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

Character stream

Lexical analysis
Syntactic analysis
Semantic analysis

Abstract syntax tree + symbol tables

Intermediate code generation

IR tree

IR flattening

Canonical IR tree

Code generation

Assembly code

IR expressions

- CONST(i) : the integer constant i
- TEMP(t) : a temporary register t. The abstract machine has an infinite number of these
- OP(e1, e2) : 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, a0, a1, ...) : 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(s1, ..., sn) : execute each stmt s_i in order
- JUMP(e) : jump to address e
- CJUMP(e, l1, l2) : jump to statement named l_i or l_j 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 → IR tree

- Each subtree of AST translated to subtree in IR tree
- Translated version of AST subtree e is IR subtree [e]
Translating if

\[
\text{if (e) s} \quad \rightarrow \quad \text{SEQ} \quad \text{CJUMP} \quad \text{LABEL(t)} \quad \text{LABEL(f)}
\]

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

How to read IR trees

- Think of SEQ nodes as blocks of stmts

\[
\text{if (e) s} \quad \rightarrow \quad \text{SEQ} \quad \text{CJUMP} \quad \text{LABEL(t)} \quad \text{LABEL(f)}
\]

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

Translating if-else

\[
\text{if (e) } s_1 \text{ else } s_2
\]

\[
\text{CJUMP}(\text{e}, \text{t}, \text{f}) \quad \text{LABEL(t)} \quad \text{LABEL(f)}
\]

\[
\text{CJUMP}(\text{e}, \text{t}, \text{f}) \quad \text{LABEL(t)} \quad \text{LABEL(f)}
\]

Translating while

\[
\text{while (e) s}
\]

\[
\text{loop: CJUMP (e}, \text{t}, \text{f})
\]

\[
\text{t: [s]}
\]

\[
\text{f: }
\]

\[
\text{= SEQ(LABEL(loop),}
\]

\[
\text{CJUMP(e}, \text{t}, \text{f),}
\]

\[
\text{LABEL(t),}
\]

\[
\text{[s],}
\]

\[
\text{JUMP(NAME(loop))}
\]

\[
\text{LABEL(f)}
\]

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

Spec → Implementation

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

\[
\text{class IfNode { …}
\]

\[
\text{IRnode translate() {}
\]

\[
\text{SeqNode ret = new SEQ();}
\]

\[
\text{ret.append(new CJUMP(e.translate(), "t", "f"));}
\]

\[
\text{ret.append(new LABEL("t"));}
\]

\[
\text{ret.append(s.translate());}
\]

\[
\text{ret.append(new LABEL("f"));}
\]

\[
\text{return ret;}
\]

\[
\text{}}
\]

\[
\text{class IfNode { …
\]

\[
\text{IRnode translate() {}
\]

\[
\text{SeqNode ret = new SEQ();}
\]

\[
\text{ret.append(new CJUMP(e.translate(), "t", "f"));}
\]

\[
\text{ret.append(new LABEL("t"));}
\]

\[
\text{ret.append(new LABEL(f));}
\]

\[
\text{ret.append(new LABEL(f));}
\]

\[
\text{return ret;}
\]

\[
\text{}}
\]

\[
\text{class IfNode { …
\]

\[
\text{IRnode translate() {}
\]

\[
\text{SeqNode ret = new SEQ();}
\]

\[
\text{ret.append(new CJUMP(e.translate(), "t", "f"));}
\]

\[
\text{ret.append(new LABEL("t"));}
\]

\[
\text{ret.append(new LABEL("f"));}
\]

\[
\text{return ret;}
\]

\[
\text{}}
\]
Problem: multiple translations

\[ v = e \]

As expression:
\[
\begin{align*}
E [e] &= \text{MOVE} \left( \text{TEMP}(t_e), v \right) \\
&\quad \text{ESEQ} \left( \text{TEMP}(t_e), \text{TEMP}(t_e) \right) \\
&\quad \text{TEMP}(t_e)
\end{align*}
\]

As statement:
\[
\begin{align*}
S [e] &= \text{SEQUENCE} \left( \text{MOVE}(T E M P(t_v)), v \right) \\
&\quad \text{TEMP}(t_e)
\end{align*}
\]

