# Principles of Compiler Construction

**Prof. Wen-jun LI**
School of Computer Science and Engineering
lnslwj@mail.sysu.edu.cn

# Lecture 9. Semantic Analysis and Intermediate Code Generation

1. Introduction
2. Types and Declarations
3. Assignments and Expressions
4. Type Checking
5. Boolean Expressions
6. Backpatching and Flow-of-Control Statements

# 1. Introduction

- Review
  - Front end vs. back end
    - $m \times n$: $m$ front ends and $n$ back ends.
  - Interface between front ends and back ends
    - Intermediate representation
    - Why IR? Extendability and optimization.
  - Semantic (static) analysis
    - The most common analysis
      - Type checking
    - Other static checking
      - Unreachable code
      - Use of uninitialized variables
      - etc.

# Static Checking

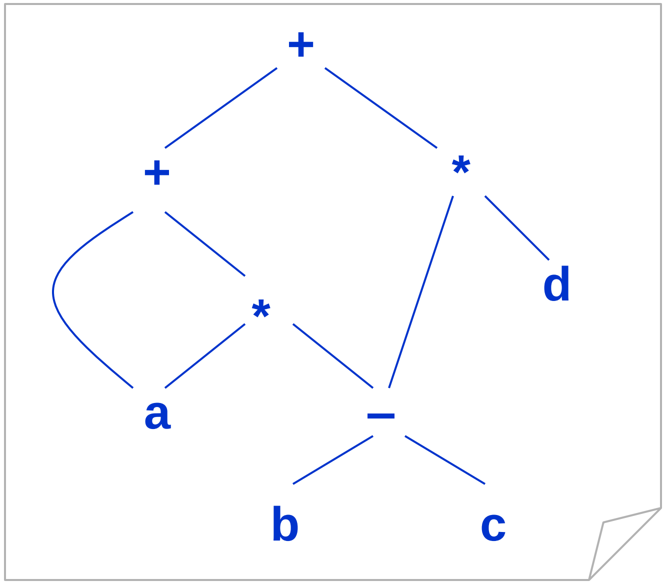
- Semantic analysis also focuses on the well-formness of source code
  - Due to the expressiveness power of Context-Free Grammars.
  - For example,
    - Number matching of actual parameters.
      - Context sensitive requirements cannot be specified using a context free grammar.
    - **break** statement must be in a loop or **switch**.
      - Requires a complicated and unnatural context free grammar.

# Intermediate Representation

- High level intermediate representations
  - AST and DAG
  - Suitable for tasks like static type checking
- Low level intermediate representations
  - 3-address code: x = y **op** z
  - Suitable for machine-dependent tasks, such as register allocation and instruction selection.
- IR choice/design are application specific
  - C language is commonly used (AT&T Bell Lab Advanced C++)

# Three-Address Code

- Compiler-generated temporary variables
  - x + y * z
  - $t_1 = y * z$
    $t_2 = x + t_1$
- An example
  - $t_1 = b - c$
    $t_2 = a * t_1$
    $t_3 = a + t_2$
    $t_4 = t_1 * d$
    $t_5 = t_3 + t_4$



# Addresses

- Addresses in 3-address code
  - Name (variables in source code)
    - May be implemented as a pointer or reference to its entry in the symbol table.
  - Constant
    - Type conversions must be considered.
  - Compiler-generated temporary
    - Useful for optimization.
    - Register allocation.

# Instructions

- Common 3-address instructions
  - x = y *op* z // arithmetic and logical
    x = *op* y // negation and conversion
    x = y // copy
  - **goto** L // unconditional jump
    **if** x **goto** L // conditional jump
    **ifFalse** x **goto** L // conditional jump
    **if** x *op* y **goto** L // relational operation
  - **param** $x_1$ // parameter passing
    **param** $x_2$
    …
    **param** $x_n$
    **call** p, n // procedure call
    y = **call** p, n // function call
    **return** y // return a value

# Instructions (cont')

- Common 3-address instructions
  - x = y[i]   // indexed copy, i is the offset
    x[i] = y
  - x = &y   // address and pointer assignment
    x = *y
    *x = y

# Three-Address Code: Example

- Source code
  - **do** i = i + 1;
    **while** (a[i] < v);
- Translation to 3-address code (symbolic labels)
  - L: $t_1 = i + 1$
    $i = t_1$
    $t_2 = i * 8$
    $t_3 = a[t_2]$
    **if** $t_3 < v$ **goto** L
- Another translation form (position numbers)
  - 100: $t_1 = i + 1$
    101: $i = t_1$
    102: $t_2 = i * 8$
    103: $t_3 = a[t_2]$
    104: **if** $t_3 < v$ **goto** 100

# Implementations of Three-Address Code

- Quadruples (quads)
  - Pros and cons?
- Triples
  - Pros and cons?
- Indirect triples
  - Pros and cons?

