Intermediate Code Generation

Lecture 10

Objectives

By the end of this lecture you should be able to:

1. Construct three-address code translations of assignments and expressions.
2. Construct three-address code translations of flow control structures.

Outline

Outline

1. Expressions and Assignments
2. Flow Control

Strategy

- We would like to construct three-address code for assignment statements and expressions.
- With each non-terminal we associate an attribute, *code*, which contains the piece of three-address code computing the expression or executing the assignment.
- With each non-terminal deriving an expression we associate an attribute, *addr*, which contains a reference to the value of the expression.
  - Recall that an address is a name (identifier), a constant, or a compiler-generated temporary.

Strategy

- We would like to construct three-address code for assignment statements and expressions.
- With each non-terminal we associate an attribute, *code*, which contains the piece of three-address code computing the expression or executing the assignment.
- With each non-terminal deriving an expression we associate an attribute, *addr*, which contains a reference to the value of the expression.
  - Recall that an address is a name (identifier), a constant, or a compiler-generated temporary.

Strategy

- We would like to construct three-address code for assignment statements and expressions.
- With each non-terminal we associate an attribute, *code*, which contains the piece of three-address code computing the expression or executing the assignment.
- With each non-terminal deriving an expression we associate an attribute, *addr*, which contains a reference to the value of the expression.
  - Recall that an address is a name (identifier), a constant, or a compiler-generated temporary.

Strategy

- We would like to construct three-address code for assignment statements and expressions.
- With each non-terminal we associate an attribute, *code*, which contains the piece of three-address code computing the expression or executing the assignment.
- With each non-terminal deriving an expression we associate an attribute, *addr*, which contains a reference to the value of the expression.
  - Recall that an address is a name (identifier), a constant, or a compiler-generated temporary.

Simple Arithmetic Expression SDD

Example

$$
\begin{array}{llll}
S & \longrightarrow & \textbf{id} = E & S.code = E.code \\
 & & & \quad \circ gen(top.get(\textbf{id}.lexeme)\ '='\ E.addr) \\
E & \longrightarrow & E_1 + E_2 & E.addr = \textbf{new}\ Temp() \\
 & & & E.code = E_1.code \circ E_2.code \\
 & & & \quad \circ gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr) \\
E & \longrightarrow & -E_1 & E.addr = \textbf{new}\ Temp() \\
 & & & E.code = E_1.code \circ gen(E.addr\ '= \textbf{minus}'\ E_1.addr) \\
E & \longrightarrow & (E_1) & E.addr = E_1.addr \\
 & & & E.code = E_1.code \\
E & \longrightarrow & \textbf{id} & E.addr = top.get(\textbf{id}.lexeme) \\
 & & & E.code = \ ''
\end{array}
$$

Addressing Array Elements

- Would like to translate k-dimensional array references of the form $A[i_1][i_2]\cdots[i_k]$.
- Like all variables, array variables have an "offset" attribute in their symbol table entry indicating their relative address; we refer to the value of this attribute as the base of the array.
- To access a particular entry as indicated above, we need to calculate its relative address.

Addressing Array Elements

- Would like to translate k-dimensional array references of the form $A[i_1][i_2]\cdots[i_k]$.
- Like all variables, array variables have an "offset" attribute in their symbol table entry indicating their relative address; we refer to the value of this attribute as the base of the array.
- To access a particular entry as indicated above, we need to calculate its relative address.

Addressing Array Elements

- Would like to translate k-dimensional array references of the form $A[i_1][i_2]\cdots[i_k]$.
- Like all variables, array variables have an "offset" attribute in their symbol table entry indicating their relative address; we refer to the value of this attribute as the base of the array.
- To access a particular entry as indicated above, we need to calculate its relative address.

