Frame layout for \texttt{sumpow and pow}

The call \texttt{sum(a, b, pow)} in \texttt{sumpow} looks like this:

\begin{verbatim}
LOCAL 0
GLOBAL _pow
LDLW 20
LDLW 16
CONST 0
GLOBAL _sum
PCALLW 4
\end{verbatim}

Here's an example that combines nesting and procedural parameters with recursion. The problem is to find a permutation \(d_1d_2\ldots d_9\) of the digits 1
to 9 such that \((d_1d_2 \ldots d_9)_{10}\) is divisible by 9 and \((d_1d_2 \ldots d_8)_{10}\) is divisible by 8, and so on. The solution is a recursive procedure \(\text{search}(k, n, \text{avail})\) that expects \(n\) to be a \(k\)-digit number already chosen, and \(\text{avail}\) to be the set of digits still unchosen – represented as a boolean function \(\text{proc avail}(x: \text{integer}): \text{boolean}\).

\[
\begin{align*}
\text{proc search}(k, n: \text{integer}; \text{proc avail}(x: \text{integer}): \text{boolean};
& \quad \text{var d, n1: integer; }
& \quad \text{proc avail1}(x: \text{integer}): \text{boolean;}
& \quad \begin{align*}
& \quad \text{begin}
& \quad \quad \text{if (}x \neq d\text{) then}
& \quad \quad \quad \text{return avail}(x)
& \quad \quad \text{else}
& \quad \quad \quad \text{return false}
& \quad \quad \text{end}
& \quad \text{end;}
& \quad \text{begin}
& \quad \quad \text{...}
& \quad \quad \text{for } d := 1 \text{ to } 9 \text{ do}
& \quad \quad \quad \text{if avail}(d) \text{ then ...}
& \quad \quad \quad \quad \text{search}(k+1, n1, \text{avail1})
& \quad \quad \quad \text{...}
& \quad \quad \text{end}
& \quad \text{end;}
& \quad \text{proc avail0}(x: \text{integer}): \text{boolean;}
& \quad \begin{align*}
& \quad \text{begin}
& \quad \quad \text{return true}
& \quad \text{end;}
& \quad \text{... search}(0, 0, \text{avail0}) ...
\end{align*}
\end{align*}
\]

Figure 6.8: Frame layout for avail1 and search.

The body of avail1 becomes

\[
\begin{align*}
\text{LDLW } & 16 - x \\
\text{LDLW } & 12; \text{LDNW } -4 - d \\
\text{JEQ } & L1 \\
\text{LDLW } & 16 - x \\
\text{LDLW } & 12; \text{LDNW } 28 - \text{static link}
\end{align*}
\]
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 ppc 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:

```ocaml
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:

```ocaml
type def\_kind =
    ConstDef of value
    | VarDef | CParamDef | VParamDef
```
where for us value = int, but more generally could incorporate other target values like floating point.

Declarations are processed by

\[
\text{check\_decl : decl} -\rightarrow \text{environment} -\rightarrow (\text{def} -\rightarrow \text{unit}) -\rightarrow \text{environment}
\]

where the \text{def} -\rightarrow \text{unit} argument is one of \text{loc\_alloc} (allocate space for a local variable, downwards in the stack frame), \text{param\_alloc} (for a parameter, upwards), \text{field\_alloc} (for record fields), \text{global\_alloc} (for global variables).

Use an abstract data type of environments, with operations

\[
\begin{align*}
\text{val empty : environment} \\
\text{val lookup : ident} -\rightarrow \text{environment} -\rightarrow \text{def} \\
\text{val define : def} -\rightarrow \text{environment} -\rightarrow \text{environment} \\
\text{val new\_block : environment} -\rightarrow \text{environment} \\
\text{val top\_block : environment} -\rightarrow \text{def list}
\end{align*}
\]

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

\[
\begin{align*}
\text{let check\_heading env (Heading (x, fparams, result)) =} \\
& \text{let pcount = ref 0 in} \\
& \text{let env′ =} \\
& \quad \text{check\_decls fparams (new\_block env) (param\_alloc pcount) in} \\
& \ldots \\
& \text{mk\_type (ProcType \{ p\_fparams = top\_block env′; p\_pcount = !pcount; \ldots \})}
\end{align*}
\]

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

\[
\text{type metrics = \{ r\_size : int; r\_align : int \}}
\]

This allows us to lay out correctly a type like \textit{record c: char; n: integer end}

![Record layout with padding](image)

\textbf{Figure 6.10: Record layout with padding}

The \textit{intermediate code generator} is much as we have seen, but includes new stuff in \texttt{gen\_addr} and \texttt{address} to use the static chain:

\[
\begin{align*}
\text{(* address - generate code for address of a definition *)} \\
\text{let address d =} \\
& \text{match d.d\_addr with} \\
& \quad \text{Global g} -\rightarrow \text{GLOBAL g} \\
& \quad | \text{Local off} -\rightarrow \\
& \quad \quad \text{if d.d\_level = level then} \\
& \quad \quad \text{LOCAL off} \\
& \quad \quad \text{else} \\
& \quad \quad \text{SEQ [schain (!level - d.d\_level); CONST off; OFFSET]} \\
& \quad \text{end}
\end{align*}
\]
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.

```haskell
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 → ST LW n
- GLOBAL x; LOADW → LD GW x
- GLOBAL x; STOREW → ST GW x
- CONST n; OFFSET; LOADW → LD N W 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.

```plaintext
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:

```plaintext
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:

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

```plaintext
var v: Vehicle;
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

```plaintext
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.

```plaintext
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.

---

[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.
(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

```plaintext
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

```plaintext
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

```plaintext
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

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

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

4.1 Show the Keiko code for the following program, explaining the purpose of each instruction.

```pascal
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:

```pascal
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 The following PICOPASCAL program is written in what is called 'continuation-passing style':

```pascal
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;
```
Subroutines

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

4.4 [2013/3] Figure 6.12 shows a program that computes

\[
\sum_{0 \leq i < 10} (i + 1)^2 = 385
\]

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

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

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

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

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

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

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

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.

```pascal
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
  ...
```
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
64 Subroutines

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.
Lab three: Subroutines

This lab asks you to add recursive subroutines with nesting, and (optionally) procedural parameters to a compiler that translates a simple 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 Using the lab kit

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:

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tree</td>
<td>Abstract syntax</td>
</tr>
<tr>
<td>Lexer</td>
<td>Lexical analyser</td>
</tr>
<tr>
<td>Parser</td>
<td>Syntax analyser</td>
</tr>
<tr>
<td>Check</td>
<td>Semantic analysis</td>
</tr>
<tr>
<td>Dict</td>
<td>Symbol tables</td>
</tr>
<tr>
<td>Keiko</td>
<td>Abstract machine code</td>
</tr>
<tr>
<td>Peepopt</td>
<td>Peephole optimiser</td>
</tr>
<tr>
<td>Kgen</td>
<td>Abstract machine code generator</td>
</tr>
</tbody>
</table>

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 `ppc`. 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
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 prac-
tical, 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.

Figure 7.1: Syntax summary
• 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 RETURNW instruction. This expects the procedure’s result 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). 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`.

- 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 \(1^k + 2^k + \ldots + n^k\), 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.

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.

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 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

---

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 instructions are not used by the Oberon compiler from which the Keiko machine is adapted. The implementation supports up to 256 different code addresses, and static links that point anywhere within a subroutine stack of up to 16MB.
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.

(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.
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.

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. For example, for the statement \( x := a[i] \), where \( x \) and \( i \) are locals and \( a \) is a global array, we might have this Keiko code:

```keiko
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. It’s easy to rewrite \( Kgen \) so that it produces the same code arranged as a list of trees instead of a list of instructions. We’ll write that tree also as

\[
\langle \text{STOREW},
\langle \text{LOADW},
\langle \text{OFFSET},
\langle \text{GLOBAL } _a),
\langle \text{LSL}, \langle \text{LOADW}, \langle \text{LOCAL 40}), \langle \text{CONST 2)}}
\langle \text{LOCAL } (-4))
\]
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, provided at least that the evaluation stack is empty at each jump or label. For example, the translation of assignment statements was expressed like this:

```ml
let rec gen_stmt =
  function ...
  | Assign(v, e) ->
    let d = get_def v in
    SEQ [gen_expr e; gen_addr d; STOREW]
```

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

```ml
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:

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

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

```
⟨OFFSET, t₁, ⟨CONST n⟩⟩
```

for what ML would render as

```
Node(OFFSET, [t₁; Node(CONST n, [])])
```
In the lab materials, there’s a tool called \textit{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.

The translation of a procedure body becomes a \textit{sequence} of operator trees. The roots of the trees are labelled with instructions that don’t produce a result, such as \textit{Storew} and conditional branches, or with labels for the branches to target. Their children are labelled with operations like \textit{Offset} and \textit{Loadw} and \textit{Local n} that do produce a result; each such Keiko instruction pops a number of values from the stack and pushes a single result, and a node labelled with that instruction will have as many children as there are values it consumes. 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:

\begin{verbatim}
set r0, _a
ldr r1, [fp, #40]
lsl r1, r1, #2
ldr r0, [r0, r1]
str r0, [fp, #-4]
\end{verbatim}

\textbf{Appendix C} gives a guide to assembly language programming for the ARM, but here is a rough interpretation of this code fragment. The first instruction, \texttt{set r0, _a}, puts the global address \texttt{_a} into register \texttt{r0}. The second instruction, \texttt{ldr r1, [fp, #40]}, loads the value of \texttt{i} into register \texttt{r1} by adding together the value of the frame pointer \texttt{fp} and the offset 40 and loading from that address. The third instruction, \texttt{lsl r1, r1, #2}, multiplies this by 4, shifting it two bits to the left, putting the result back in \texttt{r1}. The fourth instruction, \texttt{ldr r0, [r0, r1]}, adds \texttt{r0} and \texttt{r1} together to form an address, loads from this address, and puts the value in \texttt{r0}. The last instruction, \texttt{str r0, [fp, #-4]} stores this value into \texttt{x}, again using addressing relative to the frame pointer.

\textit{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.

\footnote{\texttt{We are using a slightly modified syntax for the ARM assembly language; the instruction would be written \texttt{ldr r0, \_a} in the standard syntax, but I wanted to avoid explaining why that is so.}}
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.

**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; the first couple of stages take over the role that used to be played by the peephole optimiser.

- A simplifier 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 by cheaper ones like shifting.
- A jump optimiser 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 (CSE), so that any parts of a tree that were computed twice are now computed 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. The back end proper:

- Instruction selection by tiling the tree – [Section 8.2](#).
- Register assignment, using the finite register set to hold values between instructions – [Section 8.3](#).

In reality, the different problems interact. E.g., register assignment interacts with all previous stages because of the finite register set. Not considered: 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.

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

```plaintext
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.

$$\langle \text{STOREW}, \langle \text{LOADW}, \langle \text{OFFSET}, \langle \text{GLOBAL} \_a \rangle, \langle \text{LSL}, \langle \text{LOADW}, \langle \text{LOCAL} \ 40 \rangle \rangle, \langle \text{CONST} \ 2 \rangle \rangle \rangle, \langle \text{LOCAL} \ -4 \rangle \rangle \rangle$$

$$\langle \text{RESULTW}, \langle \text{TIMES}, \langle \text{LOADW}, \langle \text{LOCAL} \ -4 \rangle \rangle, \langle \text{CONST} \ 3 \rangle \rangle \rangle$$

The back end generates code for the 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 8.2. Spare space is allocated so as to ensure that $fp$ and $sp$ are both multiples of 8, as the ABI requires.

```
_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];
  set r0, _a
  ldr r1, [fp, #40]
```
8.2 Lecture 14: Instruction selection

8.2.1 Describing the problem
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:

```
set 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 $u_0$ and $u_3$ can live in the real register $r_0$, and $u_1$ and $u_2$ can live in $r_1$, 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 8.3. 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 $u_0$, ..., $u_3$. Instruction selection involves describing the set of tiles that can be Many tilings of a tree are usually possible. For example, another tiling of the tree for $x := a[i]$ is shown in Figure 8.4.

This tiling fails to exploit the addressing hardware to add together the address and offset for the array access, and pointlessly computes the address of $x$ into a register – it’s feasible but not optimal. Here is the code that results:

```
set u0, _a
ldr u1, [fp, #40]
lsl u2, u1, #2
add u3, u0, u2
ldr u4, [u3]
```
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 find an algorithm for finding a near-optimal tiling for any tree that has one.

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 \textit{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

\[
\text{reg} \rightarrow \langle \text{BINOP Plus, reg}, \text{reg} \rangle
\]

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 \(\langle \text{BINOP Plus}, t_1, t_2 \rangle\) by first computing \(t_1\) and \(t_2\) and then executing an \textit{add} instruction.

