CS 412/413

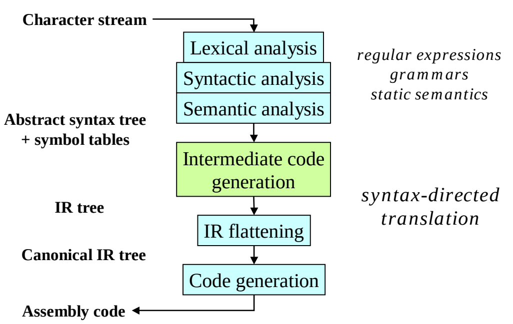
Introduction to
Compilers and Translators
Cornell University
Andrew Myers

Lecture 13: Syntax-directed translation
21 Feb 00

Administration

- Read: Appel 7, 8
- In-class prelim March 1

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(f, a_0, a_1, ...) : result of fcn f applied to arguments a_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)

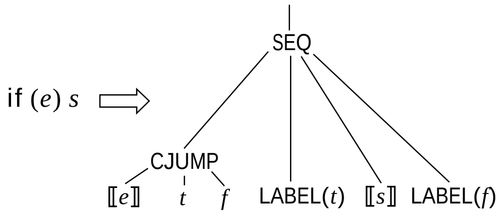
Translation

- 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 $[\![e]\!]$

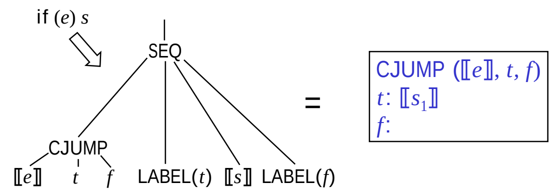
Translating if



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

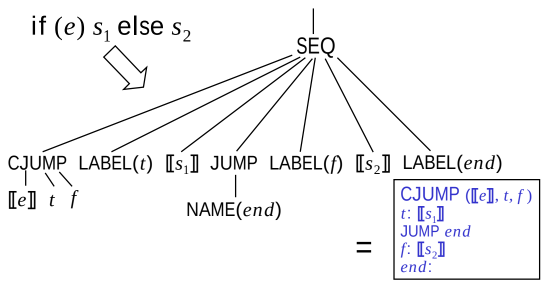
How to read IR trees

- Think of SEQ nodes as blocks of stmts



$= \text{SEQ(CJUMP(}[\![e]\!], t, f\text{), LABEL(}t\text{), } [\![s]\!]\text{, LABEL(}f\text{))}$

Translating if-else



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(); …}

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

```
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;
  } … }
```

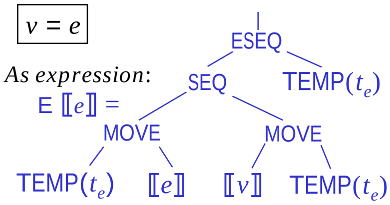

Syntax-directed translation

- 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

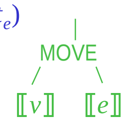
Problem: multiple translations

$v = e$

As expression: E $[\![e]\!]$ =



As statement: S $[\![e]\!]$ =



Translation functions

- E $[\![e]\!]$ is IR expr node that computes the same value as expression e (Appel: Ex)
- E $[\![s]\!]$ is IR expr node that computes the same value as statement s (Appel: Nx)
- S $[\![s]\!]$ is IR stmt node that has same side-effects as statement s (but no value)
- For boolean expr e, 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 (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); }
```

- Why is this code guaranteed to terminate?

Some examples so far

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

Translating a function

- Function body is expression e
- Translate as statement 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), E $[\![e]\!]$), LABEL(*epilogue*))

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

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

The boolean operator problem

- How to translate expression e_1 & e_2 ?
- How about E $[\![e_1 \& e_2]\!]$ = AND(e_1, e_2)?
- Problem: e_2 always evaluated
- How about E $[\![e_1 \& e_2]\!]$ =

```
ESEQ(SEQ(MOVE(TEMP(x), 0),
         CJUMP([[e1]], t1, no_set),
         LABEL(t1), CJUMP([[e2]], t2, no_set)
         LABEL(t2), MOVE(TEMP(x), 1),
         LABEL(no_set)),
     TEMP(x))
```

Current translation

- Bad IR: S $[\![$if (e_1 & e_2) $s]\!]$ =
  SEQ(CJUMP(ESEQ(SEQ(MOVE(TEMP(x), 0),
  CJUMP($[\![e_1]\!]$, t1, f),
  LABEL(t1), CJUMP($[\![e_2]\!]$, t2, f)
  LABEL(t2), MOVE(TEMP(x), 1),
  LABEL(f)),
  TEMP(x)), t, f),
  LABEL(t),
  S $[\![s]\!]$,
  LABEL(f))
- Better IR: SEQ(CJUMP($[\![e_1]\!]$, t1, f), LABEL(t1), CJUMP($[\![e_2]\!]$, t2, f), LABEL(t2), S $[\![s]\!]$, LABEL(f))

Booleans via control

- Idea: representing boolean values via control flow rather than explicitly
- For boolean expr e, 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

C $[\![$true, $l_1, l_2]\!]$ = JUMP(NAME(l_1))
C $[\![$false, $l_1, l_2]\!]$ = JUMP(NAME(l_2))
C $[\![e_1 == e_2, l_1, l_2]\!]$ = CJUMP(EQ(E $[\![e_1]\!]$, E $[\![e_2]\!]$), l_1, l_2)

Efficient translations of if and &

"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"

S $[\![$if (e) $s]\!]$ = SEQ(C $[\![e, t, f]\!]$,
LABEL(t), S $[\![s]\!]$,
LABEL(f))
C $[\![e_1$ & $e_2, l_1, l_2]\!]$ = SEQ(C $[\![e_1, t, l_2]\!]$, OR?
LABEL(t),
C $[\![e_2, l_1, l_2]\!]$)

- Now S $[\![$if (e_1 & e_2) $s]\!]$ = SEQ(C $[\![e_1$, t1, f $]\!]$, LABEL(t1), C $[\![e_2$, t2, f $]\!]$, LABEL(t2), 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. 1-2 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
- Why do we allow this in IR at all?