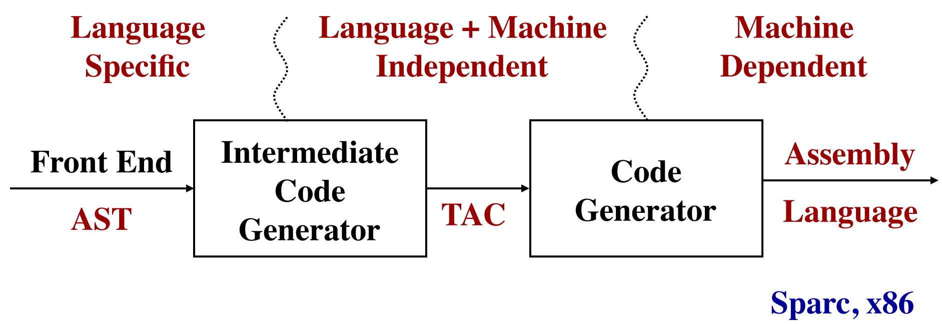
CMPT 379
Compilers

Anoop Sarkar
`http://www.cs.sfu.ca/~anoop`

TAC: Intermediate Representation



TAC: 3-Address Code

- Instructions that operate on named locations and labels: “generic assembly”
- Locations
  - Every location is some place to store 4 bytes
    - Pretend we can make infinitely many of them
  - Either on stack frame:
    - You assign offset (plus other information possibly)
  - Or global variable
    - Referred to by global name
- Labels (you generate as needed)

TAC: 3-Address Code

Addresses/Locations

- names/labels: we allow source-program names in TAC, implemented as a pointer to a symbol table entry
- constants
- temporaries

Instructions:

- assignments: *x = y op z* / *x = op y*
- copy: *x = y*
- unconditional jump: *goto L*
- conditional jumps: *if x goto L* / *ifFalse x goto L* / *if x relop y goto L*

<, ==, >=, etc.

TAC: 3-Address Code

Instructions:

- Procedure calls:
  - *param x1*
  - *param x2*
  - *…*
  - *param xn*
  - *call p, n*
- Function calls:
  - *y = call p, n*
  - *return y*

Instructions:

- Arrays:
  - *x = y[i]*
  - *x[i] = y*
- Pointers:
  - *x = &y*
  - *x = *y*
  - **x = y*

What TAC doesn't give you

- Array indexing (bounds check)
- Two or n-dimensional arrays
- Relational <=, >=, >, …
- Conditional branches other than **if** or **ifFalse**
- Field names in records/structures
  - Use base+offset load/store
- Object data and method access

Control Flow

- Consider the statement:
  while (a[i] < v) { i = i+1; }

Labels can be implemented using position numbers

```
L1:
  t1 = i
  t2 = t1 * 8
  t3 = a[ t2 ]
  ifFalse t3 < v goto L2
  t4 = i
  t4 = t4 + 1
  i = t4
  goto L1
L2: ...
```

```
100: t1 = i
101: t2 = t1 * 8
102: t3 = a[ t2 ]
103: ifFalse t3 < v goto 108
104: t4 = i
105: t4 = t4 + 1
106: i = t4
107: goto 100
108:
```



```
int gcd(int x, int y)
{
   int d;
   d = x - y;
   if (d > 0)
      return gcd(d, y);
   else if (d < 0)
      return gcd(x, -d);
   else
      return x;
}
```

```
gcd:
      t0 = x - y
      d = t0
      t1 = d
      t2 = t1 > 0
      ifFalse t2 goto L0
      param y
      param d
      t3 = call gcd, 2
      return t3
L0:
      t4 = d
      t5 = t4 < 0
      ...
```

Avoiding redundant gotos
if t2 goto L1
goto L0
L1: ...

Short-circuiting Booleans

- More complex if statements:
  - if (a or b and not c) { … }
- Typical sequence:

```
t1 = not c
t2 = b and t1
t3 = a or t2
```

- Short-circuit is possible in this case:
  - if (a and b and c) { … }
- Short-circuit sequence:

```
t1 = a
if t1 goto L0 /* sckt */
goto L4
L0: t2 = b
ifz t2 goto L1
```


More Control Flow: for loops

```
void main() {
  int i;
  for (i = 0; i < 10; i = i + 1)
    print(i);
}
```

```
main:
    t0 = 0
    i = t0
L0:
    t1 = 10
    t2 = i < t1
    ifFalse t2 goto L1
    param i, 1
    call PrintInt, 1
    t3 = 1
    t4 = i + t3
    i = t4
    goto L0
L1:
    return
```

Backpatching in Control-Flow

- Easiest way to implement the translations is to use two passes
- In one pass we may not know the target label for a jump statement
- *Backpatching* allows one pass code generation
- Generate branching statements with the targets of the jumps temporarily unspecified
- Put each of these statements into a list which is then filled in when the proper label is determined

