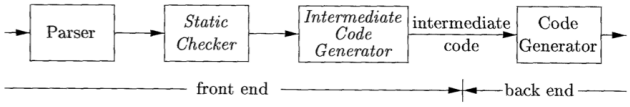
Compiler Front End

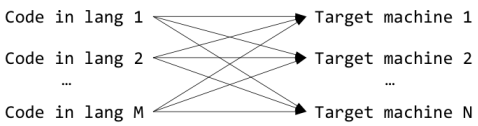
源码→前端→中间代码→后端→机器码

- The front end of a compiler analyzes a source program and creates an intermediate representation (IR, 中间表示), from which the back end generates target code
  - Details of the source language are confined to the front end, and details of the target machine to the back end

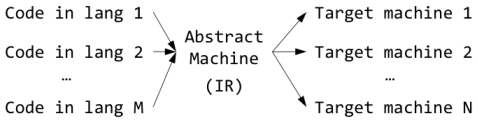


The Benefits of A Common IR

减少编译器数量

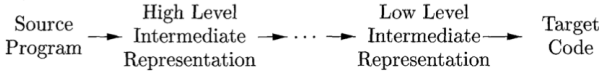


M * N compilers without a common IR



M + N compilers with a common IR

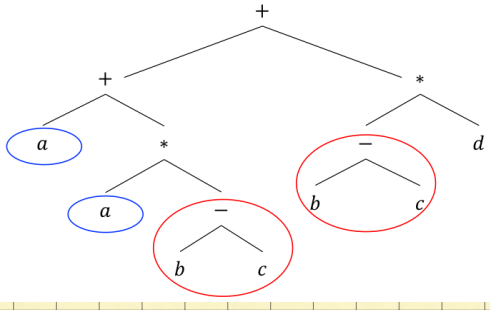
Different Levels of IRs



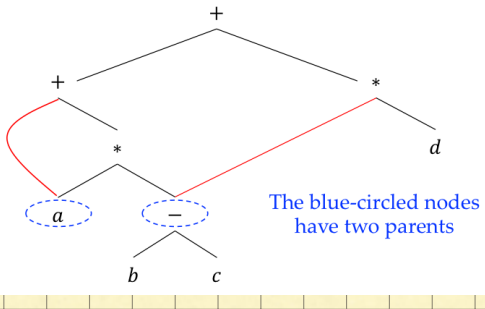
- A compiler may construct a sequence of IR's
  - High-level IR's like syntax trees are close to the source language
    - They are suitable for machine-independent tasks like static type checking
  - Low-level IR's are close to the target machines
    - They are suitable for machine-dependent tasks like register allocation and instruction selection
- Interesting fact: C is often used as an intermediate form. The first C++ compiler has a front end that generates C and a C compiler as a backend

DAG's for Expressions

- In a syntax tree, the tree for a common subexpression would be replicated as many times as the subexpression appears
  - Example: $a + a * (b - c) + (b - c) * d$



- A *directed acyclic graph* (DAG, 有向无环图) identifies the common subexpressions and represents expressions succinctly
  - Example: $a + a * (b - c) + (b - c) * d$



Constructing DAG's

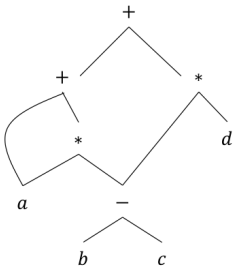
- DAG's can be constructed by the same SDD that constructs syntax trees
- **The difference:** When constructing DAG's, a new node is created if and only if there is no existing identical node

用SDD构建语法树的方式构建DAG
新建节点仅在没有相同节点

	Production	Semantic Rules
1)	$E \to E_1 + T$	$E.node =$ **new** $Node('+', E_1.node, T.node)$
2)	$E \to E_1 - T$	$E.node =$ **new** $Node('-', E_1.node, T.node)$
3)	$E \to T$	$E.node = T.node$
4)	$T \to (\ E\ )$	$T.node = E.node$
5)	$T \to$ **id**	$T.node =$ **new** $Leaf($**id**, **id**$.entry)$
6)	$T \to$ **num**	$T.node =$ **new** $Leaf($**num**, **num**$.val)$

Special "new":
Reuse existing nodes when possible

- The construction steps



1) $p_1 = Leaf($**id**, *entry-a*$)$
2) $p_2 = Leaf($**id**, *entry-a*$) = p_1$
3) $p_3 = Leaf($**id**, *entry-b*$)$
4) $p_4 = Leaf($**id**, *entry-c*$)$
5) $p_5 = Node('-', p_3, p_4)$ Node reuse
6) $p_6 = Node('*', p_1, p_5)$
7) $p_7 = Node('+', p_1, p_6)$
8) $p_8 = Leaf($**id**, *entry-b*$) = p_3$
9) $p_9 = Leaf($**id**, *entry-c*$) = p_4$
10) $p_{10} = Node('-', p_3, p_4) = p_5$
11) $p_{11} = Leaf($**id**, *entry-d*$)$
12) $p_{12} = Node('*', p_5, p_{11})$
13) $p_{13} = Node('+', p_7, p_{12})$

Three-Address Code (三地址代码)

- In three-address code, there is at most one operator on the right side of an instruction
  - Instructions are often in the form $x = y\ op\ z$
- Operands (or addresses) can be:
  - Names in the source programs
  - Constants: a compiler must deal with many types of constants
  - Temporary names generated by a compiler

最多一个操作符

操作数可为：①名称
②常量
③暂时名称

Instructions

1. Assignment instructions:
   - $x = y\ op\ z$, where op is a binary arithmetic/logical operation
   - $x = op\ y$, where op is a unary operation
2. Copy instructions: $x = y$
3. Unconditional jump instructions: goto L, where L is a label of the jump target
4. Conditional jump instructions:
   - if x goto L
   - ifFlase x goto L
   - if $x\ relop\ y$ goto L
5. Procedural calls and returns
   - param x_1
   - …
   - param x_n
   - call p, n (procedure call)
   - y = call p, n (function call)
   - return y
6. Indexed copy instructions: $x = y[i]$ $x[i] = y$
   - Here, $y[i]$ means the value in the location i memory units beyond location y
7. Address and pointer assignment instructions:
   - $x = \&y$ (set the r-value of x to be the l-value of y)
   - $x = *y$ (set the r-value of x to be the content stored at the location pointed to by y; y is a pointer whose r-value is a location)
   - $*x = y$ (set the r-value of the object pointed to by x to the r-value of y)

A variable has l-value and r-value:

- **L-value (location)** refers to the memory location, which identifies an object.
- **R-value (content)** refers to data value stored at some address in memory.

