CS 412
Introduction to Compilers

Andrew Myers
Cornell University

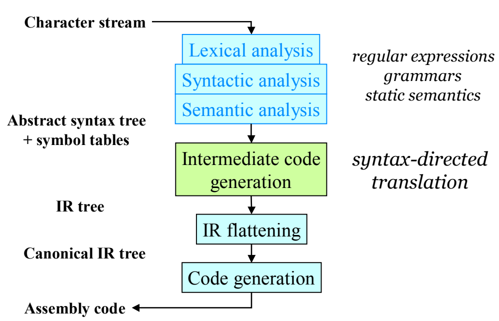
Lecture 14: Syntax-directed translation

23 Feb 01

Administration

- Read: Appel 6-8
- Prelim March 6, 7:30-9:30PM

Where we are



IR expressions

- CONST(*i*) : the integer constant *i*
- TEMP(*t*) : a temporary register *t*. The abstract machine has an infinite number of these
- OP(e_1, e_2) : one of the following operations
  - arithmetic: ADD, SUB, MUL, DIV, MOD
  - bit logic: AND, OR, XOR, LSHIFT, RSHIFT, ARSHIFT
  - comparisons: EQ, NEQ, LT, GT, LEQ, GEQ
- MEM(*e*) : contents of memory locn w/ address *e*
- CALL(e_f, e_0, e_1, ...) : result of fcn e_f applied to arguments e_i
- NAME(*n*) : address of the statement or global data location labeled *n* (TBD)
- ESEQ(*s*, *e*) : result of *e* after stmt *s* is executed

IR statements

- MOVE(*dest*, *e*) : move result of *e* into *dest*
  - *dest* = TEMP(*t*) : assign to temporary *t*
  - *dest* = MEM(*e*) : assign to memory locn *e*
- EXP(*e*) : evaluate *e*, discard result
- SEQ(s_1, ..., s_n) : execute each stmt s_i in order
- JUMP(*e*) : jump to address *e*
- CJUMP(*e*, l_1, l_2) : jump to statement named l_1 or l_2 depending on whether *e* is true or false
- LABEL(*n*) : a labeled statement (may be used in NAME, CJUMP)

Translation

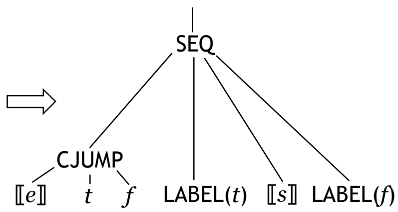
- Intermediate code generation is tree translation

  Abstract syntax tree $\Rightarrow$ IR tree

- Each subtree of AST translated to subtree in IR tree
- Translated version of AST subtree *e* is IR subtree $[\![e]\!]$

Translating if

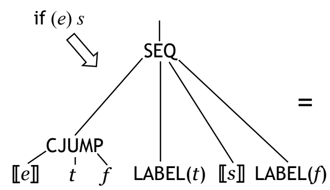
if (e) s



⟦if (e) s⟧ = SEQ(CJUMP(⟦e⟧, t, f), LABEL(t), ⟦s⟧, LABEL(f))

How to read IR trees

- Think of SEQ nodes as blocks of stmts



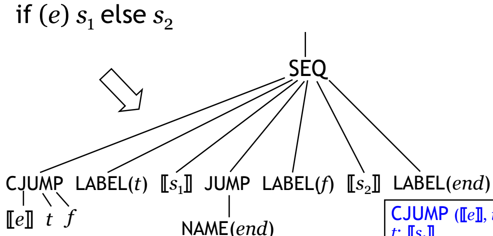
=

CJUMP (⟦e⟧, t, f)
t: ⟦s⟧
f:

= SEQ(CJUMP(⟦e⟧, t, f), LABEL(t), ⟦s⟧, LABEL(f))

Translating if-else

if (e) s_1 else s_2



=

CJUMP (⟦e⟧, t, f)
t: ⟦s_1⟧
JUMP *end*
f: ⟦s_2⟧
end:

Translating while

while (e) s

loop: CJUMP (⟦e⟧, t, f)
t: ⟦s⟧
JUMP(NAME(*loop*))
f:

