

Generation of Intermediate Code

Chapter 1, Section 1.2.4

Chapter 2, Section 2.8

Chapter 5, Section 5.1, 5.2, 5.3

Chapter 6, Section 6.1, 6.2, 6.4, 6.6

The Big Picture

- Useful machinery
 - Attribute grammars; syntax-directed definitions (SDDs)
- Abstract syntax trees (ASTs) and expression DAGs
 - Generation of ASTs and DAGs with the help of SDDs
- Three-address code
- Translation (to three-address code) of
 - Expressions
 - Flow-of-control statements
- **Project 3**: translate an AST to a string; **Projects 4 and 5**: translate an AST to three-address code

Attribute Grammars (1/2)

- For each terminal and non-terminal: zero, one, or more **attributes**
 - For each attribute: domain of possible values
- A **semantic rule** for each production

$$E \rightarrow E_1 + T$$
$$| T$$

$$E.val = E_1.val + T.val$$

$$E.val = T.val$$

$$T \rightarrow T_1 * F$$
$$| F$$

$$T.val = T_1.val * F.val$$

$$T.val = F.val$$

$$F \rightarrow (E)$$
$$| \text{const}$$

$$F.val = E.val$$

$$F.val = \text{const.lexval}$$

- Attribute *val* for each *E*, *T*, and *F* node
- Attribute *lexval* for each **const** code

Attribute Grammars (2/2)

- $X \rightarrow Y_1 Y_2 \dots Y_n$
 - The value of a **synthesized attribute for X** is computed from **$\text{Attr}(Y_1) \cup \dots \cup \text{Attr}(Y_n) \cup \text{Attr}(X)$**
 - The value of an **inherited attribute for Y_k** is computed from **$\text{Attr}(X) \cup \text{Attr}(Y_1) \cup \dots \cup \text{Attr}(Y_n)$**
- Terminals (leaf nodes) - only synthesized attributes
 - Not computed by semantic rules, but just provided by the lexical analyzer (e.g., *lexval* for each **const** code)
- Evaluation rules do not have side effects
- **Annotated** parse tree: parse tree for the context-free grammar; each node has (attribute, value) pairs; values conform to the semantic rules

Attribute Evaluation and SDDs

- Find evaluation order of attributes
 - Build dependency graph; complain about cycles in the graph; topologically sort the graph
 - Evaluate the attributes in sorted order (Section 5.2)
 - Rules do not have side effects; thus, all topological sort orders are equivalent
- **Syntax-directed definitions** (SDDs): certain side effects are permitted/desirable
 - E.g., print something; add something to a symbol table
 - Be careful with side effects and the evaluation order: make sure that all possible different orders of the side effects are equivalent

Abstract Syntax Trees (ASTs)

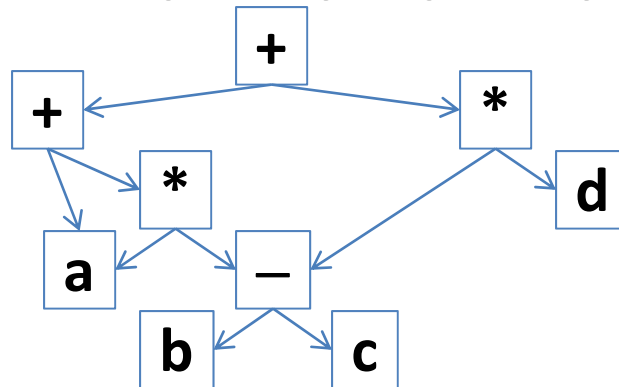
- The Dragon Book calls them just “syntax trees”
 - As opposed to “concrete syntax trees” = “parse trees”
 - Each node represents a language constructs
 - Children represent the sub-constructs
- Example: $E \rightarrow E + T$
 - Parse tree: node E with three children
 - AST: $+$ node with two children
 - Example: Parse tree and AST for $1 + a * (2 + b) * 3$
 - $E \rightarrow E + T \mid T$
 - $T \rightarrow T * F \mid F$
 - $F \rightarrow (E) \mid \text{const} \mid \text{id}$