Calculating Addresses of Array Elements

- Suppose an array is declared thus

$$T[n_1][n_2]\cdots[n_k]$$

- Assuming zero-based arrays and row-major indexing, the address of $A[i_1][i_2]\cdots[i_k]$ is given by

$$addr(A[i_1][i_2]\cdots[i_k]) =$$

$$\begin{cases} base + i_1 \times w_1 & \text{if } k = 1 \\ addr(A[i_1][i_2]\cdots[i_{k-1}]) + i_k \times w_k & \text{otherwise} \end{cases}$$

where

$$w_i = T.width \times \prod_{j=i+1}^{k} n_j$$

Calculating Addresses of Array Elements

- Suppose an array is declared thus

$$T[n_1][n_2]\cdots[n_k]$$

- Assuming zero-based arrays and row-major indexing, the address of $A[i_1][i_2]\cdots[i_k]$ is given by

$$addr(A[i_1][i_2]\cdots[i_k]) =$$

$$\begin{cases} base + i_1 \times w_1 & \text{if } k = 1 \\ addr(A[i_1][i_2]\cdots[i_{k-1}]) + i_k \times w_k & \text{otherwise} \end{cases}$$

where

$$w_i = T.width \times \prod_{j=i+1}^{k} n_j$$

Calculating Addresses of Array Elements

- Suppose an array is declared thus

$$T[n_1][n_2]\cdots[n_k]$$

- Assuming zero-based arrays and row-major indexing, the address of $A[i_1][i_2]\cdots[i_k]$ is given by

$$addr(A[i_1][i_2]\cdots[i_k]) =$$

$$\begin{cases} base + i_1 \times w_1 & \text{if } k = 1 \\ addr(A[i_1][i_2]\cdots[i_{k-1}]) + i_k \times w_k & \text{otherwise} \end{cases}$$

where

$$w_i = T.width \times \prod_{j=i+1}^{k} n_j$$

Exercise

Example

Suppose we have the declaration

```
int[3][2][4] A
```

What is the relative address of `A[1][1][2]` if the base of A is 0 and the width of `int` is 4?

Solution.

- $addr(\texttt{A[1]}) = 0 + 1 \times 4 \times 2 \times 4 = 32.$
- $addr(\texttt{A[1][1]}) = 32 + 1 \times 4 \times 4 = 48.$
- $addr(\texttt{A[1][1][2]}) = 48 + 2 \times 4 = 56.$

Exercise

Example

Suppose we have the declaration

`int[3][2][4] A`

What is the relative address of `A[1][1][2]` if the base of A is 0 and the width of `int` is 4?

Solution.

- $addr(\mathtt{A}[1]) = 0 + 1 \times 4 \times 2 \times 4 = 32.$
- $addr(\mathtt{A}[1][1]) = 32 + 1 \times 4 \times 4 = 48.$
- $addr(\mathtt{A}[1][1][2]) = 48 + 2 \times 4 = 56.$

Exercise

Example

Suppose we have the declaration

`int[3][2][4] A`

What is the relative address of `A[1][1][2]` if the base of A is 0 and the width of `int` is 4?

Solution.

- $addr(\mathtt{A}[1]) = 0 + 1 \times 4 \times 2 \times 4 = 32.$
- $addr(\mathtt{A}[1][1]) = 32 + 1 \times 4 \times 4 = 48.$
- $addr(\mathtt{A}[1][1][2]) = 48 + 2 \times 4 = 56.$

Exercise

Example

Suppose we have the declaration

```
int[3][2][4] A
```

What is the relative address of `A[1][1][2]` if the base of A is 0 and the width of `int` is 4?

Solution.

