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

- 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

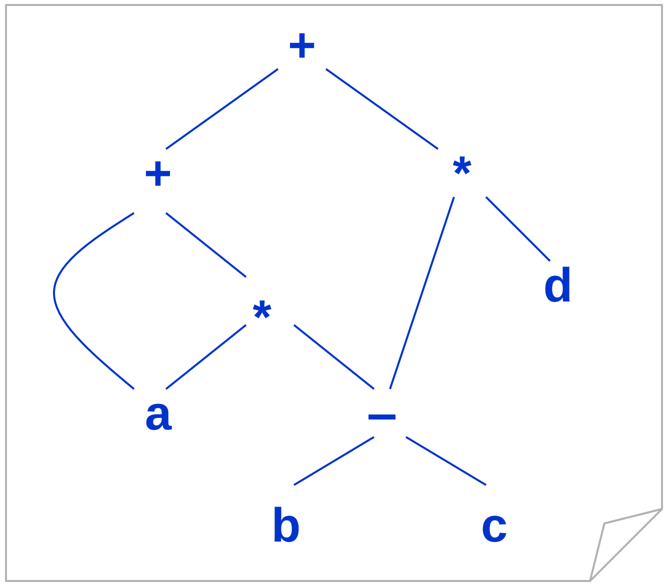
- Semantic analysis also focuses on the well-formness of source code
  - Due to the expressiveness power of Context-Free Grammars.
  - For example,
    - Number matching of actual parameters.
      - Context sensitive requirements cannot be specified using a context free grammar.
    - **break** statement 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:      t1 = i + 1
            i = t1
            t2 = i * 8
            t3 = a[t2]
            if t3 < v goto L
    ```
- Another translation form (position numbers)

  -
    ```
    100:   t1 = i + 1
    101:   i = t1
    102:   t2 = i * 8
    103:   t3 = a[t2]
    104:   if t3 < 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

In Java, array of instruction objects

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)
…	…		

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 **id ;** D | ε
T	$\rightarrow$	B C | **record {** D **}**
B	$\rightarrow$	**int** | **double**
C	$\rightarrow$	**[num]** C | ε

Translation of Type Declarations

- Computing types and their widths

$T \rightarrow B$		{ t = B.type; w = B.width }
	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 = $\underbrace{\text{array}(\textbf{num}.\text{value}, C_1.\text{type})}_{\text{Type expression}}$; C.width = **num**.value $\times$ C_1.width }
$C \rightarrow \varepsilon$		{ C.type = t; C.width = w }

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

Translation of Type Declarations (cont')

- Computing relative addresses

```
P  →          { offset = 0 }
      D
D  → T id ;   { top.put(id.lexeme, T.type, offset);
                offset += T.width }
      D1
D  → ε
```

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?

DBv1

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());
                   t = tableStack.top();  tableStack.pop();
                   offsetStack.pop();
                   tableStack.top().enter(id.name, t) }
D  →  id : T     { tableStack.top().enter(id.name, T.type, offsetStack.top());
                   offsetStack.top() += T.width }
```

DBv1

Another Example (cont')

- Field names in records

```
T  →  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();
		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();
		E.code = E_1.code || gen(E.addr '=' '**minus**' E_1.addr)

4	$E \rightarrow$ **(** E_1 **)**	E.addr = E_1.addr;
		E.code = E_1.code

5	$E \rightarrow$ **id**	E.addr = top.get(**id**.lexeme);
		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) }

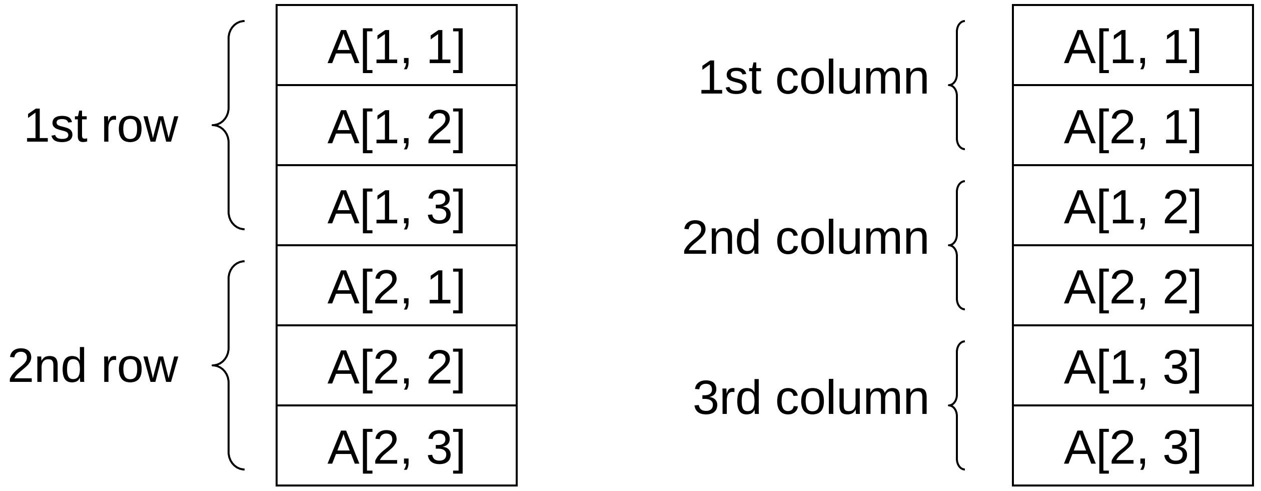
Another Example

- Declared variables