11/26/10 11

Backpatching

- S → while M '(' expr ')' M block
- expr → true
- expr → false
- expr → expr || expr
- [illegible] → ε

simply returns the current instruction number

while (true) { ... }

- 108: t0 = true
- 109: if t0 goto 111
- 110: goto - falselist
- 111: ...
- 122: goto 108
- 123: ...
  - backpatch({110}, 123)

backpatch is done by rule that uses S

11/26/10 12

Backpatching

continue is similar, generates goto 108

- S → while M ‘(‘expr’)’ M block
- expr → true
- expr → false
- expr → expr || expr
- M → ε

simply returns the current instruction number

while (true) { break; }

- 108: t0 = true
- 109: if t0 goto 111
- 110: goto -
- 111: goto -
- 122: goto 108
- 123: ...
  - backpatch({110}, 123)
  - backpatch({111}, 123)

backpatch is done by while rule

Backpatching

- S → while M ‘(‘expr’)’ M block
- expr → true
- expr → false
- expr → expr || expr
- M → ε

while (true||false) { ... }

true || false

- 100: t0 = true
- 101: if t0 goto -
- 102: t1 = false
- 103: if t1 goto 106
- 104: t0 = false
- 105: goto -
- 106: t0 = true
- 107: goto -
  - backpatch({101, 105, 107}, 109)

nextlist

backpatch is done by while rule

Backpatching

- We maintain a list of statements that need patching by future statements
- Three lists are maintained:
  - truelist: for targets when evaluation is true
  - falselist: for targets when eval is false
  - nextlist: the statement that ends the block
- These lists can be implemented as a synthesized attribute
- Note the use of marker non-terminals

Array Elements

- Array elements are numbered *0, …, n-1*
- Let *w* be the width of each array element
- Let *base* be the address of the storage allocated for the array
- Then the i^{th} element *A[i]* begins in location *base+i*w*
- The element *A[i][j]* with *n* elements in the 2nd dimension begins at: *base+(i*n+j)*w*

```
void foo(int[] arr)
     { arr[1] = arr[0] * 2 }
```

foo:
t0 = 1
t1 = 4
t2 = t1 * t0
t3 = arr + t2
t4 = *(t3)
t5 = 0
t6 = 4
t7 = t6 * t5
t8 = arr + t7
t9 = *(t8)
t10 = 2
t11 = t9 * t10
t4 = t11

Wrong

foo:
t0 = 1
t1 = 4
t2 = t1 * t0
t3 = arr + t2
t4 = 0
t5 = 4
t6 = t5 * t4
t7 = arr + t6
t8 = *(t7)
t9 = 2
t10 = t8 * t9
*(t3) = t10

Correct

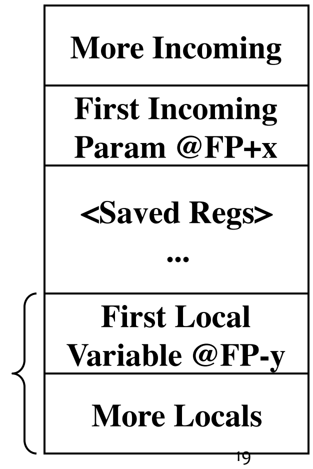
Array References

Translation of Expressions

- S → id = E
  - $$.code = concat($3.code, $1.lexeme = $3.addr)
- E → E + E
  - $$.addr = new Temp(); $$.code = concat ($1.code, $3.code, $$.addr = $1.addr + $3.addr)
- E → - E
  - $$.addr = new Temp(); $$.code = concat ($2.code, $$.addr = - $2.addr)
- E → (E)
  - $$.addr = $2.addr; $$.code = $2.code
- E → id
  - $$.addr = symtbl($1.lexeme); $$.code = ‘’

Function arguments

- Compute offsets for all incoming arguments, local variables and temporaries
  - Incoming arguments are at offset x, x+4, x+8, …
  - Locals+Temps are at –y, -y-4, -y-8,…
- Compute → **Frame Size**



Computing Location Offsets

```
class A {
 void f (int a /* @x+4 */,
            int b /* @x+8 */,
            int c /* @ x+12 */) {
     int s    // @-y-4
     if (c > 0) {
            int t …     // @-y-8
     } else {
            int u       // @-y-12
            int t …     // @-y-16
     }
  }
}
```

Location offsets for temporaries are ignored on this slide

← You could reuse @-y-8 here, but okay if you don't

```
int factorial(int n)
{
 if (n <=1 ) return 1;
 return n*factorial(n-1);
}

void main()
{
   print(factorial(6));
}
```

```
factorial:
      t0 = 1
      t1 = n lt t0
      t2 = n eq t0
      t3 = t1 or t2
      ifFalse t3 goto L0
      t4 = 1
      return t4
L0:
      t5 = 1
      t6 = n - t5
      param t6
      t7 = call factorial, 1
      t8 = n * t7
      return t8
```

t3 = n <= 1

Implementing TAC

- Quadruples:

  t1 = - c
  t2 = b * t1
  t3 = - c
  t4 = b * t3
  t5 = t2 + t4
  a = t5

- Triples

  1. - c
  2. b * (1)
  3. - c
  4. b * (3)
  5. (2) + (4)
  6. a = (5)

Implementing TAC

- Indirect Triples
  1. - c
  2. b * (1)
  3. - c
  4. b * (3)
  5. (2) + (4)
  6. a = (5)

Instruction List:
(1)
(2)
(3)
(4)
(5)
(6)

can be re-ordered by the code optimizer

- Static Single Assignment (SSA)

instead of:

a = t1
b = a + t1
a = b + t1

the SSA form has:

a1 = t1
b1 = a1 + t1
a2 = b1 + t1

a variable is never reused

Correctness vs. Optimizations

- When writing backend, correctness is paramount
  - Efficiency and optimizations are secondary concerns at this point
- Don't try optimizations at this stage

Basic Blocks

- Functions transfer control from one place (the caller) to another (the called function)
- Other examples include any place where there are branch instructions
- A *basic block* is a sequence of statements that enters at the start and ends with a branch at the end
- Remaining task of code generation is to create code for basic blocks and branch them together

Summary

- TAC is one example of an intermediate representation (IR)
- An IR should be close enough to existing machine code instructions so that subsequent translation into assembly is trivial
- In an IR we ignore some complexities and differences in computer architectures, such as limited registers, multiple instructions, branch delays, load delays, etc.