- $addr(\mathtt{A}[1]) = 0 + 1 \times 4 \times 2 \times 4 = 32.$
- $addr(\mathtt{A}[1][1]) = 32 + 1 \times 4 \times 4 = 48.$
- $addr(\mathtt{A}[1][1][2]) = 48 + 2 \times 4 = 56.$

Array Declarations

Example

Recall how arrays are represented.

$$
\begin{array}{lcll}
T & \longrightarrow & BC & C.t = B.type; C.w = B.width \\
 & & & T.type = C.type; T.width = C.width \\
B & \longrightarrow & \textbf{int} & B.type = integer; B.width = 4 \\
B & \longrightarrow & \textbf{float} & B.type = float; B.width = 8 \\
C & \longrightarrow & [\textbf{num}]C_1 & C_1.t = C.t; C_1.w = C.w \\
 & & & C.type = array(\textbf{num}.value, C_1.type) \\
 & & & C.width = \textbf{num}.value \times C_1.width \\
C & \longrightarrow & \varepsilon & C.type = C.t; C.width = C.w
\end{array}
$$

Translating Array References

Suppose we want to augment the CFG for assignments and expressions with the following rules.

$$
\begin{array}{lcl}
S & \longrightarrow & L = E \\
E & \longrightarrow & L \\
L & \longrightarrow & \textbf{id}\ [E] \mid L\ [E]
\end{array}
$$

How would we augment the SDD?

Strategy

- For the non-terminal deriving array references, we associate three synthesized attributes.
  1. *array* holds a pointer to the symbol-table entry of the array name.
     - The symbol-table entry contains the array base, width, and type, among others.
  2. *addr* holds the temporary which contains the offset to be added to the array base.
  3. *type* holds the type of the (sub-)array expression derived by the non-terminal.

Strategy

- For the non-terminal deriving array references, we associate three synthesized attributes.
  1. *array* holds a pointer to the symbol-table entry of the array name.
     - The symbol-table entry contains the array base, width, and type, among others.
  2. *addr* holds the temporary which contains the offset to be added to the array base.
  3. *type* holds the type of the (sub-)array expression derived by the non-terminal.

Strategy

- For the non-terminal deriving array references, we associate three synthesized attributes.
    1. *array* holds a pointer to the symbol-table entry of the array name.
        - The symbol-table entry contains the array base, width, and type, among others.
    2. *addr* holds the temporary which contains the offset to be added to the array base.
    3. *type* holds the type of the (sub-)array expression derived by the non-terminal.

Strategy

- For the non-terminal deriving array references, we associate three synthesized attributes.
  1. *array* holds a pointer to the symbol-table entry of the array name.
     - The symbol-table entry contains the array base, width, and type, among others.
  2. *addr* holds the temporary which contains the offset to be added to the array base.
  3. *type* holds the type of the (sub-)array expression derived by the non-terminal.

Array SDD

Example

$$
\begin{array}{llll}
S & \longrightarrow & L = E & S.code = E.code \\
 & & & \quad \circ gen(L.array.base'['L.addr'] =' \; E.addr) \\
E & \longrightarrow & L & E.addr = \textbf{new}Temp() \\
 & & & E.code = L.code \\
 & & & \quad \circ gen(E.addr\,' =' \; L.array.base'['L.addr']') \\
L & \longrightarrow & \textbf{id}\ [E] & L.array = top.get(\textbf{id}.lexeme); L.addr = \textbf{new}Temp() \\
 & & & L.type = L.array.type.elem \\
 & & & L.code = E.code \\
 & & & \quad \circ gen(L.addr\,' =' \; E.addr\,' *' \; L.type.width) \\
L & \longrightarrow & L_1\ [E] & L.array = L_1.array; L.type = L_1.type.elem \\
 & & & t = \textbf{new}Temp(); L.addr = \textbf{new}Temp() \\
 & & & L.code = L1.code \circ E.code \\
 & & & \quad \circ gen(t\,' =' \; E.addr\,' *' \; L.type.width) \\
 & & & \quad \circ gen(L.addr\,' =' \; L_1.addr\,' +' \; t)
\end{array}
$$

