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

\[
E \rightarrow E_1 + T
| \ T
\]

\[
T \rightarrow T_1 * F
| \ F
\]

\[
F \rightarrow ( \ E )
| \ const
\]

- Attribute \textit{val} for each \(E\), \(T\), and \(F\) node
- Attribute \textit{lexval} for each \texttt{const} code
Attribute Grammars (2/2)

• $X \rightarrow Y_1 Y_2 \ldots Y_n$
  – The value of a synthesized attribute for $X$ is computed from $\text{Attr}(Y_1) \cup \ldots \cup \text{Attr}(Y_n) \cup \text{Attr}(X)$
  – The value of an inherited attribute for $Y_k$ is computed from $\text{Attr}(X) \cup \text{Attr}(Y_1) \cup \ldots \cup \text{Attr}(Y_n)$

• Terminals (leaf nodes) - only synthesized attributes
  – Not computed by semantic rules, but just provided by the lexical analyzer (e.g., $\text{lexval}$ for each $\text{const}$ 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

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

- The parser has already entered all ids into symbol tables (one table per scope); \text{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: \( a + a \times (b - c) + (b - c) \times d \)

• Use a similar SDD as for ASTs – but reuse nodes
  – \textit{newNode}(op, left, right) checks if there already exists a node with label \textit{op}, and children \textit{left} and \textit{right}; returns this node if it already exists
  – \textit{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. \( x = y \ op \ z \)
  – 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
  – \( x = y \text{ op } z \): 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
  – 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 \textit{code} for \( S \) and \( E \): sequence of three-address instructions
  – Attribute \textit{addr} for \( E \): the “address” (program variable or temporary) that will hold the value of \( E \)
S → id = E ;
    S.code = E.code || gen(id.symtbl_entry "=" E.addr)

E → E₁ + E₂
    E.addr = newTemp()
    E.code = E₁.code || E₂.code ||
                 gen(E.addr "=" E₁.addr "+" E₂.addr)

E → - E₁
    E.addr = newTemp()
    E.code = E₁.code || gen(E.addr "=" "-" E₁.addr)

E → ( E₁ )
    E.addr = E₁.addr   E.code = E₁.code

E → id
    E.addr = id.symtbl_entry   E.code = " "
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 $t_1 = -y; x = t_1$;

$x = y + z$ produces $t_1 = y + z; x = t_1$;

$x = y + z + w$ produces $t_1 = y + z; t_2 = t_1 + w; x = t_2$;

$x = y + -z$ produces $t_1 = -z; t_2 = y + t_1; x = t_2$;
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

\[ S \rightarrow E \; ; \; \text{(we will consider only SgExprStatement)} \]

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

\[ E \rightarrow E_1 + E_2 \mid E_1 - E_2 \mid E_1 \times E_2 \mid E_1 / E_2 \mid E_1 \ll E_2 \mid E_1 \gg E_2 \]

\[ E \rightarrow + E_1 \mid - E_1 \]

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

\[ E \rightarrow (E_1) \]

\[ E \rightarrow E_1 = E_2 \mid E_1 += E_2 \mid E_1 -= E_2 \mid E_1 \ll= E_2 \mid E_1 \gg= E_2 \]

\[ E \rightarrow ++ E_1 \mid E_1 ++ \mid -- E_1 \mid E_1 -- \]
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” (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

- \[ E \rightarrow ++ \text{id} \mid ++ \text{id}[E_1] \mid \text{id}++ \mid \text{id}[E_1]++ \mid ... \]

- 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 C semantics!
  - 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)
• $S \rightarrow E$
  – $S.code = E.code$

• $E \rightarrow E_1 + E_2$ (and similar binary operators)
  – $E.addr = \text{newTemp}()$ and $E.code = E_1.code \ || \ || E_2.code \ || \ \text{gen}(E.addr "=" E_1.addr "+" E_2.addr)$
    \text{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 $E.code = E_1.code \ || \ \text{gen}(E.addr "=" "-" E_1.addr)$

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

- $E \rightarrow \text{const}$
  - $E.\text{addr} = \text{const}.\text{lexval}$ and $E.\text{code} = " "$

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

- $E \rightarrow \text{id}[E_1]$
  - $E.\text{addr} = \text{newTemp}()$ and $E.\text{code} = E_1.\text{code}$  
    
    gen($E.\text{addr} \ "=\ " \text{id}.\text{symtbl\_entry} \ [" E_1.\text{addr }"]$)

- Need $x = y[z]$ 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

• $E \rightarrow \text{id} = E_1$
  - $E.\text{addr} = E_1.\text{addr}$
  - $E.\text{code} = E_1.\text{code} \ '|' \ gen(\text{id}.\text{symtbl_entry} "=" E_1.\text{addr})$

