CS412/CS413

Introduction to Compilers
Tim Teitelbaum

Lecture 18: Intermediate Code
29 Feb 08
Summary: Semantic Analysis

• Check errors not detected by lexical or syntax analysis

• Scope errors:
  – Variables not defined
  – Multiple declarations

• Type errors:
  – Assignment of values of different types
  – Invocation of functions with different number of parameters or parameters of incorrect type
  – Incorrect use of return statements
Semantic Analysis

• Type checking
  – Use type checking rules
  – Static semantics = formal framework to specify type-checking rules

• There are also control flow errors:
  – Must verify that a break or continue statement is always enclosed by a while (or for) statement
  – Java: must verify that a break X statement is enclosed by a for loop with label X
  – Can easily check control-flow errors by recursively traversing the AST
Where We Are

Source code (character stream) → Lexical analysis

Token stream → Syntactic Analysis

Abstract syntax tree → Semantic Analysis

Abstract syntax tree + symbol tables, types → Intermediate Code Generation

Intermediate Code
Intermediate Code

- Usually two IRs:

  **High-level IR**
  - Language-independent
  - (but closer to language)

  **Low-level IR**
  - Machine independent
  - (but closer to machine)

  - Languages:
    - C
    - Fortran
    - Pascal
    - Pentium
    - Java bytecode
    - PowerPC
High-level IR

• **Tree node structure**, essentially **ASTs**
• **High-level constructs common to many languages**
  - Expression nodes
  - Statement nodes

• **Expression nodes for:**
  - Integers and program variables
  - Binary operations: \texttt{e1 \ OP \ e2}
    • Arithmetic operations
    • Logic operations
    • Comparisons
  - Unary operations: \texttt{OP \ e}
  - Array accesses: \texttt{e1[e2]}
High-level IR

- **Statement nodes:**
  - Block statements (statement sequences): \((s_1, \ldots, s_N)\)
  - Variable assignments: \(v = e\)
  - Array assignments: \(e_1[e_2] = e_3\)
  - If-then-else statements: \(\text{if } c \text{ then } s_1 \text{ else } s_2\)
  - If-then statements: \(\text{if } c \text{ then } s\)
  - While loops: \(\text{while } (c) \ s\)
  - Function call statements: \(f(e_1, \ldots, e_N)\)
  - Return statements: \(\text{return } \text{or } \text{return } e\)

- **May also contain:**
  - For loop statements: \(\text{for } (v = e_1 \text{ to } e_2) \ s\)
  - **Break** and **continue** statements
  - Switch statements: \(\text{switch}(e) \{ \ v_1: s_1, \ldots, v_N: s_N \ \}\)
Low-Level IR

• Low-level representation is essentially an instruction set for an abstract machine

• Alternatives for low-level IR:
  – Three-address code or quadruples (Dragon Book):
    \[ a = b \text{ OP } c \]
  – Tree representation (Tiger Book)
  – Stack machine (like Java bytecode)
Three-Address Code

• In this class: **three-address code**
  
  \[ a = b \text{ OP } c \]

• Has at most three addresses (may have fewer)

• Also named **quadruples** because can be represented as: (a, b, c, OP)

• Example:
  
  \[
  a = (b+c)*(-e); \quad t1 = b + c \quad t2 = - e \quad a = t1 * t2 
  \]
Low IR Instructions

• Assignment instructions:
  - Binary operations: \( a = b \text{ OP } c \)
    • arithmetic: ADD, SUB, MUL, DIV, MOD
    • logic: AND, OR, XOR
    • comparisons: EQ, NEQ, LT, GT, LEQ, GEQ
  - Unary operation \( a = \text{ OP } b \)
    • Arithmetic MINUS or logic NEG
  - Copy instruction: \( a = b \)
  - Load /store: \( a = *b, *a = b \)
  - Other data movement instructions
Low IR Instructions, cont.

• Flow of control instructions:
  - `label L`: label instruction
  - `jump L`: Unconditional jump
  - `cjump a L`: conditional jump

• Function call
  - `call f(a_1, ..., a_n)`
  - `a = call f(a_1, ..., a_n)`
  - Is an extension to quads

• …IR describes the Instruction Set of an abstract machine
Example

```c
m = 0;
if (c == 0) {
    m = m + n*n;
} else {
    m = m + n;
}
```

```c
m = 0
t1 = (c == 0)
fjump t1 falseb
t2 = n * n
m = m + t2
jump end
label falseb
m = m + n
label end
```
How To Translate?

• May have nested language constructs
  - Nested if and while statements

• Need an algorithmic way to translate

• Solution:
  - Start from the AST representation
  - Define translation for each node in the AST in terms of a (recursive) translation of its constituents
Notation

• Use the notation $T[e] = \text{low-level IR of high-level IR construct } e$
• $T[e]$ is sequence of low-level IR instructions
• If $e$ is expression (or statement expression), $T[e]$ represents a value
• Denote by $t = T[e]$ the low-level IR of $e$, whose result value is stored in $t$
• For variable $v$, define $T[v]$ to be $v$, i.e., $t = T[v]$ is copy instruction $t = v$
Translating Expressions

- Binary operations: \[ t = T[ e_1 \ OP \ e_2 ] \]
  (arithmetic operations and comparisons)

  \[
  \begin{align*}
  t_1 &= T[ e_1 ] \\
  t_2 &= T[ e_2 ] \\
  t &= t_1 \ OP \ t_2
  \end{align*}
  \]