左值：内存地址
右值：数据值

Example

- Source code: `do i = i + 1; while (a[i] < v);`

```
L:   t1 = i + 1
     i = t1
     t2 = i * 8
     t3 = a [ t2 ]
     if t3 < v goto L
```

(a) Symbolic labels.

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

(b) Position numbers.

Assuming each array element takes 8 units of space

Representation of Instructions

- In a compiler, three-address instructions can be implemented as objects/records with fields for the operator and the operands
- Three typical representations:
  - Quadruples (四元式表示方法)
  - Triples (三元式表示方法)
  - Indirect triples (间接三元式表示方法)

Quadruples

- A *quadruple* has four fields
  - General form: *op* arg_1 arg_2 *result*
  - *op* contains an internal code for the operator
  - arg_1, arg_2, *result* are addresses (operands)
  - Example: $x = y + z$ ➔ + y z x
- Some exceptions:
  - Unary operators like *x = minus y* or *x = y* do not use arg_2
  - *param* operators use neither arg_2 nor *result*
  - Conditional/unconditional jumps put the target label in *result*

Quadruples Example

- Assignment statement: $a = b * -c + b * -c$

```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
a = t5
```

(a) Three-address code

	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
	...			

Temporaries

(b) Quadruples

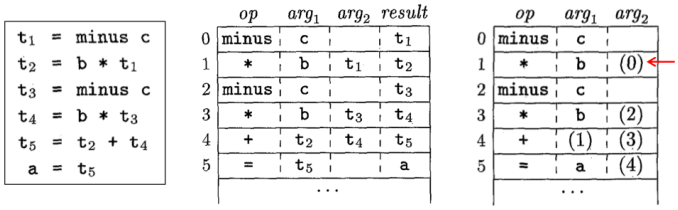
The result field is used primarily for temporary names
Temporary names waste space (symbol table entries)

result主要存暂用名，会占用多余空间.

Triples

- A *triple* has only three fields: op, arg_1, arg_2
- We refer to the result of an operation $x\ op\ y$ by its position without generating temporary names (an optimization over quadruples)

result换为位置.

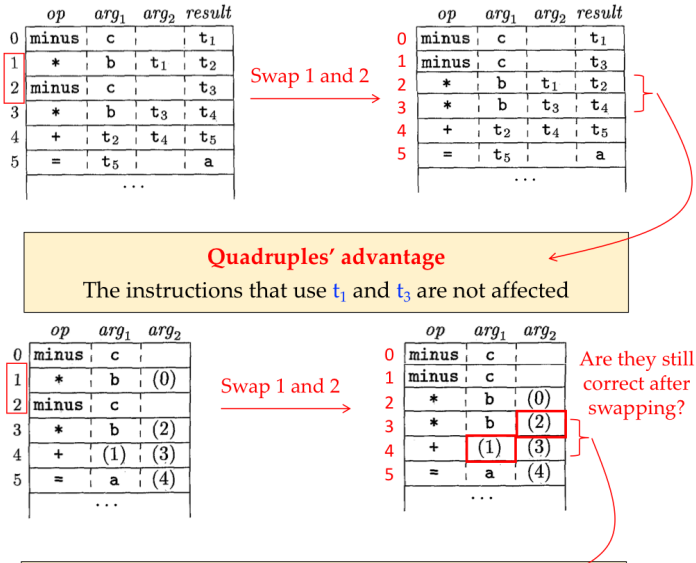


	Three-address code		Quadruples				Triples		
			op	arg_1	arg_2	result	op	arg_1	arg_2
	t_1 = minus c	0	minus	c		t_1	minus	c	
	t_2 = b * t_1	1	*	b	t_1	t_2	*	b	(0)
	t_3 = minus c	2	minus	c		t_3	minus	c	
	t_4 = b * t_3	3	*	b	t_3	t_4	*	b	(2)
	t_5 = t_2 + t_4	4	+	t_2	t_4	t_5	+	(1)	(3)
	a = t_5	5	=	t_5		a	=	a	(4)

Quadruples vs. Triples

- In optimizing compilers, instructions are often moved around

quadruple的指令位置更自由



Quadruples' advantage

The instructions that use t_1 and t_3 are not affected

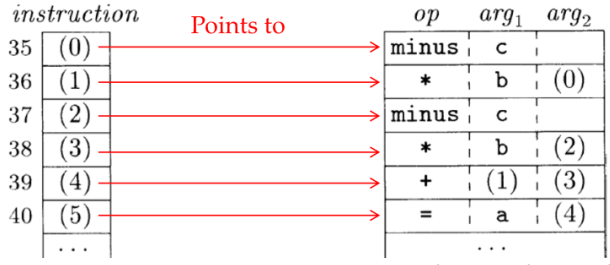
Triples' problem

The instructions now refer to wrong results; The positions need to be updated.

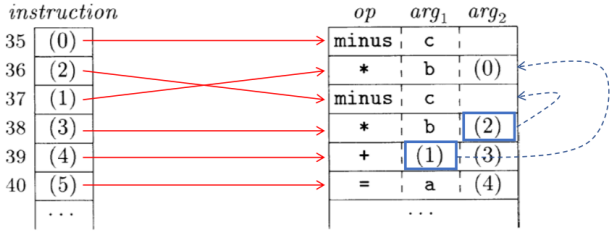
Indirect Triples (间接三元式)

- *Indirect triples* consist of a list of pointers to triples

将位置移出，改为指针



- An optimization can move an instruction by reordering the *instruction* list



Swapping pointers! The triples are not affected.