Space consuming
Flexibility to optimizations

# 1) Quadruples

- Source code
  - $a = b * - c + b * - c$
- Three-address code
  - $t_1$ = **minus** c
    $t_2 = b * t_1$
    $t_3$ = **minus** c
    $t_4 = b * t_3$
    $t_5 = t_2 + t_4$
    $a = t_5$
- Quads

| | op | $arg_1$ | $arg_2$ | result |
|---|---|---|---|---|
| 0 | **minus** | c | | $t_1$ |
| 1 | * | b | $t_1$ | $t_2$ |
| 2 | **minus** | c | | $t_3$ |
| 3 | * | b | $t_3$ | $t_4$ |
| 4 | + | $t_2$ | $t_4$ | $t_5$ |
| 5 | = | $t_5$ | | a |
| ... | ... | | | |

# 2) Triples

- Three-address code
  - $t_1$ = **minus** c
    $t_2$ = b * $t_1$
    $t_3$ = **minus** c
    $t_4$ = b * $t_3$
    $t_5$ = $t_2$ + $t_4$
    a = $t_5$

| | op | $arg_1$ | $arg_2$ |
|---|---|---|---|
| 0 | **minus** | c | |
| 1 | * | b | (0) |
| 2 | **minus** | c | |
| 3 | * | b | (2) |
| 4 | + | (1) | (3) |
| 5 | = | a | (4) |
| … | … | | |

# 3) Indirect Triples

- Three-address code
  - $t_1$ = **minus** c
    $t_2$ = b * $t_1$
    $t_3$ = **minus** c
    $t_4$ = b * $t_3$
    $t_5$ = $t_2$ + $t_4$
    a = $t_5$

| | |
|---|---|
| 35 | (0) |
| 36 | (1) |
| 37 | (2) |
| 38 | (3) |
| 39 | (4) |
| 40 | (5) |
| … | … |

| | op | $arg_1$ | $arg_2$ |
|---|---|---|---|
| 0 | **minus** | c | |
| 1 | * | b | (0) |
| 2 | **minus** | c | |
| 3 | * | b | (2) |
| 4 | + | (1) | (3) |
| 5 | = | a | (4) |
| … | … | | |

In Java, array of instruction objects

# 2. Types and Declarations

- Declaration
  - Literals: implicitly
  - Variables: explicitly
  - Other names: explicitly
- Type checking in strong-typing languages
  - Type compatibility
  - Type inference
  - Implicit type conversion
  - Resolving overloading operators

# Simplified Grammar

- Declare only one name at a time

$$D \rightarrow T\ \textbf{id ;}\ D \mid \varepsilon$$

$$T \rightarrow B\ C \mid \textbf{record \{}\ D\ \textbf{\}}$$

$$B \rightarrow \textbf{int} \mid \textbf{double}$$

$$C \rightarrow \textbf{[ num ]}\ C \mid \varepsilon$$

# Translation of Type Declarations

- Computing types and their widths

$T \rightarrow B$ { $t$ = B.type; $w$ = B.width }

$\quad\quad C$ { T.type = C.type; T.width = C.width }

$B \rightarrow$ **int** { B.type = INTEGER; B.width = 4 }

$B \rightarrow$ **double** { B.type = DOUBLE; B.width = 8 }

$C \rightarrow$ **[ num ]** $C_1$ { C.type = array(**num**.value, $C_1$.type);

C.width = **num**.value $\times$ $C_1$.width }

$C \rightarrow \varepsilon$ { C.type = t; C.width = w }

Type expression

Just try it: **int[2][3]**
What is T.type and T.width?

# Translation of Type Declarations (cont')

- Computing relative addresses

$P \rightarrow$ { offset = 0 }

$\quad D$

$D \rightarrow T$ **id ;** { top.put(**id**.lexeme, T.type, offset);

offset += T.width }

$\quad D_1$

$D \rightarrow \varepsilon$

top denotes the current symbol table

Embedded actions can be removed with markers

DBv1

# Another Example

- Enter types and their widths

| | | | |
|---|---|---|---|
| P | $\rightarrow$ | | { offset = 0 } |
| | | D | |
| D | $\rightarrow$ | D **;** D | |
| D | $\rightarrow$ | **id :** T | { table.enter(id.name, T.type, offset); offset += T.width } |
| T | $\rightarrow$ | **integer** | { T.type = INTEGER; T.width = 4 } |
| T | $\rightarrow$ | **real** | { T.type = REAL; T.width = 8 } |
| T | $\rightarrow$ | **array [ num ] of** $T_1$ | { T.type = array(**num**.value, $T_1$.type); T.width = **num**.value $\times$ $T_1$.width } |
| T | $\rightarrow$ | **^** $T_1$ | { T.type = pointer($T_1$.type); T.width = 4 } |

Just try it: **k: array [5] of ^real**
What are the side effects?