```
S → id := E ;     { p = symbolTable.lookup(id.name);
                    if (p == null)  throw new SomeException();
                    emit(p  '='  E.place) }
E → E_1 + E_2     { E.place = new Temp();
                    emit(E.place  '='  E_1.place  '+'  E_2.place) }
E → – E_1         { E.place = new Temp();
                    emit(E.place  '='  'minus'  E_1.place) }
E → ( E_1 )       { E.place = E_1.place }
E → 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



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

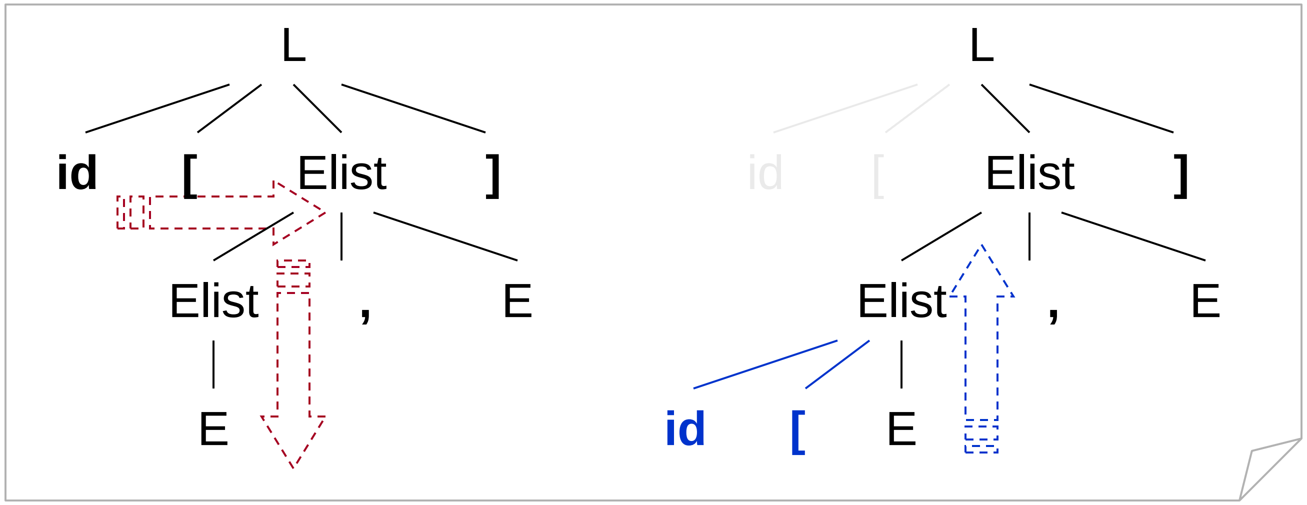
  - $A[i_1, i_2]$
    - $base + ((i_1 - low_1) \times n_2 + i_2 - low_2) \times w$
    - $((i_1 \times n_2) + i_2) \times w + (base - (low_1 \times n_2 + low_2) \times w)$
  - $A[i_1, i_2, \ldots, i_k]$
    - $((\ldots((i_1 \times n_2 + i_2) \times n_3 + i_3)\ldots) \times n_k + i_k) \times w +$
     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