For compactness in describing the set of feasible trees, we use multiple non-terminals: \textit{stmt} – the root of a tree; \textit{reg} – a value in a register; \textit{addr} – an address for \textit{ldr} or \textit{str}; \textit{rand} – a register or constant. For example, we could cover the \textit{STOREW} by either one of two productions:

- \textit{stmt} \rightarrow \langle \text{STOREW, reg}, \langle \text{LOCAL } n \rangle \rangle, corresponding to the instruction
  
  \[
  \text{str } \text{reg}, [\text{fp, } #n];
  \]

- \textit{stmt} \rightarrow \langle \text{STOREW, reg}_1, \text{reg}_2 \rangle, corresponding to the instruction

\[
\text{add u5, fp, } #\text{#}8
\]

\[
\text{str u4, } [\text{u5}]
\]
8.2 Lecture 14: Instruction selection

But it’s more compact to write one production $stmt \rightarrow \langle STOREW, reg, addr \rangle$, and provide productions for $addr$ that are shared by $LOADW$ and $STOREW$.

$$\text{stmt} \rightarrow \langle \text{STOREW}, \text{reg}, \text{addr} \rangle \quad \{ \text{str reg, addr} \}$$

$$\text{addr} \rightarrow \langle \text{LOCAL n} \rangle \quad \{ \text{[fp, #n]} \}$$

Figure 8.4: Alternative tiling of an operator tree

$$\text{addr} \rightarrow \langle \text{reg} \rangle \quad \{ \text{[reg]} \}$$

Figure 8.5 shows a set of productions relevant to the 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

$$\text{reg} \rightarrow \langle \text{OFFSET, reg$_1$, rand} \rangle \quad \{ \text{add reg, reg$_1$, rand} \}$$

that generate each operation on its own and can have inputs and output in registers, though this will not always give the best code. Local variables in a large stack frame are a case in point. If the frame is small, then loading from a local variable, $\langle LOADW, \langle \text{LOCAL n} \rangle \rangle$, is covered by the rules

$$\text{reg} \rightarrow \langle \text{LOADW, addr} \rangle \quad \{ \text{ldr reg, addr} \}$$

$$\text{addr} \rightarrow \langle \text{LOCAL n} \rangle \quad \{ \text{[fp, #n]} \}$$

But the second of these has a side-condition that $n$ 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:

$$\text{addr} \rightarrow \text{reg} \quad \{ \text{[reg]} \}$$

$$\text{reg} \rightarrow \langle \text{LOCAL n} \rangle \quad \{ \text{set ip, #n; add reg, fp, ip} \}$$

(The last of these applies even if $n$ is large.) The resulting code uses the scratch register ip:

$$\text{set ip, #n}$$

$$\text{add u1, fp, ip}$$
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 works from the root towards the leaves, 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 using the algorithm.

Actually, the tiling shown in Figure 8.3 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 → ⟨OFFSET, reg₁, ⟨LSL, reg₂, ⟨CONST n⟩⟩⟩ { [reg₁, reg₂, LSL #n] }
```

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

### 8.2.2 Implementing greedy selection

For RISC machines, 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. For the ARM, we will use functions

- \( e_{\text{reg}} : \text{optree} \rightarrow \text{operand} \rightarrow \text{operand} \)
- \( e_{\text{rand}} : \text{optree} \rightarrow \text{operand} \)
- \( e_{\text{addr}} : \text{optree} \rightarrow \text{operand} \)
corresponding to the non-terminals `reg`, `rand`, `addr`, `stmt` and `call`. The function `e_call` is used only for procedure calls, so we'll return to it later. Note that each of these functions takes an `optree` as a parameter, and three of them return an `operand` as a result; we will define that type a bit later. The function `e_reg` takes an additional parameter of type `operand` which, like other details, we'll return to later; for now, we can notice that this parameter is usually the special `operand` value `anyreg`.

**Analysing the input:** Initially, let's ignore the values 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 8.6 shows the parts of the functions that apply the productions shown in Figure 8.5.

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 `e_stmt` where we match a tree with the pattern `⟨Storew, t₁, t₂⟩`, then make recursive calls `e_reg t₁ anyreg` and `e_addr t₂`. The other rules are reflected in the other functions in a similar way; note that `e_rand` and `e_addr` each have a catch-all case at the end that defers to `e_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 8.5) 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 if it cannot. To become 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 `operand`, intended to represent a fragment of code that doesn't amount to a complete instruction.

```
type operand = (* Value) Syntax *)
   Const of int (* val #val *)
| Register of reg (* [reg] reg *)
| Index of reg * int (* val + [reg] [reg, #val] *)
| Index₂ of reg * reg * int (* [r₁] + [r₂] ≪ n [r₁, r₂, LSL #n] *)
| Global of symbol (* lab lab *)
| Label of codelab (* lab .Lab *)
```

The function `e_reg` also takes an `operand` value as a parameter, so that we can specify (if we care) which register should contain the result, and it generates code to evaluate the tree into that register.

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⟩
```

let rec e_stmt t =
match t with
| ⟨STOREW, t₁, t₂⟩ →
e_reg t₁ anyreg;
e_addr t₂;
| ...

and e_reg t r =
match t with
  (CONST n) when fits_move n → ()
| (LOCAL n) when fits_add n → ()
| (GLOBAL x) → ()
| (LOADW, t₁) →
e_addr t₁
| (OFFSET, t₁, t₂) →
e_reg t₁ anyreg;
e_rand t₂
| (BINOP Lsl, t₁, t₂) →
e_reg t₁ anyreg;
e_rand t₂
| ...

and e_rand t =
match t with
  (CONST k) when fits_immed k → ()
| _ → e_reg t anyreg

and e_addr t =
match t with
  (LOCAL n) when fits_offset n → ()
| (OFFSET, t₁, (CONST n)) when fits_n →
e_reg t₁ anyreg
| (OFFSET, t₁, t₂) →
e_reg t₁ anyreg;
e_reg t₂ anyreg
| _ →
e_reg t anyreg

Figure 8.6: Code generator skeleton
which produces a piece of assembler syntax \([fp, \#n]\), corresponding to the operand value \(Index(R_{fp}, n)\), with \(R_{fp}\) being a value of type \(reg\) that denotes the frame pointer \(fp\). This operand is what \(e_{addr}\) should return if the rule applies, so that an appropriate case in that function is

\[
\text{let rec } e_{addr} t =
\]
\[
\begin{array}{l}
\text{match } t \text{ with } \ldots \\
| (\text{LOCAL } n) \text{ when } \text{fits_offset } n \to \\
\quad Index(R_{fp}, n)
\end{array}
\]

This value is returned to the caller of \(e_{addr}\), which will make the operand part of a larger instruction. So let's look at a call to \(e_{addr}\) from the function \(e_{stmt}\):

\[
\text{let rec } e_{stmt} t =
\]
\[
\begin{array}{l}
\text{match } t \text{ with } \ldots \\
| (\text{STOREW}, t_1, t_2) \to \\
\quad \text{let } v_1 = e_{reg} t_1 \text{ anyreg in } \\
\quad \text{let } v_2 = e_{addr} t_2 \text{ in } \\
\quad \text{gen } "str" [v_1; v_2]
\end{array}
\]

This calls \(e_{reg}\) and \(e_{addr}\) to form two operands, one a register and the other an address, and then uses a subroutine \(gen\) to emit an instruction containing the two operands. The operands in question could be \(Register(R_{fp})\) and \(Index(R_{fp}, -4)\), and in that case the call \(\text{gen } "str" [v_1; v_2]\) would output the instruction

\[
\text{str } r1, [fp, \#-4]
\]

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

\[
\begin{array}{l}
\text{gen : string } \to \text{ operand list } \to \text{ unit} \\
\text{gen_reg : string } \to \text{ operand list } \to \text{ operand}
\end{array}
\]

The difference between them is that \(\text{gen_reg}\) generates an instruction that puts its result in a register, and it returns as an operand the register that has been chosen for the result. The details of how a register is chosen, and how we keep track of which registers are occupied, we can leave for later.

With these conventions, 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 \(e_{reg}\):

\[
\text{let rec } e_{reg} t r =
\]
\[
\begin{array}{l}
\text{match } t \text{ with } \ldots \\
\quad (\text{CONST } n) \text{ when } \text{fits_move } n \to \\
\quad \text{gen_reg } "mov" [r; Const n] \\
| (\text{LOCAL } n) \text{ when } \text{fits_add } n \to \\
\quad \text{gen_reg } "add" [r; Register(R_{fp}); Const n] \\
| (\text{GLOBAL } x) \to \\
\quad \text{gen_reg } "set" [r; Global x] \\
| (\text{LOADW}, t_1) \to \\
\quad \text{let } v_1 = e_{addr} t_1 \text{ in } \\
\quad \text{gen_reg } "ldr" [r; v_1] \\
| (\text{OFFSET}, t_1, t_2) \to \\
\end{array}
\]
let $v_1 = e_{reg} t_1 \ anyreg$ in
let $v_2 = e_{rand} t_2 \ in$
$gen_{reg} "add"[r; v_1; v_2]$
| (BINOP Lsl, t_1, t_2) →
  let $v_1 = e_{reg} t_1 \ anyreg$ in
  let $v_2 = e_{rand} t_2 \ in$
  $gen_{reg} "lsl"[r; v_1; v_2]$
| ...

The interaction between $gen_{reg}$ and the special operand value $anyreg$ needs a bit of explanation. Typically, the argument $r$ passed to $e_{reg}$ will be $anyreg$, 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 $e_{reg}$.

For example, consider the tree

$$t = \langle LOADW, \langle LOCAL 40 \rangle \rangle.$$  

The call $e_{reg} t \ anyreg$ matches the first case in $e_{reg}$, and results in a call of $e_{addr} \langle LOCAL 40 \rangle$. This, as we have seen before, returns the operand value $Index(R_{fp}, 40)$. So now $e_{reg}$ makes the call

$$gen_{reg} "ldr"[anyreg; Index(R_{fp}, 40)]$$

It’s the job of the register allocator (to which $gen_{reg}$ is part of the 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 the operand value $Register(R1)$, which also becomes the result of $e_{reg}$.

Working along the same lines, we can complete the function $e_{addr}$ to output code, incorporating the case we covered before.

let rec $e_{addr} t =$
match $t$ with
  $(\text{LOCAL } n) \text{ when fits_offset } n →$
  $Index(R_{fp}, n)$
| $(\text{OFFSET, } t_1, (\text{CONST } n)) \text{ when fits_offset } n →$
  let $v_1 = e_{reg} t_1 \ anyreg$ in
  $Index(\text{reg_of } v_1, n)$
| $(\text{OFFSET, } t_1, t_2) →$
  let $v_1 = e_{reg} t_1 \ anyreg$ in
  let $v_2 = e_{reg} t_2 \ anyreg$ in
  $Index(\text{reg_of } v_1, \text{reg_of } v_2, 0)$
| _ →
  let $v_1 = e_{reg} t \ anyreg$ in
  $Index(\text{reg_of } v_1, 0)$

The aim here is to use the addressing modes to get an addition for free, whether it comes from an OFFSET operation or a LOCAL node. The function $\text{reg_of}$ extracts the register $r$ from an operand $Register\ r$. 
There’s also a function \( e_{\text{rand}} \) that prepares the second operand of arithmetic instructions; this can be a register or a constant.

\[
\text{and } e_{\text{rand}} = \\
\text{function} \\
\quad (\text{CONST } k) \text{ when } \text{fits_immed } k \rightarrow \text{Const } k \\
| \ t \rightarrow e_{\text{reg}} t \text{ anyreg}
\]

The condition when \( \text{fits_immed } k \) 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. (The definition of \( \text{fits_immed} \) is a bit complicated, and is omitted here.) If a tree \( \langle \text{CONST } k \rangle \) 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,

\[
\text{add } u1, u0, \#3
\]

while the expression \( x+31416 \) produces a \texttt{set} and an \texttt{add}:

\[
\begin{align*}
\text{set } u2, \#31416 \\
\text{add } u1, u0, u2
\end{align*}
\]

Similar checks \( \text{fits_offset} \) and \( \text{fits_move} \) 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 \( e_{\text{stmt}} \). Again, a few example rules:

\[
\text{let } e_{\text{stmt}} = \\
\text{function} \\
\quad \langle \text{STOREW, } t_1, t_2 \rangle \rightarrow \\
\quad \text{let } v_1 = e_{\text{reg}} t_1 \text{ anyreg in} \\
\quad \text{let } v_2 = e_{\text{addr}} t_2 \text{ in} \\
\quad \text{gen "str" } [v_1; v_2] \\
| \langle \text{LABEL lab} \rangle \rightarrow \text{emit_lab lab} \\
| \langle \text{JUMP lab} \rangle \rightarrow \text{gen "b" [Label lab]} \\
| \langle \text{JUMPC (Eq, lab), } t_1, t_2 \rangle \rightarrow \\
\quad \text{let } v_1 = e_{\text{reg}} t_1 \text{ anyreg in} \\
\quad \text{let } v_2 = e_{\text{rand}} t_2 \text{ in} \\
\quad \text{gen "cmp" } [v_1; v_2]; \\
\quad \text{gen "beq" [Label lab]} \\
| \ldots
\]

Working together, these functions will take an optree and output assembly language code for it. They rely on \( \text{gen} \) and \( \text{gen_reg} \) to keep track of which registers are in use, so that \( \text{gen_reg} \) never tries to put a value in a register that is already occupied by another value that is still wanted. In principle, this can be achieved with a simple scheme where we keep track of the set of registers that are free. Then \( \text{gen_reg} \) can choose a free register and remove it from the set, and both \( \text{gen} \) and \( \text{gen_reg} \) can look for registers among the inputs of the instruction and return them to the free set. This covers at least the common case where each value is used once.
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.

