C ♭ Code Patterns

Preamble

A key observation is that all expressions can be evaluated and, whatever type it has, the value can be held in a register. We can therefore traverse the parts of the AST which represent expressions and keep track of which register has been used.

We will have a number of traversal functions. There will be different functions for different parts of the language, and different functions for different contexts (e.g. the LHS of an assignment is different from the RHS).

The registers `r0`-`r3` are used as scratch registers (i.e. very short term usage) and they are not preserved across function calls. Registers `r4`-`r11` are used for evaluating expressions, and they must be preserved across function calls. Register `r12` is also known as `fp` and is used as a *frame pointer*; it provides access to local variables and the formal parameters of the current method. Register `r13` is also known as `sp` which is short for *stack pointer*. The stack grows downward in memory. Register `r14` is also known as `lr`; it is the *link register* and holds the return address after control is transferred to a function or method by the `bl` (*branch and link*) instruction. Register `r15` is also known as `pc`, short for *program counter*. Loading a value into `pc` causes an immediate control transfer.

Suggested Traversal Functions

The functions listed below mostly have `void` return types. One exception is an int type, where the integer specifies the number of a register which will hold the value of an expression after the generated code is executed. The other exception is the `Loc` type; it is a class which defines how to access a location in memory. There are three possibilities for a Loc: a named global constant, an offset plus register pair, and a register plus a scaled index register.

Function	Purpose
`int GenExpression(n)`	Translates subtree `n` representing an expression and returns the number of the register which holds the expression's value
`Loc GenVariable(n)`	Translates subtree `n` representing a variable (e.g. the LHS of an assignment) and returns a `Loc` value.
`void GenStatement(n)`	Translates one statement.
`void GenConditional( n,TL,FL)`	Evaluates expression `n` and generates conditional branches to label `TL` and `FL` according to the expression being true/false.
`void GenMethod(n)`	Translates a method declaration; `n` is a node with the `Method` tag.
`void GenProgram(n)`	Translates the entire program; n is a node with the `Program` tag.
`void GenConstDefn(n)`	Translates a constant definition; `n` is a node with the `Const` tag.

Many of the ARM coding patterns are discussed below. The usual strategy for a traversal function is to inspect the root node's tag to choose the pattern; then appropriate traversal functions are called for subtrees of this node, and finally ARM code is generated which combines the results from the subtrees.

Note that not all subtrees generate code or are even visited. For example, subtrees which represent datatype definitions or local variable declarations were useful only during the type checking phase. They do not directly get translated into code.

Translating Arithmetic Expressions

The general strategy is to (1) force the operands into registers, and (2) generate the appropriate instruction which leaves the result in a register. In the examples below, register `r4` is used for the first (left) operand and `r5` for the second (right) operand, if there is one.

`Unary Minus`	`MVN r4,r4`	Can re-use the same register
`Add`	`ADD r4,r4,r5`	Can re-use one of the operand registers for the result
`Sub`	`SUB r4,r4,r5`	
`Mul`	`MULSL r6,r4,r5`	Should use a new register (not either of the input registers) for the result.

Unfortunately, the ARM architecture does not provide an integer divide instruction, so we cannot implement the `Div` and `Mod` operations directly. We call a support routine instead.

`Div`	`MOV r0,r4`
	`MOV r1,r5`
	`BL  cb.DivMod`
	`@ result in r0`
	`MOV r4,r0`
		The `DivMod` function implements both division and remainder. The two operands are passed in registers `r0` and `r1`. The division result is returned in `r0` and the remainder result in `r1`.

`Mod`	`MOV r0,r4`
	`MOV r1,r5`
	`BL  cb.DivMod`
	`@ result in r1`
	`MOV r4,r1`	

The simplest kinds of expressions are constants and variables.

`IntConst`	`LDR r4,=12345`
	`MOV r4,#123`
	`MVN r4,#123`
		Can be used for any 32-bit constant.
		`MOV` can be used for constants 0 to 255.
		This loads -123 (use `MVN` for constants 0 to -255).

`Ident`	`LDR r4,[fp,#-40]`	If `Ident` represents an `int` variable or a string or an array, the value or the address of the string/array needs to be loaded. The example assumes it is a local.

Comparisons and Boolean Expressions

A comparison or a Boolean expression can only be used to control a while loop or if statement in C♭. Therefore we never need to generate a Boolean as a 0 or 1 value. We only use the values to decide whether to branch or not to branch.

