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
Outline

• Useful machinery
  – Syntax-directed definitions (SDDs)

• Program representations
  – Abstract syntax trees (ASTs)
  – Expression DAGs
  – 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
Syntax Directed Definitions (SDDs)

• For each terminal and non-terminal: zero, one, or more attributes
  – For each attribute: set of possible values
  – Very similar to attribute grammars (CSE 6341)

• A semantic rule for each production

\[
\begin{align*}
E & \rightarrow E_1 + T & E.val = E_1.val + T.val \\
    & \mid T & E.val = T.val \\
T & \rightarrow T_1 * F & T.val = T_1.val * F.val \\
    & \mid F & T.val = F.val \\
F & \rightarrow (E) & F.val = E.val \\
    & \mid \text{const} & F.val = \text{const}.lexval
\end{align*}
\]

– Attribute \( val \) for each \( E, T, \) and \( F \) node
– Attribute \( lexval \) for each \( \text{const} \) code
Syntax Directed Definitions

• An attribute of a non-terminal X can be either **synthesized** or **inherited** (but not both)
  – **Synthesized attribute X.a**: computed from attributes of X’s children (this is an oversimplification)
  – **Inherited attribute X.a**: computed from attributes of X’s parent (this is an oversimplification)

• Attributes for terminals (leaf nodes)
  – Not computed by semantic rules, but just provided by the lexical analyzer (e.g., `lexval` for each `const` code)

• Certain **side effects** are permitted in SSDs
  – Unlike with attribute grammars
  – E.g., print something; add something to a symbol table
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 construct
  – 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

- $E \rightarrow E_1 + T \quad E.\text{node} = \text{newNode}(+, E_1.\text{node}, T.\text{node})$
- $E \rightarrow T \quad E.\text{node} = T.\text{node}$
- $T \rightarrow T_1 * F \quad T.\text{node} = \text{newNode}(*, T_1.\text{node}, F.\text{node})$
- $T \rightarrow F \quad T.\text{node} = F.\text{node}$
- $F \rightarrow (E) \quad F.\text{node} = E.\text{node}$
- $F \rightarrow \text{const} \quad F.\text{node} = \text{newLeaf(\text{const}, \text{const.\lexval})}$
- $F \rightarrow \text{id} \quad F.\text{node} = \text{newLeaf(\text{id}, \text{id.symtbl_entry})}$

The parser has already entered all ids into the symbol table; \text{id.symtbl_entry} points to the entry in the table

AST construction can be done during parsing (no parse tree built) or after it (first build parse tree, then AST)
Expression DAGs

• Directed acyclic graph: common sub-expressions are not replicated
  – Example: \( a + a \ast (b - c) + (b - c) \ast d \)

• 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
Another Representation: Three-Address Code

- AST is a high-level IR
  - 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 (no while-do, switch, ...)
Addresses and Instructions

• “Address”: a program variable, a constant, or a compiler-generated temporary variable

• Instructions
  – \( x = y \text{ op } z \): binary operator \( \text{op} \); \( y \) and \( z \) are variables, temporaries, or constants; \( x \) is a variable or a temporary
  – \( x = \text{op } y \): unary operator \( \text{op} \); \( y \) 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
  – More later: arrays, flow-of-control

• Each instruction contains at most three “addresses”
  • Thus, three-address code
Translation of Expressions: Toy Example

• A simple grammar for assignments and expressions
  – Ambiguous, but it doesn’t 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
  – Synthesized attribute \textit{code} for \( S \) and \( E \): sequence of three-address instructions
  – Synthesized attribute \textit{addr} for \( E \): the “address” (program variable or temp) that will hold the value of \( E \)
Toy Example: SDD for Translation

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

$S.code = E.code \ || \ \text{id}.\text{symtbl}\_entry \ "=" E.addr$  \ || \text{is concatenation}

$E \rightarrow E_1 + E_2$

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

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

$E.addr "=" E_1.addr "+" E_2.addr$

$E \rightarrow -E_1$

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

$E.code = E_1.code \ || E.addr "=" "-" E_1.addr$

$E \rightarrow (E_1)$

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

$E \rightarrow \text{id}$

$E.addr = \text{id}.\text{symtbl}\_entry \quad E.code = " "$
Examples of Code Generation