In addition to these, it is helpful in even a simple compiler to include a stage that implements 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, common sub-expression elimination transforms the statement \( x := x \times x \) from the single tree:

\[
\langle \text{STOREW},
\langle \text{TIMES},
\langle \text{LOADW}, (\text{GLOBAL } x) \rangle,
\langle \text{LOADW}, (\text{GLOBAL } x) \rangle\rangle,
\langle \text{GLOBAL } x \rangle\rangle
\]

into a sequence of three trees that communicate via two temporaries:

\[
\langle \text{DEFTEMP 1, (GLOBAL } x) \rangle
\langle \text{DEFTEMP 2, LOADW, (TEMP 1)} \rangle
\langle \text{STORE, (TIMES, (TEMP 2), (TEMP 2)), (TEMP 1)} \rangle
\]

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

\[
\text{set t1, } \_x
\text{ldr t2, [t1]}
\text{mul u1, t2, t2}
\text{str u1, [t1]}
\]

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 DAG that have more than one parent and compute their values into temps. For example, take a tree for \((x-y) \times (x-y) + x\) (see Figure 8.7). Turn it into a DAG by sharing subtrees (Figure 8.8), then use temps to turn it back into trees:

\[
\langle \text{DEFTEMP 1, (LOADW, (LOCAL } x) \rangle
\langle \text{DEFTEMP 2, MINUS, (TEMP 1), (LOADW, (LOCAL } y) \rangle\rangle
\langle \text{PLUS, (TIMES, (TEMP 2), (TEMP 2)), (TEMP 1)} \rangle
\]
8.3 Lecture 15: Common sub-expressions

Figure 8.7: Expression tree

Figure 8.8: Directed acyclic graph

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 sharing:

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

(Note that the forms \(\langle\text{DEFTEMP} \ n, \_\rangle\) and \(\langle\text{TEMP}, \ n\rangle\) 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.

<table>
<thead>
<tr>
<th>Value</th>
<th>Operation</th>
<th>Ref. count</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>⟨LOCAL x⟩</td>
<td>1</td>
</tr>
<tr>
<td>[2]</td>
<td>⟨LOADW, [1]⟩</td>
<td>2</td>
</tr>
<tr>
<td>[3]</td>
<td>⟨LOCAL y⟩</td>
<td>1</td>
</tr>
<tr>
<td>[4]</td>
<td>⟨LOADW, [3]⟩</td>
<td>1</td>
</tr>
<tr>
<td>[5]</td>
<td>⟨MINUS, [2], [3]⟩</td>
<td>2</td>
</tr>
<tr>
<td>[6]</td>
<td>⟨TIMES, [5], [3]⟩</td>
<td>1</td>
</tr>
<tr>
<td>[7]</td>
<td>⟨PLUS, [6], [2]⟩</td>
<td>0</td>
</tr>
</tbody>
</table>

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

\[ y := x - y; z := x - y \]

\begin{align*}
\langle \text{STORE}, \\
\langle \text{MINUS}, \langle \text{LOADW}, \langle \text{LOCAL x} \rangle \rangle, \\
\langle \text{LOADW}, \langle \text{LOCAL y} \rangle \rangle, \\
\langle \text{LOCAL y} \rangle \rangle \\
\langle \text{STORE}, \\
\langle \text{MINUS}, \langle \text{LOADW}, \langle \text{LOCAL x} \rangle \rangle, \\
\langle \text{LOADW}, \langle \text{LOCAL y} \rangle \rangle, \\
\langle \text{LOCAL z} \rangle \rangle
\end{align*}

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Operation</th>
<th>Ref. count</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>⟨LOCAL x⟩</td>
<td>1</td>
</tr>
<tr>
<td>[2]</td>
<td>⟨LOADW, [1]⟩</td>
<td>2</td>
</tr>
<tr>
<td>[3]</td>
<td>⟨LOCAL y⟩</td>
<td>2</td>
</tr>
<tr>
<td>[4]</td>
<td>⟨LOADW, [3]⟩</td>
<td>1</td>
</tr>
<tr>
<td>[5]</td>
<td>⟨MINUS, [2], [3]⟩</td>
<td>2</td>
</tr>
<tr>
<td>[6]</td>
<td>⟨STOREW, [5], [3]⟩</td>
<td>0</td>
</tr>
</tbody>
</table>

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 \text{LOADW} node [4] that references \( y \), that is, removing it from the hash table while leaving it part of the DAG.
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 \langle Loadw, [3] \rangle 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.

\[
\begin{align*}
[7] & \quad \langle \text{Loadw}, [3] \rangle & 1 \\
[8] & \quad \langle \text{Minus}, [2], [7] \rangle & 1 \\
[9] & \quad \langle \text{Local} \ z \rangle & 1 \\
[10] & \quad \langle \text{Storew}, [8], [3] \rangle & 0
\end{align*}
\]

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:

\[
\begin{align*}
\langle \text{Deftemp 1}, \langle \text{Loadw}, \langle \text{Local} \ x \rangle \rangle \rangle \\
& \quad \text{ldr} \ t1, [fp, \ #x] \\
\langle \text{Deftemp 2}, \langle \text{Minus}, \langle \text{Temp} \ 1 \rangle, \langle \text{Loadw}, \langle \text{Local} \ y \rangle \rangle \rangle \rangle \\
& \quad \text{ldr} \ u1, [fp, \ #y] \\
& \quad \text{sub} \ t2, t1, u1 \\
\langle \text{Storew}, \langle \text{Temp} \ 2 \rangle, \langle \text{Local} \ y \rangle \rangle \\
& \quad \text{stw} \ t2, [fp, \ #y] \\
\langle \text{Storew}, \langle \text{Minus}, \langle \text{Temp} \ 1 \rangle, \langle \text{Temp} \ 2 \rangle \rangle, \langle \text{Local} \ z \rangle \rangle \\
& \quad \text{sub} \ u2, t1, t2 \\
& \quad \text{stw} \ u2, [fp, \ #z]
\end{align*}
\]

An example: Consider the procedure

\[
\begin{align*}
\text{proc} \ \text{swap(i, j: integer);} \\
& \quad \text{var} \ t: \text{reg integer;} \\
& \quad \text{begin} \\
& \quad \quad \ t := a[i]; \\
& \quad \quad \ a[i] := a[j]; \\
& \quad \quad \ a[j] := t \\
& \quad \text{end;}
\end{align*}
\]

where t lives in a register and a is global. Initial code is the three trees,

\[
\begin{align*}
\langle \text{Storew}, \\
& \quad \langle \text{Loadw}, \\
& \quad \quad \langle \text{Offset}, \langle \text{Global} \ a \rangle, \\
& \quad \quad \quad \langle \text{Lsl}, \langle \text{Loadw}, \langle \text{Local} \ 40 \rangle \rangle, \langle \text{Const} \ 2 \rangle \rangle \rangle, \\
& \quad \quad \langle \text{Regvar} \ 0 \rangle \rangle \\
\langle \text{Storew}, \\
& \quad \langle \text{Loadw}, \\
& \quad \quad \langle \text{Offset}, \langle \text{Global} \ a \rangle, \\
& \quad \quad \quad \langle \text{Lsl}, \langle \text{Loadw}, \langle \text{Local} \ 44 \rangle \rangle, \langle \text{Const} \ 2 \rangle \rangle \rangle,
\end{align*}
\]
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,

\[
\langle \text{Deftemp} \, 1, \langle \text{Global} \, _a \rangle \rangle
\]

\[
\langle \text{Deftemp} \, 2,
\quad \langle \text{Offset}, \langle \text{Temp} \, 1 \rangle, \langle \text{LSL}, \langle \text{LOADW}, \langle \text{Local} \, 40 \rangle \rangle, \langle \text{Const} \, 2 \rangle \rangle \rangle
\]

\[
\langle \text{Storew}, \langle \text{LOADW}, \langle \text{Temp} \, 2 \rangle \rangle, \langle \text{Regvar} \, 0 \rangle \rangle
\]

\[
\langle \text{Deftemp} \, 3,
\quad \langle \text{Offset}, \langle \text{Temp} \, 1 \rangle, \langle \text{LSL}, \langle \text{LOADW}, \langle \text{Local} \, 44 \rangle \rangle, \langle \text{Const} \, 2 \rangle \rangle \rangle
\]

\[
\langle \text{Storew}, \langle \text{LOADW}, \langle \text{Temp} \, 3 \rangle \rangle, \langle \text{Temp} \, 2 \rangle \rangle
\]

\[
\langle \text{Storew}, \langle \text{LOADW}, \langle \text{Regvar} \, 0 \rangle \rangle, \langle \text{Temp} \, 3 \rangle \rangle
\]

from which we can generate the code,

\[
\langle \text{Deftemp} \, 1, \langle \text{Global} \, _a \rangle \rangle
\]

\[
\quad \text{set} \ t1, \ _a
\]

\[
\langle \text{Deftemp} \, 2,
\quad \langle \text{Offset}, \langle \text{Temp} \, 1 \rangle, \langle \text{LSL}, \langle \text{LOADW}, \langle \text{Local} \, 40 \rangle \rangle, \langle \text{Const} \, 2 \rangle \rangle \rangle
\]

\[
\quad \text{ldr} \ u1, \ [\text{fp}, \ #40]
\quad \text{lsl} \ u2, \ u1, \ #2
\quad \text{add} \ t2, \ t1, \ u2
\]

\[
\langle \text{Storew}, \langle \text{LOADW}, \langle \text{Temp} \, 2 \rangle \rangle, \langle \text{Regvar} \, 0 \rangle \rangle
\]

\[
\langle \text{Deftemp} \, 3,
\quad \langle \text{Offset}, \langle \text{Temp} \, 1 \rangle, \langle \text{LSL}, \langle \text{LOADW}, \langle \text{Local} \, 44 \rangle \rangle, \langle \text{Const} \, 2 \rangle \rangle \rangle
\]

\[
\quad \text{ldr} \ v0, \ [t2]
\]

\[
\langle \text{Deftemp} \, 3,
\quad \langle \text{Offset}, \langle \text{Temp} \, 1 \rangle, \langle \text{LSL}, \langle \text{LOADW}, \langle \text{Local} \, 44 \rangle \rangle, \langle \text{Const} \, 2 \rangle \rangle \rangle
\]

\[
\quad \text{ldr} \ u3, \ [\text{fp}, \ #44]
\quad \text{lsl} \ u4, \ u3, \ #2
\quad \text{add} \ t3, \ t1, \ u4
\]

\[
\langle \text{Storew}, \langle \text{LOADW}, \langle \text{Temp} \, 3 \rangle \rangle, \langle \text{Temp} \, 2 \rangle \rangle
\]

\[
\quad \text{ldr} \ u5, \ [t3]
\quad \text{str} \ u5, \ [\text{t2}]
\]

\[
\langle \text{Storew}, \langle \text{LOADW}, \langle \text{Regvar} \, 0 \rangle \rangle, \langle \text{Temp} \, 3 \rangle \rangle
\]

\[
\quad \text{str} \ v0, \ [\text{t3}]
\]

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. To avoid the over-enthusiastic identification of common sub-expressions, it’s necessary to introduce an interaction between CSE and instruction selection.
Procedure calls: It’s convenient to allocate each procedure call to a temp, even when they cannot be shared; then calls will no longer be nested inside other operations. This requires some care in the presence of side effects; e.g., \( y + \text{nasty}(x) \) may not be equivalent to
\[
t := \text{nasty}(x); \ y + t
\]
if \text{nasty} has a side effect on \( x \), and we should pay attention to what the language manual says.