`Equals`
`NotEquals`
`LessThan`
`GreaterThan`
`LessOrEqual`
`GreaterOrEqual`
	`BEQ L1`
	`BNE L2`
	`BLT L3`
	`BGE L4`
	`BLE L5`
	`BGE L6`
		An expression where the operator is `==` `!=` `<` `>` `<=` or `>=`

`And` `Or`		No explicit code is generated for these operators; the effect is achieved via control flow which links the evaluation of the left and right operands. See the course slides for the strategy (Intermediate Code Generation, part 2)

Method Calls

We will evaluate and push each actual parameter onto the stack. For consistency between everyone's C♭ compiler and for consistency with the course slides, the parameters should be processed in reverse order.

If a result is returned, it will be returned in register `r0`. (If the method result is a string, an array or a `struct`[1] then `r0` holds the address of the value being returned.)

For example, the statement

```
    k = justDoIt( 37, "hello" );
```

could be translated into code similar to the following:

```
    LDR  r4, =_S.5      @ _S.5 is a label for the string constant
    STR  r4, [sp,#-4]!  @ push r4 onto stack
    LDR  r4, =37        @ or MOV r4,#37 is good here
    STR  r4, [sp,#-4]!  @ push r4 onto stack
    BL   justDoIt
    ADD  sp, sp, #8     @ pop 8 bytes off stack
    MOV  r4, r0         @ move result out of scratch register
    STR  r4, [fp, #-40] @ store result in local variable k
    ...
_S.5: .asciz "hello"    @ located after end of current method
```

The code can be improved, especially the last two instructions.

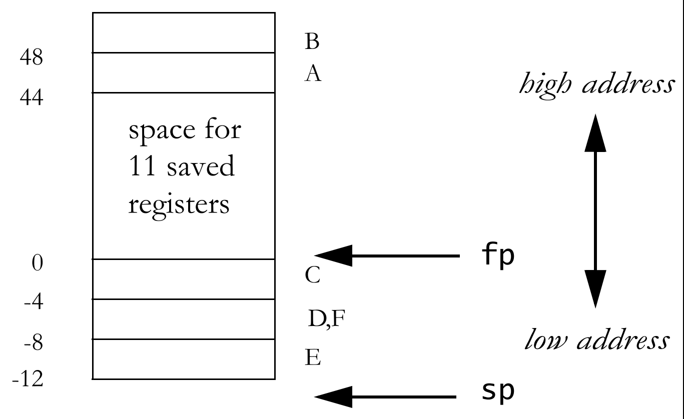
Note that, except for `struct` values, all actual parameters being pushed are integers or addresses and are 4 bytes in size.

1. If you implement methods which accept struct parameters and/or return struct results, you are eligible for bonus marks. It is suggested that your compiler should refuse to compile such methods. Leave this tricky stuff until everything else is working!

Method Prolog and Epilog

In a pass over the AST before code generation for the method begins, it is necessary to figure out where each formal parameter and each of the method's local variables will be located relative to the frame pointer. A tiny example will illustrate how the locations are calculated.

```
void FOO( int A, int B ) {
    int C;
    if (A > B) {
        int D, E;
        ...
    } else {
        int F;
        ...
    }
    ....
}
```



The formal parameters A and B are accessed via the memory addresses `[fp,#44]` and `[fp,#48]` respectively. Variable C is at `[fp,#-4]`. Variables D and F share the location `[fp,#-8]`. Variable E is at `[fp,#12]`.

The prolog code for the example would be the following:

```
FOO:
    STMFD sp!, {r4-r12,lr} @ push all registers onto stack
    MOV   fp, sp           @ set up frame pointer
    SUB   sp, sp, #12      @ reserve 12 bytes for local variables
```

The epilog code needs to undo all of the above. It is the following.

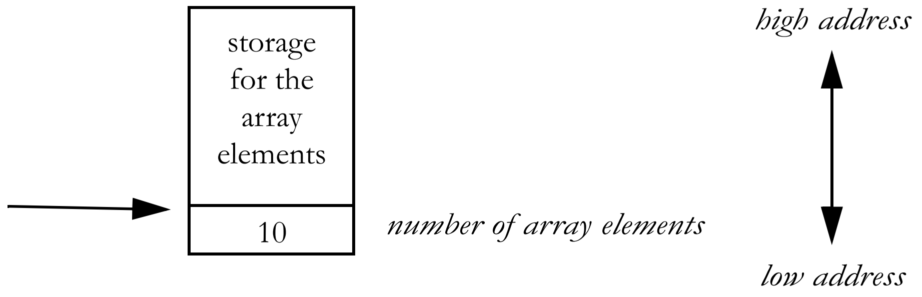
```
L37:
    MOV   sp, fp           @ this has effect of popping the locals
    LDMFD sp!, {r4-r12,pc} @ reload saved registers AND RETURN!
```