- Unary operations: \[ t = T[ \ OP \ e ] \]

  \[
  \begin{align*}
  t_1 &= T[ e ] \\
  t &= \ OP \ t_1
  \end{align*}
  \]
Translating Boolean Expressions

• $t = T[e_1 \text{ OR } e_2]$

$$
\begin{align*}
t1 &= T[e_1] \\
t2 &= T[e_2] \\
t &= t1 \text{ OR } t2
\end{align*}
$$

• … but how about short-circuit OR, for which we should compute $e_2$ only if $e_1$ evaluates to false
Translating Short-Circuit OR

• Short-circuit OR: \( t = T[ e_1 \text{ SC-OR } e_2 ] \)

\[
\begin{align*}
  t &= T[ e_1 ] \\
  \text{tjump } t \text{ Lend} \\
  t &= T[ e_2 ] \\
  \text{label } \text{Lend}
\end{align*}
\]

• … how about short-circuit AND?
Translating Short-Circuit AND

- Short-circuit AND: \( t = T[ e1 \ SC-AND \ e2 ] \)

\[
\begin{align*}
t &= T[ e1 ] \\
fjump t &\text{ Lend} \\
t &= T[ e2 ] \\
\text{label} &\text{ Lend}
\end{align*}
\]
Array and Field Accesses

• Array access: \[ t = T[v[e]] \]

\[ t1 = T[e] \]
\[ t = v[t1] \]

• Field access: \[ t = T[e1.f] \]

\[ t1 = T[e1] \]
\[ t = t1.f \]
Nested Expressions

• In these translations, expressions may be nested;
• Translation recurses on the expression structure

• Example: \( t = T[ (a \ - \ b) \ * \ (c \ + \ d) ] \)

\[
\begin{align*}
    t1 &= a \\
    t2 &= b \\
    t3 &= t1 - t2 \\
    t4 &= b \\
    t5 &= c \\
    t5 &= t4 + t5 \\
    t &= t3 \ * \ t5
\end{align*}
\]

\[
\begin{align*}
    T[ (a \ - \ b) ] \\
    T[ (c \ + \ d) ]
\end{align*}
\]
Translating Statements

• Statement sequence: \( T[ s1; s2; \ldots; sN ] \)

\[
\begin{align*}
T[ s1 ] \\
T[ s2 ] \\
\ldots \\
T[ sN ]
\end{align*}
\]

• IR instructions of a statement sequence = concatenation of IR instructions of statements
Assignment Statements

- Variable assignment: $T[ v = e ]$
  
  $t = T[ e ]$
  $v = t$

  [alternatively]
  $v = T[ e ]$

- Array assignment: $T[ v[e1] = e2 ]$
  
  $t1 = T[ e1 ]$
  $t2 = T[ e2 ]$
  $v[t1] = t2$
Translating If-Then-Else

• $T[ \text{if (e) then } s_1 \text{ else } s_2 ]$

  \[ t_1 = T[ e ] \]
  \[ \text{fjump } t_1 \text{ Lfalse} \]
  \[ T[ s_1 ] \]
  \[ \text{jump } \text{Lend} \]
  \[ \text{label } \text{Lfalse} \]
  \[ T[ s_2 ] \]
  \[ \text{label } \text{Lend} \]
Translating If-Then

- $T[ \text{if (e) then s} ]$

```plaintext
t1 = T[ e ]
fjump t1 Lend
T[ s ]
label Lend
```

if-then
    \( / \)
    \( / \)
    e
    s
While Statements

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

```
label Ltest
t1 = T[ e ]
fjump t1 Lend
T[ s ]
jump Ltest
label Lend
```

```
while
  \/_
  e s
```
Switch Statements

- \( T[\; \text{switch} \; (e) \{ \; \text{case} \; v_1: \; s_1, \; \ldots, \; \text{case} \; v_N: \; s_N \; \} \; ] \)

\[
\begin{align*}
t &= T[e] \\
c &= t \neq v_1 \\
t &\text{jump } c \; L2 \\
T[s_1] \\
&\text{jump } Lend \\
\text{label } L2 \\
c &= t \neq v_2 \\
t &\text{jump } c \; L3 \\
T[s_2] \\
&\text{jump } Lend \\
\ldots \\
\text{label } LN \\
c &= t \neq v_N \\
t &\text{jump } c \; Lend \\
T[s_N] \\
&\text{label } Lend
\end{align*}
\]
Call and Return Statements

• \( T[ \text{call } f(e_1, e_2, \ldots, e_N) ] \)

\[
\begin{align*}
t_1 &= T[ e_1 ] \\
t_2 &= T[ e_2 ] \\
&\vdots \\
t_N &= T[ e_N ] \\
\text{call } f(t_1, t_2, \ldots, t_N)
\end{align*}
\]

• \( T[ \text{return } e ] \)

\[
\begin{align*}
t &= T[ e ] \\
\text{return } t
\end{align*}
\]
Nested Statements

• Same for statements as expressions: recursive translation

• Example: $T[\text{ if } c \text{ then if } d \text{ then } a = b ]$

  $t1 = c$
  $\text{fjump } t1 \text{ Lend1}$  
  $t2 = d$
  $\text{fjump } t2 \text{ Lend2}$
  $t3 = b$
  $T[ a = b ]$
  $a = t3$
  $\text{label Lend2}$
  $\text{label Lend1}$
IR Lowering Efficiency

t1 = c
fjump t1 Lend1

t2 = d
fjump t2 Lend2

t3 = b
a = t3
label Lend2
label Lend1

if-then
    c
if-then
    d =
        a
        b

fjump c Lend
fjump d Lend
a = b
Label Lend