Static Single-Assignment Form

- Static single-assignment form (SSA, 静态单赋值形式) is an IR that facilitates certain code optimizations
- In SSA, each name receives a single assignment

```
p = a + b          p1 = a + b
q = p - c          q1 = p1 - c
p = q * d          p2 = q1 * d
p = e - p          p3 = e - p2
q = p + q          q2 = p3 + q1
```

(a) Three-address code. (b) Static single-assignment form.

SSA中所有赋值都是对不同名字的变量的. 如 p_1 与 p_2 不是一个变量

- The same variable may be defined in two control-flow paths

```
if ( flag ) x = -1; else x = 1;
y = x * a;
```

(x → X_1, x → X_2)

Which name should we use in y = x * a?

- SSA uses a notational convention called ϕ-function to combine the two definitions of x

```
if ( flag ) x1 = -1; else x2 = 1;
x3 = φ(x1, x2); // x1 if control flow passes through the true path; otherwise x2
y = x3 * a;
```

φ函数会根据不同控制流路径返回不同参数值.

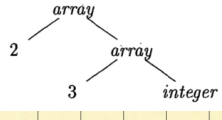
Types and Type Checking

- *Data type* or simply *type* tells a compiler or interpreter how the programmers intend to use the data
- The usefulness of type information
  - Find faults in the source code
  - Determine the storage needed for a name at runtime
  - Calculate the address of an array element
  - Insert type conversions
  - Choose the right version of some arithmetic operator (e.g., `fadd`, `iadd`)
- *Type checking* (类型检查) uses logical rules to make sure that the types of the operands match the type expectation by an operator

操作数类型符合操作符.

Type Expressions (类型表达式)

- Types have structure, which can be represented by *type expressions*
  - A type expression is either a basic type, or
  - Formed by applying a *type constructor* (类型构造算子) to a type expression
- $array(2, array(3, integer))$ is the type expression for `int[2][3]`
  - *array* is a type constructor with two arguments: a number, a type expression



①基本类型
②类型表达式上应用类型构造算子

The Definition of Type Expression

- A basic type is a type expression
  - *boolean*, *char*, *integer*, *float*, and *void*, …
- A type name (e.g., name of a class) is a type expression
- A type expression can be formed
  - By applying the *array* type constructor to a number and a type expression
  - By applying the *record* type constructor to the field names and their types
  - By applying the → type constructor for function types

- If s and t are type expressions, then their Cartesian product $s \times t$ is a type expression (this is introduced for completeness, can be used to represent a list of types such as function parameters)
- Type expressions may contain type variables (e.g., those generated by compilers) whose values are type expressions

Type Equivalence

- Type checking rules usually have the following form

If two type expressions are equivalent
then return a given type
else return **type_error**

Code under analysis:
`a + b`

- The key is to define when two type expressions are equivalent
  - The main difficulty arises from the fact that most modern languages allow the naming of user-defined types
    - In C/C++, type naming is achieved by the `typedef` statement

Name Equivalence (名等价)

类型表达式的name即为基本类型

- Treat named types as basic types; names in type expressions are not replaced by the exact type expressions they define
- Two type expressions are name equivalent if and only if they are identical (represented by the same syntax tree, with the same labels)

```
typedef struct {
    int data[100];
    int count;
} Stack;
```

```
typedef struct {
    int data[100];
    int count;
} Set;
```

```
Code under analysis:
Stack x, y;
Set r, s;
x = y;  ✓
r = s;  ✓
x = r;  ✗
```

http://web.eecs.utk.edu/~bvanderz/teaching/cs365Sp14/notes/types.html

Structual Equivalence (结构等价)

类型表达式的name被其具体的组成递归代替.

- For named types, replace the names by the type expressions and recursively check the substituted trees

```
typedef struct {
    int data[100];
    int count;
} Stack;
```

```
typedef struct {
    int data[100];
    int count;
} Set;
```

```
Code under analysis:
Stack x, y;
Set r, s;
x = y;  ✓
r = s;  ✓
x = r;  ✓
```

Declarations (变量声明)

- The grammar below deals with basic, array, and record types
  - Nonterminal D generates a sequence of declarations
  - T generates basic, array, or record types
  - A record type is a sequence of declarations for the fields of the record, surrounded by curly braces
  - B generates one of the basic types: `int` and `float`
  - C generates sequences of one or more integers, each surrounded by brackets

$$
\begin{array}{lcl}
D & \rightarrow & T\ \textbf{id}\ ;\ D \mid \epsilon \\
T & \rightarrow & B\ C \mid \textbf{record}\ '\{'\ D\ '\}' \\
B & \rightarrow & \textbf{int} \mid \textbf{float} \\
C & \rightarrow & \epsilon \mid [\ \textbf{num}\ ]\ C
\end{array}
$$

Storage Layout for Local Names (局部变量的存储布局)

- From the type of a name, we can decide the amount of memory needed for the name at run time
  - The *width* (宽度) of a type: # memory units needed for an object of the type
  - For data of varying lengths, such as strings, or whose size cannot be determined until run time, such as dynamic arrays, we only reserve a fixed amount of memory for a pointer to the data
- For local names of a function, we always assign contiguous bytes*
  - For each such name, at compile time, we can compute a relative address
  - Type information and relative addresses are stored in symbol table

* This follows the principle of proximity and is mainly for performance considerations.