Our treatment, allowing the transformation, is to generate the following initial code for the statement \( z := y + \text{nasty}(x) \) (see lab4/test/nasty.p).

\[
\langle \text{STOREW}, \ \\
\langle \text{PLUS}, \ \\
\langle \text{LOADW}, \langle \text{GLOBAL}\_y \rangle \rangle, \ \\
\langle \text{CALL} 1, \ \\
\langle \text{GLOBAL}\_\text{nasty} \rangle, \ \\
\langle \text{STATLINK}, \langle \text{CONST} 0 \rangle \rangle, \ \\
\langle \text{ARG} 0, \langle \text{LOADW}, \langle \text{GLOBAL}\_x \rangle \rangle \rangle \rangle, \ \\
\langle \text{GLOBAL}\_z \rangle \rangle
\]

The compiler next transforms the code so that the result of the call is assigned to a temp.

\[
\langle \text{DEFTEMP} 1, \ \\
\langle \text{CALL} 1, \ \\
\langle \text{GLOBAL}\_\text{nasty} \rangle, \ \\
\langle \text{STATLINK}, \langle \text{CONST} 0 \rangle \rangle, \ \\
\langle \text{ARG} 0, \langle \text{LOADW}, \langle \text{GLOBAL}\_x \rangle \rangle \rangle \rangle \rangle, \ \\
\langle \text{STOREW}, \langle \text{PLUS}, \langle \text{LOADW}, \langle \text{GLOBAL}\_y \rangle \rangle, \langle \text{TEMP} 1 \rangle, \langle \text{GLOBAL}\_z \rangle \rangle \rangle
\]

Flatten the code.

\[
\langle \text{ARG} 0, \langle \text{LOADW}, \langle \text{GLOBAL}\_x \rangle \rangle \rangle, \ \\
\langle \text{STATLINK}, \langle \text{CONST} 0 \rangle \rangle, \ \\
\langle \text{DEFTEMP} 1, \langle \text{CALL} 1, \langle \text{GLOBAL}\_\text{nasty} \rangle \rangle \rangle, \ \\
\langle \text{STOREW}, \langle \text{PLUS}, \langle \text{LOADW}, \langle \text{GLOBAL}\_y \rangle \rangle, \langle \text{TEMP} 1 \rangle, \langle \text{GLOBAL}\_z \rangle \rangle \rangle
\]

Translate each tree.

\[
\langle \text{ARG} 0, \langle \text{LOADW}, \langle \text{GLOBAL}\_x \rangle \rangle \rangle, \ \\
\text{set} \ r0, \ _x \ \\
\text{ldr} \ r0, \ [r0] \ \\
\langle \text{STATLINK}, \langle \text{CONST} 0 \rangle \rangle \ \\
\langle \text{DEFTEMP} 1, \langle \text{CALL} 1, \langle \text{GLOBAL}\_\text{nasty} \rangle \rangle \rangle, \ \\
\text{bl} \ _\text{nasty} \ \\
\langle \text{STOREW}, \ \\
\langle \text{PLUS}, \langle \text{LOADW}, \langle \text{GLOBAL}\_y \rangle \rangle, \langle \text{TEMP} 1 \rangle, \langle \text{GLOBAL}\_z \rangle \rangle \rangle, \ \\
\text{set} \ r1, \ _y \ \\
\text{ldr} \ r1, \ [r1] \ \\
\text{add} \ r0, \ r1, \ r0 \ \\
\text{set} \ r0, \ _z \ \\
\text{str} \ r0, \ [r1]
\]
For another example, let’s consider this procedure call, which occurs in the test case lab4/test/sumpower.p.

\[
\text{return } \text{sum}(a, b, \text{pow})
\]

The front end translates this statement into the tree,

\[
\langle \text{RESULTW}, \\
\langle \text{CALL 4}, \\
\langle \text{GLOBAL _sum}, \\
\langle \text{STATLINK}, \langle \text{CONST 0} \rangle), \\
\langle \text{ARG 0}, \langle \text{LOADW}, \langle \text{LOCAL 40} \rangle) \rangle, \\
\langle \text{ARG 1}, \langle \text{LOADW}, \langle \text{LOCAL 44} \rangle) \rangle, \\
\langle \text{ARG 2}, \langle \text{GLOBAL _pow} \rangle), \\
\langle \text{ARG 3}, \langle \text{LOCAL 0} \rangle) \rangle \rangle \rangle
\]

We can see that, because \text{sum} is a higher-order function, the front end has implemented the three parameters with four low-level arguments. The procedure \text{sum} is global, however, so the static link for the call is zero.

Next, the process of CSE (or a simpler pass with the same effect) lifts out the procedure call and assigns it to a temp, even though it is not shared.

\[
\langle \text{DEFTEMP 1}, \\
\langle \text{CALL 4}, \\
\langle \text{GLOBAL _sum}, \\
\langle \text{STATLINK}, \langle \text{CONST 0} \rangle), \\
\langle \text{ARG 0}, \langle \text{LOADW}, \langle \text{LOCAL 40} \rangle) \rangle, \\
\langle \text{ARG 1}, \langle \text{LOADW}, \langle \text{LOCAL 44} \rangle) \rangle, \\
\langle \text{ARG 2}, \langle \text{GLOBAL _pow} \rangle), \\
\langle \text{ARG 3}, \langle \text{LOCAL 0} \rangle) \rangle \rangle \rangle
\]

\[
\langle \text{RESULTW}, \langle \text{TEMP 1} \rangle \rangle
\]

This transformation is particularly helpful if procedure calls are nested, because it lets the code generator treat each call in isolation from the others. Before selecting instructions, the compiler flattens the trees into the sequence,

\[
\langle \text{ARG 3}, \langle \text{LOCAL 0} \rangle) \\
\langle \text{ARG 2}, \langle \text{GLOBAL _pow} \rangle) \\
\langle \text{ARG 1}, \langle \text{LOADW}, \langle \text{LOCAL 44} \rangle) \rangle \\
\langle \text{ARG 0}, \langle \text{LOADW}, \langle \text{LOCAL 40} \rangle) \rangle \\
\langle \text{STATLINK}, \langle \text{CONST 0} \rangle) \\
\langle \text{DEFTEMP 1}, \langle \text{CALL 4}, \langle \text{GLOBAL _sum} \rangle) \\
\langle \text{RESULTW}, \langle \text{TEMP 1} \rangle \rangle
\]

These trees can then be translated individually.

\[
\langle \text{ARG 3}, \langle \text{LOCAL 0} \rangle) \\
\text{mov r3, fp} \\
\langle \text{ARG 2}, \langle \text{GLOBAL _pow} \rangle) \\
\text{set r2, _pow} \\
\langle \text{ARG 1}, \langle \text{LOADW}, \langle \text{LOCAL 44} \rangle) \rangle \\
\text{ldr r1, [fp, #44]} \\
\langle \text{ARG 0}, \langle \text{LOADW}, \langle \text{LOCAL 40} \rangle) \rangle \\
\text{ldr r0, [fp, #40]}
\]
Each \langle \text{Arg} \, n, \ldots \rangle tree generates a single instruction. Note that \langle \text{Local} \, 0 \rangle is the \textit{value} of the frame pointer, whereas \langle \text{Loadw}, \langle \text{Local} \, 40 \rangle \rangle is the \textit{contents} of a word in the stack frame. The static link is known to be zero, so we can be sure that the called procedure will never look at it, and there is no need for any code to pass it. The result of the call is in register \( r0 \), and that is where the \textit{Resultw} operation must leave it; so there is no code needed for the \textit{Resultw} tree either.

### 8.4 Lecture 16: Register allocation

The remaining component needed for a working compiler is some way of assigning real registers \( r0, r1, \ldots \) to the symbolic registers \( ui \) and \( tj \) 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 \textit{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 that 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 \( e\_reg \) that takes an \textit{optree}, possibly outputs some instructions, and returns an \textit{operand} value that denotes the register containing the result. We'll add an additional argument, also of type \textit{operand}:

\[
e\_reg : \text{optree} \rightarrow \text{operand} \rightarrow \text{operand}
\]

This added argument specifies which register should be chosen for the result, and the \textit{operand} that is returned indicates which register was actually used. We allow the special value \textit{anyreg} as the argument if we don't care.

For example, the \texttt{mul} instruction on the ARM requires its two operands in registers, so we make the rule,

\[
\begin{align*}
\text{let } & \text{reg } e\_reg \ t \ r = \\
\text{match } & t \text{ with } \ldots \\
| \langle \text{Binop} \, \text{Times}, \ t_1, \ t_2 \rangle \rightarrow \\
& \text{let } v_1 = \text{e\_reg } t_1 \text{ anyreg in} \\
& \text{let } v_2 = \text{e\_reg } t_2 \text{ anyreg in} \\
& \text{gen\_reg } \text{"mul" } [r; v_1; v_2]
\end{align*}
\]

We use \textit{anyreg} twice in the recursive calls to denote the fact that we don't care what registers are used for the two operands of the \texttt{mul} instruction; we do care that they are different, but we will deal with that in a moment. The final call to \textit{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 \textit{e\_reg}.
This subroutine *gen_reg* both emits an instruction and looks after allocating a register for the result.

```plaintext
let gen_reg op (r :: rands) =
  List.iter release rands
let r' = fix_reg r in
emit op (r' :: rands);
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 *anyreg* then *r'* will be the first available real register.) The call to *emit* 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 permits instructions that use the same register as both input and output.

In addition to *e_reg*, there are other mutually recursive subroutines *e_rand*, *e_addr* and *e_stmt*. They deal with registers in a similar way via *gen_reg* and (for *e_stmt*) a similar function *gen* that does not have a result register.

**Temps:** To implement temps, we need a rule in *e_stmt* and another in *e_reg*. In *e_stmt*, we compile ⟨*Deftemp n, t₁*⟩ by evaluating *t₁* into any register (this time preferring a callee-save register by specifying *tempreg*) and remembering that as the location of the temp:

```plaintext
let rec e_stmt =
  function . . .
  | ⟨*Deftemp n, t₁*⟩ →
    let v₁ = e_reg t₁ tempreg in
    def_temp n (reg_of v₁)
```

In *e_reg* we compile ⟨*TEMP n*⟩ into an optional move instruction:

```plaintext
let rec e_reg t₁ r =
  match t with . . .
  | ⟨*TEMP n*⟩ →
    gen_move r (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. 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,

\[ \langle \text{Arg} 1, \langle \text{Const} 3 \rangle \rangle, \langle \text{Arg} 0, \langle \text{Loadw}, \langle \text{Local} x \rangle \rangle \rangle, \langle \text{Call} \langle \text{Global} f \rangle \rangle. \]

Then for \( i < 4 \) we implement \( \langle \text{Arg} i, t \rangle \) like this:

```plaintext
let rec e_stmt =
  function . . .
  | \langle Arg i, t \rangle \rightarrow
     spill_temps [R i];
     e_reg t (R i)
```

so that the tree is evaluated into the specific register. Then the Call operation is implemented by

```plaintext
| \langle Call n, \langle Global f \rangle \rangle \rightarrow
  spill_temps volatile;
  gen "bl" [Global f]
```

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 \( r_0 \ldots r_3 \).

An example: The expression \( g(a) + g(b) \) is the simplest that causes a spill, generating the instruction \( \text{mov r4, r0} \) shown below.

```plaintext
\langle \text{Arg} 0, \langle \text{Loadw}, \langle \text{Local} 40 \rangle \rangle \rangle
  ldr r0, [fp, #40]
\langle \text{DefTemp} 1, \langle \text{Call} 1, \langle \text{Global} _g \rangle \rangle \rangle
  bl _g
\langle \text{Arg} 0, \langle \text{Loadw}, \langle \text{Local} 44 \rangle \rangle \rangle
  mov r4, r0
  ldr r0, [fp, #44]
\langle \text{DefTemp} 2, \langle \text{Call} 1, \langle \text{Global} g \rangle \rangle \rangle
  bl _g
\langle \text{Storew}, \langle \text{Plus}, \langle \text{Temp} 1 \rangle, \langle \text{Temp} 2 \rangle \rangle, \langle \text{Local} -4 \rangle \rangle
  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.

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 8.3 and 8.4** 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, 35–39, 41–43 and 48 in Appendix D.)

5.2 The ARM has a multiply instruction \( \text{mul } r1, r2, r3 \) 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 Consider the following data type and procedure:

```
type ptr = pointer to rec;
  rec = record data: integer; next: ptr; end;

proc sum(p: ptr): integer;
  var q: ptr; s: integer;
begin
  q := p; s := 0;
  while q <> nil do
    s := s + q↑.data;
    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.

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

\[
\begin{align*}
& s := s + q↑.data; \\
& q := q↑.next
\end{align*}
\]

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.)

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

5.6 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.

5.7 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.

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

\[
\text{movm } [r1], [r2]
\]

with the effect \( \text{mem}_4[r1] \leftarrow \text{mem}_4[r2] \); 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 the tree grammar in Figure 14.2 of the handout 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.
Machine code

5.9  [part of 2012/3, edited]
(a) Show the trees that represent the statement
\[
a[i][j] := a[i] + i
\]
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 elim-
nination 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.

5.10  The following procedure swaps two specified elements of a global
array and has a local variable \(t\) that lives in a register.

\begin{verbatim}
var a: array 10 of integer;
proc swap(i, j: integer);
var t: integer;
begin
  t := a[i]; a[i] := a[j]; a[j] := t
end;
\end{verbatim}
The procedure is translated into assembly language for a RISC machine by a
compiler whose intermediate representation is operator trees. Parameters
\(i\) and \(j\) are available in the procedure body at offsets 40 and 44 from the frame
pointer, and \(t\) is allocated to register \(r4\).

(a) Show the sequence of trees that corresponds to the three assignments
in the body of procedure \texttt{swap}. Omit array bounds checks and simplify
as much as possible.

(b) The compiler implements common sub-expression elimination by value
numbering in basic blocks. Show the DAG that results from applying
value numbering to the trees in your answer to part (a), assuming perfect
alias analysis, and the sequence of trees that results from introducing
temps for shared nodes in the DAG.

(c) List the aliasing assumptions made in generating the DAG and how they
could be justified, identifying any that might be missed by a compiler
that approximates alias analysis conservatively.

(d) Show assembly language for a typical RISC machine corresponding to
the transformed sequence of trees, making clear the effect of each in-
struction.