AST Construction Specified with a SDD

- $E \rightarrow E_1 + T$ $E.node = newNode(+, E_1.node, T.node)$
- $E \rightarrow T$ $E.node = T.node$
- $T \rightarrow T_1 * F$ $T.node = newNode(*, T_1.node, F.node)$
- $T \rightarrow F$ $T.node = F.node$
- $F \rightarrow (E)$ $F.node = E.node$
- $F \rightarrow \mathbf{const}$ $F.node = newLeaf(\mathbf{const}, \mathbf{const.lexval})$
- $F \rightarrow \mathbf{id}$ $F.node = newLeaf(\mathbf{id}, \mathbf{id.symtbl_entry})$
- The parser has already entered all ids into symbol tables (one table per scope); *symtbl_entry* points to the entry in the appropriate symbol table
- Do not need to explicitly construct the underlying parse tree – AST construction can be done during parsing

Expression DAGs

- Directed acyclic graph: common sub-expressions are not replicated
 - Example: $a + a * (b - c) + (b - c) * d$



- Use a similar SDD as for ASTs – but reuse nodes
 - *newNode*(*op*, *left*, *right*) checks if there already exists a node with label *op*, and children *left* and *right*; returns this node if it already exists
 - *newLeaf* is modified in a similar way

Three-Address Code

- ASTs and expression DAGs are high-level IRs
 - Close to the source language
 - Suitable for tasks such as type checking
- Three-address code is a lower-level IR
 - Closer to the target language (i.e., assembly code)
 - Suitable for tasks such as code generation/optimization
- Basic ideas
 - A small number of simple instructions: e.g. $x = y \text{ op } z$
 - A number of **compiler-generated temporary variables**
 - $a = b + c + d$; in source code $\rightarrow t = b + c; a = t + d$;
 - Simple flow of control – conditional and unconditional jumps to labeled statements

Addresses and Instructions

- “Address”: a program variable, a constant, or a compiler-generated temporary variable
- Instructions
 - **x = y op z**: binary operator *op*; *y* and *z* are variables, temporaries, or constants; *x* is a variable or a temporary
 - **x = op y**: unary operator *op*; *y* is a variable, a temporary, or a constant; *x* is a variable or a temporary
 - **x = y**: copy instruction; *y* is a variable, a temporary, or a constant; *x* is a variable or a temporary
 - Arrays, flow-of-control (more later ...)
 - Each instruction contains at most three “addresses”
 - Thus, **three-address code**

Translation of Expressions

- A simple grammar for assignments and expressions
 - Ambiguous, but it does not matter – parsing is finished

$S \rightarrow \text{id} = E ;$

$E \rightarrow E_1 + E_2$

$E \rightarrow - E_1$

$E \rightarrow (E_1)$

$E \rightarrow \text{id}$

- Two attributes
 - Attribute *code* for S and E : sequence of three-address instructions
 - Attribute *addr* for E : the “address” (program variable or temporary) that will hold the value of E

SDD for Translation

$S \rightarrow \mathbf{id} = E ;$

$S.code = E.code \quad || \quad \text{gen}(\mathbf{id}.syntbl_entry \text{ "=" } E.addr)$

$E \rightarrow E_1 + E_2$

$E.addr = \text{newTemp}()$

$E.code = E_1.code \quad || \quad E_2.code \quad ||$

$\text{gen}(E.addr \text{ "=" } E_1.addr \text{ "+" } E_2.addr)$

$E \rightarrow - E_1$

$E.addr = \text{newTemp}()$

$E.code = E_1.code \quad || \quad \text{gen}(E.addr \text{ "=" } \text{"-"} E_1.addr)$