An SDT for Computing Types and Their Widths

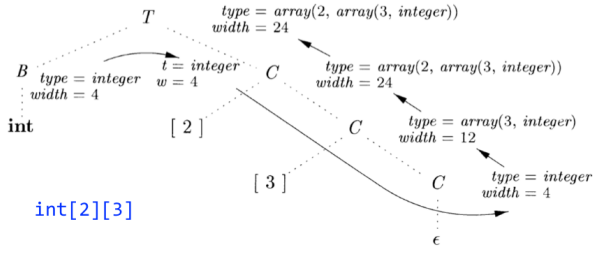
- Synthesized attributes: *type*, *width*
- Global variables t and w pass type and width information from a B node in a parse tree to the node for the production $C \rightarrow \epsilon$
  - In an SDD, t and w would be C's inherited attributes (the SDD is L-attributed)*

$T \rightarrow B$	$\{ t = B.type; w = B.width; \}$
$\quad C$	$\{ T.type = C.type;\ T.width = C.width; \}$
$B \rightarrow \textbf{int}$	$\{ B.type = integer;\ B.width = 4; \}$
$B \rightarrow \textbf{float}$	$\{ B.type = float;\ B.width = 8; \}$
$C \rightarrow \epsilon$	$\{ C.type = t;\ C.width = w; \}$
$C \rightarrow [\ \textbf{num}\ ]\ C_1$	$\{ C.type = array(\textbf{num}.value,\ C_1.type);$ $C.width = \textbf{num}.value \times C_1.width; \}$

This SDT can be implemented during recursive-descent parsing

Translation Process Example

- **Recall the translation during recursive-descent parsing**
  - Use the arguments of function $A()$ to pass nonterminal A's inherited attributes*
  - Evaluate and Return the synthesized attributes of A when the $A()$ completes

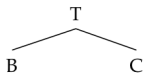


* In our example, we use global variables t and w

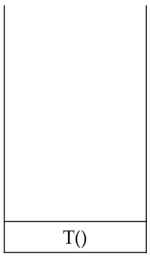
Translation Process Example

$T \rightarrow B\ C \mid \textbf{record}\ '\{'\ D\ '\}'$
$B \rightarrow \textbf{int} \mid \textbf{float}$
$C \rightarrow \epsilon \mid [\ \textbf{num}\ ]\ C$

Input string: `int[2][3]`

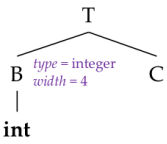


Step 1: Rewrite T using $T \rightarrow BC$

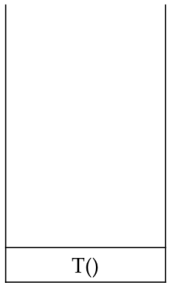


Call stack

Input string: `int[2][3]`



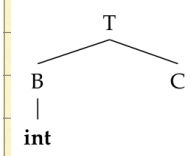
Step 3:
- B() returns
- Execute semantic action



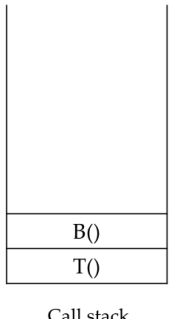
Call stack

$B \rightarrow \textbf{int}$ $\{\ B.type = integer;\ B.width = 4;\ \}$

Input string: `int[2][3]`



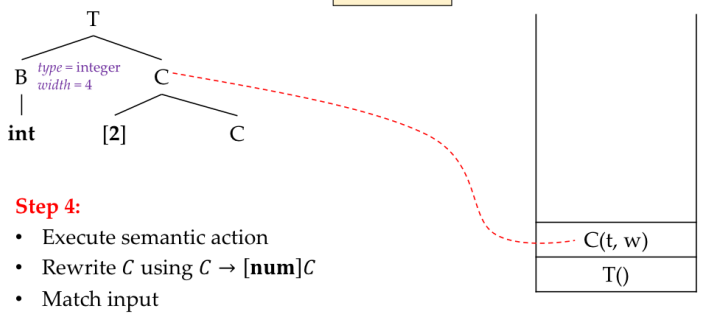
Step 2:
- Rewrite B using $B \rightarrow$ **int**
- Match input



Call stack

Input string: `int[2][3]`

t = integer
w = 4



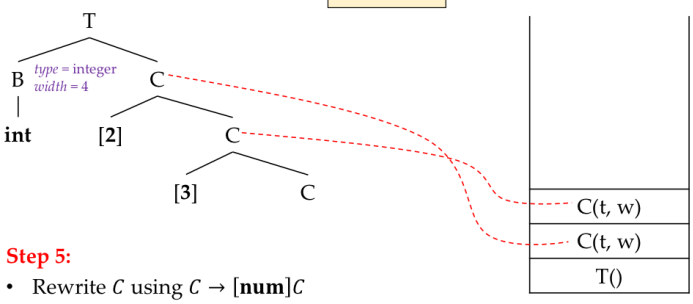
Step 4:
- Execute semantic action
- Rewrite C using $C \rightarrow [\textbf{num}]C$
- Match input