\texttt{x = y;} produces one three-address instruction

Left: a pointer to the symbol table entry for \texttt{x}

Right: a pointer to the symbol table entry for \texttt{y}

For convenience, we will write this as \texttt{x = y}

\texttt{x = - y;} produces \texttt{t1 = - y; x = t1;}

\texttt{x = y + z;} produces \texttt{t1 = y + z; x = t1;}

\texttt{x = y + z + w;} produces \texttt{t1 = y + z; t2 = t1 + w; x = t2;}

\texttt{x = y + - z;} produces \texttt{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: $E \rightarrow \ldots \mid E_1 = E_2 \mid E_1 ++ \mid \text{id}[E_1]$
  – 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)

\[ E \rightarrow \text{id} \mid \text{const} \]

\[ E \rightarrow E_1 + E_2 \mid E_1 - E_2 \mid \ldots \] (all binary arithmetic/bitwise)

\[ E \rightarrow + E_1 \mid - E_1 \mid \ldots \] (all unary arithmetic/bitwise)

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

\[ E \rightarrow (E_1) \]

\[ E \rightarrow E_1 = E_2 \mid E_1 += E_2 \mid E_1 -= E_2 \mid \ldots \] (all assignments)

\[ E \rightarrow ++ E_1 \mid E_1 ++ \mid -- E_1 \mid E_1 -- \]
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 $x=5$ we have
    • the value (or r-value) of expr $x$ is the int value 5
    • the l-value of expression $x$ is the “chunk of memory” (typically, 4 bytes) in which the variable resides

• L-values: only for $E \rightarrow \text{id} \mid \text{id}[E_1]$
  – 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

- \( E \rightarrow ++ \text{id} \mid ++ \text{id}[E_1] \mid \text{id}++ \mid \text{id}[E_1]++ \mid -- \text{id} \mid \ldots \)

- Semantics of ++ (also --): postfix and prefix
  - \( \text{id}++ \): (1) produce a value obtained by reading \( \text{id} \); (2) immediately after that, increment the value of \( \text{id} \)
  - \( \text{id}[E_1]++ \): (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
  - \( ++\text{id} \): completely equivalent to \((\text{id} += 1)\) – next slide
  - \( ++\text{id}[E] \): completely equivalent to \((\text{id}[E] += 1)\)

- Note: this is not the semantics of C
  - Undefined order: (ANSI C doc, p. 67): \( a[i++] = i \); is bad ...
Modified Full Grammar for Project 4

- $E \rightarrow \text{id} = E_1 \mid \text{id}[E_1] = E_2 \mid \text{id} += E_2 \mid \text{id}[E_1] += E_2 \mid \ldots$

- Semantics of assignment operators
  - $\text{id} = E_1$: produces the new value of $\text{id}$
  - $\text{id}[E_1] = E_2$: (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
  - $\text{id} += E_1$ is equivalent to $(\text{id} = \text{id} + E_1)$
  - $\text{id}[E_1] += E_2$ is equivalent to $(\text{id}[E_1] = \text{id}[E_1] + E_2)$, 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)
  - $E.addr = \text{newTemp}()$ and $E.code = E_1.code \text{ || } E_2.code \text{ || } E.addr "=" E_1.addr "+" E_2.addr$  
    But C semantics defines no order

- $E \rightarrow + E_1$
  - $E.addr = E_1.addr$ and $E.code = E_1.code$

- $E \rightarrow - E_1$
  - $E.addr = \text{newTemp}()$ and $E.code = E_1.code \text{ || } E.addr "=" "-" E_1.addr$

- $E \rightarrow \text{id}$
  - $E.addr = \text{newTemp}()$ and $E.code = E.addr "=" \text{id.symtbll_entry}$
SDD for Translation

• $E \rightarrow const$
  – $E.addr = const.lexval$ and $E.code = " "$

• $E \rightarrow (E_1)$
  – $E.addr = E_1.addr$ and $E.code = E_1.code$

• $E \rightarrow id[E_1]$
  – $E.addr = newTemp()$ and $E.code = E_1.code$ || $E.addr = id.symtbl_entry [" E_1.addr "]$

  • Here we use $x = y[z]$ instructions in the three-address code
    – $y$ is an array variable
    – $z$ is a variable, a temporary, or a constant
    – $x$ is a variable or a temporary
SDD for Translation

- $E \rightarrow id = E_1$
  - $E.addr = E_1.addr$
  - $E.code = E_1.code \text{ || } id.symtbl_entry "=" E_1.addr$

- $E \rightarrow id[E_1] = E_2$
  - $E.addr = E_2.addr$
  - $E.code = E_1.code \text{ || } E_2.code \text{ || } id.symtbl_entry "[" E_1.addr "]" "=" E_2.addr$
  - Here we use $x[y] = z$ instructions
    - $x$ is an array variable
    - $y$ and $z$ are variables, temporaries, or constants
SDD for Translation