Notes:

1. The `LDMFD` instruction in the epilog loads the saved `r14` value into the `pc` register instead. That has the effect of a jump back to the caller of the method.
2. Any return statement inside the method body should be implemented as a jump to the label (L37 in the example) which is placed immediately before the method's epilog.
3. It is usually unnecessary to save all the registers `r4` through `r11`. Only those which are actually used inside the method body need be saved. However it is an extra complication to get that right; the code shown above is good enough.
4. The proper ARM calling convention does not use a frame pointer, `fp`, register because all formal parameters and locals can be accessed relative to the `sp` register. However, the `sp` register is changing when parameters are being pushed onto the stack for another method call and keeping track of how much `sp` has changed is one more detail which is easy to get wrong!

Allocating Arrays – the `new` Operation and `Length` Property

The expression `new int[10]` for example should evaluate to an address inside a block of memory in the heap area of the program's address space which is 44 bytes in size. The block of memory is word-aligned and organized as shown below.



The ARM code to generate for `new int[10]` is as follows.

```
MOV r0, #44      @ number of bytes of heap space needed
BL  cb.Malloc    @ request space from the malloc routine
MOV r1, #10      @ number of array elements
STR r1, [r0],#4  @ store 10 in first word, then advance r0 by 4
```

In general, the number of bytes needed is the number of elements times their size plus 4. On return from the `cb.Malloc` function, `r0` holds the address of the allocated block of memory. The caller now stores the number of array elements in the first word and advances `r0` by 4 bytes so that `r0` holds the address of the first array element.

Subsequently, if register `r4` holds the address of an array, the `Length` property can be evaluated via the following load instruction:

```
LDR r4, [r4, #-4]    @ load the array's length prefix
```

String Constants

A statement like

```
s = "abc";
```

has to store the address of the string constant into variable s. The string constant is represented in C♭ using the same coding as C, which is the coding scheme supported in the ARM assembly language. The string constant can be located in either the text area or the data area of the program; it does not matter. A compiler would normally collect all the string constants for a compilation unit, removing any duplicates as they are encountered, and place them in a read-only section of the data area.

The supplied code (in file `GenCode.cs`) provides methods for accumulating string constant values and assigning labels to them. It does not remove duplicates. Using these methods, the example statement above would be translated into code similar to the following

```
LDR   r4, =_S.43     @ label attached to string constant
STR   r4, [fp,#-40]  @ store in s
```

and after the program has been translated, the following definition is added to the data area

```
_S.43: .asciz "abc"
```

Input-Output Statements

All I/O is performed by calls to support routines. The call

```
    cbio.read(out x);
```

should be translated into the following ARM code sequence

```
    @ code to load address of x into register r0
    bl   cb.ReadInt
```

The calls

```
    cbio.write(x);  // x is an int value
    cbio.write(s);  // s is a string value
```

should similarly be translated in these ARM code sequences

```
    @ code to load value of x into register r0
    bl   cb.WriteInt

    @ code to load value of s (an address) into r0
    bl   cb.WriteString
```

Note that output of strings does not have quite the same semantics as the C# equivalent. In C#, strings can contain null bytes whereas C♭ adopts the same implementation as the C language and uses a null byte to mark the end of a string.

Summary of the Support Function Package

The following functions are provided in an ARM assembly language file named `CbRuntime.s`.

Function	Input Params		Result
`cb.Malloc`	`r0`	number of bytes	`r0` = address of a block of memory of the specified size obtained from the heap
`cb.ReadInt`			An integer is read from standard input and returned in `r0`.
`cb.WriteInt`	`r0`	a value	The integer in register `r0` is written to standard output
`cb.WriteString`	`r0`	address of a string	The string, up to but not including a null byte, is written to the standard output.
`cb.DivMod`	`r0` `r1`	an integer an integer	The integer in `r0` is divided by the integer in `r1`. On return, `r0` holds the quotient and `r1` holds the remainder.

`cb.MemCopy`	`r0`
	`r1`
	`r2`
		dest address
		src address
		an integer
			Copies `r2` bytes of memory from the source address specified by `r1` to the destination address specified by `r0`. (Needed for assigning `struct` values.)

`cb.StrLen`	`r0`	address of a string	`r0` holds the string length. Note: the string must be null terminated