Call stack

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

Input string: `int[2][3]`

t = integer
w = 4

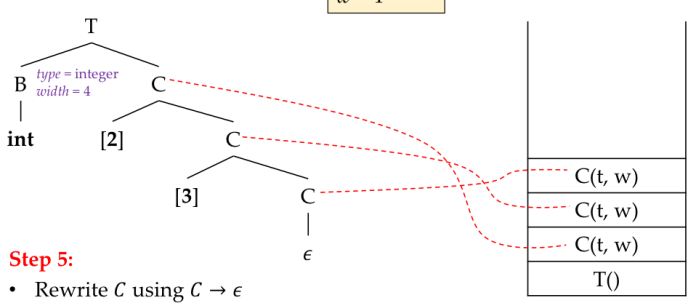


Step 5:
- Rewrite C using $C \rightarrow [\textbf{num}]C$
- Match input

Call stack

Input string: `int[2][3]`

t = integer
w = 4

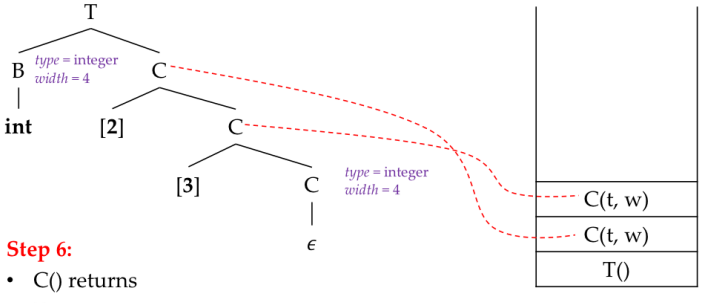


Step 5:
- Rewrite C using $C \rightarrow \epsilon$

Call stack

Input string: `int[2][3]`

t = integer
w = 4



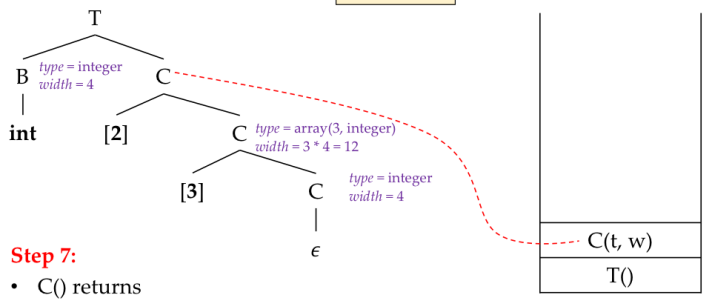
Step 6:
- C() returns
- Execute semantic action

Call stack

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

Input string: `int[2][3]`

t = integer
w = 4



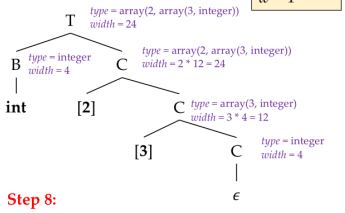
Step 7:
- C() returns
- Execute semantic action

Call stack

$C \rightarrow [\ \textbf{num}\ ]\ C_1$ $\{\ C.type = array(\textbf{num}.value,\ C_1.type);\ C.width = \textbf{num}.value \times C_1.width;\ \}$

Input string: `int[2][3]`

t = integer
w = 4



Step 8:
- T() returns
- Execute semantic action

Call stack

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

Sequences of Declarations

局部变量在在分离的符号表中，使用offset来寻找

- When dealing with a procedure, local variables should be put in a separate symbol table; their declarations can be processed as a group
  - Name, type, and relative address of each variable should be stored
- The translation scheme below handles a sequence of declarations
  - *offset*: the next available relative address; *top*: the current symbol table

$$
\begin{array}{lll}
P \rightarrow & & \{\ offset = 0;\ \} \\
 & D & \\
D \rightarrow & T\ \mathbf{id}\ ; & \{\ top.put(\mathbf{id}.lexeme,\ T.type,\ offset); \\
 & & \quad offset\ =\ offset + T.width;\ \} \\
 & D_1 & \\
D \rightarrow & \epsilon &
\end{array}
$$

Computing relative addresses of declared names

Fields in Records and Classes

假设
①同一record中的field name不一样
②offset是相对于record的data area
每一个record type都有一个符号表

- Two assumptions:
  - The field names within a record must be distinct
  - The offset for a field name is relative to the data area (数据区) for that record
- For convenience, we use a symbol table for each record type
  - Store both type and relative address of fields
- A record type has the form $record(t)$
  - *record* is the type constructor
  - t is a symbol table object, holding info about the fields of this record type

$$
\begin{array}{lll}
T \rightarrow & \mathbf{record}\ '\{' & \{\ Env.push(top);\ top = \mathbf{new}\ Env(); \\
 & & \quad Stack.push(offset);\ offset = 0;\ \} \\
 & D\ '\}' & \{\ T.type = record(top);\ T.width = offset; \\
 & & \quad top = Env.pop();\ offset = Stack.pop();\ \}
\end{array}
$$

- The class *Env* implements symbol tables
- $Env.push(top)$ and $Stack.push(offset)$ save the current symbol table and offset; later, they will be popped to continue with other translation
- The translation scheme can be adapted to deal with classes

Type Checking

强类型：程序运行时不会发生类型错误
弱类型：反之

- To do type checking, a compiler needs to assign a type expression to each component of the source program
- The compiler then determines whether the type expressions conform to a collection of logical rules (i.e., the *type system*)
  - A *sound* type system allows us to determine statically that type errors cannot occur at run time
- A language is *strongly typed* if the compiler guarantees that the programs it accepts will run without type errors (sound type system)
  - **Strongly typed:** Java (`double a; ~~int b = a~~; //cannot compile`)
  - **Weakly typed:** C/C++ (`double a; int b = a; //implicit conversion`)

Rules for Type Checking

- **Type synthesis (类型合成)**
  - Build up the type of an expression from the types of subexpressions
    - **Typical form: if** f has type $s \to t$ **and** x has type s, **then** expression $f(x)$ has type t
    - **Example:** $f(x) = -x$ (can be generalized to multi-argument cases)
- **Type inference (类型推导)**
  - Determine the type of a language construct from the way it is used
    - **Typical form: if** $f(x)$ is an expression, **then:** as f has type $\alpha \to \beta$ (α, β represent two types), x has type α
    - **Example:** let *null* be a function that tests whether a list is empty, then from the usage $null(x)$, we can tell that x must be a list

类型合成：
$f: s \to t$ ，$x: s$ ，则 $f(x): t$

类型推导：
$f(x)$，则若 $f: \alpha \to \beta$，则 $x: \alpha$。

Type Conversions

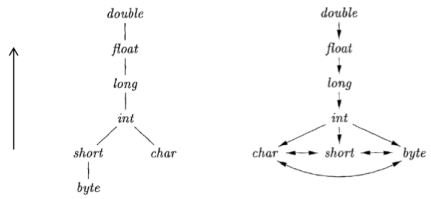
- Consider an expression $x * i$, where x is a float and i is an integer
  - The representation (the way of organizing 0/1 bits) of integers and floating-point numbers is different
  - Different machine instructions are used for operations on integers an floats
  - Convert integers to floats: t_1 = (float) i t_2 = x fmul t_1
- Type conversion SDT for a simple case (using type synthesis)
  - $E \to E_1 + E_2$

```
{   if(E1.type = integer and E2.type = integer) E.type = integer;
    else if(E1.type = float and E2.type = integer) E.type = float;
     ...
}
```