# Another Example (cont')

- Declarations in nested procedures

```
P →               { tableStack.push(new Table(null));
                    offsetStack.push(0) }
     D            { addWidth(tableStack.top(), offsetStack.top());
                    tableStack.pop();
                    offsetStack.pop() }
D →  D ; D
D →  proc id ;    { tableStack.push(new Table(tableStack.top()));
                    offsetStack.push(0) }
     D1 ; S       { addWidth(tableStack.top(), offsetStack.top());
                    tableStack.pop();
                    offsetStack.pop();
                    tableStack.top().enter(id.name, tableStack.top() }
D →  id : T       { tableStack.top().enter(id.name, T.type, offsetStack.top());
                    offsetStack.top() += T.width }
```

# Another Example (cont')

- Field names in records

$T \rightarrow$ **record** { tableStack.push(**new** Table(**null**));
offsetStack.push(0) }
D **end** { T.type = record(tableStack.top());
T.width = offsetStack.top();
tableStack.pop();
offsetStack.pop() }

# 3. Assignments and Expressions

- Intermediate code generation
  - Code concatenation
    - gen(...)
    - ||
    - No side effects
  - Incremental generation
    - DBv1: emit(...)
    - DBv2: overloading gen(...)
    - Side effects

# Translation of Expressions

- Code concatenation (syntax-directed definition)

| | Productions | Semantic Rules |
|---|---|---|
| 1 | $S \rightarrow$ **id = E ;** | S.code = E.code \|\| gen(top.get(**id**.lexeme) '=' E.addr) |
| 2 | $E \rightarrow E_1 + E_2$ | E.addr = **new** Temp();<br>E.code = $E_1$.code \|\| $E_2$.code \|\| gen(E.addr '=' $E_1$.addr '+' $E_2$.addr) |
| 3 | $E \rightarrow - E_1$ | E.addr = **new** Temp();<br>E.code = $E_1$.code \|\| gen(E.addr '=' '**minus**' $E_1$.addr) |
| 4 | $E \rightarrow$ **(** $E_1$ **)** | E.addr = $E_1$.addr;<br>E.code = $E_1$.code |
| 5 | $E \rightarrow$ **id** | E.addr = top.get(**id**.lexeme);<br>E.code = ' ' |

# Translation of Expressions (cont')

- Incremental translation (translation scheme)

| | | |
|---|---|---|
| $S \rightarrow$ | **id = ** E **;** | { gen(top.get(**id**.lexeme) '=' E.addr) } |
| $E \rightarrow$ | $E_1$ **+** $E_2$ | { E.addr = **new** Temp(); gen(E.addr '=' $E_1$.addr '+' $E_2$.addr) } |
| $E \rightarrow$ | **–** $E_1$ | { E.addr = **new** Temp(); gen(E.addr '=' '**minus**' $E_1$.addr) } |
| $E \rightarrow$ | **(** $E_1$ **)** | { E.addr = $E_1$.addr } |
| $E \rightarrow$ | **id** | { E.addr = top.get(**id**.lexeme) } |

# Another Example

- Declared variables

$S \rightarrow$ **id := E ;** { p = symbolTable.lookup(**id**.name);
**if** (p == **null**) **throw new** SomeException();
emit(p '=' E.place) }

$E \rightarrow E_1 + E_2$ { E.place = **new** Temp();
emit(E.place '=' $E_1$.place '+' $E_2$.place) }

$E \rightarrow - E_1$ { E.place = **new** Temp();
emit(E.place '=' '**minus**' $E_1$.place) }

$E \rightarrow ( E_1 )$ { E.place = $E_1$.place }

$E \rightarrow$ **id** { p = symbolTable.lookup(**id**.name);
**if** (p == **null**) **throw new** SomeException();
E.place = p }

# Addressing Array Elements

- 2-dimensional array layout
  - Row major vs. column major

| | |
|---|---|
| 1st row | A[1, 1] |
| | A[1, 2] |
| | A[1, 3] |
| 2nd row | A[2, 1] |
| | A[2, 2] |
| | A[2, 3] |

| | |
|---|---|
| 1st column | A[1, 1] |
| | A[2, 1] |
| 2nd column | A[1, 2] |
| | A[2, 2] |
| 3rd column | A[1, 3] |
| | A[2, 3] |