Exercise

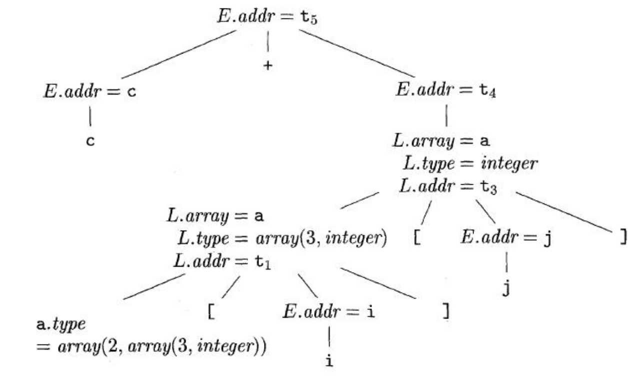
Example

Give the annotated parse tree and the generated three-address code for the string `c + a[i][j]`, where `a` is a 2×3 array of `int`. Use the expression SDD and the array SDD .

Exercise: Tree

Example

Expression SDD and array SDD.



©Aho et al. (2007)

Exercise: Three-Address Code

Example

Expression SDD and array SDD .

```
t1 = i * 12
t2 = j * 4
t3 = t1 + t2
t4 = a[t3]
t5 = c + t4
```

Outline

Boolean Expressions

- By default, execution flows from one instruction to the textually next instruction.
- Control structures use boolean expressions to alter the flow of execution.
- Boolean expressions have two roles requiring different translation schemes:
  1. as expressions evaluating to `true` or `false` and
  2. as controllers of execution flow.
- We focus on the second.

Boolean Expressions

- By default, execution flows from one instruction to the textually next instruction.
- Control structures use boolean expressions to alter the flow of execution.
- Boolean expressions have two roles requiring different translation schemes:
  1. as expressions evaluating to `true` or `false` and
  2. as controllers of execution flow.
- We focus on the second.

Boolean Expressions

- By default, execution flows from one instruction to the textually next instruction.
- Control structures use boolean expressions to alter the flow of execution.
- Boolean expressions have two roles requiring different translation schemes:
  1. as expressions evaluating to `true` or `false` and
  2. as controllers of execution flow.
- We focus on the second.

Boolean Expressions

- By default, execution flows from one instruction to the textually next instruction.
- Control structures use boolean expressions to alter the flow of execution.
- Boolean expressions have two roles requiring different translation schemes:
  1. as expressions evaluating to `true` or `false` and
  2. as controllers of execution flow.
- We focus on the second.

Boolean Expressions

- By default, execution flows from one instruction to the textually next instruction.
- Control structures use boolean expressions to alter the flow of execution.
- Boolean expressions have two roles requiring different translation schemes:
  1. as expressions evaluating to `true` or `false` and
  2. as controllers of execution flow.
- We focus on the second.

Boolean Expressions

- By default, execution flows from one instruction to the textually next instruction.
- Control structures use boolean expressions to alter the flow of execution.
- Boolean expressions have two roles requiring different translation schemes:
    1. as expressions evaluating to `true` or `false` and
    2. as controllers of execution flow.
- We focus on the second.

A CFG for Programs

We assume the following CFG for control of flow statements

$$
\begin{array}{lcl}
P & \longrightarrow & S \\
S & \longrightarrow & \mathbf{id}{=}E \\
 & \longrightarrow & \mathbf{if}\ (B)\ S \\
 & \longrightarrow & \mathbf{if}\ (B)\ S\ \mathbf{else}\ S \\
 & \longrightarrow & \mathbf{while}\ (B)\ S \\
 & \longrightarrow & S\ S
\end{array}
$$

SDD: Program

$$P \longrightarrow S$$
$$S.next = newlabel()$$
$$P.code = S.code \circ label(S.next)$$

- $newlabel()$ generates a new label.
- $label(l)$ attaches label l to the next instruction.