综合时选用更大的类型

Widening and Narrowing

- Type conversion rules vary from language to language
- Java distinguishes between *widening* conversions (类型拓宽) and *narrowing* conversions (类型窄化)
- *Widening* conversions preserve information and can be done automatically by the compiler (*implicit* type conversions, or *coercions*)
- *Narrowing* conversions lose information and require programmers to write code to cause the conversion (*explicit* type conversions, or *casts*)



(a) Widening conversions (b) Narrowing conversions

拓宽是自动完成的，保护了信息
窄化是需要增加操作的，会损失信息

SDT for Type Conversion

- $\max(t_1, t_2)$ takes two types t_1 and t_2 and returns the maximum (or least upper bound) of the two types in the widening hierarchy
- $widen(a, t, w)$ generates type conversions if needed to widen an address a of type t into a value of type w

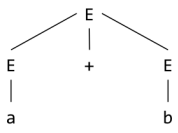
```
Addr widen(Addr a, Type t, Type w)
    if ( t = w ) return a;
    else if ( t = integer and w = float ) {
        temp = new Temp();
        gen(temp '=' '(float)' a);
        return temp;
    }
    else error;
}
```

```
E → E1 + E2   { E.type = max(E1.type, E2.type);
                a1 = widen(E1.addr, E1.type, E.type);
                a2 = widen(E2.addr, E2.type, E.type);
                E.addr = new Temp();
                gen(E.addr '=' a1 '+' a2); }
```

Example

- a + b (suppose a is of *int* type and b is of *float* type)

```
Addr widen(Addr a, Type t, Type w)
    if ( t = w ) return a;          3
    else if ( t = integer and w = float ) {
        temp = new Temp();
        gen(temp '=' '(float)' a);  2
        return temp;
    }
    else error;
}
```



Generated code:

```
temp = (float) a    2
temp2 = temp + b    5
```

```
E → E1 + E2  { E.type = max(E1.type, E2.type);
               a1 = widen(E1.addr, E1.type, E.type);
               a2 = widen(E2.addr, E2.type, E.type);
               E.addr = new Temp();
               gen(E.addr '=' a1 '+' a2); }
```

- $E.type = max(\texttt{int}, \texttt{float}) = \texttt{float}$ — 1
- $a_1 = widen(\texttt{a}, \texttt{int}, \texttt{float}) = \texttt{temp}$ — 2
- $a_2 = widen(\texttt{b}, \texttt{float}, \texttt{float}) = \texttt{b}$ — 3
- $E.addr = \texttt{new } Temp() = \texttt{temp2}$ — 4
- 5

Expressions and Arrays

- An expression with more than one operator: $a + b * c$
  - Translate into multiple instructions with at most one operator per instruction

$$a + b * c \Rightarrow \begin{aligned} t_1 &= b * c \\ t_2 &= a + t_1 \end{aligned}$$

- An array reference $A[i][j]$ will expand into a sequence of three-address instructions that calculate an address for the reference

SDD for Expressions

PRODUCTION	SEMANTIC RULES
$S \rightarrow \textbf{id} = E\ ;$	$S.code = E.code \mid\mid gen(top.get(\textbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \textbf{new}\ Temp()$ $E.code = E_1.code \mid\mid E_2.code \mid\mid gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ - E_1$	$E.addr = \textbf{new}\ Temp()$ $E.code = E_1.code \mid\mid gen(E.addr\ '='\ '\textbf{minus}'\ E_1.addr)$
$\mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid$ **id**	$E.addr = top.get(\textbf{id}.lexeme)$ $E.code = ''$

S.code and *E.code* denote three-address code

E.addr denotes the address that will hold the value of *E*

top denotes the current symbol table; *get* returns the address of **id** (a variable)

gen generates three-address instructions

All attributes are synthesized. This S-attributed SDD can be implemented during bottom-up parsing.

PRODUCTION	SEMANTIC RULES
$S \rightarrow \textbf{id} = E\ ;$	$S.code = E.code \mid\mid gen(top.get(\textbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \textbf{new}\ Temp()$ $E.code = E_1.code \mid\mid E_2.code \mid\mid gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ - E_1$	$E.addr = \textbf{new}\ Temp()$ → Temporary name generated by compiler $E.code = E_1.code \mid\mid gen(E.addr\ '='\ '\textbf{minus}'\ E_1.addr)$
$\mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid$ **id**	$E.addr = top.get(\textbf{id}.lexeme)$ → Check the symbol-table entry for **id** and save its address in *E.addr* $E.code = ''$

PRODUCTION	SEMANTIC RULES
$S \rightarrow \textbf{id} = E\ ;$	$S.code = E.code \mid\mid gen(top.get(\textbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \textbf{new}\ Temp()$ $E.code = E_1.code \mid\mid E_2.code \mid\mid gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ - E_1$	$E.addr = \textbf{new}\ Temp()$ $E.code = E_1.code \mid\mid gen(E.addr\ '='\ '\textbf{minus}'\ E_1.addr)$
$\mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid$ **id**	$E.addr = top.get(\textbf{id}.lexeme)$ $E.code = ''$

- Generate instructions when seeing operations.
- Then concatenate instructions.

