Compilers I - Chapter 4: Generating Better Code

- Lecturers:
  - Paul Kelly (phjk@doc.ic.ac.uk)
    - Office: room 304, William Penney Building
  - Naranker Dulay (nd@doc.ic.ac.uk)
    - Office: room 562
- Materials:
  - Textbook
  - Course web pages (http://www.doc.ic.ac.uk/~phjk/Compilers)
  - Piazza (http://piazza.com/imperial.ac.uk/fall2016/221)

Overview

- We have seen a simple code generator which handles the basic components of common programming languages - statements and expressions
- We will need to handle more:
  - declarations (constants, records etc)
  - storage management
  - procedures and functions
- But first we will examine ways of producing better-quality output code
  - The main issue is the effective use of registers
- ***At this stage we are looking for simple, fast algorithms which do a reasonably good job: optimizing compilers use more powerful (slower!) techniques, which we will cover later***

The plan

- A simple language with assignments, loops etc.
- A stack-based instruction set and its code generator
- Code generation for a machine with registers:
  - an unbounded number of registers
  - a fixed number of registers
  - avoiding running out of registers
  - register allocation across multiple statements
- Conditionals and Boolean expressions

Code generation with an unbounded number of registers

- We will concentrate on using registers well in arithmetic expressions:
  - Initially assume there will always be enough registers.
  - Invent a scheme to handle cases where we run out of registers
  - Modify the translator to evaluate expressions in the order which minimises the number of registers needed

- When we look at sequences of assignments ("basic blocks") it is clear that better code results if *variables* are kept in registers as well as nameless intermediate values. We will examine the *graph colouring* approach to register allocation which addresses this.

Instruction set for example machine with registers

Model: Stack machine as before, augmented with some number of registers R0, R1, . . .

- **Instruction set data type:**

```
data Instruction =
Add reg reg                    (Add r1 r2:  r1:= r1+ r2)
| Sub reg reg | ...                   (similar)
| Load reg name                (reg := value at location name)
| LoadImm reg num                 (load constant into reg)
| Store reg name               (store reg at location name)
| Push reg                     (push reg onto stack)
| Pop reg        (remove value from stack, and put it in the reg)
| CompEq reg reg  (subtract reg from reg and set reg to 1
                   if the result was zero, 0 otherwise)
| JTrue reg label              (if reg = 1 jump to label)
| JFalse reg label             (if reg = 0 jump to label)
| Define label                 (set up destination for jump)
```

Storage allocation for intermediate values in expressions

- The code generator given earlier would translate the expression:

  (100*3) + ((200*2) + 300) + (400 + (500*3))

- to the stack machine assembly code:

```
PushImm 100,
PushImm 3,
Mul,
PushImm 200,
PushImm 2,
Mul,
PushImm 300,
Add,
Add,
PushImm 400,
PushImm 500,
PushImm 3,
Mul,
Add,
Add
```

- If you feed this into the stack machine simulator

(http://www.doc.ic.ac.uk/~phjk/Compilers/Haskell/StackMachine.hs)

- you can get the trace:

Stack slot:	*3*	*2*	*1*	*0*
PushImm 100,				100
PushImm 3,			3	100
Mul,				300
PushImm 200,			200	300
PushImm 2,		2	200	300
Mul,			400	300
PushImm 300,		300	400	300
Add,			700	300
Add,				1000
PushImm 400,			400	1000
PushImm 500,		500	400	1000
PushImm 3,	3	500	400	1000
Mul,		1500	400	1000
Add,			1900	1000
Add				2900

How the stack machine uses memory

- The first instruction puts 100 into slot 0
- the second puts 3 into slot 1
- the third adds the contents of slots 0 and 1 and puts the result in slot 0

We see that if the computer knows where the stack pointer starts, it can work out beforehand where everything will be

- This shows the way towards using registers efficiently

Stack slot:	*3*	*2*	*1*	*0*
PushImm 100,				100
PushImm 3,			3	100
Mul,				300
PushImm 200,			200	300
PushImm 2,		2	200	300
Mul,			400	300
PushImm 300,		300	400	300
Add,			700	300
Add,				1000
PushImm 400,			400	1000
PushImm 500,		500	400	1000
PushImm 3,	3	500	400	1000
Mul,		1500	400	1000
Add,			1900	1000
Add				2900

```
LoadImm R0 100,
LoadImm R1 3,
Mul R0 R1,
LoadImm R1 100,
LoadImm R2 2,
Mul R1 R2,
LoadImm R2 300,
Add R1 R2,
Add R0 R1,
LoadImm R1 400,
LoadImm R2 500,
LoadImm R3 3,
Mul R2 R3,
Add R1 R2,
Add R0 R1
```