# Addressing Array Elements

- Relative address of array elements
  - $A[i]$
    - $base + (i - low) \times w$
    - $i \times w + (base - low \times w)$

Constant for optimization

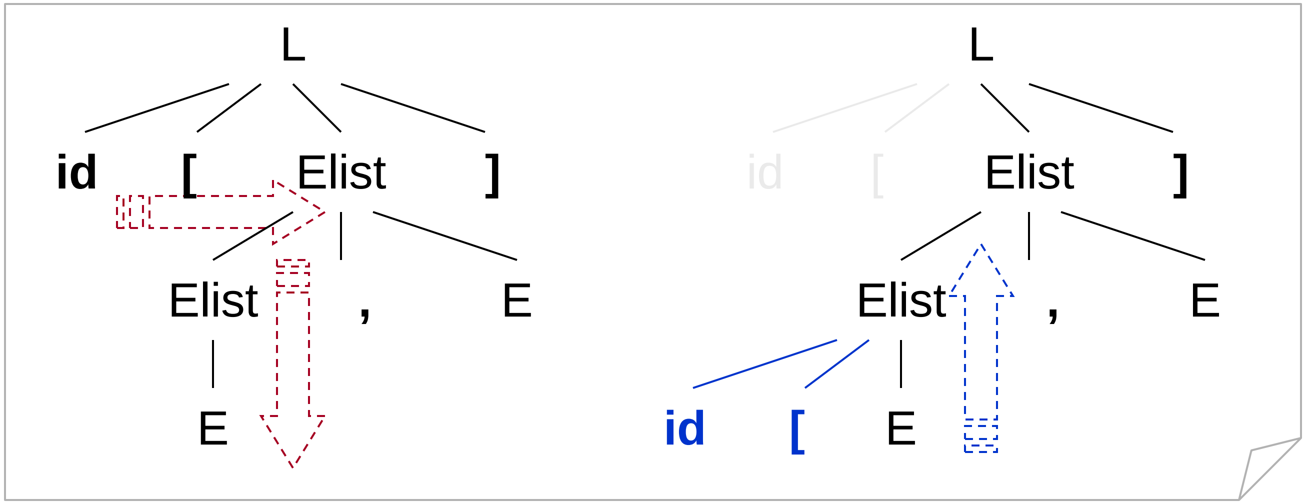
  - $A[i_1, i_2]$
    - $base + ((i_1 - low_1) \times n_2 + i_2 - low_2) \times w$
    - $((i_1 \times n_2) + i_2) \times w + (base - (low_1 \times n_2 + low_2) \times w)$
  - $A[i_1, i_2, \ldots, i_k]$
    - $((\ldots((i_1 \times n_2 + i_2) \times n_3 + i_3)\ldots) \times n_k + i_k) \times w +$
      $base - ((\ldots((low_1 \times n_2 + low_2) \times n_3 + low_3)\ldots) \times n_k + low_k) \times w$

# Addressing Tips

- For each increment of a new dimension, addressing is calculated recursively, e.g. from k to k + 1
  - For variable part V: $V \times n_{k+1} + i_{k+1}$
  - For constant part C: $C \times n_{k+1} + low_{k+1}$

# Grammar for Array References

- Array references in Pascal: a[2, 3]
  - L → **id [** Elist **]** | **id**
  - Elist → Elist **,** E | E
- Grammar transformation (why?)
  - L → Elist **]** | **id**
  - Elist → Elist **,** E | **id [** E



# Translation Scheme

- Addressing array elements in Pascal

| | |
|---|---|
| (1) $S \rightarrow L$ **:=** $E$ | { **if** (L.offset == **null**) emit(L.place '=' E.place) **else** emit(L.place '[' L.offset ']' '=' E.place) } |
| (2) $E \rightarrow E_1$ **+** $E_2$ | { E.place = **new** Temp(); emit(E.place '=' $E_1$.place '+' $E_2$.place) } |
| (3) $E \rightarrow$ **(** $E_1$ **)** | { E.place = $E_1$.place } |
| (4) $E \rightarrow L$ | { **if** (L.offset == **null**) E.place = L.place **else** { E.place = **new** Temp(); emit(E.place '=' L.place '[' L.offset ']') } } |
| (5) $L \rightarrow$ Elist **]** | { L.place = **new** Temp(); emit(L.place '=' constant(Elist.array)); L.offset = **new** Temp(); emit(L.offset '=' Elist.place '*' width(Elist.array) } |
| (6) $L \rightarrow$ **id** | { L.place = **id**.place; L.offset = **null** } |