Inefficiency in the SDD

PRODUCTION	SEMANTIC RULES
$S \rightarrow \textbf{id} = E\ ;$	$S.code = E.code \mid\mid gen(top.get(\textbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \textbf{new}\ Temp()$ $E.code = E_1.code \mid\mid E_2.code \mid\mid gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ - E_1$	$E.addr = \textbf{new}\ Temp()$ $E.code = E_1.code \mid\mid gen(E.addr\ '='\ '\textbf{minus}'\ E_1.addr)$
$\mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid$ **id**	$E.addr = top.get(\textbf{id}.lexeme)$ $E.code = ''$

Code attributes can be very long strings (as the expressions can be arbitrarily complex)

Redundant parts waste memory!

code这个属性可能会很长，会导致内存浪费

Incremental Translation Scheme

- In the SDT below, *gen* not only generates a three-address instruction, but also appends it to the sequence of instructions generated so far
  - In comparison, in the previous SDD, the *code* attribute can be long strings after concatenations

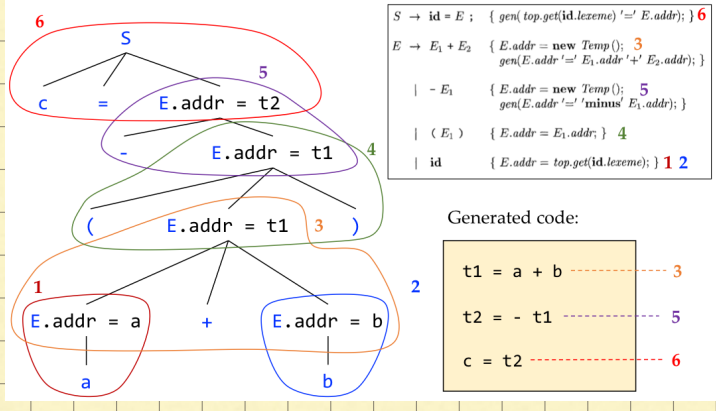
$S \rightarrow$ **id** = E ;	{ *gen*(*top.get*(**id**.*lexeme*) '=' E.*addr*); }
$E \rightarrow E_1 + E_2$	{ E.*addr* = **new** *Temp*(); *gen*(E.*addr* '=' E_1.*addr* '+' E_2.*addr*); }
| - E_1	{ E.*addr* = **new** *Temp*(); *gen*(E.*addr* '=' '**minus**' E_1.*addr*); }
| (E_1)	{ E.*addr* = E_1.*addr*;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

Why this incremental approach can guarantee the correct order of instructions?

$S \rightarrow$ **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_1	{ E.*addr* = **new** *Temp*(); *gen*(E.*addr* '=' '**minus**' E_1.*addr*); }
| (E_1)	{ E.*addr* = E_1.*addr*; }
| **id**	{ E.*addr* = *top.get*(**id**.*lexeme*); }

This postfix SDT can be implemented in bottom-up parsing where subexpressions are always handled first (e.g., the code of E_1 and E_2 is generated before E)

Example: Translating c = -(a+b)



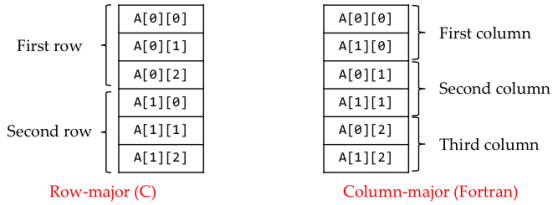
$S \rightarrow$ **id** = E ;	{ *gen*(*top.get*(**id**.*lexeme*) '=' E.*addr*); } 6
$E \rightarrow E_1 + E_2$	{ E.*addr* = **new** *Temp*(); 3 *gen*(E.*addr* '=' E_1.*addr* '+' E_2.*addr*); }
| - E_1	{ E.*addr* = **new** *Temp*(); 5 *gen*(E.*addr* '=' '**minus**' E_1.*addr*); }
| (E_1)	{ E.*addr* = E_1.*addr*; } 4
| **id**	{ E.*addr* = *top.get*(**id**.*lexeme*); } 1 2

Generated code:

```
t1 = a + b ---------- 3
t2 = - t1 ---------- 5
c = t2 ---------- 6
```

Addressing Array Elements

- Array elements can be accessed quickly if they are stored consecutively
- For an array A with n elements, the relative address of $A[i]$ is:
  - $\boldsymbol{base + i * w}$ (*base* is the relative address of $A[0]$, w is the width of an element)
- For a 2D array A (row-major layout), the relative address of $A[i_1][i_2]$ is:
  - $\boldsymbol{base + i_1 * w_1 + i_2 * w_2}$ (w_1 is the width of a row, w_2 is the width of an element)



- Further generalize to k-dimensional array A (row-major layout), the relative address of $A[i_1][i_2] \ldots [i_k]$ is:
  - $\boldsymbol{base + i_1 * w_1 + i_2 * w_2 + \cdots + i_k * w_k}$ (w's can be generalized as above)

Translation of Array References

- The main problem in generating code for array references is to relate the address-calculation formula to the grammar
  - The relative address of $A[i_1][i_2] \ldots [i_k]$ is $base + i_1 * w_1 + i_2 * w_2 + \cdots + i_k * w_k$
  - Productions for generating array references: $L \rightarrow L$ [E] | **id** [E]

gen() 不止生成三地址指令，还将其加入到已生成的指令之后.

自底向上生成.

Row-major: 最右先变
Col-major: 最左先变

SDT for Array References

$L \rightarrow \textbf{id}\ [\ E\ ]$ { $L.array = top.get(\textbf{id}.lexeme)$;
$L.type = L.array.type.elem$;
$L.addr = \textbf{new}\ Temp()$;
$gen(L.addr\ '='\ E.addr\ '*'\ L.type.width)$; }

$|\ L_1\ [\ E\ ]$ { $L.array = L_1.array$;
$L.type = L_1.type.elem$;
$t = \textbf{new}\ Temp()$;
$L.addr = \textbf{new}\ Temp()$;
$gen(t\ '='\ E.addr\ '*'\ L.type.width)$;
$gen(L.addr\ '='\ L_1.addr\ '+'\ t)$; }

A is a 2*3 array of integers
Translate A[i][j]

L.type is the type of A's element: array(3, int)

A[i] → Reduce using prod. #1 → L

L[j] → Reduce using prod. #2 → L

L.type is the type of A[i]'s element: int

L.array: a pointer to the symbol-table entry for the array name

L.array.base: the base address of the array

L.addr: a temporary for computing the offset for the array reference

L.type: the type of the subarray generated by *L*

t.elem: for any array type *t*, *t.elem* gives the element type

- The semantic actions of L-productions compute offsets
- The address of an array element is *base + offset*

