Summary: Intermediate Code Generation

1 Intermediate Code Generation

- Rather than generate assembler directly from parse trees, many approaches generate an intermediate representation
- The intermediate representation is machine-independent, and a second step translates it to machine-specific assembler.
- This means that work to port to a new machine is reduced.
- Optimisation on intermediate code, not on object code.
- Two common formats:
  - Postfix notation
  - Quadruples

2 Postfix Notation

- Also called suffix notation or reverse polish notation
- Used as an intermediate representation between the parse tree and the generated assembler code.

2.1 Mapping program forms to Postfix

2.1.1 Mathematical and boolean expressions

- a + b => a b +
- a * (b + c) => a b c + *
- a == b => a b ==

2.1.2 Unary operators

- -a => a _

2.1.3 Assignment

- a = a + b => a_i a b + =

2.1.4 Goto statements

- goto L1 => L1 jump_to

2.1.5 If statements

- if <p> then <inst1> else <inst2> =>
  <p> L1 jump_if_false <inst1> L2 jump_to L1: <inst2> L2:

2.2 Mapping from Postfix to Assembler

Take each symbol in the postfix code in turn.

- If it is a reference to a variable in an expression, generate:
  push dword [a]
- if it is a reference to a variable on the LHS of an assignment, generate:
  push dword a
- if it is a reference to a constant (e.g., 2), generate:
  push dword 2
- For mathematical operators (integers):

+	pop eax pop ebx add ebx, eax push ebx
-	pop eax pop ebx sub ebx, eax push ebx
*	pop eax pop ebx mul ebx, eax push ebx
/	pop eax pop ebx idiv ebx, eax push ebx

- Unary Minus:
  mov ebx, 0
  pop eax
  sub ebx, eax
  push ebx
- Assignment:
  pop ebx ; top of stack is value to store
  pop eax ; new top of stack is location to store it in
  mov dword [eax], ebx
- Unconditional jumps: … L1 jump_to =>
  push dword L2
  pop eax
  jmp near eax
- Conditional jumps: … L1 jump_if_false =>
  push dword L1
  pop eax
  jz near eax
- Labels: … L1: … => L:

Example of If Statement:

… L1 jump_if_false instr1 L2 jump_toL1: instr2 L2:

```
…
push dword L1
pop  eax
jz   near eax (*)
"code for inst1"
push dword L2
pop  eax
jmp  near eax
L1:
"code for inst2"
L2:
```

3 Quadruples Notation

- Re-represents parse tree as sequence of quadruples
- Each quadruple has form: (operator, arg1, arg2, result)
- The 'result' field represents where the result of the operation is stored.
- Typically, the result is some temporary variable

3.1 Mapping program forms to Quadruples

3.1.1 Mathematical and boolean expressions

- a + b => (+, a, b, tmp1)
- a * (b + c) => (+, b, c, tmp1), (*, a, tmp1, tmp2)
- a < b => (<, a, b, tmp)

3.1.2 Unary operators

- -a => (uminus, a, , tmp)

3.1.3 Assignment

- a = a + b => (+, a, b, tmp), (:=, tmp, , a)

3.1.4 Declarations

- int a, b => nothing
- int a=5 => (:=, 5, , a)

3.1.5 Array reference

- a = x[i] => ([ ]=, x, i, tmp1), (:=, y, , tmp1)
- x[i] = a => ([ ]=, x, i, tmp1), (:=, tmp1, , y)

3.1.6 Type conversion (itof = 'int to float')

- a = 1; b = a + 0.7 => (:=, 1, , a), (itof, a, , tmp1),
  (+ tmp1, 0.7, tmp2), (:=, tmp2, , b)

3.1.7 Unconditional Jump

(jmp, <jump_address>, ,)

3.1.8 Conditional Jump

```
(<, i, 5, t1)  ; condition
(jtrue, L10, t1, )
```

3.1.9 For loop

- for (i=0, i<5, i++) do <body> end

```
=>   1 (:=, 0,  , i)   ; initialisation
     2 (<,  i, 5, t1)  ; condition
     3 (jfalse, 7, t1,)
     … <body of loop>
     5 (+, i, 1, i) ; Iteration
     6 (jmp, 2, , )
     7
```

3.1.10 If statement

- if (a < b) then c = 1 else c = 2

```
=>
1    (<, a, b, t1)
2    (jfalse, 5, t1, )
3    (:=, 1, , c)
4    (jmp, 6, , )
5    (:=, 2, , c)
6
```

3.1.11 while statement

- while (a < b) do a = a +1

```
=>
1    (<, a, b, t1)
2    (jfalse, 5, t1, )
3    (+, a, 1, tmp1)
3    (:=, tmp1, , a)
4    (jmp, 1, , )
5
```

3.2 From Quads to Assembler

A list of quads can be converted to assembler using the following functions:

```
convertQuads(quad *quads) {
     for i=0, quads.length {
          switch (quads[i][0]) {
            '+':
               ADD(quad[i][1], quad[i][2], quad[i][3]);
               break;
            '*':
               MUL(quad[i][1], quad[i][2], quad[i][3]);
               break;
            '-':
               SUB(quad[i][1], quad[i][2], quad[i][3]);
               break;
            '/':
               DIV(quad[i][1], quad[i][2], quad[i][3]);
               break;
            '_':
               NEG(quad[i][1], quad[i][3]);
               break;
          ...
     }
}

int CAC (opd *x, opd *y) {
  if (AC==y) return 1;
  if (AC!=x) {
     if (AC!=NULL) GEN ("MOV", AC, "EAX");
     GEN ("MOV", "EAX", x);
     AC=x;
  }
  return 0;
}
```

```
ADD (opd *x, opd *y, opd *z) {
     if (CAC (x, y))
          GEN("ADD", "EAX", x)
     else
          GEN ("ADD", "EAX", y);
     AC=z;
}
```

```
MUL (opd *x, opd *y, opd *z) {
     if (CAC (x, y))
          GEN("MUL", "EAX", x)
     else
          GEN ("MUL", "EAX", y);
     AC=z;
}
```

```
SUB (opd *x, opd *y, opd *z) {
     CAC (x, null))
     GEN("SUB", "EAX", y)
     AC=z;
}
```

```
DIV (opd *x, opd *y, opd *z) {
     CAC (x, null))
     GEN("DIV", "EAX", y)
     AC=z;
}
```

```
Neg (opd *x, opd *z) {
     CAC (x, null))
     GEN("NEG", "EAX")
     AC=z;
}
```