L is a simple id (if L.offset is null)
or an array reference

**L.place** = base – C * w
**L.offset** = V * w

# Translation Scheme (cont')

- Addressing array elements in Pascal (cont')

(7) Elist $\rightarrow$ $Elist_1$ **,** E { t = **new** Temp();
m = $Elist_1$.ndim + 1;
emit(t '=' $Elist_1$.place '*' limit($Elist_1$.array, m));
emit(t '+=' E.place);
Elist.array = $Elist_1$.array;
Elist.place = t;
Elist.ndim = m }

(8) Elist $\rightarrow$ **id [** E { Elist.array = **id**.place;
Elist.place = E.place;
Elist.ndim = 1 }

**Elist.array** = base
**Elist.place** = V
**Elist.ndim** = dimensions

# Another Translation Scheme

- Array references in C/C++: a[2][3]
  - For all n, $low_n = 0$
  - Addressing formula
    - A[i]
      - $base + i \times w$
    - $A[i_1][i_2]$
      - $base + i_1 \times w_1 + i_2 \times w_2$
      - $w_1$ is the width of a row
      - $w_2$ is the width of an element in a row
    - $A[i_1][i_2]\ldots[i_k]$
      - $base + i_1 \times w_1 + i_2 \times w_2 + \ldots + i_k \times w_k$

Java does NOT use
row-major storage for arrays

# Another Translation Scheme (cont')

- Translation scheme

| | | | |
|---|---|---|---|
| S | $\rightarrow$ | **id = E ;** | { gen(top.get(**id**.lexeme) '=' E.addr) } |
| S | $\rightarrow$ | L **=** E **;** | { gen(L.array.base '[' L.addr ']' '=' E.addr) } |
| E | $\rightarrow$ | $E_1$ **+** $E_2$ | { E.addr = **new** Temp();<br>gen(E.addr '=' $E_1$.addr '+' $E_2$.addr) } |
| E | $\rightarrow$ | **id** | { E.addr = top.get(**id**.lexeme) } |
| E | $\rightarrow$ | L | { E.addr = **new** Temp();<br>gen(E.addr '=' L.array.base '[' L.addr ']') } |
| L | $\rightarrow$ | **id [** E **]** | { L.array = top.get(**id**.lexeme);<br>L.type = L.array.type.element;<br>L.addr = **new** Temp();<br>gen(L.addr '=' E.addr '*' L.type.width) } |
| L | $\rightarrow$ | $L_1$ **[** E **]** | { L.array = $L_1$.array;<br>L.type = $L_1$.type.element;<br>t = **new** Temp();<br>L.addr = **new** Temp();<br>gen(t '=' E.addr '*' L.type.width);<br>gen(L.addr '=' $L_1$.addr '+' t) } |

L only for array reference
E.addr = E.place
L.array.base = L.place
L.addr = L.offset

# 4. Type Checking

- Strong typing vs. weak typing
  - Strongness is relative
- Type definitions
  - Primitive types: enumeration of constant
  - Composite types: type expressions
  - Type of functions: signatures
    - **if** f has type $s \rightarrow t$ **and** x has type s **then** expression f(x) has type t

# Translation Scheme: An Example

- Type checking, inference and implicit casting

```
E  → E1 * E2  { E.place := new Temp();
                if (E1.type == TK_INT && E2.type == TK_INT) {
                   emit(E.place  '='  E1.place  '*int'  E2.place);
                   E.type = TK_INT;
                } elsif (E1.type == TK_REAL && E2.type == TK_REAL) {
                   emit(E.place  '='  E1.place  '*real'  E2.place);
                   E.type = TK_REAL;
                } elsif (E1.type == TK_INT && E2.type == TK_REAL) {
                   t := new Temp();
                   emit(t  '='  'int2real'  E1.place);
                   emit(E.place  '='  t  '*real'  E2.place);
                   E.type = TK_REAL;
                } elsif (...) {  ...  }
              }
```

# 5. Boolean Expressions

- Boolean expressions are used in
  - Flows of control
  - Computing logical values