SDD: If-Then

$S \longrightarrow \textbf{if } (B)\ S_1$

$$\begin{array}{rcl} B.true & = & newlabel() \\ B.false & = & S_1.next = S.next \\ S.code & = & B.code \\ & \circ & label(B.true) \\ & \circ & S_1.code \end{array}$$

- $B.true$ is the label to which control flows if the value of B is `true`.
- $B.false$ is the label to which control flows if the value of B is `false`.

SDD: If-Then-Else

$$S \longrightarrow \textbf{if } (B)\ S_1 \textbf{ else } S_2$$

$$\begin{array}{rcl}
B.true & = & newlabel() \\
B.false & = & newlabel() \\
S_1.next & = & S_2.next = S.next \\
S.code & = & B.code \\
 & \circ & label(B.true) \\
 & \circ & S_1.code \\
 & \circ & gen('\texttt{goto}'\ S.next) \\
 & \circ & label(B.false) \\
 & \circ & S_2.code
\end{array}$$

SDD: While-Loop

$$S \longrightarrow \textbf{while}\ (B)\ S_1$$

$$\begin{aligned}
B.true &= newlabel() \\
B.false &= S.next \\
S_1.next &= newlabel() \\
S.code &= label(S1.next) \\
&\circ\ B.code \\
&\circ\ label(B.true) \\
&\circ\ S_1.code \\
&\circ\ gen('\texttt{goto}'\ S_1.next)
\end{aligned}$$

SDD: Sequence

$S \longrightarrow S_1 \; S_2$

$$
\begin{aligned}
S_1.next &= newlabel() \\
S_2.next &= S.next \\
S.code &= S_1.code \\
&\circ\ label(S_1.next) \\
&\circ\ S_2.code
\end{aligned}
$$

A Grammar for Boolean Expressions

In what follows, we assume the following CFG for boolean expressions.

$$
\begin{array}{lcl}
B & \longrightarrow & B \,||\, B \\
 & \longrightarrow & B \;\&\&\; B \\
 & \longrightarrow & !\, B \\
 & \longrightarrow & (B) \\
 & \longrightarrow & E \;\mathbf{rel}\; E \\
 & \longrightarrow & \mathbf{true} \\
 & \longrightarrow & \mathbf{false}
\end{array}
$$

Short-Circuit Evaluation

- Short-circuit evaluation of boolean expressions does not evaluate the second operand of a binary boolean operator if the value of the expression can be determined from the first operand alone.
  - For `||`, this happens when the first operand evaluates to `true`.
  - For `&&`, this happens when the first operand evaluates to `false`.
- Some programming languages have only short-circuit operators (e.g. Lisp), others have two sets of operators (e.g. Java).
- One should be careful with which one to choose since operand evaluation may have important side-effects.
- With short-circuit evaluation, boolean operators translate into jumps in three-address code.

Short-Circuit Evaluation

- Short-circuit evaluation of boolean expressions does not evaluate the second operand of a binary boolean operator if the value of the expression can be determined from the first operand alone.
  - For `||`, this happens when the first operand evaluates to `true`.
  - For `&&`, this happens when the first operand evaluates to `false`.
- Some programming languages have only short-circuit operators (e.g. Lisp), others have two sets of operators (e.g. Java).
- One should be careful with which one to choose since operand evaluation may have important side-effects.
- With short-circuit evaluation, boolean operators translate into jumps in three-address code.

Short-Circuit Evaluation

- Short-circuit evaluation of boolean expressions does not evaluate the second operand of a binary boolean operator if the value of the expression can be determined from the first operand alone.
  - For `||`, this happens when the first operand evaluates to `true`.
  - For `&&`, this happens when the first operand evaluates to `false`.
- Some programming languages have only short-circuit operators (e.g. Lisp), others have two sets of operators (e.g. Java).
- One should be careful with which one to choose since operand evaluation may have important side-effects.
- With short-circuit evaluation, boolean operators translate into jumps in three-address code.