(e) Assess whether the code generated is optimal under reasonable as-
sumptions, and suggest any possible improvements.
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 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:

\[
\text{set } r1, _a \\
\text{ldr } r2, [fp, #40] \\
\text{lsl } r2, r2, #2 \\
\text{add } r1, r1, r2 \\
\text{ldr } r0, [r1] \\
\]

Here the base address of \(a\) is put in \(r1\) and the quantity \(4i\) 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 \text{ldr} instruction:

\[
\text{set } r1, _a \\
\text{ldr } r2, [fp, #40] \\
\text{lsl } r2, r2, #2 \\
\text{ldr } r0, [r1, r2] \\
\]

There is a better code sequence still that performs the scaling also as part of the final instruction:

\[
\text{set } r1, _a \\
\text{ldr } r2, [fp, #40] \\
\text{ldr } r0, [r1, r2, LSL #2] \\
\]

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

```c
set r1, _a
ldr r2, [fp, #40]
add r1, r1, r2, LSL #2
```

To make the instruction selector generate this code, you will need to add a new kind of operand that includes $r2, \text{LSL} \, #2$ as an instance.

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 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

```c
ocamlc -l ../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

```
⟨Offset, t1, ⟨Const n⟩⟩
```

and these are expanded by `nodexp` into the form

```
Node (Offset, [t1; Node (Const n, [])]),
```

which is expressed directly in terms of the type of optrees,

```
type optree = Node of inst * optree 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 `ocamlex` and `ocamlyacc`.

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
9.1 Building and testing the compiler

(* 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}
  .ltorg
  @ End
]]*)

Figure 9.1: Compiler test case

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 test0 to do this for all the test cases.

- 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.

- 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 no more powerful than a Raspberry Pi would provide the computing service for an entire university.

Here are the commands executed by make test1-print:

```bash
1 $ make test1-print
2 arm-linux-gnueabihf-gcc -marm -march=armv6 -c pas0.c -o pas0.o
```
Stage by stage:

- First, the file `pas0.c` is compiled with the ARM C compiler [line 2].
- Next, our compiler `ppc` 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 next command extracts the expected code from the source file and compares it with the code that was actually produced [line 6]. Here, there are no differences, but as you improve the code generator, you will start to see longer sequences being replaced by shorter ones.

- The file `b.s` is next assembled and linked with the library `pas0.o` to create an ARM executable `b.out` [line 7]; 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 8].

- Finally, the expected output is extracted from the test case and compared with the actual output [line 9]. 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.\footnote{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.}

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, 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
9.2 More addressing modes

The addresses that appear in load and store instructions are compiled by the function $e_{addr}$ shown in Figure 9.2. As it stands, this function implements three rules. The first,

\[
addr \rightarrow \langle \text{LOCAL } n \rangle \ {\langle \text{fp, } \#n \rangle} ,
\]

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,

\[
addr \rightarrow \langle \text{OFFSET, } \text{reg, } \langle \text{CONST } n \rangle \rangle \ {\langle \text{reg, } \#n \rangle} ,
\]

matches in other places where an address includes a constant offset, using the same addressing mode. The address that is written $[\text{reg, } \#n]$ in assembly language is represented internally in the compiler by the operand value $\text{Index} (\text{reg, } n)$. As you will see, both these rules are qualified in the implemented code generator with the condition $\text{fits_offset } n$ to ensure that the offset $n$ will fit in the 12-bit offset field of an $\text{ldr}$ or $\text{str}$ instruction. A third rule,

\[
addr \rightarrow \text{reg} \ {\langle \text{reg} \rangle} ,
\]

matches all other addresses, arranging for the address to be computed into a register, then using the indirect address $[\text{reg}]$, or $\text{Index} (\text{reg, } 0)$.

We can do better by adding additional rules. The machine provides another addressing mode, written $[\text{reg}_1, \text{reg}_2]$ in the assembler, that takes the sum of two registers; this is represented in the compiler by an (existing) operand value $\text{Index}_2 (\text{reg}_1, \text{reg}_2, 0)$ – where the 0 represent the fact that the value of $\text{reg}_2$ is not shifted. The rule,

\[
addr \rightarrow \langle \text{OFFSET, } \text{reg}_1, \text{reg}_2 \rangle \ {\langle \text{reg}_1, \text{reg}_2 \rangle} ,
\]

uses this mode for addresses that are the sum of two others. Better still, we can also add the rule,
addr → ⟨OFFSET, reg₁, ⟨BINOP Lsl, reg₂, ⟨CONST n⟩⟩⟩
{ [reg₁, reg₂, LSL #n] },

that matches sums where the offset is shifted in order to multiply it by a power of two and produces an address that we can represent with the operand value Index₂(reg₁, reg₂, n).

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 -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.

9.3 Shifter operands

Another function, e.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 introducing a new constructor Shift (reg, n) for the type operand, defined in target.mli and target.ml. That, in turn, will mean extending other definitions in the compiler that deal with operand values. 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 to compute addresses of array elements in a way that can use this rule.
9.4 Multiplying by a constant

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:

\[
\begin{align*}
\text{mov} & \ r1, \ #n \\
\text{mul} & \ r0, \ r0, \ r1
\end{align*}
\]

(Additionally, the \text{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 \text{lsl} instructions (which are really \text{mov} instructions that use a shifter operand). We can also multiply \( r0 \) by 3 with an instruction that adds twice \( r0 \) to itself:

\[
\text{add} \ r0, \ r0, \ r0, \ \text{LSL} \ #1
\]

Again, we can multiply by 15 with a reverse-subtract instruction that takes \( r0 \) away from 16 times itself:

\[
\text{rsb} \ r0, \ r0, \ r0, \ \text{LSL} \ #4
\]

And we can add 10 times \( r0 \) to \( r1 \) by first computing 5 times \( r0 \) in \( r2 \), then multiplying by 2 as we add:

\[
\begin{align*}
\text{add} & \ r2, \ r0, \ r0, \ \text{LSL} \ #2 \\
\text{add} & \ r1, \ r1, \ r2, \ \text{LSL} \ #1
\end{align*}
\]

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 \textit{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 \textit{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 \text{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 \text{mul} instructions you can eliminate.

A small detail: after computing a value into a register \( r1 \), it's tempting to multiply it by 3 (say) by writing

\[
\text{gen_reg}^\text{"add"} \ r [\text{Register} \ r1; \ \text{Shift} (r1, 1)]
\]

but this will lead to an inconsistency in the state of the register allocator. Normally, the \text{gen_reg} function looks after the details of register allocation, first freeing the registers used by the input operands of the instruction, then allocating a register to hold the result. Here, however, the automatic treatment
of register allocation would give the wrong result, because \texttt{gen_reg} would free register \texttt{r1} twice. The net result might be that the usage count of the register could go negative, and once that happens all bets about the future sanity of register allocation are off the table. The cure is to insert an extra call of \texttt{reserve_reg}:

\begin{verbatim}
reserve_reg r1;
gen_reg "add" r [Register r1; Shift (r1, 1)]
\end{verbatim}

This bumps up the usage count of register \texttt{r1} to compensate for the subsequent double decrement.

9.5 Going the other way

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 \texttt{(* and *)} in the compiler source.

For example, the rule for \texttt{offset} in the original implementation of \texttt{e_addr} can be deleted, at the expense of computing more addresses into registers, and even the rule for \texttt{local} \texttt{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 \texttt{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?
6.1 A certain programming language has the following abstract syntax for expressions and assignment statements:

\[
\text{type } \text{stmt} = \begin{cases} 
\text{Assign of expr * expr} & (\ast \text{Assignment } e_1 := e_2 *) \\
\text{expr} & \{ e\text{.guts : expr\_guts; e\text{.size : int} } \\
\text{and } \text{expr\_guts} = 
\text{Var of name} & (\ast \text{Variable (name, address) }) \\
\text{Sub of expr * expr} & (\ast \text{Subscript } e_1[e_2] *) \\
\text{Binop of op * expr * expr} & (\ast \text{Binary operator } e_1 \ op \ e_2 *) \\
\text{and } \text{name} = \{ x\text{.name : ident; x\text{.addr : symbol} } \\
\text{and } \text{op} = \{ \text{Plus | Minus | Times | Divide } \\
\end{cases}
\]

Each expression is represented by a record \( e \) with a component \( e\text{.guts} \) that indicates the kind of expression, and a component \( e\text{.size} \) that indicates the size of its value. Each variable \( \text{Var } x \) is annotated with its address \( x\text{.name} \).

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:

\[
\text{type } \text{code} = \begin{cases} 
\text{Const of int} & (\ast \text{Push constant } *) \\
\text{Global of symbol} & (\ast \text{Push symbolic address } *) \\
\text{Load} & (\ast \text{Pop address, push contents } *) \\
\text{Store} & (\ast \text{Pop address, pop value, store } *) \\
\text{Binop of op} & (\ast \text{Pop two operands, push result } *) \\
\text{Seq of code list} & (\ast \text{Sequence of code fragments } *) \\
\end{cases}
\]

(a) Defining whatever auxiliary functions are needed, give the definition of a function \( \text{gen_stmt} : \text{stmt} \rightarrow \text{code} \) that returns the code for an assignment statement. Do not attempt any optimisation at this stage.
Revision problems

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

\[ a[i,j] := b[i,j] \times b[1,1] \]

where the variables \( a, b, i, j \) are declared by

```pascal
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 \textit{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:

```pascal
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 *)
10 Revision problems

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

(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:

```pascal
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.
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. OCaml 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” 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 program-
ing 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 num-
ber of other features that we shall use: in particular, there is a module sys-
tem that allows programs to be split cleanly into independent pieces, with
the source code of each module kept in a separate file and compiled inde-
pendently 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 Ob-
jective 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:

\[
\text{let rec} \quad \text{reverse} = \\
\quad \text{function} \\
\quad \quad [\,] \rightarrow [\,] \\
\quad \quad x :: xs \rightarrow \text{reverse} \; xs @ [x];;
\]

The keywords let rec introduce a recursive function definition; reverse is de-
efined 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 \(\text{reverse} [\,]\)
is [], and the value of \(\text{reverse} (x :: xs)\) 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. Like every top-level phrase in an Objective
CAML program, the definition ends with a double semicolon ;; . The Objective
CAML programs in this course have been formatted for ease of reading, using
different styles of type and special symbols like \(\rightarrow\); when the programs
are entered into a computer, an ASCII form of the language is used, and the
arrow symbol is typed as \(\rightarrow\), for example. Section A.10 on page 124 explains
the correspondence between the two forms.

Objective CAML has a strong, polymorphic type system which does not re-
quire 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

\(\alpha \; \text{list} \rightarrow \alpha \; \text{list}\)

Here, \(\alpha\) 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:

- \(\text{int}\), integers of 31 bits.
- \(\text{char}\), characters, written between single quotes like this: ‘a’, ‘b’, ‘c’.
- \(\text{string}\), 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.

\[
\text{type } expr = \\
\text{Constant of int} \\
\mid \text{Variable of string} \\
\mid \text{Binop of op} \times expr \times expr
\]

\[
\text{and } op = \text{Plus} \mid \text{Minus} \mid \text{Times} \mid \text{Divide};
\]

This definition introduces two types \(expr\) and \(op\). The two definitions are joined by the keyword \(\text{and}\), so that each definition may refer to the other one. The type \(op\) simply consists of the four values \(\text{Plus}\), \(\text{Minus}\), \(\text{Times}\), and \(\text{Divide}\), but the type \(expr\) has a more elaborate recursive structure. An \(expr\) may be simply a constant \(\text{Constant } n\) for some integer \(n\), or a variable \(\text{Variable } x\) named by the string \(x\), but it can also be a compound expression \(\text{Binop } (w, a, b)\), where \(w\) is an operator and \(a\) and \(b\) are other expressions. For example, the expression

\[
x \times (y + 1)
\]

would be represented as the following value of type \(expr\):

\[
\text{Binop } (\text{Times}, \text{Variable } "x", \\
\text{Binop } (\text{Plus}, \text{Variable } "y", \text{Constant } 1)).
\]

Constructors like \(\text{Binop}\) and \(\text{Plus}\) 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 \(\text{eval}\) that gives the value of an arithmetic expression (at least, if it doesn't contain any variables).

\[
\text{let } do\_\text{binop } w \ a \ b = \\
\text{match } w \text{ with} \\
\quad \text{Plus } \rightarrow a + b \\
\quad \text{Minus } \rightarrow a - b \\
\quad \text{Times } \rightarrow a \ast b \\
\quad \text{Divide } \rightarrow a/b;
\]

\[
\text{let rec } \text{eval } = \\
\quad \text{function} \\
\quad \text{Constant } n \rightarrow n \\
\quad \text{Binop } (w, e_1, e_2) \rightarrow \\
\quad \quad \text{do\_\text{binop } } w (\text{eval } e_1) (\text{eval } e_2) \\
\quad \text{Variable } x \rightarrow \\
\quad \quad \text{failwith } "\text{sorry, I don't do variables}";
\]
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:

```ocaml
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

\[
  f(n) = f(0) + f(1) + \ldots + f(n-1) \quad (n > 0),
\]

and the sum on the right-hand side is computed by `g n`.

