CS302 Languages and Compiler Design II

Lecture 6
Intermediate Languages

Why have one?
• Simplify compilation task by dividing into stages.
• Ease porting to new source or target languages.
• Ease implementation of code transformations.

Must bridge source and object code styles.

Source code is primarily **hierarchical**.
• expressions
• structured control statements
• (perhaps) local scoping

Object code is primarily **linear**.
• explicit intermediate values
• explicit labels and jumps
• flat name space
Source Code -> ?? -> Object Code

Intermediate language is a compromise
- maintain some hierarchy for ease of generating I.L.
- linearize somewhat for ease of generating object code.
- many possibilities.

Linear machine-like code
- expressions are linearized, with intermediate results in temporaries.
- use explicit labels & jumps
- flat address space
- for PCAT project, we’ll use enriched SPARC assembler code.

I.R. trees
- expressions remain in tree form.
- use explicit labels & jumps
- locally-scoped temporaries.

Stack machine code
- expressions are linearized with intermediate results on stack
- use explicit labels & jumps
Specifying 3-address code

General form: three operands, one operator

\[ X := Y \text{ op } Z \]

Typical operators:

\[ A := B \]
\[ A := B \text{ op } C \]
\[ \text{goto L} \]
\[ \text{if0 A goto L} \]
\[ A := \text{addr B} \]
\[ A := *B \]

Operands: named variables, temporaries, labels.

Assume an abstract instruction-generation function:

\[ \text{gen(result, operator, arg1, arg2)} \]

- can produce strings, “quads”, or whatever.

Quite ad-hoc: we’ll add new instructions when we need to.
Example: 3-Addr Code for \( a := -b \times (c + 3) \)
Example: 3-Addr Code for \texttt{while a > 5 do a := b + 2}
Syntax-directed translation of Expressions

\[
\begin{align*}
E & := V \\
E & := N \\
E & := E + E \\
E & := E - E \\
E & := E > E \\
V & := VAR \\
N & := NUM
\end{align*}
\]

\[
\begin{align*}
E & .place = \text{newtemp}(); \\
E & .code = V .code @ [\text{gen}(E .place, *, V .place, _)] \\
E & .place = N .place; \\
E & .code = N .code \\
E & .place = \text{newtemp}(); \\
E & .code = E1 .code @ E2 .code @ [\text{gen}(E .place, +, E1 .place, E2 .place)] \\
E & .place = \text{newtemp}(); \\
E & .code = E1 .code @ [\text{gen}(E .place, \text{uminus}, E1 .place)] \\
E & .place = \text{newtemp}(); \\
E & .code = E1 .code @ E2 .code @ [\text{gen}(E .place, >, E1 .place, E2 .place)] \\
V & .place = \text{newtemp}(); \\
V & .code = [\text{gen}(V .place, \text{addr}, \text{VAR.var}, _)] \\
N & .place = \text{newtemp}(); \\
N & .code = [\text{gen}(N .place, \text{const}, \text{NUM.num}, _)]
\end{align*}
\]
Contrast Syntax-directed Evaluation of Expressions

\[
\begin{align*}
E & := V \quad \text{E.val} = \ast V.\text{loc} \\
E & := N \quad \text{E.val} = N.\text{val} \\
E & := E \ '+' E \quad \text{E.val} = E_1.\text{val} + E_2.\text{val} \\
E & := ' - ' E \quad \text{E.val} = - E_1.\text{val} \\
E & := E \ ' > ' E \quad \text{E.val} = (E_1.\text{val} > E_2.\text{val}) \ ? 1 : 0 \\
V & := \text{VAR} \quad \text{V.\text{loc}} = \text{lookup(VAR.\text{var})} \\
N & := \text{NUM} \quad \text{N.\text{val}} = \text{NUM.\text{num}}
\end{align*}
\]
Syntax-directed translation of Statements

S := V `:=` E  
S.code = E.code @ V.code @ 
[gen(*V.place,:=,E.place,_)]

S := WHILE E DO S1  
S.code = 
let begin=newlabel() 
end = newlabel() 
in [gen(begin,:,_,_)] @ 
E.code @ 
[gen(end,if0,E.place,_)] @ 
S1.code @ 
[gen(begin,goto,_,_), 
gen(end,:,_,_)]

generates:
L1: if0 E goto L2 
S1 
goto L1 
L2:

S := S1 `;` S2  
S.code = S1.code @ S2.code
Contrast: Syntax-directed Evaluation of Statements

\[ S \ := \ V \ := \ E \quad s() \{ \text{update}(V.\text{loc}, E.\text{val}); \} \]

\[ S \ := \ \text{WHILE} \ E \ \text{DO} \ S_1 \quad s() \{ \text{while} (E.\text{val} <> 0) \]
  \[ S_1(); \]
  \} \]

\[ S \ := \ S_1 \ ; \ S_2 \quad s() \{ s_1(); \]
  \[ s_2(); \}
  \} \]