# Computing Logical Values

- **a < b** equals to **if (a < b) then 1 else 0**

| | | | |
|---|---|---|---|
| E | $\rightarrow$ | $E_1$ **or** $E_2$ | { E.place = **new** Temp(); |
| | | | emit(E.place '=' $E_1$.place '**or**' $E_2$.place) } |
| E | $\rightarrow$ | $E_1$ **and** $E_2$ | { E.place = **new** Temp(); |
| | | | emit(E.place '=' $E_1$.place '**and**' $E_2$.place) } |
| E | $\rightarrow$ | **not** $E_1$ | { E.place = **new** Temp(); |
| | | | emit(E.place '=' '**not**' $E_1$.place) } |
| E | $\rightarrow$ | **(** $E_1$ **)** | { E.place = $E_1$.place } |
| E | $\rightarrow$ | **$id_1$ relop $id_2$** | { E.place = **new** Temp(); |
| | | | emit('**if**' **$id_1$**.place **relop**.op **$id_2$**.place '**goto**' currentStmt+3); |
| | | | emit(E.place '=' '**0**'); |
| | | | emit('**goto**' currentStmt+2); |
| | | | emit(E.place '=' '**1**') } |
| E | $\rightarrow$ | **true** | { E.place = **new** Temp(); |
| | | | emit(E.place '=' '**1**') } |
| E | $\rightarrow$ | **false** | { E.place = **new** Temp(); |
| | | | emit(E.place '=' '**0**') } |

DBv1

# Computing Logical Values: An Example

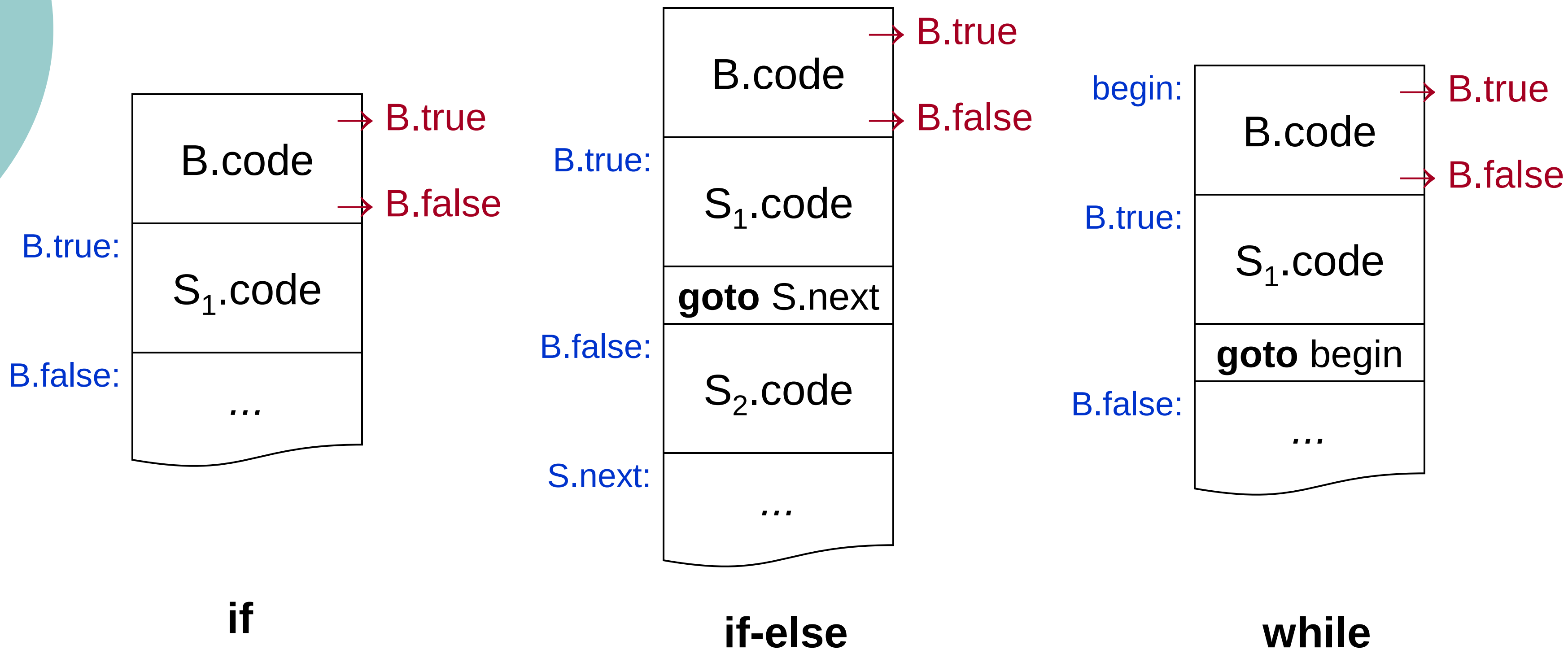
- Source code
  - a < b **or** c < d **and** e < f
- Intermediate code