Short-Circuit Evaluation

- Short-circuit evaluation of boolean expressions does not evaluate the second operand of a binary boolean operator if the value of the expression can be determined from the first operand alone.
  - For `||`, this happens when the first operand evaluates to `true`.
  - For `&&`, this happens when the first operand evaluates to `false`.
- Some programming languages have only short-circuit operators (e.g. Lisp), others have two sets of operators (e.g. Java).
- One should be careful with which one to choose since operand evaluation may have important side-effects.
- With short-circuit evaluation, boolean operators translate into jumps in three-address code.

Short-Circuit Evaluation

- Short-circuit evaluation of boolean expressions does not evaluate the second operand of a binary boolean operator if the value of the expression can be determined from the first operand alone.
  - For `||`, this happens when the first operand evaluates to `true`.
  - For `&&`, this happens when the first operand evaluates to `false`.
- Some programming languages have only short-circuit operators (e.g. Lisp), others have two sets of operators (e.g. Java).
- One should be careful with which one to choose since operand evaluation may have important side-effects.
- With short-circuit evaluation, boolean operators translate into jumps in three-address code.

Short-Circuit Evaluation

- Short-circuit evaluation of boolean expressions does not evaluate the second operand of a binary boolean operator if the value of the expression can be determined from the first operand alone.
  - For `||`, this happens when the first operand evaluates to `true`.
  - For `&&`, this happens when the first operand evaluates to `false`.
- Some programming languages have only short-circuit operators (e.g. Lisp), others have two sets of operators (e.g. Java).
- One should be careful with which one to choose since operand evaluation may have important side-effects.
- With short-circuit evaluation, boolean operators translate into jumps in three-address code.

SDD: Disjunction

$B \longrightarrow B_1 \mid\mid B_2$

$$
\begin{aligned}
B_1.true &= B.true \\
B_1.false &= newlabel() \\
B_2.true &= B.true \\
B_2.false &= B.false \\
B.code &= B_1.code \\
&\circ\ label(B_1.false) \\
&\circ\ B_2.code
\end{aligned}
$$

SDD: Conjunction

$B \longrightarrow B_1 \ \&\& \ B_2$

$$\begin{array}{lcl} B_1.true & = & newlabel() \\ B_1.false & = & B.false \\ B_2.true & = & B.true \\ B_2.false & = & B.false \\ B.code & = & B_1.code \\ & \circ & label(B_1.true) \\ & \circ & B_2.code \end{array}$$

SDD: Negation

$B \longrightarrow !\, B_1$

$$\begin{array}{lcl} B_1.true & = & B.false \\ B_1.false & = & B.true \\ B.code & = & B_1.code \end{array}$$

SDD: Parenthesized Expression

$B \longrightarrow (B_1)$

$$\begin{aligned} B_1.true &= B.true \\ B_1.false &= B.false \\ B.code &= B_1.code \end{aligned}$$

SDD: Relational Expressions

$B \longrightarrow E_1 \ \mathbf{rel} \ E_2$

$$
\begin{aligned}
B.code &= E_1.code \\
&\circ \ E_2.code \\
&\circ \ gen('\texttt{if}' \ E_1.addr \ \mathbf{rel}.op \ E_2.addr \ '\texttt{goto}' \ B.true) \\
&\circ \ gen('\texttt{goto}' \ B.false)
\end{aligned}
$$

SDD: true

$B \longrightarrow \textbf{true}$

$$B.code \quad = \quad gen('\texttt{goto}'\ B.true)$$

SDD: false

$B \longrightarrow \textbf{false}$

$$B.code \quad = \quad gen(' \texttt{goto} ' \; B.false)$$

Short-Circuit Translation

Example

The expression

```
if (x < 1000 || x > 200 && x != y) x = 0;
```

translates into

```
    if x < 100 goto L2
    goto L3
L3: if x > 200 goto L4
    goto L1
L4: if x != y goto L2
    goto L1
L2: x = 0
L1:
```