$E \rightarrow (E_1)$

$E.addr = E_1.addr \quad E.code = E_1.code$

$E \rightarrow \mathbf{id}$

$E.addr = \mathbf{id}.syntbl_entry \quad E.code = \text{" "}$

Simple Examples

$x = y$ produces one three-address instruction

Left: a pointer to the symbol table entry for x

Right: a pointer to the symbol table entry for y

For convenience, we will write this as $x = y$

$x = -y$ produces $t1 = -y; x = t1;$

$x = y + z$ produces $t1 = y + z; x = t1;$

$x = y + z + w$ produces $t1 = y + z; t2 = t1 + w; x = t2;$

$x = y + -z$ produces $t1 = -z; t2 = y + t1; x = t2;$

More Complex Expressions & Assignments

- All binary & unary operators are handled similarly
- We run into more interesting issues with
 - Expressions that have **side effects**
 - Arrays
- Example: $E \rightarrow \dots \mid E_1 = E_2 \mid E_1 ++ \mid \text{id} [E_1]$
 - In C, we can write $x = y = z + z$: maybe it should be translated to $t1 = z + z; y = t1; x = y; ?$
 - How should we translate $x = y = z++ + w$? How about $a[v = x++] = y = z++ + w$? Or $i = i++ + 1$? Or $a[i++] = i$?

Full Grammar for Project 4

$S \rightarrow E$; (we will consider only SgExprStatement)

$E \rightarrow \text{id} \mid \text{const}$

$E \rightarrow E_1 + E_2 \mid E_1 - E_2 \mid E_1 * E_2 \mid E_1 / E_2 \mid E_1 \ll E_2 \mid E_1 \gg E_2$

$E \rightarrow + E_1 \mid - E_1$

$E \rightarrow \text{id} [E_1]$ (only 1-dimensional arrays)

$E \rightarrow (E_1)$

$E \rightarrow E_1 = E_2 \mid E_1 += E_2 \mid E_1 -= E_2 \mid E_1 \ll= E_2 \mid E_1 \gg= E_2$

$E \rightarrow ++ E_1 \mid E_1 ++ \mid -- E_1 \mid E_1 --$

L-values of Expressions

- An expression E has an **l-value** if this expression can appear on the left-hand-side of an assignment
 - The type of an l-value is always “a chunk of memory”
 - E.g. x is a local **int** variable in **main**; after stmt $x=5$
 - the *value* (or *r-value*) of expr x is the **int** value 5
 - the *l-value* of expression x is the “chunk of memory” (2 bytes or 4 bytes) in which the variable resides
- L-values: only for $E \rightarrow \mathbf{id} \mid \mathbf{id}[E_1]$
 - Also for (E_1) , if E_1 has an l-value; let's ignore this case ...
- The parser (or semantic analyzer) guarantees that
 - the left operand of an assign. operator has an l-value
 - the operand of pre/post $++$ or $--$ has an l-value

Modified Full Grammar for Project 4

- $E \rightarrow ++ \text{id} \mid ++ \text{id}[E_1] \mid \text{id} ++ \mid \text{id}[E_1] ++ \mid \dots$
- Semantics of ++ (also --): postfix and prefix
 - **id ++**: (1) produce a value obtained by reading **id**; (2) immediately after that, increment the value of **id**
 - **id[E₁] ++**: (1) evaluate E_1 to get an index value; any side effects due to E_1 occur; (2) produce a value obtained by reading the array element; (3) immediately after that, increment the array element
 - **++ id**: completely equivalent to **(id += 1)**
 - **++ id[E]**: completely equivalent to **(id[E] += 1)**
- Note: this is not the C semantics!
 - Undefined order: (ANSI C doc, p. 67): **a[i++] = i**; is bad ...

Modified Full Grammar for Project 4

- $E \rightarrow \mathbf{id} = E_1 \mid \mathbf{id}[E_1] = E_2 \mid \mathbf{id} += E_2 \mid \mathbf{id}[E_1] += E_2 \mid \dots$
- Semantics of assignment operators
 - $\mathbf{id} = E_1$: produces the new value of \mathbf{id}
 - $\mathbf{id}[E_1] = E_2$: (1) evaluate E_1 to get an index value; any side effects due to E_1 occur; (2) evaluate E_2 to a value; any side effects due to E_2 occur; (3) modify the array element; (4) produce the new value of the element
 - Again, this is not the semantics of C
 - $\mathbf{id} += E_1$ is equivalent to $(\mathbf{id} = \mathbf{id} + E_1)$
 - $\mathbf{id}[E_1] += E_2$ is equivalent to $(\mathbf{id}[E_1] = \mathbf{id}[E_1] + E_2)$, except that the evaluation of E_1 happens only once (and thus all of its side effects occur once)