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

```ocaml
let rec f = function ...
```

can always be replaced by the equivalent form

```ocaml
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
Functions \( \mathit{fst} \) and \( \mathit{snd} \) are provided for extracting the first component \( x \) and the second component \( y \) of an ordered pair \((x, y)\): matching.

\[
\begin{align*}
\mathit{fst} & : \alpha \times \beta \rightarrow \alpha \\
\mathit{snd} & : \alpha \times \beta \rightarrow \beta
\end{align*}
\]

These functions can be defined by pattern matching like this:

\[
\begin{align*}
\text{let } \mathit{fst} \ (x, y) &= x; \\
\text{let } \mathit{snd} \ (x, y) &= y;
\end{align*}
\]

Components of bigger \( n \)-tuples can also be extracted by pattern matching; for example, here is a function that extracts the second component of a 3-tuple:

\[
\text{let } \mathit{second3} \ (x, y, z) = y;
\]

As an alternative to \( n \)-tuples, Objective CAML also provides record types. A type definition like

\[
\text{type } \mathit{def} = \{ \mathit{d.key} : \text{string}; \mathit{d.value} : \text{int} \};
\]

defines a new record type \( \mathit{def} \) with selectors \( \mathit{d.key} \) and \( \mathit{d.value} \). Values of this record type can be constructed by expressions like

\[
\{ \mathit{d.key} = "\text{foo}"; \mathit{d.value} = 3 \}
\]

and if \( d \) is such a value, its components can be extracted as \( d.\mathit{d.key} \) and \( d.\mathit{d.value} \). 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

\[
\text{type } \mathit{couple} = \text{string} \times \text{int};
\]

introduces \( \mathit{couple} \) as an alternative name for the type \( \text{string} \times \text{int} \).

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:

\[
\text{val reverse : } \alpha \text{ list } \rightarrow \alpha \text{ list;;}
\]

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:

\[
\text{(∗ memory.mli ∗)}
\]

\[
\text{type memory;;}
\]

\[
\text{val empty : memory;;}
\]

\[
\text{val insert : string } \rightarrow \text{ int } \rightarrow \text{ memory } \rightarrow \text{ memory;;}
\]

\[
\text{val lookup : string } \rightarrow \text{ memory } \rightarrow \text{ int;;}
\]

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:

\[
\text{(∗ memory.ml ∗)}
\]

\[
\text{type def = } \{ \text{d.key : string; d.value : int } \};;
\]

\[
\text{type memory = def list;;}
\]

\[
\text{let empty = [ ];;;}
\]

\[
\text{let insert x v t = } \{ \text{d.key = x; d.value = v } \} :: t;;
\]

\[
\text{let rec lookup x =}
\]

\[
\text{function}
\]

\[
\text{[ ] } \rightarrow \text{ raise Not_found}
\]

\[
\text{| d :: t } \rightarrow
\]

\[
\text{if d.d.key = x then d.d.value else lookup x t;;}
\]

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
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 \texttt{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 \texttt{raise} is the Objective CAML facility for handling exceptions raised during the evaluation of an expression: for example, one might use \texttt{lookup} in the following expression, which returns the value associated with "Mike" in the table \texttt{age}, or 21 if there is no such value:

\begin{verbatim}
try lookup "Mike" age with Not\_found → 21;;
\end{verbatim}

Though we shall make little use of them, Objective CAML also provides exceptions with arguments; thus \texttt{failwith s} is equivalent to \texttt{raise(Failure s)}, where \texttt{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 \texttt{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 \texttt{Failure} exception with a \texttt{try} block, we shall never do so.

\section*{A.6 References and mutable fields}

In addition to the purely functional data types we've seen so far, Objective CAML provides \textit{reference types}, whose values can be changed. These values or \textit{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 \texttt{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.
A.6 References and mutable fields

• 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:

```ocaml
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:

```ocaml
let divide a b = 
  let rec div 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 divide1 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:

```ocaml
(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:

```ocaml
(* onemem.mli *)
```
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:

```
let rec eval e =
  let v = match e.e_guts with
    Binop (op, 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 `e1` as `e1.e_value`. The only fly in the ointment is that even an unevaluated 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 ← 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

---

Here is the source code for the OCaml program that generates this text:

```ocaml
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:

```
let rec eval e =
  let v = match e.e_guts with
    Binop (op, 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 `e1` as `e1.e_value`. The only fly in the ointment is that even an unevaluated 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 ← t` and `!x` standing for `x.contents`.

---

This source code is designed to be read naturally and can be used to generate the text as shown above.
A.7 Library functions

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:

- $\text{length } xs = n$
- $xs @ ys = [x_0; \ldots; x_{n-1}; y_0; \ldots; y_{m-1}]$
- $\text{hd } xs = x_0$
- $\text{tl } xs = [x_1; \ldots; x_{n-1}]$
- $\text{nth } xs i = x_i$
- $\text{rev } xs = [x_{n-1}; \ldots; x_1; x_0]$

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

- $\text{concat } xss = xs_0 @ xs_1 @ \cdots @ xs_{n-1}$.

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 map2 : (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

- $\text{map } f xs = [f x_0; f x_1; \ldots; f x_{n-1}]$. 
Because Objective CAML is not a purely functional language, it sometimes matters in what order the terms \( f x_0, f x_1, \ldots, f x_{n-1} \) 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 \( \text{map} \) 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 \( \text{map} \) that has this property:

\[
\begin{array}{l}
\text{let rec } \text{map} f = \\
\quad \text{function} \\
\quad \quad [ ] \rightarrow [ ] \\
\quad \quad \mid \ x :: xs \rightarrow \text{let } y = f \ x \ \text{in } y :: \text{map} f \ xs; \\
\end{array}
\]

It is the \texttt{let} expression that ensures that the value of \( f x \) is determined before the recursive call \( \text{map} f \ xs \) is evaluated. The left-to-right evaluation is useful for tasks like defining the function \( \text{totals} \) that returns a list of running totals, like this:

\[
\text{totals} [1; 2; 3; 4; 5] = [1; 3; 6; 10; 15]
\]

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:

\[
\begin{array}{l}
\text{let } \text{totals} \ xs = \\
\quad \text{let } t = \text{ref } 0 \ \text{in} \\
\quad \text{let } f \ x = (t := !t + x; !t) \ \text{in} \\
\quad \text{map} f \ xs; \\
\end{array}
\]

In this simple example, it might be easier and clearer to define \( \text{totals} \) directly by recursion:

\[
\begin{array}{l}
\text{let rec } \text{totals} \ xs = \\
\quad \text{let } t = \text{ref } 0 \ \text{in} \\
\quad \text{let } f \ x = (t := !t + x; !t) \ \text{in} \\
\quad \text{tot} 1 \ t' :: \text{totals} f \ xs; \\
\end{array}
\]

In more complicated situations, however, it is good to have the additional flexibility that a left-to-right \( \text{map} \) provides.

The function \( \text{iter} \) is similar to \( \text{map} \), in that evaluating \( \text{iter} f \ xs \) entails evaluating \( f x_0, f x_1, \ldots, f x_{n-1} \) 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 \( \text{iter} \) is the \texttt{unit} value \( () \).

The two functions \( \text{fold_left} \) and \( \text{fold_right} \) combine all the elements of a list using a binary operation:

\[
\begin{array}{l}
\quad \text{fold_left} f \ a \ xs = f \ldots(f \ (f \ a \ x_0) \ x_1) \ldots \ x_{n-1} \\
\quad \text{fold_right} f \ a \ xs = f \ x_0 \ldots(f \ x_1 \ldots(f \ x_{n-1} \ a) \ldots)
\end{array}
\]

Thus \( \text{fold_left} \) combines the elements of the list starting from the left, and \( \text{fold_right} \) starts from the right. Here are the recursive definitions:

\[\text{The function } \text{fold_right} \text{ takes its arguments in a different order from the function } \text{foldr} \text{ of HASKELL, so that } \text{fold_right} f \ a \ xs = \text{foldr} f \ a \ xs.\]
let rec fold_left f a ys =  
  match ys with  
  | [] → a  
  | x :: xs → fold_left f (f x a) 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 $f(fxy)z = fx(fyz)$ for all $x$, $y$, $z$, and $a$ is a left and right identity element for $f$, so that $fax = x = fxa$, then there is no difference between the results returned by `fold_left` and `fold_right`. This is true, for example, if $fxy = x + y$ 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 → int → int`, we can therefore define a function `sum : int list → 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 (function b x → f x b) a xs.
```

The function `accum` therefore has the type

```
val accum : (α → β → β) → α list → β → β
```

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 x0 y0; f x1 y1; ...; f x(n-1) y(n-1)],

iter_2 f xs ys = f x0 y0; f x1 y1; ...; f x(n-1) y(n-1).
```

A number of deal with lists of ordered pairs of type `(α * β)` list:

```
val combine : α list → β list → (α * β) list;;
val assoc : α → (α * β) list → β;;
val remove_assoc : α → (α * β) list → (α * β) 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 = [(x0, y0); (x1, y1); ...; (xn-1, yn-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; \( \text{assoc} \) raises the exception \texttt{Not_found} if no such value exists. If the list \( ps \) contains more than one pair \((u, v)\) with \( u = x \), then \( \text{assoc} \ x \ ps \) returns the value of \( v \) from the first such pair. Thus \( \text{assoc} \) may be defined as follows:

\[
\text{let rec assoc} \ x =
\begin{align*}
\text{function} & \quad [\] \rightarrow \text{raise Not_found} \\
& \quad (u, v) :: ps \rightarrow \text{if } u = x \text{ then } v \text{ else } \text{assoc} \ x \ ps
\end{align*}
\]

There is another function \( \text{remove_assoc} \) that removes the first pair \((u, v)\), if any, with a specified component \( u = x \):

\[
\text{let rec remove_assoc} \ x =
\begin{align*}
\text{function} & \quad [\] \rightarrow [\] \\
& \quad (u, v) :: ps \rightarrow \\
& \quad \quad \text{if } u = x \text{ then } ps \text{ else } (u, v) :: \text{remove_assoc} \ x \ ps
\end{align*}
\]

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 \texttt{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 \( \alpha \ \text{option} \), defined as follows:

\[
\text{type } \alpha \ \text{option} = \text{Some of } \alpha \ | \ \text{None};;
\]

For example, a language might have a \texttt{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 \texttt{return} statements by in the abstract syntax tree like this:

\[
\text{type } \text{stmt} = \ldots \\
\quad \text{| Return of expr option} \\
\quad \ldots
\]

Then we could use \texttt{return (Some e)} for the \texttt{return} statement that contains expression \( e \), and \texttt{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, mutable data structures of \texttt{arrays} and \texttt{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 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 $\alpha$ has the built-in type $\alpha$ array. The module Array provides the following operations on arrays:

\[
\begin{align*}
\text{val create} & : \text{int} \rightarrow \alpha \rightarrow \alpha \text{array};; \\
\text{val length} & : \alpha \rightarrow \text{int};;
\end{align*}
\]

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 $a.(i) \leftarrow y$.

Strings belong to the built-in type string. There is a library module String that provides the following interface:

\[
\begin{align*}
\text{val create} & : \text{int} \rightarrow \text{string};; \\
\text{val length} & : \text{string} \rightarrow \text{int};; \\
\text{val sub} & : \text{string} \rightarrow \text{int} \rightarrow \text{int} \rightarrow \text{string};;
\end{align*}
\]

A string of length $n$ is created by String.create $n$; all characters of this string are initially undefined. Strings are also created as the value of string constants appearing in Objective CAML programs. The $i$'th character of a string $s$ is written $s.[i]$ for $i = 0, 1, \ldots, n-1$, and it can be set to a character $c$ by evaluating $s.[i] \leftarrow c$. 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 $s_1 \neg s_2$. 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 $s$ $i$ $j$.

### A.8 Mappings and hash tables

[How to use the standard library modules.]