= SEQ(LABEL(*loop*),
CJUMP(⟦e⟧, t, f),
LABEL(t),
⟦s⟧,
JUMP(NAME(*loop*))
LABEL(f))

Spec → Implementation

```
abstract class Node { abstract IRnode translate(); …}
```

⟦if (e) s⟧ = SEQ(CJUMP(⟦e⟧, t, f), LABEL(t), ⟦s⟧, LABEL(f))

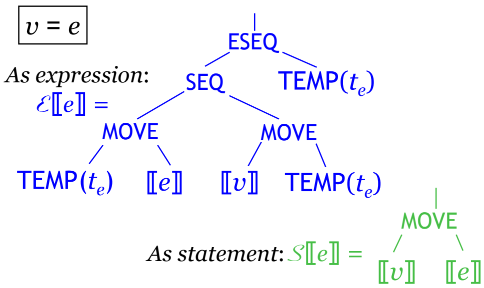
```
class IfNode { …
  IRnode translate() {
    SeqNode ret = new SEQ();
    ret.append(new CJUMP(e.translate(), "t", "f"));
    ret.append(new LABEL("t"));
    ret.append(s.translate());
    ret.append(new LABEL("f"));
    return ret;
  } … }
```

need fresh labels here

Syntax-directed translation

- Translation of any expression or statement expressed in terms of translations of subexpressions
- Can write down translations formally
  - precise specification of what compiler does
  - converts directly to an implementation
  - allows proof that compiler works correctly

Problem: multiple translations



Translation functions

- $\mathcal{E}[\![e]\!]$ is IR expr node that computes the same value as expression e (Appel: Ex)
- $\mathcal{E}[\![s]\!]$ is IR expr node that computes the same value as statement s (Appel: Ex)
- $\mathcal{S}[\![s]\!]$ is IR stmt node with side-effects of s (Appel: Nx)

- For boolean expr e, $\mathcal{C}[\![e, l_1, l_2]\!]$ is IR **statement** node that jumps to label l_1 if e evaluates to true and to l_2 if e evaluates to false (Appel: Cx)

Implementing translations

```
abstract class Node { ...
   abstract IRnode translateE();
   abstract IRnode translateS();
   abstract IRnode translateC(); … }
class Assignment {
   Expr variable, value;
      IRnode translateS() {
         return new MOVE(translateE(variable),
                                translateE(value)); }
      IRnode translateE() { TEMP t = freshTemp();
         return new ESEQ(new SEQ(new MOVE(t,
            value.translateE()), new MOVE(…)), t); }
```


Some examples so far

$\mathcal{E}[\![v]\!]$ = TEMP(v) (for local variable only!)
$\mathcal{E}[\![e_1+e_2]\!]$ = ADD($\mathcal{E}[\![e_1]\!]$ + $\mathcal{E}[\![e_2]\!]$)
$\mathcal{S}[\![v = e]\!]$ = MOVE($\mathcal{E}[\![v]\!]$, $\mathcal{E}[\![e]\!]$)
$\mathcal{E}[\![v = e]\!]$ = ESEQ(SEQ(MOVE(TEMP(t),$\mathcal{E}[\![e]\!]$),
MOVE($\mathcal{E}[\![v]\!]$, TEMP(t))),
TEMP(t))
$\mathcal{S}[\![$if (e) $s]\!]$ = SEQ(CJUMP($\mathcal{E}[\![e]\!]$, t, f),
LABEL(t), $\mathcal{S}[\![s]\!]$, LABEL(f))
$\mathcal{E}[\![$if (e) $s]\!]$ = ESEQ(SEQ(...?...), TEMP(t))
$\mathcal{E}[\![s_1;...;s_n]\!]$ = ?