SDD for Translation

- $S \rightarrow E$
 - $S.code = E.code$
- $E \rightarrow E_1 + E_2$ (and similar binary operators)
 - $E.addr = newTemp()$ and $E.code = E_1.code \parallel E_2.code \parallel$
 $gen(E.addr "=" E_1.addr "+" E_2.addr)$ *But C semantics defines no order*
- $E \rightarrow + E_1$
 - $E.addr = E_1.addr$ and $E.code = E_1.code$
- $E \rightarrow - E_1$
 - $E.addr = newTemp()$ and $E.code = E_1.code \parallel$
 $gen(E.addr "=" "-" E_1.addr)$
- $E \rightarrow id$
 - $E.addr = newTemp()$ and $E.code = gen(E.addr "="$
 $id.symtbl_entry)$

SDD for Translation

- $E \rightarrow \mathbf{const}$
 - $E.addr = \mathbf{const.lexval}$ and $E.code = ""$
- $E \rightarrow \mathbf{(E_1)}$
 - $E.addr = E_1.addr$ and $E.code = E_1.code$
- $E \rightarrow \mathbf{id[E_1]}$
 - $E.addr = newTemp()$ and $E.code = E_1.code \mid \mid$
 $gen(E.addr \text{ "=" } \mathbf{id.symtbl_entry} \text{ "[" } E_1.addr \text{ "]"})$
 - Need $\mathbf{x = y[z]}$ instructions in the three-address code;
y is a variable; z is a variable, a temporary, or a constant; x is a variable or a temporary

SDD for Translation

- $E \rightarrow \mathbf{id} = E_1$
 - $E.addr = E_1.addr$
 - $E.code = E_1.code \ || \ \text{gen}(\mathbf{id.symtbl_entry} \ \mathbf{"="} \ E_1.addr)$
- $E \rightarrow \mathbf{id}[E_1] = E_2$
 - $E.addr = E_2.addr$
 - $E.code = E_1.code \ || \ E_2.code \ ||$
 $\text{gen}(\mathbf{id.symtbl_entry} \ \mathbf{"["} \ E_1.addr \ \mathbf{"]"} \ \mathbf{"="} \ E_2.addr)$
 - Need $\mathbf{x[y] = z}$ instructions; x is a variable; y and z are variables, temporaries, or constants

SDD for Translation

- $E \rightarrow \mathbf{id} += E_1$
 - Treat this exactly as $\mathbf{id} = \mathbf{id} + E_1$ (i.e., combination of the rules for $E \rightarrow E_1 + E_2$ and $E \rightarrow \mathbf{id} = E_1$)
- $E \rightarrow \mathbf{id}[E_1] += E_2$
 - $E.addr = newTemp()$
 - $E.code = E_1.code \ ||$
 $gen(E.addr "=" \mathbf{id}.symtbl_entry "[" E_1.addr "]") \ ||$
 $E_2.code \ || gen(E.addr "=" E.addr "+" E_2.addr) \ ||$
 $gen(\mathbf{id}.symtbl_entry "[" E_1.addr "]" "=" E.addr)$
- $E \rightarrow ++ \mathbf{id}$: special case of $\mathbf{id} += E_1$
- $E \rightarrow ++ \mathbf{id}[E_1]$: special case of $\mathbf{id}[E_1] += E_2$

SDD for Translation

- $E \rightarrow \mathbf{id ++}$
 - $E.addr = newTemp()$
 - $E.code = gen(E.addr "=" \mathbf{id.symtbl_entry}) \mid \mid$
 $gen(\mathbf{id.symtbl_entry} "=" \mathbf{id.symtbl_entry} "+" "1")$
- $E \rightarrow \mathbf{id}[E_1] ++$
 - $E.addr = newTemp()$
 - $E.code = E_1.code \mid \mid$
 $gen(E.addr "=" \mathbf{id.symtbl_entry} "[" E_1.addr "]") \mid \mid$
 $gen(E.addr "=" E.addr "+" "1") \mid \mid$
 $gen(\mathbf{id.symtbl_entry} "[" E_1.addr "]" "=" E.addr) \mid \mid$
 $gen(E.addr "=" E.addr "-" "1")$