### A.9 Formatted output

The module print provides a general facility for formatted output via the following three functions:

\[
\begin{align*}
(* \ printf & - \text{print on standard output} *) \\
\text{val printf} & : \text{string} \rightarrow \text{arg list} \rightarrow \text{unit};; \\
(* \ fprintf & - \text{print to a file} *) \\
\text{val fprintf} & : \text{out_channel} \rightarrow \text{string} \rightarrow \text{arg list} \rightarrow \text{unit};;
\end{align*}
\]

---

1. 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.
2. This is one of very few places in Objective CAML where an undefined value is created.
3. The notations $s.[i]$ and $s.[i] \leftarrow c$ are abbreviations for String.get $s$ $i$ and String.set $s$ $i$ $c$. 
124 OCaml basics

(* 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 \( \rightarrow \) have been used in place of ASCII combinations like \( \text{->} \). 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

A.10 Computer representation of OCaml programs

<table>
<thead>
<tr>
<th>Symbol</th>
<th>ASCII equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>≠</td>
<td>&lt;&gt;</td>
</tr>
<tr>
<td>≤</td>
<td>&lt;=</td>
</tr>
<tr>
<td>≥</td>
<td>&gt;=</td>
</tr>
<tr>
<td>→</td>
<td>-&gt;</td>
</tr>
<tr>
<td>←</td>
<td>&lt;-&gt;</td>
</tr>
<tr>
<td>⌢</td>
<td>¬</td>
</tr>
</tbody>
</table>

Table A.1: ASCII equivalents for special symbols

would actually look like this when submitted to the Objective CAML compiler:

```ocaml
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:

- 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 env₀ and y₁ of 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..

---

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.
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, LEQ, GT, GEQ, NEQ, AND, OR

These instructions are represented in OCaml programs by BINOP Plus, BINOP Minus, ..., 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 **Monop Not**. 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 PROC 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 In**

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 In**

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** and **Return** instructions apply to procedures that don’t return a result, and the **Pcallw** and **Returnw** instructions apply 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.
RETURNW

Return from a procedure, preserving a one-word result. The instruction expects to find the result on the evaluation stack of the returning procedure, and – together with the PCALLW instruction that was used to invoke it – transfers this result to the evaluation of the calling 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 \((\text{code}, \text{env})\) 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 \((\text{code}, \text{env})\) pair.
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 \textit{Ldgw} and \textit{Stgw}, the conditional branches \textit{Jeq} etc. (all mentioned above), and the following memory operations:

\textit{Ldlw n} \\
Load local word, equivalent to \textit{Local n; Loadw}.

\textit{Stlw n} \\
Store local word, equivalent to \textit{Local n; Storew}.

\textit{Ldnw n} \\
Load indexed word, equivalent to \textit{Const n; Offset; Loadw}.

\textit{Stnw n} \\
Store indexed word, equivalent to \textit{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

\texttt{MODULE Main 0 0} \\
\texttt{IMPORT Lib 0} \\
\texttt{ENDHDR}

These give the program the name \texttt{Main} and record that it uses library routines from a module \texttt{Lib}, which provides the routines \texttt{Lib.Print} and \texttt{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 \texttt{begin} and \texttt{end} towards the end of the source code) into a procedure called \texttt{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 \textit{code} of Keiko code sequences, but just printed out when they are needed.
PROC P n 0 0
This begins a procedure named P that uses n bytes of stack space for local variables. The two zeroes fill spaces that could be used to give a map of where heap pointers are stored in the stack frame (not needed, because we’re not using a garbage collector), and to specify the number of words of evaluation stack the procedure might need (which could be used to detect stack overflow). 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.
A rough guide to ARM

A

Incoming args
| fp+56: beyond first 4
| Space to save
| fp+40: args from r0 - r3
| Saved values of
- fp: r4 - r10, fp, sp, lr
A

Outgoing args
- sp: beyond first 4

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.
- 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.

\footnote{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.}

\footnote{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 is 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:

```
    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.

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:

```
set r0, #12345678  # set r0 to an integer constant
```

Getting the value of static variable `x` into `r1` requires two instructions:

```
set 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 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
```

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 `.ltorg` directive appears in the compiler's output). The `set` 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:

\[
\text{ldr } r0, [r2, r4, LSL #2]
\]

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 \(r0\).

These shifts are available not only in instructions that address memory, but also for the operands of arithmetic instructions. For example, this instruction:

\[
\text{add } r0, r1, r2, \text{LSL } #4
\]

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:

\[
\text{add } r0, r1, r2, \text{LSL } r4
\]

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:

\[
\text{lsr } r1, r2, \#4
\]

\[
\text{mov } r1, r2, \text{LSL } #4
\]

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 \(r2\):

\[
\begin{align*}
\text{cmp } r1, r2 & \quad \text{- compare } r1 \text{ and } r2 \\
\text{mov } r0, \#0 & \quad \text{- set } r0 \text{ to } 0 \\
\text{movlt } r0, \#1 & \quad \text{- reset } r0 \text{ to } 1 \text{ if } r1 < r2
\end{align*}
\]

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.

[1] 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 \(0 \leq r3 < 10\) on line 1234:

\[
\begin{align*}
\text{cmp } r3, \#10 & \quad \text{- compare } r3 \text{ with array bound} \\
\text{seths } r0, \#1234 & \quad \text{- conditionally load line number into } r0 \\
\text{b1hs check} & \quad \text{- conditionally call error routine}
\end{align*}
\]

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 \(r3 \geq 10\) 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, 37, 41 and 42) 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 five non-terminals: reg corresponds to expressions computed in registers; rand and addr to the operands of arithmetic and load/store instructions respectively; call matches procedure calls; and stmt matches at the roots of optrees.

The instruction ‘?mov reg1, reg1’ denotes a mov instruction that may be elided if reg could 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 = reg1’ implies that the sub-tree rooted at reg1 should be computed into the same register as the tree at reg.

D.2 Expressions

1. \texttt{reg} \rightarrow \langle \texttt{CONST} k \rangle \quad \{ \texttt{mov reg, #} k \} \ (\text{when fits.move} k)
2. \texttt{reg} \rightarrow \langle \texttt{CONST} k \rangle \quad \{ \texttt{set reg, #} k \}
3. \texttt{reg} \rightarrow \langle \texttt{LOCAL} 0 \rangle \quad \{ ?\texttt{mov reg, fp} \}
4. \texttt{reg} \rightarrow \langle \texttt{LOCAL} n \rangle \quad \{ \texttt{add reg, fp, #} n \} \ (\text{when fits.add} n)
5. \texttt{reg} \rightarrow \langle \texttt{LOCAL} n \rangle \quad \{ \texttt{set ip, #} n; \texttt{add reg, fp, ip} \}
6. \texttt{reg} \rightarrow \langle \texttt{GLOBAL} x \rangle \quad \{ \texttt{set reg, x} \}
7. \texttt{reg} \rightarrow \langle \texttt{TEMP} n \rangle \quad \{ ?\texttt{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, reg₁)  \{neg reg, reg₁\}
13. reg → (MONOP Not, reg₁)  \{eor reg, reg₁, #1\}
14. reg → (MONOP BitNot, reg₁)  \{mvn reg, reg₁\}
15. reg → (OFFSET, reg₁, ⟨CONST n⟩)  \{add reg, reg₁, #n\} (when fits \_add\_n)
16. reg → (OFFSET, reg₁, rand)  \{add reg, reg₁, rand\}
17. reg → (BINOP Plus, reg₁, rand)  \{add reg, reg₁, rand\}
18. reg → (BINOP Minus, reg₁, rand)  \{sub reg, reg₁, rand\}
19. reg → (BINOP And, reg₁, rand)  \{and reg, reg₁, rand\}
20. reg → (BINOP Or, reg₁, rand)  \{orr reg, reg₁, rand\}
21. reg → (BINOP Lsl, reg₁, rand)  \{lsl reg, reg₁, rand\}
22. reg → (BINOP Lsr, reg₁, rand)  \{lslr reg, reg₁, rand\}
23. reg → (BINOP Asr, reg₁, rand)  \{asr reg, reg₁, rand\}
24. reg → (BINOP BitAnd, reg₁, rand)  \{and reg, reg₁, rand\}
25. reg → (BINOP BitOr, reg₁, rand)  \{orr reg, reg₁, rand\}
26. reg → (BINOP Times, reg₁, reg₂)  \{mul reg, reg₁, reg₂\}
27. reg → (BINOP Eq, reg₁, rand)  \{cmp reg₁, rand;
          mov reg, #0; moveq reg, #1\}
28. reg → (BINOP Neq, reg₁, rand)  \{cmp reg₁, rand;
          mov reg, #0; movne reg, #1\}
29. reg → (BINOP Gt, reg₁, rand)  \{cmp reg₁, rand;
          mov reg, #0; movgt reg, #1\}
30. reg → (BINOP Geq, reg₁, rand)  \{cmp reg₁, rand;
          mov reg, #0; movge reg, #1\}
31. reg → (BINOP Lt, reg₁, rand)  \{cmp reg₁, rand;
          mov reg, #0; movlt reg, #1\}
32. reg → (BINOP Leq, reg₁, rand)  \{cmp reg₁, rand;
          mov reg, #0; movle reg, #1\}
33. reg → (BOUND, reg₁, rand)  \{cmp reg₁, rand;
          sets r0, #line; blhs check\}
          \(target \text{reg} = \text{reg}_1\)
34. reg → (NCHECK, reg₁)  \{cmp reg₁, #0;
          seteq r0, #line; ble nullcheck\}
          \(target \text{reg} = \text{reg}_1\)
D.3 Operands

35. \( \text{rand} \rightarrow \langle \text{CONST } k \rangle \) \#k \ (when \text{fits\_immed } k)

36. \( \text{rand} \rightarrow \langle \text{BINOP Lsl, reg, (CONST } n \rangle \) \reg, \text{LSL } \#n \ (when \ n < 32)

37. \( \text{rand} \rightarrow \langle \text{BINOP Lsl, reg}_{1}, \text{reg}_{2} \rangle \) \reg_{1}, \text{LSL } \reg_{2}

38. \( \text{rand} \rightarrow \reg \) \reg

D.4 Addresses

39. \( \text{addr} \rightarrow \langle \text{LOCAL } n \rangle \) \[fp, \#n\] (when \text{fits\_offset } n)

40. \( \text{addr} \rightarrow \langle \text{OFFSET, reg, (CONST } n \rangle \) \[\reg, \#n\] (when \text{fits\_offset } n)

41. \( \text{addr} \rightarrow \langle \text{OFFSET, reg}_{1}, \text{reg}_{2} \rangle \) \[\reg_{1}, \reg_{2}\]

42. \( \text{addr} \rightarrow \langle \text{OFFSET, reg}_{1}, \langle \text{BINOP Lsl, reg}_{2}, (\text{CONST } n) \rangle \) \[\reg_{1}, \reg_{2}, \text{LSL } \#n\] (when \ n < 32)

43. \( \text{addr} \rightarrow \reg \) \reg

D.5 Procedure calls

44. \( \text{call} \rightarrow \langle \text{GLOBAL } f \rangle \) \bl f

45. \( \text{call} \rightarrow \reg \) \blx \reg

D.6 Statements

46. \( \text{stmt} \rightarrow \langle \text{DEFTEMP } n, \reg \rangle \) (set \text{temp}_{n} = \reg)

47. \( \text{stmt} \rightarrow \langle \text{STOREW, reg, (REGVAR } n \rangle \) (target \reg = \text{regvar}_{n})

48. \( \text{stmt} \rightarrow \langle \text{STOREW, reg, addr} \rangle \) \text{str } \reg, \text{addr}

49. \( \text{stmt} \rightarrow \langle \text{STOREC, reg, (REGVAR } n \rangle \) (target \reg = \text{regvar}_{n})

50. \( \text{stmt} \rightarrow \langle \text{STOREC, reg, addr} \rangle \) \text{strb } \reg, \text{addr}

51. \( \text{stmt} \rightarrow \langle \text{CALL } k, \text{call} \rangle \)

52. \( \text{stmt} \rightarrow \langle \text{DEFTEMP } n, (\text{CALL } k, \text{call}) \rangle \) \?mov \reg, r0

53. \( \text{stmt} \rightarrow \langle \text{RESULTW, reg} \rangle \) (target \reg = r0)

54. \( \text{stmt} \rightarrow \langle \text{LABEL } \text{lab} \rangle \) \.Llab: \)

55. \( \text{stmt} \rightarrow \langle \text{JUMP } \text{lab} \rangle \) \b .Llab

56. \( \text{stmt} \rightarrow \langle \text{JUMPC (Eq, lab), reg, rand} \rangle \) \text{cmp } \reg, \text{rand}; \text{beq .Llab}

57. \( \text{stmt} \rightarrow \langle \text{JUMPC (Lt, lab), reg, rand} \rangle \) \text{cmp } \reg, \text{rand}; \text{blt .Llab}

58. \( \text{stmt} \rightarrow \langle \text{JUMPC (Gt, lab), reg, rand} \rangle \) \text{cmp } \reg, \text{rand}; \text{bgt .Llab}

59. \( \text{stmt} \rightarrow \langle \text{JUMPC (Leq, lab), reg, rand} \rangle \) \text{cmp } \reg, \text{rand}; \text{ble .Llab}

60. \( \text{stmt} \rightarrow \langle \text{JUMPC (Geq, lab), reg, rand} \rangle \) \text{cmp } \reg, \text{rand}; \text{bge .Llab} \)
61. \textit{stmt} \rightarrow \langle \text{JUMPC} (\text{Neq}, \text{lab}), \text{reg}, \text{rand} \rangle \{ \text{cmp reg, rand; bne .Llab} \}

62. \textit{stmt} \rightarrow \langle \text{JCASE} (\text{table}, \text{deflab}), \text{reg} \rangle \{ \text{cmp reg, #n; ldrlo pc, [pc, reg, LSL #2]; b deflab} \}

\text{(where \(n = \text{length table, followed by jump table})

63. \textit{stmt} \rightarrow \langle \text{ARG i, (TEMP n)} \rangle \{ \text{mov ri, temp_n} \} \text{ (when \(i < 4\))}

64. \textit{stmt} \rightarrow \langle \text{ARG i, reg} \rangle \text{ (target reg = ri, when \(i < 4\))}

65. \textit{stmt} \rightarrow \langle \text{ARG i, reg} \rangle \{ \text{str reg, [sp, o]} \}

\text{(where \(o = 4 \times i - 16, \text{when } i \geq 4\))}

66. \textit{stmt} \rightarrow \langle \text{STATLINK, (CONST 0)} \rangle \text{ (no-op)}

67. \textit{stmt} \rightarrow \langle \text{STATLINK, reg} \rangle \text{ (target reg = r4)}
Appendix E