Translating a function

- Function body is expression e
- Translate as statement $\mathcal{E}[\![e_1+e_2]\!]$?
- How to translate return statement?
- Idea: introduce return value register TEMP(RV), *function epilogue* label
- Function body e translated as

SEQ(MOVE(TEMP(RV), $\mathcal{E}[\![e]\!]$), LABEL(*epilogue*))

- return e translated as $\mathcal{S}[\![$return $e]\!]$ =

SEQ(MOVE(TEMP(RV), $\mathcal{E}[\![e]\!]$), JUMP(*epilogue*))

The boolean operator problem

- How to translate expression e_1 & e_2 ?
- How about $\mathcal{E}[\![e_1 \& e_2]\!]$ = AND($\mathcal{E}[\![e_1]\!]$, $\mathcal{E}[\![e_2]\!]$)?
- How about $\mathcal{E}[\![e_1 \& e_2]\!]$ =

ESEQ(SEQ(MOVE(TEMP(x), 0),
CJUMP($[\![e_1]\!]$, t1, no_set),
LABEL(t1), CJUMP($[\![e_2]\!]$, t2, no_set)
LABEL(t2), MOVE(TEMP(x), 1),
LABEL(no_set)),
TEMP(x))
?

Current translation

- Bad IR: $\mathcal{S}[\![\text{if } (e_1 \,\&\, e_2)\ s]\!]$ =

```
SEQ(CJUMP(ESEQ(SEQ(MOVE(TEMP(x), 0),
                   CJUMP(E[[e1]], t1, f),
                   LABEL(t1), CJUMP(E[[e2]], t2, f)
                   LABEL(t2), MOVE(TEMP(x), 1),
                   LABEL(f)),
              TEMP(x)), t, f),
    LABEL(t),
    S[[s]],
    LABEL(f))
```

- Better IR: SEQ(CJUMP($\mathcal{E}[\![e_1]\!]$, t1, f), LABEL(t1), CJUMP($\mathcal{E}[\![e_2]\!]$, t2, f), LABEL(t2), $\mathcal{S}[\![s]\!]$, LABEL(f))

Booleans via control

- Idea: representing boolean values via control flow rather than explicitly
- For boolean expr e, $\mathcal{C}[\![e, l_1, l_2]\!]$ is IR **statement** node that jumps to label l_1 if e evaluates to true and to l_2 if e evaluates to false

$\mathcal{C}[\![\text{true}, l_1, l_2]\!]$ = JUMP(NAME(l_1))

$\mathcal{C}[\![\text{false}, l_1, l_2]\!]$ = JUMP(NAME(l_2))

$\mathcal{C}[\![e_1 == e_2, l_1, l_2]\!]$ = CJUMP(EQ($\mathcal{E}[\![e_1]\!]$, $\mathcal{E}[\![e_2]\!]$), l_1, l_2)

Efficient translations of if and &

"$\mathcal{C}[\![e, l_1, l_2]\!]$ is IR **statement** node that jumps to label l_1 if *e* evaluates to true and to l_2 if *e* evaluates to false"

Can use to improve if translation:

$\mathcal{S}[\![\text{if } (e)\ s]\!]$ = SEQ($\mathcal{C}[\![e, t, f]\!]$, LABEL($t$), $\mathcal{S}[\![s]\!]$, LABEL(f))

$\mathcal{C}[\![e_1 \,\&\, e_2, l_1, l_2]\!]$ = SEQ($\mathcal{C}[\![e_1, t, l_2]\!]$, LABEL($t$), $\mathcal{C}[\![e_2, l_1, l_2]\!]$)

Now: $\mathcal{S}[\![\text{if } (e_1 \,\&\, e_2)\ s]\!]$ =
SEQ($\mathcal{C}[\![e_1$, t1, f $]\!]$, LABEL(t1), $\mathcal{C}[\![e_2$, t2, f $]\!]$,
LABEL(t2), $\mathcal{S}[\![s]\!]$, LABEL(f)): efficient

Progress

- Now have rules for transforming AST into intermediate representation
- Can apply this to AST of each function defn to get IR for function
- Intermediate representation has many features not found in real assembly code
  - arbitrarily deep expression trees vs. <5 deep
  - ability to perform statements with side-effects as part of an expression (ESEQ, CALL); undefined behavior
  - CJUMP is two-way jump rather than fall-through
- Next time: flattening IR (canonical form)