Flow of Control - Expressions

- Boolean expressions
 - Role 1: conditions of ifs and loops
 - Role 2: assign to a boolean variable (let's ignore it ...)
- $B \rightarrow B_1 \ || \ B_2 \ | \ B_1 \ \&\& \ B_2 \ | \ ! \ B_1 \ | \ (B_1) \ | \ E_1 \ \text{rel} \ E_2 \ | \ \text{true} \ | \ \text{false}$
 - $\text{rel.op} \in \{ <, <=, ==, !=, >, >= \}$
 - $||$ and $\&\&$ are left-associative
 - $||$ has the lowest precedence, then $\&\&$, then $!$
- Short-circuit evaluation
 - $B_1 \ || \ B_2$ first evaluates B_1 ; if true, B_2 is not evaluated
 - $B_1 \ \&\& \ B_2$ first evaluates B_1 ; if false, B_2 is not evaluated

Flow of Control - Statements

$P \rightarrow S$ – this is the complete program

$S \rightarrow E ;$ – this we have already defined

$S \rightarrow \text{if } (B) S_1 \mid \text{if } (B) S_1 \text{ else } S_2$

$S \rightarrow \text{while } (B) S_1 \mid \text{do } S_1 \text{ while } (B) \mid \text{for } (E_1; B ; E_2) S_1$

$S \rightarrow S_1 S_2$ – to be able to construct sequences

Example: **if (x < 100 || x > 200 && x != y) x = 0;**

if (x < 100) goto L2;

if (!(x > 200)) goto L1;

if (!(x != y)) goto L1;

L2: x = 0;

L1: ...

Note: for simplicity of presentation, all examples in the rest of the slides assume $E.addr = id.symtbl_entry$ for production $E \rightarrow id$. In reality, there will be additional temporary variables due to $E.addr = newTemp()$

Three-Address Instructions

- New instructions
 - **goto L**: unconditional jump to the three-address instruction with label L
 - **if (x relop y) goto L**: x and y are variables, temporaries, or constants; relop $\in \{ <, <=, ==, !=, >, >= \}$
 - **if (!(x relop y)) goto L**: x and y are variables, temporaries, or constants
 - Strictly speaking, these are redundant ...
- The labels are symbolic names
 - We will just generate label names L1, L2, ... using a helper function *newLabel()*, in the same way we generate temporaries with names t1, t2, ... using a helper function *newTemp()*

SDD for Translation

- $P \rightarrow S$
 - $S.next = newLabel()$
 - $P.code = S.code \ || \ label(S.next) \ || \ gen("noop")$
 - Inside the code for S , we will have jump instructions to label $S.next$ (provided as an inherited attribute)
- $S \rightarrow E ;$
 - $S.code = E.code$
 - Example: For a program $x = y + z + w;$ the result is

```
t1 = y + z;  
t2 = t1 + w;  
x = t2;  
L1: noop;
```

SDD for Translation

- $S \rightarrow \text{if } (B) S_1$
 - $B.true = \text{newLabel}()$
 - $B.false = S.next$
 - $S_1.next = S.next$
 - $S.code = B.code \parallel \text{label}(B.true) \parallel S_1.code$
 - Example: For a program **if (a < b) x = y + z + w;**

```
if (a < b) goto L2;  
goto L1; B.code
```

```
B.true L2: t1 = y + z;  
t2 = t1 + w;  
x = t2;
```

```
B.false L1: noop;
```

SDD for Translation

- $S \rightarrow \text{if } (B) S_1 \text{ else } S_2$
 - $B.true = newLabel()$ and $B.false = newLabel()$
 - $S_1.next = S.next$ and $S_2.next = S.next$
 - $S.code = B.code \ || \ label(B.true) \ || \ S_1.code \ || \ gen(\text{"goto"} \ S.next) \ || \ label(B.false) \ || \ S_2.code$
 - Example: **if (x < 0) y = 1; else y=2;**
if (x < 0) goto L2;
goto L3;
 $B.true$ **L2: y = 1;**
goto L1;
 $B.false$ **L3: y = 2;**
L1: noop;