Code listings

E.1 Lab one

E.1.1 lexer.mll

1 (* lab1/lexer.mll *)
2 (* Copyright (c) 2017 J. M. Spivey *)
3 {
4  open Lexing
5  open Tree
6  open Keiko
7  open Parser
8  (* |lineno| -- line number for use in error messages *)
9  let lineno = ref 1
10 (* |make_hash| -- create hash table from list of pairs *)
11  let make_hash n ps =
12  let t = Hashtbl.create n in
13    List.iter (fun (k, v) -> Hashtbl.add t k v) ps;
14    t
15 (* |kwtable| -- a little table to recognize keywords *)
16  let kwtable =
17    make_hash 64
18      [ ("begin", BEGIN); ("do", DO); ("if", IF ); ("else", ELSE);
19        ("end", END); ("then", THEN); ("while", WHILE); ("print", PRINT);
20          ("newline", NEWLINE); ("and", MULOP And); ("div", MULOP Div);
21            ("or", ADDOP Or); ("not", MONOP Not); ("mod", MULOP Mod);
22              ("true", NUMBER 1); ("false", NUMBER 0) ]
23 (* |idtable| -- table of all identifiers seen so far *)
24  let idtable = Hashtbl.create 64
25 (* |lookup| -- convert string to keyword or identifier *)
26  let lookup s =
27    try Hashtbl.find kwtable s with
Not_found ->
    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 []
}

rule token =
parse
  ['A'-'Z''a'-'z'][A-Za-z0-9'_'-']* 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 Leq }
  "" >= "" { RELOP Geq }
  "" := "" { ASSIGN }
  [' ''	']+  { 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
/* punctuation and keywords */
%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

let optflag = ref false

(* |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]  

(* |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))

(* |accum| -- fold_left with arguments swapped *)

let rec accum f xs a =
match xs with
[] -> a
| y::ys -> accum f ys (f y a)

(* |lookup_def| -- find definition of a name, give error is none *)

let lookup_def x env =
erm_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
  accum declare vs env0

(* [check_decls] -- check a sequence of declarations *)
let check_decls ds env0 =
  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]
E.2 Lab two 151

| 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]

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

let translate (Program (ds, ss)) =
  let code = gen_stmt ss in
  printf "PROC MAIN 0 0 0\n" [];
  Keiko.output (if !optflag then Peepopt.optimise code else code);
  printf "RETURN\n" [];
  printf "END\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))

(* |accum| -- fold_left with arguments swapped *)
let rec accum f xs a =
  match xs with
  | [] -> a
  | y::ys -> accum f ys (f y a)

(* |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

(* |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_lab = ""; d_off = 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_lab = sprintf "$_" [fStr x]; d_off = 0 } in
    add_def d env

(* |declare_proc| -- declare a procedure *)
let declare_proc (Proc (p, formals, body)) lev env =
    let lab = sprintf "$$_$" [fStr p.x_name; fNum (label ())] in
    let d = { d_tag = p.x_name;
             d_kind = ProcDef (List.length formals); d_level = lev;
             d_lab = lab; d_off = 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;
    let env' =
        accum (fun (i, x) -> declare_local x lev (arg_base + 4*i))
        (serialize formals) (new_block env) in
    let env'' =

accum (fun (i, x) -> declare_local x lev (loc_base - 4*(i+1)))
  (serialize vars) env' in
150   let env''' =
     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'''

(* annotate -- check and annotate a program *)
let annotate (Program (Block (vars, procs, body))) =
  let env = accum declare_global vars empty in
  let env' = 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

1 (* 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_lab
  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"
      | 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 "PROC $ $ 0 0\n" [fStr d.d_lab; 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

let translate (Program (Block (vars, procs, body))) =
  level := 0;
  printf "PROC MAIN 0 0 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.ml

(* lab4/target.ml *)
(* Copyright (c) 2017 J. M. Spivey *)

open Optree

(* |reg| -- type of Risc86 registers *)
type reg = R of int | R_fp | R_sp | R_pc | R_ip | 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

(* |operand| -- type of operands for assembly instructions *)
type operand = (* VALUE ASM SYNTAX *)
  Const of int (* val #val *)
  | Register of reg (* [reg] reg *)
  | Index of reg * int (* [reg]+val [reg, #val] *)
  | Index2 of reg * reg * int (* [r1]+[r2]<<n [r1, r2, LSL #n] *)
  | Global of symbol (* lab lab *)
  | Label of codelab (* lab lab *)

(* |fRand| -- format operand for printing *)
val fRand : operand -> Print.arg

(* |reg_of| -- extract register (or R_none) from operand *)
val reg_of : operand -> reg

(* |emit| -- emit an assembly language instruction *)
val emit : string -> operand list -> unit

(* |move_reg| -- emit a register-to-register move *)
val move_reg : 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 any register used by a value *)
let release =
function
  Register reg -> release_reg reg
| Index (reg, off) -> release_reg reg
| Index2 (r1, r2, n) -> release_reg r1; release_reg r2
| _ -> ()

let fix_reg r = Register (get_reg (reg_of r))

(* |gen_reg| -- emit instruction with result in a register *)
let gen_reg op rands =
  List.iter release (List.tl rands);
  let r' = fix_reg (List.hd rands) in
  emit op (r' :: List.tl rands);
  r'

(* |gen| -- emit an instruction *)
let gen op rands =
  List.iter release rands;
  emit op rands

(* |gen_move| -- move value to specific register *)
let gen_move dst src =
  if reg_of dst = R_any || reg_of dst = R_temp || dst = src then
    src
  else
    gen_reg "mov" [dst; 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
e_X t, each generating code for a tree t, leaving the value in
a register, or as an operand for another instruction, etc. *)

let anyreg = Register R_any
let anytemp = Register R_temp
let rec e_reg t r =

let binary op t1 t2 =
    let v1 = e_reg t1 anyreg in
    let v2 = e_rand t2 in
    gen_reg op [r; v1; v2]

let unary op t1 =
    let v1 = e_reg t1 anyreg in
    gen_reg op [r; v1]

match t with
    <CONST k> when fits_move k ->
        gen_reg "mov" [r; Const k]
    | <CONST k> ->
        gen_reg "set" [r; Const k]
    | <LOCAL 0> ->
        gen_move r (Register R_fp)
    | <LOCAL n> when fits_add n ->
        gen_reg "add" [r; Register R_fp; Const n]
    | <LOCAL n> ->
        emit "set" [Register R_ip; Const n];
        gen_reg "add" [r; Register R_fp; Register R_ip]
    | <GLOBAL x> ->
        gen_reg "set" [r; Global x]
    | <TEMP n> ->
        gen_move r (Register (Regs.use_temp n))
    | <(LOADW|LOADC), <REGVAR i>> ->
        let rv = List.nth stable i in
        reserve_reg rv; gen_move r (Register rv)
    | <LOADW, t1> ->
        let v1 = e_addr t1 in
        gen_reg "ldr" [r; v1]
    | <LOADC, t1> ->
        let v1 = e_addr t1 in
        gen_reg "ldrb" [r; v1]
    | <MONOP Uminus, t1> ->
        unary "neg" t1
    | <MONOP Not, t1> ->

(* |e_reg| -- evaluate expression with result in specified register *)

(* returns |Register| *)

(* Binary operation *)

Binary operation

(* Unary operation *)

Unary operation

(* Comparison with boolean result *)

Comparison with boolean result
let v1 = e_reg t1 anyreg in
  gen_reg "eor" [r; v1; Const 1]
| <MONOP BitNot, t1> -> unary "mvn" t1
| <OFFSET, t1, <CONST n>> when fits_add n ->
  (* Allow add for negative constants *)
    let v1 = e_reg t1 anyreg in
  gen_reg "add" [r; v1; Const n]
| <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> -> binary "lsl" t1 t2
| <BINOP Lsr, t1, t2> -> binary "lsr" t1 t2
| <BINOP Asr, t1, t2> -> binary "asr" t1 t2
| <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 = e_reg t1 anyreg in
    let v2 = e_reg t2 anyreg in
  gen_reg "mul" [r; v1; v2]
| <BINOP Eq, t1, t2> -> compare "moveq" t1 t2
| <BINOP Neq, t1, t2> -> compare "movne" t1 t2
| <BINOP Gt, t1, t2> -> compare "movgt" t1 t2
| <BINOP Geq, t1, t2> -> compare "movge" t1 t2
| <BINOP Lt, t1, t2> -> compare "movlt" t1 t2
| <BINOP Leq, t1, t2> -> compare "movle" t1 t2
| <BOUND, t1, t2> ->
  let v1 = e_reg t1 r in
  let v2 = e_rand t2 in
  release v2;
  emit "cmp" [v1; v2];
  emit "seths" [Register (R 0); Const !line];
  emit "blhs" [Global "check"];
  v1
| <NCHECK, t1> ->
  let v1 = e_reg t1 r in
  emit "cmp" [v1; Const 0];
  emit "seteq" [Register (R 0); Const !line];
  emit "bleq" [Global "nullcheck"];
  v1
| <w, @args> ->
  failwith (sprintf "eval $" [fInst w])

(* |e_rand| -- evaluate to form second operand *)
and e_rand =
  (* returns |Const| or |Register| *)
function
  <CONST k> when fits_immed k -> Const k
  t -> e_reg t anyreg

(* |e_addr| -- evaluate to form an address for ldr or str *)
and e_addr =
(* returns |Index| *)
function
  <LOCAL n> when fits_offset n ->
    Index (R_fp, n)
  <OFFSET, t1, <CONST n>> when fits_offset n ->
    let v1 = e_reg t1 anyreg in
    Index (reg_of v1, n)
  t ->
    let v1 = e_reg t anyreg in
    Index (reg_of v1, 0)

(* |e_call| -- execute procedure call *)
let e_call =
function
  <GLOBAL f> ->
    gen "bl" [Global f]
  t ->
    let v1 = e_reg t anyreg in
    gen "blx" [v1]

(* |e_stmt| -- generate code to execute a statement *)
let e_stmt t =

(* Conditional jump *)
let condj op lab t1 t2 =
  let v1 = e_reg t1 anyreg in
  let v2 = e_rand t2 in
  gen "cmp" [v1; v2];
  gen op [Label lab] in

(* Procedure call *)
let call t =
  spill_temps volatile; (* Spill any remaining temps *)
  e_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 = e_reg t1 anytemp in
    Regs.def_temp n (reg_of v1)
| <(STOREW|STOREC), t1, <REGVAR i>> ->  
   let rv = List.nth stable i in  
   release (e_reg t1 (Register rv))  
| <STOREW, t1, t2> ->  
   let v1 = e_reg t1 anyreg in  
   let v2 = e_addr t2 in  
   gen "str" [v1; v2]  
| <STOREC, t1, t2> ->  
   let v1 = e_reg t1 anyreg in  
   let v2 = e_addr t2 in  
   gen "strb" [v1; v2]  
| <RESULTW, t1> ->  
   release (e_reg t1 (Register (R 0)))  
| <LABEL lab> -> emit_lab lab  
| <JUMP lab> -> gen "b" [Label lab]  
| <JUMPC (Eq, lab), t1, t2> -> condj "beq" lab t1 t2  
| <JUMPC (Lt, lab), t1, t2> -> condj "blt" lab t1 t2  
| <JUMPC (Gt, lab), t1, t2> -> condj "bgt" lab t1 t2  
| <JUMPC (Leq, lab), t1, t2> -> condj "ble" lab t1 t2  
| <JUMPC (Geq, lab), t1, t2> -> condj "bge" lab t1 t2  
| <JUMPC (Neq, lab), t1, t2> -> condj "bne" 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 itself. *)  
   let v1 = e_reg t1 anyreg in  
   emit "cmp" [v1; Const (List.length table)];  
   gen "ldrlo" [Register R_pc; Index2 (R_pc, reg_of v1, 2)];  
   gen "b" [Label deflab];  
   List.iter (fun lab -> emit ".word" [Label 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 (Register r) (Register r1))  
| <ARG i, t1> when i < 4 ->  
   let r = R i in  
   spill_temps [r];  
   ignore (e_reg t1 (Register r))  
| <ARG i, t1> when i >= 4 ->  
   need_stack (4*i-12);  
   let v1 = e_reg t1 anyreg in  
   gen "str" [v1; Index (R_sp, 4*i-16)]  
| <STATLINK, t1> ->
let r = R 10 in
spill_temps [r];
ignore (e_reg t1 (Register r))

| <w, @ts> ->
  failwith (sprintf "e_stmt $" [fInst w])

(* |process| -- generate code for a statement, or note a line number *)
let process =
  function
    <LINE n> ->
      if !line <> n then
        emit_comment (Source.get_line n);
        line := n
      | t ->
        if !debug > 0 then emit_tree t;
        e_stmt t;
      if !debug > 1 then emit_comment (Regs.dump_regs ())

(* |translate| -- translate a procedure body *)
let translate lab nargs fsize nregv code =
  Target.start_proc lab nargs fsize;
  Regs.get_regvars nregv;
  (try List.iter process code with exc ->)
      (* Code generation failed, but let's see how far we got *)
      Target.flush_proc (); raise exc;
  Target.end_proc ()