| | | | |
|---|---|---|---|
| 100: | **if** a < b **goto** 103 | 108: | **if** e < f **goto** 111 |
| 101: | $t_1 = 0$ | 109: | $t_3 = 0$ |
| 102: | **goto** 104 | 110: | **goto** 112 |
| 103: | $t_1 = 1$ | 111: | $t_3 = 1$ |
| 104: | **if** c < d **goto** 107 | 112: | $t_4 = t_2$ **and** $t_3$ |
| 105: | $t_2 = 0$ | 113: | $t_5 = t_1$ **or** $t_4$ |
| 106: | **goto** 108 | 114: | … |
| 107: | $t_2 = 1$ | | |

# Short-Circuit Evaluation

- Flow-of-Control Statements
  - $S \rightarrow$ **if** **(** B **)** $S_1$
  - $S \rightarrow$ **if** **(** B **)** $S_1$ **else** $S_2$
  - $S \rightarrow$ **while** **(** B **)** $S_1$
- Short-circuit evaluation for **&&** and **||**
  - For higher evaluation efficiency
  - And ...

# Generated Code Illustration



# Syntax-Directed Definition for Flow-of-Control Statements

Where does **S.next** come from?

| Productions | Semantic Rules |
|---|---|
| $P \rightarrow S$ | S.next = **new** Label();<br>P.code = S.code \|\| label(S.next) |
| $S \rightarrow$ **assign** | S.code = **assign**.code |
| $S \rightarrow S_1\ S_2$ | $S_1$.next = **new** Label();<br>$S_2$.next = S.next;<br>S.code = $S_1$.code \|\| label($S_1$.next) \|\| $S_2$.code |
| $S \rightarrow$ **if (** B **)** $S_1$ | B.true = **new** Label();<br>B.false = $S_1$.next = S.next;<br>S.code = B.code \|\| label(B.true) \|\| $S_1$.code |
| $S \rightarrow$ **if (** B **)** $S_1$ **else** $S_2$ | B.true = **new** Label();<br>B.false = **new** Label();<br>$S_1$.next = $S_2$.next = S.next;<br>S.code = B.code \|\| label(B.true) \|\| $S_1$.code \|\| gen('**goto**' S.next) \|\| label(B.false) \|\| $S_2$.code |
| $S \rightarrow$ **while (** B **)** $S_1$ | begin = **new** Label();<br>B.true = **new** Label();<br>B.false = S.next;<br>$S_1$.next = begin;<br>S.code = label(begin) \|\| B.code \|\| label(B.true) \|\| $S_1$.code \|\| gen('**goto**' begin) |

Avoid redundant **gotos**

# Syntax-Directed Definition for Booleans

Short-Circuit Evaluation

| Productions | Semantic Rules |
|---|---|
| $B \rightarrow B_1$ **\|\|** $B_2$ | $B_1$.true = B.true;<br>$B_1$.false = **new** Label();<br>$B_2$.true = B.true;<br>$B_2$.false = B.false;<br>B.code = $B_1$.code \|\| label($B_1$.false) \|\| $B_2$.code |
| $B \rightarrow B_1$ **&&** $B_2$ | $B_1$.true = **new** Label();<br>$B_1$.false = B.false;<br>$B_2$.true = B.true;<br>$B_2$.false = B.false;<br>B.code = $B_1$.code \|\| label($B_1$.true) \|\| $B_2$.code |
| $B \rightarrow$ **!** $B_1$ | $B_1$.true = B.false;<br>$B_1$.false = B.true;<br>B.code = $B_1$.code |
| $B \rightarrow E_1$ **relop** $E_2$ | B.code = $E_1$.code \|\| $E_2$.code<br>\|\| gen('**if**' $E_1$.addr **relop**.op $E_2$.addr '**goto**' B.true)<br>\|\| gen('**goto**' B.false) |
| $B \rightarrow$ **true** | B.code = gen('**goto**' B.true) |
| $B \rightarrow$ **false** | B.code = gen('**goto**' B.false) |

# Syntax-Directed Translation: An Example

- Source code
  - **if** (x < 100 || x > 200 && x != y)  x = 0
- Intermediate code

|  |  |
|---|---|
|  | **if** x < 100 **goto** $L_2$ |
|  | **goto** $L_3$ |
| $L_3$: | **if** x > 200 **goto** $L_4$ |
|  | **goto** $L_1$ |
| $L_4$: | **if** x != y **goto** $L_2$ |
|  | **goto** $L_1$ |
| $L_2$: | x = 0 |
| $L_1$: | … |

# 6. Backpatching and Flow-of-Control Statements

- In SDD for Flow-of-Control Statements
  - Where does **S.next** come from?
  - Only after all intermediate code are generated, can **S.next** be computed.
- In SDD for Booleans
  - Where do **B.true** and **B.false** come from?
  - Must be provided by the context of the boolean expressions.
  - The context depends on the result of **S.next**.