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 \( e \), \( C \ [e] \) 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

\[
\begin{align*}
E [v] &= \text{TEMP}(v) \quad \text{(variable)} \\
E [v_1 + v_2] &= \text{ADD}(E [v_1], E [v_2]) \\
S [v = e] &= \text{MOVE}(E [v], E [e]) \\
E [v = e_1 & e_2] &= \text{ESEQ}( \text{SEQ}(\text{MOV}E(\text{TEMP}(t), E [e_1]), \text{MOV}E(E [v_1], \text{TEMP}(t))), \text{TEMP}(t_1)) \\
S [if (e) s] &= \text{SEQ}(\text{CJUMP}(E [e], t_1, \text{no_set}), \text{LABEL}(t_1), \text{CJUMP}(E [e_2], t_2, \text{no_set}) \text{LABEL}(t_2), \text{MOV}E(\text{TEMP}(t), 1), \text{LABEL}(\text{no_set})), \text{TEMP}(t))
\end{align*}
\]

Translating a function

- Function body is expression \( e \)
- Translate as statement \( E \ [e] \) ?
- How to translate return statement?
- Idea: introduce return value register \( \text{TEMP}(R V) \), function epilogue label
- Function body \( e \) translated as \( \text{SEQ}(\text{MOV}E(\text{TEMP}(R V), E [e]), \text{LABEL}(\text{epilogue})) \)
- return \( e \) translated as \( S \ [\text{return } e] = \text{SEQ}(\text{MOV}E(\text{TEMP}(R V), E [e]), \text{JUMP}(\text{epilogue})) \)

The boolean operator problem

- How to translate expression \( e_1 & e_2 \) ?
- About \( E \ [e_1 & e_2] = \text{AND}(e_1, e_2) \) ?
- Problem: \( e_2 \) always evaluated
- How about \( E \ [e_1 & e_2] = \text{ESEQ}(\text{SEQ}(\text{MOV}E(\text{TEMP}(x), 0), \text{CJUMP}(E [e_1], t_1, \text{no_set}), \text{LABEL}(t_1), \text{CJUMP}(E [e_2], t_2, \text{no_set}) \text{LABEL}(t_2), \text{MOV}E(\text{TEMP}(x), 1), \text{LABEL}(\text{no_set})), \text{TEMP}(x)) \)
Current translation

- Bad IR: $S \{ \text{if } (e_1 \& e_2) s \} =$
  
  \[\text{SEQ(CJUMP(ESEQ(MOVE(TEMP(x), 0)),)}\]
  \[\text{CJUMP(}[e_1], t_1, f, \)}
  \[\text{LABEL(t1), CJUMP(}[e_2], t_2, f, \)}
  \[\text{LABEL(t2), MOVE(TEMP(x), 1),} \)
  \[\text{LABEL(f), TEMP(x)) \} ; t, f)\]
  \[\text{LABEL(t),} \]
  \[\text{S \{s\}, Label(f))\]

- Better IR: $\text{SEQ(CJUMP(}[e_1], t_1, f, Label(t1),)}$
  \[\text{CJUMP(}[e_2], t_2, f, Label(t2), S \{s\}, Label(f))\]

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
  \[C \{[true, l_1, l_2]\} = \text{JUMP(NAME}(l_1))\]
  \[C \{[false, l_1, l_2]\} = \text{JUMP(NAME}(l_2))\]
  \[C \{e_1 == e_2, l_1, l_2\} = \text{CJUMP(EQ}[e_1, E[e_2]], l_1, l_2)\]

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"

$S \{\text{if } (e) s \} =$

\[\text{SEQ(CJUMP([e], t, f)}, \text{LABEL(t), S \{s\}, Label(f))}\]

\[C \{e_1 \& e_2, l_1, l_2\} =$

\[\text{SEQ(CJUMP([e_1], t_1, f)}, \text{LABEL(t1),)}\]
\[\text{CJUMP([e_2], t_2, f)}, \text{LABEL(t2), S \{s\}, Label(f))}; \text{efficient}\]

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?