Here is the code which results from working out where all the operands will be at compile-time

The translation function

- The translation function `transExp` requires an extra parameter which specifies the register in which the result is to be left:

```
transExp :: exp -> reg -> [instruction]
```

- The code generated by `transExp e Ri` can use registers R_i, R_{i+1} upwards, but must leave the other registers ($R_0 . . . R_{i-1}$) intact

- The easy cases….

```
transExp (Const n) r = [LoadImm r n]
transExp (Ident x) r = [Load r x]
```

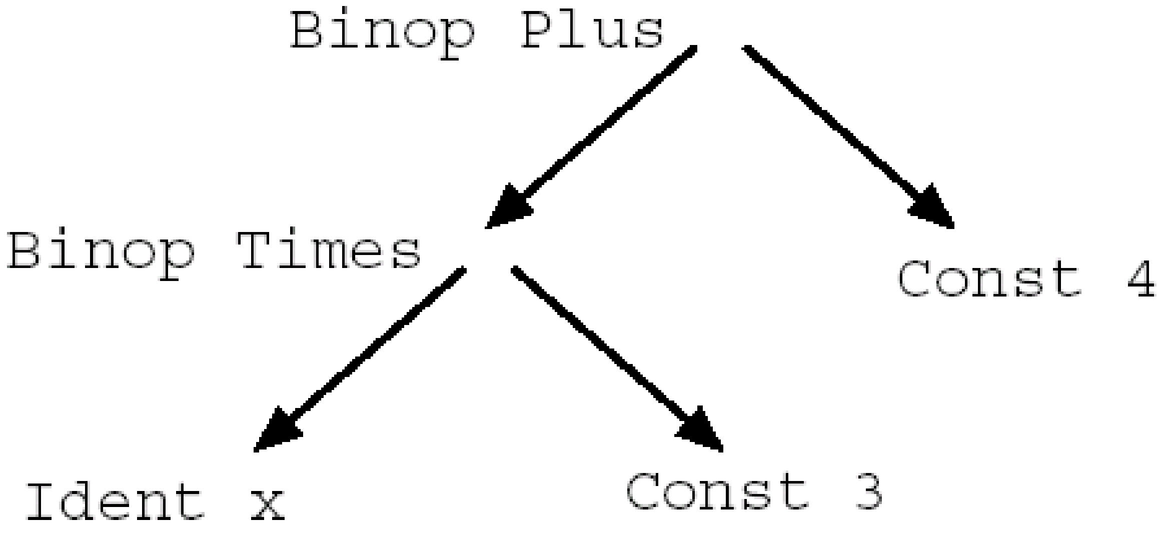
The translation function…

```
transExp (Binop op e1 e2) r
  = transExp e1 r ++
    transExp e2 (r+1) ++
    transBinop op r (r+1)
where
    transBinop Plus r1 r2 = [Add r1 r2]
    transBinop Minus r1 r2 = [Sub r1 r2]
etc.
```

- Parameter r is used to track where the stack pointer *would* point

Example: `x * 3 + 4`

- **AST:**



- **Walkthrough:**

```
transExp (Binop Plus (Binop Times (Ident "x")
                                  (Const 3))
                     (Const 4))
         0 (deliver result to register 0)
```

Example: x * 3 + 4 ...

- transExp (Binop Plus (Binop Times (Ident "x")(Const 3))(Const 4)) 0

Using the definition of transExp, we unfold this to get:

- (transExp (Binop Times (Ident "x")(Const 3)) 0) ++
  (transExp (Const 4) 1) ++
  [Add R0 R1]

Unfolding again, this reduces to:

```
[Load R0 "x",
LoadImm R1 3,
Mul R0 R1,       (R0 := x*3)
LoadImm R1 4,    (R1 := 4)
Add R0 R1]       (R0 := (x*3)+4)
```

How might this be improved?

How might use of registers be improved?

- **Example:** `x * 3 + 4`
- We used two registers – can we get away with fewer?
- How about:

```
[Load R0 "x",
 MulImm R0 3,
 AddImm R0 4]
```

Instead of loading the constant into a register, use an instruction that takes an immediate operand – eg in Intel's IA32:

```
movl x,%eax
imull $3,%eax
addl $4,%eax
```

- This is clearly better because it involves fewer instructions, and uses fewer registers
- How to fix the translator to do this?