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



DBv1

Translation Scheme

- Addressing array elements in Pascal

```
(1) S → L := E       { if (L.offset == null)  emit(L.place  '='  E.place)
                       else emit(L.place  '['  L.offset  ']'  '='  E.place) }
(2) E → E1 + E2      { E.place = new Temp();
                       emit(E.place  '='  E1.place  '+'  E2.place) }
(3) E → ( E1 )       { E.place = E1.place }
(4) E → L            { if (L.offset == null)  E.place = L.place
                       else {
                            E.place = new Temp();
                            emit(E.place  '='  L.place  '['  L.offset  ']')
                       } }
(5) L → 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 → 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  → Elist₁ , E  { t = new Temp();
                           m = Elist₁.ndim + 1;
                           emit(t '=' Elist₁.place '*' limit(Elist₁.array, m));
                           emit(t '+=' E.place);
                           Elist.array = Elist₁.array;
                           Elist.place = t;
                           Elist.ndim = m }
(8) Elist  → 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 → id = E ;       { gen(top.get(id.lexeme) '=' E.addr) }
S → L = E ;        { gen(L.array.base '[' L.addr ']' '=' E.addr) }
E → E₁ + E₂        { E.addr = new Temp();
                     gen(E.addr '=' E₁.addr '+' E₂.addr) }
E → id             { E.addr = top.get(id.lexeme) }
E → L              { E.addr = new Temp();
                     gen(E.addr '=' L.array.base '[' L.addr ']') }
L → id [ E ]       { L.array = top.get(id.lexeme);
                     L.type = L.array.type.element;
                     L.addr = new Temp();
                     gen(L.addr '=' E.addr '*' L.type.width) }
L → L₁ [ E ]       { L.array = L₁.array;
                     L.type = L₁.type.element;
                     t = new Temp();
                     L.addr = new Temp();
                     gen(t '=' E.addr '*' L.type.width);
                     gen(L.addr '=' L₁.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  →  E1 or E2        { E.place = new Temp();
                        emit(E.place '=' E1.place 'or' E2.place) }
E  →  E1 and E2       { E.place = new Temp();
                        emit(E.place '=' E1.place 'and' E2.place) }
E  →  not E1          { E.place = new Temp();
                        emit(E.place '=' 'not' E1.place) }
E  →  ( E1 )          { E.place = E1.place }
E  →  id1 relop id2   { E.place = new Temp();
                        emit('if' id1.place relop.op id2.place 'goto' currentStmt+3);
                        emit(E.place '=' '0');
                        emit('goto' currentStmt+2);
                        emit(E.place '=' '1') }
E  →  true            { E.place = new Temp();
                        emit(E.place '=' '1') }
E  →  false           { E.place = new Temp();
                        emit(E.place '=' '0') }
```

Computing Logical Values: An Example

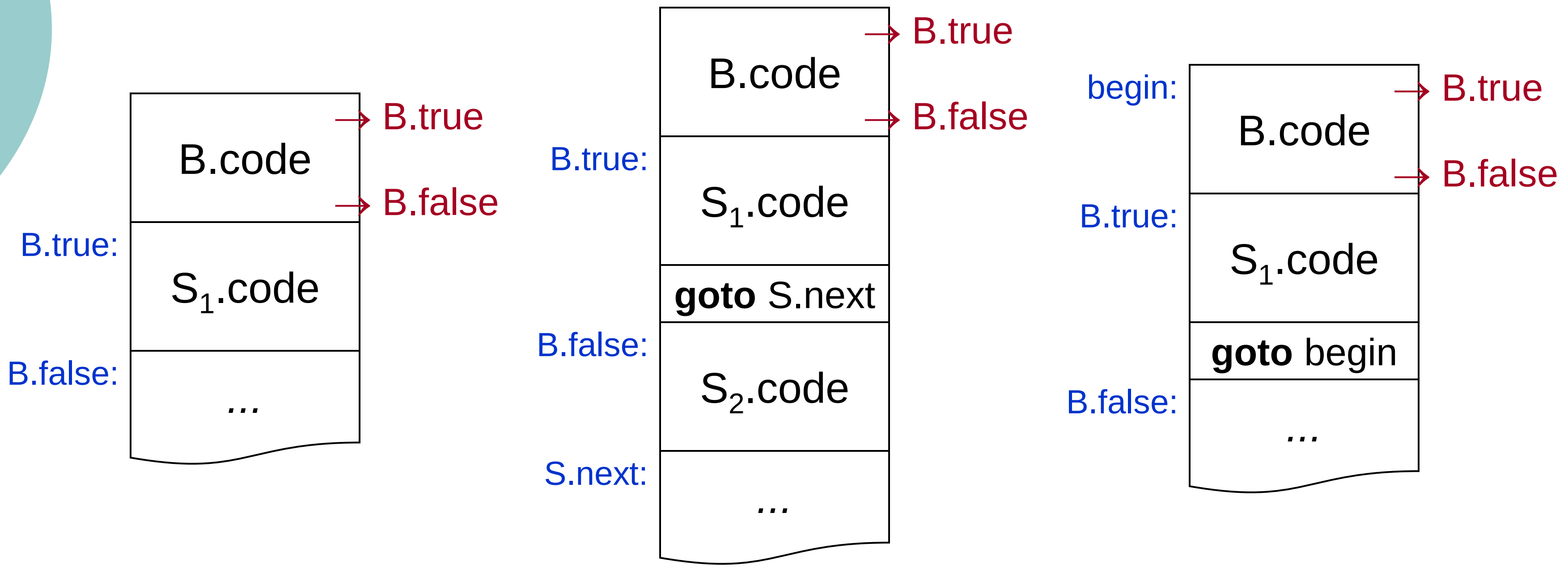
- Source code
  - a < b **or** c < d **and** e < f
- Intermediate code

```
100:    if a < b goto 103
101:    t₁ = 0
102:    goto 104
103:    t₁ = 1
104:    if c < d goto 107
105:    t₂ = 0
106:    goto 108
107:    t₂ = 1
108:    if e < f goto 111
109:    t₃ = 0
110:    goto 112
111:    t₃ = 1
112:    t₄ = t₂ and t₃
113:    t₅ = t₁ or t₄
114:    …
```