• $E \rightarrow \text{id}[E_1] = E_2$
  - $E.\text{addr} = E_2.\text{addr}$
  - $E.\text{code} = E_1.\text{code} \ '|' \ E_2.\text{code} \ '|'$
    gen(\text{id}.\text{symtbl_entry} [" E_1.\text{addr }"] "=" E_2.\text{addr})$
  • Need $x[y] = z$ instructions; $x$ is a 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 $E \rightarrow \text{id} = E_{1}$)

- $E \rightarrow \text{id}[E_{1}] += E_{2}$
  - $E.\text{addr} = \text{newTemp}()$
  - $E.\text{code} = E_{1}.\text{code} ||$
    - gen($E.\text{addr} \"=\" \text{id}.\text{symtbl\_entry }\"[\" E_{1}.\text{addr }\"]\") ||$ $E_{2}.\text{code} ||$ gen($E.\text{addr} \"=\" E.\text{addr }\"+\" E_{2}.\text{addr }\") ||$ gen($\text{id}.\text{symtbl\_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

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

- $E \rightarrow \text{id}[E_1] \text{ ++}$
  - $E.\text{addr} = \text{newTemp}()$
  - $E.\text{code} = E_1.\text{code} | |$
    
    \text{gen}(E.\text{addr} \ "=\" \text{id}.\text{symtbl\_entry} \ "[\" E_1.\text{addr}\"]\") | |$
    
    \text{gen}(E.\text{addr} \ "=\" E.\text{addr} \ "\text{+}\" \ "1") | |$
    
    \text{gen}(\text{id}.\text{symtbl\_entry} \ "[\" E_1.\text{addr}\"]\" \ "=\" E.\text{addr}) | |$
    
    \text{gen}(E.\text{addr} \ "=\" E.\text{addr} \ "\text{-}\" \ "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 ...)

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

• Short-circuit evaluation
  – $B_1 \lor B_2$ first evaluates $B_1$; if true, $B_2$ is not evaluated
  – $B_1 \land B_2$ first evaluates $B_1$; if false, $B_2$ is not evaluated
Flow of Control - Statements

\[ P \rightarrow S \rightarrow E ; \rightarrow if (B) S_1 | if (B) S_1 else S_2 \]

\[ S \rightarrow while (B) S_1 | do S_1 while (B) | for (E_1; B ; E_2) S_1 \]

\[ S \rightarrow S_1 S_2 \rightarrow to be able to construct sequences \]

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 ∈ { <, <=, ==, !=, >, >= }`

• 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

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

    - \( B.true \)
      
      \[
      L2: \ t1 = y + z; \\
      t2 = t1 + w; \\
      x = t2;
      \]
    
    - \( B.false \)
      
      \[
      L1: \text{noop};
      \]
SDD for Translation

• $S \rightarrow \textbf{if } (B) \ S_1 \ 	extbf{else } S_2$
  
  – $B.true = \text{newLabel}()$ and $B.false = \text{newLabel}()$
  
  – $S_1.next = S.next$ and $S_2.next = S.next$
  
  – $S.code = B.code \mid \mid \text{label}(B.true) \mid \mid S_1.code \mid \mid \text{gen}("goto" \ S.next) \mid \mid \text{label}(B.false) \mid \mid S_2.code$
  
  – Example: \textbf{if } (x < 0) \ y = 1; \ 	extbf{else } y=2;

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

  \text{goto} \ L3;

  B.true \ L2: \ y = 1;

  \text{goto} \ L1;

  B.false \ L3: \ y = 2;

  L1: \ \text{noop};
SDD for Translation

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

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

B.true
L3: y = 1;
goto L2;
L1: noop;
```
SDD for Translation

- $S \rightarrow S_1 \; S_2$
  - $S_1.next = \text{newLabel}()$
  - $S_2.next = S.next$
  - $S.code = S_1.code \; \lor \; \text{label}(S_1.next) \; \lor \; S_2.code$
  - 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

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

• \( B \rightarrow \text{true} \) or \( B \rightarrow \text{false} \)
  
  – \( B.code = \text{gen("goto" } B.true) \) or \( \text{gen("goto" } B.false) \)
SDD for Translation

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

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

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

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

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

```
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;
L2: z = 2;
L1: noop;
```
Potential Improvements

• Redundant **gotos**
  – Example: \( \text{if } (x < 100 \text{ || } x > 200 \text{ && } x != y) \ x = 0; \)
  
  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”
  – Need new type of instruction: \( \text{if } (! (x \text{ relop } y)) \text{ goto L} \)