- \( E \rightarrow \text{id} += E_1 \)
  - Treat this exactly as \( \text{id} = \text{id} + E_1 \) (i.e., combination of the rules for \( E \rightarrow E_1 + E_2 \) and \( E \rightarrow \text{id} = E_1 \))

- \( E \rightarrow \text{id}[E_1] += E_2 \)
  - \( E.\text{addr} = \text{newTemp()} \)
  - \( E.\text{code} = E_1.\text{code} \mid | \mid E.\text{addr} "=" \text{id}.\text{symtbl}._\text{entry} "[" E_1.\text{addr }"]" \mid | \mid E_2.\text{code} \mid | \mid E.\text{addr} "=" E.\text{addr }"+" E_2.\text{addr} \mid | \mid \text{id}.\text{symtbl}._\text{entry} "[" E_1.\text{addr }"]" "=" E.\text{addr} \)

- \( E \rightarrow ++ \text{id} \): special case of \( \text{id} += E_1 \)

- \( E \rightarrow ++ \text{id}[E_1] \): special case of \( \text{id}[E_1] += E_2 \)
SDD for Translation

• $E \rightarrow \text{id} \; ++$
  
  $- \; E.\text{addr} = \text{newTemp}()$
  
  $- \; E.\text{code} = E.\text{addr} \; "=\; \text{id}.\text{symtbl\_entry}\) \; ||$
  
  $\text{id}.\text{symtbl\_entry} \; "=\; \text{id}.\text{symtbl\_entry} \; +\; "1"$

• $E \rightarrow \text{id}[E_1] \; ++$
  
  $- \; E.\text{addr} = \text{newTemp}()$
  
  $- \; E.\text{code} = E_1.\text{code} \; ||$
  
  $E.\text{addr} \; "=\; \text{id}.\text{symtbl\_entry} \; [\; "E_1.\text{addr} \; \]" \; ||$
  
  $E.\text{addr} \; "=\; E.\text{addr} \; +\; "1" \; ||$
  
  $\text{id}.\text{symtbl\_entry} \; [\; "E_1.\text{addr} \; \]" \; =\; E.\text{addr}\) \; ||$
  
  $E.\text{addr} \; "=\; E.\text{addr} \; -\; "1"$
A Few Examples to Try at Home

• \( x = y+z; \)
• \( w = x = y+z; \)
• \( a[x=y+z] = x; \)
• \( a[x] = x = y+z; \)
• \( x += y+z; \)
• \( x += x = y+z; \)
• \( x = ++y; \)
• \( x = y++; \)
Flow of Control - Expressions

• Boolean expressions
  – Role 1: conditions of ifs and loops
  – Role 2: assign to a boolean variable (let’s ignore it ...)

• $B \rightarrow B_1 \text{ || } B_2 \text{ || } B_1 \&\& B_2 \text{ || } ! B_1 \text{ || } (B_1) \text{ || } E_1 \text{ rel } E_2 \text{ || } \text{true} \text{ || } \text{false}$
  – $\text{rel.op} \in \{ <, <=, ==, !=, >, >= \}$
  – || and && are left-associative
  – || has the lowest precedence, then &&, then !

• Short-circuit evaluation
  – $B_1 \text{ || } B_2$ 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:
\[
\text{if } (x < 100 \mid \mid x > 200 \&\& x \neq y) \ x = 0; \\
\text{if } (x < 100) \ \text{goto} \ L2; \\
\text{if } (!(x > 200)) \ \text{goto} \ L1; \\
\text{if } (!(x \neq y)) \ \text{goto} \ L1; \\
L2: \ x = 0; \\
L1: \ ...
\]

Note: for simplicity of presentation, all examples in the rest of the slides assume \( E.\text{addr} = \text{id.symtblEntry} \) for production \( E \rightarrow \text{id} \). In reality, there will be additional temporary variables due to \( E.\text{addr} = \text{newTemp}() \).
Three-Address Instructions

• New instructions
  – \textbf{goto L}: unconditional jump to the three-address instruction with label L
  – \textbf{if (x relop y) goto L}: x and y are variables, temporaries, or constants; \textit{relop} \in \{ <, \leq, =, \neq, >, \geq \}

• The labels are symbolic names
  – We will just generate label names L1, L2, ... using a helper function \texttt{newLabel()}, in the same way we generate temporaries with names t1, t2, ... using a helper function \texttt{newTemp()}
SDD for Translation

• \( P \rightarrow S \)
  – \( S.next = newLabel() \)
  – \( P.code = S.code \) || label\((S.next)\) || "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
    
    \[
    \begin{align*}
    t1 &= y + z; \\
    t2 &= t1 + w; \\
    x &= t2; \\
    \end{align*}
    