Using immediate operands

- The modification required to the translator is small—we need to add a rule to catch the special case:

```
 transExp (Binop op e1 (Const n)) r
    = transExp e1 r ++ transBinopImm op r n
```

where

```
 transBinopImm Plus r n = [AddImm r n]
```

etc.

- If the operator is commutative, we can catch another case:

```
 transExp (Binop op (Const n) e2) r
   | commutative op = transExp e2 r ++ transBinopImm op r n
```

The general idea here is to use pattern-matching to find opportunities to use instructions – see "instruction selection" in the textbooks

- **Problem:** We don’t have an unbounded number of registers

- Before we see how to overcome this problem in the register machine case, we introduce the accumulator machine—a machine with only *one* register, its “accumulator”.

- The solution to the problem will be to combine the two techniques.

The Accumulator Machine

- This machine has a stack, and just one register, the accumulator. The stack is used for intermediate values as before, but arithmetic etc. instructions are always of the form:

```
Acc := Acc + Store[SP];
SP := SP+1;
```

- The instruction set:

```
data Instruction =
    Add | Sub | Mul | Div... |
    AddImm num | ... |
    CompEq | ... |
    Push |
    Pop |
    Load name |
    LoadImm num |
    Store name |
    Jump label |
    Jtrue label |
    JFalse label |
    Define label
```

The accumulator machine…

What the instructions do:

- `Add:`

  Acc:=Acc+Store[SP]; SP:=SP+1;
- `Push:`

  SP:=SP-1; Store[SP]:=Acc;
- `Pop:`

  Acc:=Store[SP]; SP:=SP+1;
- `Load name:`

  Acc:=Store[name];
- `Store name:`

  Store[name]:=Acc;

Translator for accumulator machine:

- The translator transExp generates code to evaluate an expression and leave its value in the accumulator:

```
transExp (Const n) = [LoadImm n]
transExp (Ident x) = [Load x]
transExp (Binop op e1 e2)
  = transExp e2 ++
    [Push] ++
    transExp e1 ++
    transBinop op
where
  transBinop Plus = [Add]
  etc.
```

(Note that `e2` has to be evaluated before `e1` so that it forms the right-hand operand of the binary operator)

- For ‘e1 + e2’, push value of e2 onto stack while evaluating e1

Translator for machine with limited register set

- A neat solution to the problem of running out of registers is to combine the register machine and accumulator strategies:
  - While free registers remain, use the register machine strategy
  - When the limit is reached (ie. when there is *one* register left), revert to the accumulator strategy, using the last register as the accumulator
- The effect is that most expressions get the full benefit of registers, while unusually large expressions are handled correctly

Code generation with limited registers - Implementation:

```
transExp (Const n) r = [LoadImm r n]
transExp (Ident x) r = [Load r x]

transExp (Binop op e1 e2) r
= if r == MAXREG then
  transExp e2 r ++
     [Push r] ++
     transExp e1 r ++
     transBinopStack op r

  elseif r < MAXREG
  transExp e1 r ++
  transExp e2 (r+1) ++
  transBinop op r (r+1)
```

if no more registers, use stack for e2 while evaluating e1

Arithmetic instruction gets one operand from stack (next slide)

If there are more registers, use register r to hold e1. Evaluate e2 leaving result in register r+1

Arithmetic instruction gets both operands from registers

- If there were enough registers, generate an arithmetic instruction which takes register operands:

```
 transBinop Plus r1 r2 = [Add r1 r2]
     transBinop Minus r1 r2 = [Sub r1 r2]
     etc.
```

- If there is only one register left, we need an arithmetic instruction which takes one operand from the top of the stack, as in the accumulator machine:

```
     transBinopStack Add r = [AddStack r]
     transBinopStack Sub r = [SubStack r]
```

  etc .

- The `AddStack r` instruction is simply:

```
 r := r+Store[SP];  SP:=SP+1;
```

Conclusion

- In the last chapter we saw a code generator for simple statements and expressions.
- In this chapter we looked at ways of improving code for expressions
- We looked at using stacks, accumulators and registers — three ways of managing storage for intermediate values
- The stack approach is not very efficient. Adding just one register ("accumulator") much reduces the number of memory references
- With several registers, main memory for intermediate values is rarely needed. If you do run out, it is efficient enough to revert to the accumulator scheme.
- Optimising control structures (if-then-else, while, for) is trickier