CMPT 755
Compilers
Anoop Sarkar
http://www.cs.sfu.ca/~anoop
TAC: Intermediate Representation
TAC: 3-Address Code

• Instructions that operate on named locations and labels
  – Mini-ISA or “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)
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
        int t; /* @x+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
TAC Instructions (I)

- Assignment
- rhs can be
  - Location
  - String Constant
  - Integer Constant
  - Label

Example:

```c
Code.Append(
    new LoadStringConstant(
        /*t3=*/GenTempVar(), "Hello")
);```

t2 := t1;
t3 := "Hello"
t5 := 42;
t7 := _L1;
TAC Instructions (II)

- Arithmetic
  - Binary add, sub, multiply, divide, modulo
- Equality (eq)
- Relational (lt)
- Logical (and, or)

- Labels and branches:
  - Insert label in TAC stream
    _L4:
  - Unconditional branch
    goto _L4
  - Conditional branch
    ifz t1 goto _L3
TAC Instructions (III)

• Preparing function calls
  – param t1;
  – (eval left to right)
  – (push right to left)
  – pop n

• Calling methods
• Label vs. Address
  – call
• Void vs. nonvoid
  – T1 = call _L3
  – call t3 (akin to jump return)
• Defining functions
  – BeginFunc <n>
    • Enter function, specify or forward-declare stack frame size
  – EndFunc
  – Return
  – Return t3

• Loads and Stores
  – Optional integer offset
  – Examples:
    t2 = *(t4)
    *(t5+4) = t6

• Unary minus, logical not
  t2 := not t3
What TAC doesn’t give you

- Array indexing (bounds check)
- Two or n-dimensional arrays
- Relational $\leq$, $\geq$, $>$, ...
- Conditional branches other than \texttt{ifz}
- Field names in records/structures
  - Use base+offset load/store
- Object data and method access
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:
    BeginFunc 32;
    _tmp0 := x - y;
    d := _tmp0;
    _tmp1 := 0;
    _tmp2 := _tmp1 < d;
    ifz _tmp2 goto _L0;
    param y #1;
    param d #0;
    _tmp3 := call _gcd;
    pop 8;
    return _tmp3;
    goto _L1;
_L0:
    _tmp4 := 0;
    ....
_L1:
    EndFunc;
int factorial(int n)
{
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}

void main()
{
    Print(factorial(6));
}
Short-circuiting Booleans

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

- Typical sequence:
  \[ t_1 := \text{not } c \]
  \[ t_2 := b \text{ and } t_1 \]
  \[ t_3 := a \text{ or } t_2 \]

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

- Short-circuit sequence:
  \[ t_1 := a \]
  ifz \( t_1 \) goto \( _L0 \) /* sckt */
  goto \( _L4 \)
  \( _L0: t_2 := b \)
  ifz \( t_2 \) goto \( _L1 \)
void main() {
    int i;
    for (i = 0; i < 10; i = i + 1)
        Print(i);
}

main:
    BeginFunc 24 ;
    _tmp0 := 0 ;
    i := _tmp0 ;
    _L0:
        _tmp1 := 10 ;
        _tmp2 := i < _tmp1 ;
        ifz _tmp2 goto _L1 ;
        param i #0 ;
        call _PrintInt ;
        pop 4 ;
        _tmp3 := 1 ;
        _tmp4 := i + _tmp3 ;
        i := _tmp4 ;
        goto _L0 ;
    _L1:
        EndFunc ;
```c
void foo(int[:] arr) {
    arr[1] = arr[0] * 2;
}
```

```c
_foo:  BeginFunc 48 ;
    _tmp0 := 1 ;
    _tmp1 := 4 ;
    _tmp3 := arr + _tmp2 ;
    _tmp4 := *( _tmp3 ) ;
    _tmp5 := 0 ;
    _tmp6 := 4 ;
    _tmp7 := _tmp6 * _tmp5 ;
    _tmp8 := arr + _tmp7 ;
    _tmp9 := *( _tmp8 ) ;
    _tmp10 := 2 ;
    _tmp11 := _tmp9 * _tmp10 ;
    _tmp4 := _tmp11 ;  \textbf{Wrong}
    EndFunc ;

_foo:  BeginFunc 44;
    _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 ;
    EndFunc ;  \textbf{Correct}
```
Backpatching

• 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 us to do it in one pass
• 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
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