    L1: noop;
SDD for Translation

- **$S \rightarrow \text{if} \ (B) \ S_1$**
  - $B.true = \text{newLabel()}$
  - $B.false = S.next$
  - $S_1.next = S.next$
  - $S.code = B.code \ | \ | \ \text{label}(B.true) \ | \ | \ S_1.code$
  - Example: For a program `if (a < b) x = y + z + w;`
    ```
    \[
    \begin{align*}
    \text{if} \ (a < b) & \text{ goto } L2; \\
    \text{goto } L1; \\
    \end{align*}
    \]
    ```
    ```
    B.true
    \[
    \begin{align*}
    L2: & \text{ t1 = y + z; } \\
    \text{t2 = t1 + w; } \\
    \text{x = t2; } \\
    \end{align*}
    \]
    ```
    ```
    B.false
    \[
    \begin{align*}
    L1: & \text{ noop; } \\
    \end{align*}
    \]
    ```
SDD for Translation

• $S \rightarrow \textbf{if} \ (B) \ S_1 \ \textbf{else} \ S_2$
  
  – $B.\text{true} = \text{newLabel}()$ and $B.\text{false} = \text{newLabel}()$
  
  – $S_1.\text{next} = S.\text{next}$ and $S_2.\text{next} = S.\text{next}$
  
  – $S.\text{code} = B.\text{code} \ || \ \text{label}(B.\text{true}) \ || \ S_1.\text{code} \ || \ "\text{goto}" \ S.\text{next} \ || \ \text{label}(B.\text{false}) \ || \ S_2.\text{code}$
  
  – Example: \textbf{if} (x < 0) \ y = 1; \ \textbf{else} \ y=2;
    \textbf{if} (x < 0) \ \text{goto L2};
    \text{goto L3};
    \textit{B.true} \textbf{L2:} \ y = 1;
    \text{goto L1};
    \textit{B.false} \textbf{L3:} \ y = 2;
    \textbf{L1:} \ \text{noop};
SDD for Translation

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

\textbf{begin} L2: if (x < 0) goto L3;

go to L1;

$B\.true$

L3: $y = 1$;

go to L2;

L1: \text{noop};
SDD for Translation

• $S \rightarrow S_1 S_2$
  – $S_1.next = newLabel()$
  – $S_2.next = S.next$
  – $S.code = S_1.code \mid \mid \text{label}(S_1.next) \mid \mid S_2.code$
  – Example: if (x < 0) y = 1; if (z < 2) w = 3;
    if (x < 0) goto L3;
    goto L2;
    L3: y = 1;

$S_1.next \quad L2: \text{if (z < 2) goto L4;}
    \quad \text{goto L1;}
    \quad L4: w = 3;
    \quad L1: \text{noop;}$
SDD for Translation

- $B \rightarrow E_1 \text{ rel } E_2$
  - $B.code = E_1.code | | E_2.code | |
    "if" $E_1.addr \text{ rel.op } E_2.addr "goto" B.true | |
    "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 $B \rightarrow \text{false}$
  - $B.code = "goto" B.true$ or $"goto" B.false$
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

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

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

L2: z = 2;
L1: noop;
```
SDD for Translation

• \( B \rightarrow B_1 \land \land B_2 \)
  
  – \( B_1\text{.true} = \text{newLabel()} \) and \( B_1\text{.false} = B\text{.false} \)
  
  – \( B_2\text{.true} = B\text{.true} \) and \( B_2\text{.false} = B\text{.false} \)
  
  – \( B\text{.code} = B_1\text{.code} \lor \lor \text{label}(B_1\text{.true}) \lor \lor B_2\text{.code} \)
  
  – Example: \( \text{if (x<0 && y<1) z=2;} \) – \( B\text{.false} \) is L1, \( B\text{.true} \) is L2

\[
\begin{align*}
\text{if (x < 0) goto L3;} & \quad B_1\text{.code} \\
\text{goto L1;} & \\
\text{L3: if (y < 1) goto L2;} & \quad B_2\text{.code} \\
\text{goto L1;} & \\
\text{L2: z = 2;} & \\
\text{L1: noop;} &
\end{align*}
\]
SDD for Translation

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

```plaintext
if (x < 0) goto L3;
goto L2;
L3: if (y < 1) goto L1;
goto L2;
L2: z = 2;
L1: noop;
```

$B_1.\text{code}$
Potential Improvements

• Redundant *gotos*
  – Example: `if (x < 100 || x > 200 && x != y) x = 0;
    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: noop;

• Possible optimization (Section 6.6.5)
  – Use an artificial label “fall” – meaning “don’t create a jump; instead, just fall through”