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



if **if-else** **while**

Syntax-Directed Definition for Flow-of-Control Statements

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

Productions	**Semantic Rules**

P → S	S.next = **new** Label();
	P.code = S.code || label(S.next)

S → **assign**	S.code = **assign**.code
S → S_1 S_2	S_1.next = **new** Label(); S_2.next = S.next; S.code = S_1.code || label(S_1.next) || S_2.code

S → **if (** B **)** S_1	B.true = **new** Label();
	B.false = S_1.next = S.next;
	S.code = B.code || label(B.true) || S_1.code

S → **if (** B **)** S_1 **else** S_2	B.true = **new** Label();
	B.false = **new** Label();
	S_1.next = S_2.next = S.next;
	S.code = B.code || label(B.true) || S_1.code || gen('**goto**' S.next) || label(B.false) || S_2.code

S → **while (** B **)** S_1	begin = **new** Label();
	B.true = **new** Label();
	B.false = S.next;
	S_1.next = begin;
	S.code = label(begin) || B.code || label(B.true) || S_1.code || gen('**goto**' begin)

Avoid redundant **goto**s

Syntax-Directed Definition for Booleans

Short-Circuit Evaluation

Productions	**Semantic Rules**
$B \rightarrow B_1$ **||** B_2	B_1.true = B.true; B_1.false = **new** Label(); B_2.true = B.true; B_2.false = B.false; B.code = B_1.code || label(B_1.false) || B_2.code
$B \rightarrow B_1$ **&&** B_2	B_1.true = **new** Label(); B_1.false = B.false; B_2.true = B.true; B_2.false = B.false; B.code = B_1.code || label(B_1.true) || B_2.code
$B \rightarrow$ **!** B_1	B_1.true = B.false; B_1.false = B.true; B.code = B_1.code
$B \rightarrow E_1$ **relop** E_2	B.code = E_1.code || E_2.code || gen('**if**' E_1.addr **relop**.op E_2.addr '**goto**' B.true) || 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 L2
        goto L3
L3:     if x > 200 goto L4
        goto L1
L4:     if x != y goto L2
        goto L1
L2:     x = 0
L1:     …
```

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

Backpatching for Flow-of-Control Statements

- Translation scheme

```
S → if ( B ) M S₁       { backpatch(B.trueList, M.instruction);
                          S.nextList = merge(B.falseList, S₁.nextList); }
S → if ( B ) M₁ S₁ N else M₂ S₂
                        { backpatch(B.trueList, M₁.instruction);
                          backpatch(B.falseList, M₂.instruction);
                          S.nextList = merge(S₁.nextList, N.nextList, S₂.nextList); }
S → while M₁ ( B ) M₂ S₁
                        { backpatch(B.trueList, M₂.instruction);
                          backpatch(S₁.nextList, M₁.instruction);
                          S.nextList = B.falseList;
                          emit('goto'  M₁.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);
                          emit('goto __'); }
L → L₁ M S              { backpatch(L₁.nextList, M.instruction);
                          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!