# Design Motivation and Solution

- Motivation
  - One-pass code generation
- Solution
  - Using backpatching
- It is a general approach to dealing with initial values which must be computed at the end.

# Backpatching for Boolean Expressions

- Translation scheme

| | |
|---|---|
| $B \rightarrow B_1$ **\|\|** $M\ B_2$ | { backpatch($B_1$.falseList, M.instruction);<br>B.trueList = merge($B_1$.trueList, $B_2$.trueList);<br>B.falseList = $B_2$.falseList; } |
| $B \rightarrow B_1$ **&&** $M\ B_2$ | { backpatch($B_1$.trueList, M.instruction);<br>B.trueList = $B_2$.trueList;<br>B.falseList = merge($B_1$.falseList, $B_2$.falseList); } |
| $B \rightarrow$ **!** $B_1$ | { B.trueList = $B_1$.falseList;<br>B.falseList = $B_1$.trueList; } |
| $B \rightarrow$ **(** $B_1$ **)** | { B.trueList = $B_1$.trueList;<br>B.falseList = $B_1$.falseList; } |
| $B \rightarrow E_1$ **relop** $E_2$ | { B.trueList = **new** List(nextInstruction);<br>B.falseList = **new** List(nextInstruction + 1);<br>emit('**if**' $E_1$.addr relop.op $E_2$.addr '**goto** __');<br>emit('**goto** __'); } |
| $B \rightarrow$ **true** | { B.trueList = **new** List(nextInstruction);<br>emit('**goto** __'); } |
| $B \rightarrow$ **false** | { B.falseList = **new** List(nextInstruction);<br>emit('**goto** __'); } |
| $M \rightarrow \varepsilon$ | { M.instruction = nextInstruction; } |

# Backpatching for Flow-of-Control Statements

- Translation scheme

| | | |
|---|---|---|
| S | → **if (** B **)** M $S_1$ | { backpatch(B.trueList, M.instruction);<br>S.nextList = merge(B.falseList, $S_1$.nextList); } |
| S | → **if (** B **)** $M_1$ $S_1$ N **else** $M_2$ $S_2$ | |
| | | { backpatch(B.trueList, $M_1$.instruction);<br>backpatch(B.falseList, $M_2$.instruction);<br>S.nextList = merge($S_1$.nextList, N.nextList, $S_2$.nextList); } |
| S | → **while** $M_1$ **(** B **)** $M_2$ $S_1$ | |
| | | { backpatch(B.trueList, $M_2$.instruction);<br>backpatch($S_1$.nextList, $M_1$.instruction);<br>S.nextList = B.falseList;<br>emit('**goto**' $M_1$.instruction); } |
| S | → **{** L **}** | { S.nextList = L.nextList; } |
| S | → A **;** | { S.nextList = **new** List(); // Assignment or Atom } |
| M | → ε | { M.instruction = nextInstruction; } |
| N | → ε | { N.nextList = **new** List(nextInstruction);<br>emit('**goto** __'); } |
| L | → $L_1$ M S | { backpatch($L_1$.nextList, M.instruction);<br>L.nextList = S.nextList; } |
| L | → S | { L.nextList = S.nextList; } |

# Exercise 9.1

- What is the translation result of input token string: **x := A[y, z]**?
  - Tips: use the translation scheme for Pascal.

DBv1, pp.486

# Exercise 9.2

- What is the translation result of input token string: **c + a[i][j]**?
  - Tips: use the translation scheme for C/C++.

DBv2, pp.385

# Exercise 9.3

- What is the translation result of input token string: **x < 100 || x > 200 && x != y**?
  - Tips: use the translation scheme for boolean expressions with backpatching.
  - Suppose that the start position of the generated code is 100.

DBv2, pp.412-414

# Further Reading

- Dragon Book, 2nd Edition (DBv2)
  - Comprehensive Reading:
    - Section 6.2 on introduction to intermediate representations.
    - Section 6.5 on type checking.
    - Section 6.3, 6.4, 6.6 and 6.7 on translations of various program constructs.
  - Skip Reading:
    - Section 6.1 on AST and DAG.
    - Section 6.8 and 6.9 on translations of switches and procedures.

# Enjoy the Course!