SDD for Translation

- $S \rightarrow \text{while } (B) S_1$
 - $begin = newLabel()$
 - $B.true = newLabel()$
 - $B.false = S.next$
 - $S_1.next = begin$
 - $S.code = \text{label}(begin) \ || \ B.code \ || \ \text{label}(B.true) \ || \ S_1.code \ || \ \text{gen}(\text{"goto" } begin)$
 - Example: **while (x < 0) y = 1;**

begin **L2: if (x < 0) goto L3;**

goto L1;

B.true **L3: y = 1;**

goto L2;

L1: noop;

SDD for Translation

- $S \rightarrow S_1 S_2$
 - $S_1.next = newLabel()$
 - $S_2.next = S.next$
 - $S.code = S_1.code \ || \ label(S_1.next) \ || \ S_2.code$
 - Example: **if (x < 0) y = 1; if (z < 2) w = 3;**
 - if (x < 0) goto L3;**
 - goto L2;**
 - L3: y = 1;**
 - $S_1.next$ **L2: if (z < 2) goto L4;**
 - goto L1;**
 - L4: w = 3;**
 - L1: noop;**

SDD for Translation

- $B \rightarrow E_1 \text{ rel } E_2$
 - $B.code = E_1.code \ || \ E_2.code \ || \ \text{gen}(\text{"if" } E_1.addr \ \text{rel.op} \ E_2.addr \ \text{"goto" } B.true) \ || \ \text{gen}(\text{"goto" } B.false)$
 - Example: **if (x+1 < 0) y = 1;** – $B.false$ is L1, $B.true$ is L2
t1 = x+1;
if (t1 < 0) goto L2;
goto L1;
L2: y = 1;
L1: noop;
- $B \rightarrow \text{true}$ or $B \rightarrow \text{false}$
 - $B.code = \text{gen}(\text{"goto" } B.true)$ or $\text{gen}(\text{"goto" } B.false)$

SDD for Translation

- $B \rightarrow B_1 \ || \ B_2$
 - $B_1.true = B.true$ and $B_1.false = newLabel()$
 - $B_2.true = B.true$ and $B_2.false = B.false$
 - $B.code = B_1.code \ || \ label(B_1.false) \ || \ B_2.code$
 - Example: `if (x<0 || y<1) z=2;` – $B.false$ is L1, $B.true$ is L2

```
if (x < 0) goto L2;  
goto L3;
```

$B_1.code$

```
L3: if (y < 1) goto L2;  
goto L1;
```

$B_2.code$

```
L2: z = 2;
```

```
L1: noop;
```

SDD for Translation

- $B \rightarrow B_1 \ \&\& \ B_2$
 - $B_1.true = newLabel()$ and $B_1.false = B.false$
 - $B_2.true = B.true$ and $B_2.false = B.false$
 - $B.code = B_1.code \ || \ label(B_1.true) \ || \ B_2.code$
 - Example: `if (x<0 && y<1) z=2;` – $B.false$ is L1, $B.true$ is L2

```
if (x < 0) goto L3;  
goto L1;
```

$B_1.code$

```
L3: if (y < 1) goto L2;  
goto L1;
```

$B_2.code$

```
L2: z = 2;
```

```
L1: noop;
```

SDD for Translation

- $B \rightarrow ! B_1$
 - $B_1.true = B.false$ and $B_1.false = B.true$
 - $B.code = B_1.code$
 - Example: **if (!(x<0 && y<1)) z=2;**
 - $B.false = B_1.true = L1$, $B.true = B_1.false = L2$

```
if (x < 0) goto L3;  
goto L2;  
L3: if (y < 1) goto L1;  
goto L2;
```

$B_1.code$

```
L2: z = 2;
```

```
L1: noop;
```

Potential Improvements

- Redundant **gotos**

- Example: **if (x < 100 || x > 200 && x != y) x = 0;**

- if (x < 100) goto L2;**

- if (x < 100) goto L2;**

- goto L3;**

- if (!(x < 200)) goto L1;**

- L3: if (x > 200) goto L4;**

- if (!(x != y)) goto L1;**

- goto L1;**

- L2: x = 0;**

- L4: if (x != y) goto L2;**

- L1: noop;**

- goto L1;**

- L2: x = 0;**

- L1: noop;**

- Possible optimization (Section 6.6.5)

- Use an artificial label “fall” – meaning “don’t create a jump; instead, just fall through”