$E \rightarrow E_1 + E_2$ { $E.addr = \textbf{new}\ Temp()$;
$gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$; }

$|\ \textbf{id}$ { $E.addr = top.get(\textbf{id}.lexeme)$; }

$|\ L$ { $E.addr = \textbf{new}\ Temp()$;
$gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']')$; }

Instruction of the form $x = a[i]$

Array references can be part of an expression

$S \rightarrow \textbf{id} = E\ ;$ { $gen(top.get(\textbf{id}.lexeme)\ '='\ E.addr)$; }

$|\ L = E\ ;$ { $gen(L.addr.base\ '['\ L.addr\ ']'\ '='\ E.addr)$; }

Instruction of form $a[i] = x$

Array references can appear at the LHS of an assignment statement

$E \rightarrow E_1 + E_2$ { $E.addr = \textbf{new}\ Temp()$; 6
$gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$; }

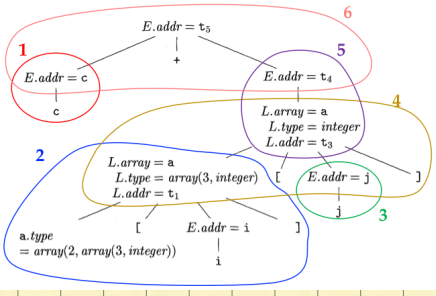
$|\ \textbf{id}$ { $E.addr = top.get(\textbf{id}.lexeme)$; } 1 3

$|\ L$ { $E.addr = \textbf{new}\ Temp()$; 5
$gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']')$; }

$L \rightarrow \textbf{id}\ [\ E\ ]$ { $L.array = top.get(\textbf{id}.lexeme)$;
$L.type = L.array.type.elem$; 2
$L.addr = \textbf{new}\ Temp()$;
$gen(L.addr\ '='\ E.addr\ '*'\ L.type.width)$; }

$|\ L_1\ [\ E\ ]$ { $L.array = L_1.array$;
$L.type = L_1.type.elem$;
$t = \textbf{new}\ Temp()$;
$L.addr = \textbf{new}\ Temp()$; 4
$gen(t\ '='\ E.addr\ '*'\ L.type.width)$;
$gen(L.addr\ '='\ L_1.addr\ '+'\ t)$; }

Translating `c + a[i][j]`



Generated code:

```
t1 = i * 12      ------ 2
t2 = j * 4       ------ 4
t3 = t1 + t2
t4 = a[t3]       ------ 5
t5 = c + t4      ------ 6
```

Control Flow

- Boolean expressions are often used to alter the flow of control or compute logical values
- Grammar: $B \rightarrow B \parallel B \mid B \,\&\&\, B \mid !\, B \mid (B) \mid E\ \textbf{rel}\ E \mid \textbf{true} \mid \textbf{false}$
- Given the expression $B_1 \parallel B_2$, if B_1 is true, then the expression is true without having to evaluate B_2. In other words, B_1 or B_2 may not need to be evaluated fully.*
- In *short-circuit* code, the boolean operators &&, ||, ! translate into jumps. The operators do not appear in the code.

If B_1 or B_2 has side effect (e.g., changing the value of a global variable), then the effect may not occur

Short-Circuit Code Example

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

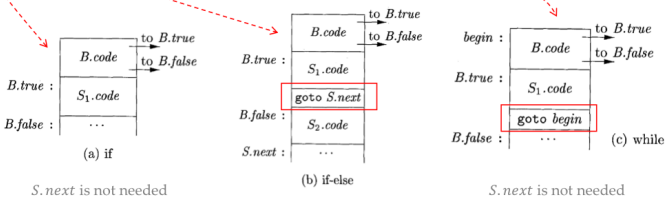
```
        if x < 100 goto L2
        ifFalse x > 200 goto L1
        ifFalse x != y goto L1
L2:     x = 0
L1:
```

Flow-of-Control Statements

- Grammar:
  - $S \rightarrow$ **if** (B) S_1
  - $S \rightarrow$ **if** (B) S_1 **else** S_2
  - $S \rightarrow$ **while** (B) S_1

Inherited attributes:

- $B.true$: the label to which control flows if B is true
- $B.false$: the label to which control flows if B is false
- $S.next$: the label for the instruction immediately after the code for S



SDD for Flow-of-Control Statements

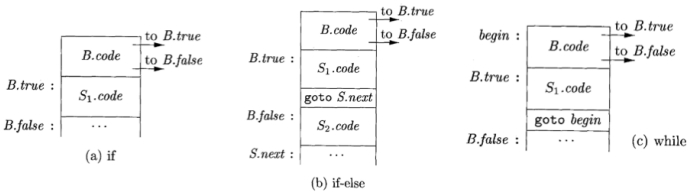
Production	Semantic Rules
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$
$S \rightarrow$ **assign**	$S.code =$ **assign**$.code$
$S \rightarrow$ **if** (B) S_1	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$
$S \rightarrow$ **if** (B) S_1 **else** S_2	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\mid\mid label(B.true) \mid\mid S_1.code$ $\mid\mid gen('goto'\ S.next)$ $\mid\mid label(B.false) \mid\mid S_2.code$
$S \rightarrow$ **while** (B) S_1	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \mid\mid B.code$ $\mid\mid label(B.true) \mid\mid S_1.code$ $\mid\mid gen('goto'\ begin)$
$S \rightarrow S_1\ S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \mid\mid label(S_1.next) \mid\mid S_2.code$

Illustrated by previous figures

Illustrated by previous figure

Translating Boolean Expressions in Flow-of-Control Statements

- A boolean expression B is translated into three-address instructions that evaluate B using conditional and unconditional jumps to one of two labels: $B.true$ and $B.false$
  - $B.true$ and $B.false$ are two inherited attributes. Their value depends on the context of B (e.g., *if* statement, *if-else* statement, *while* statement)

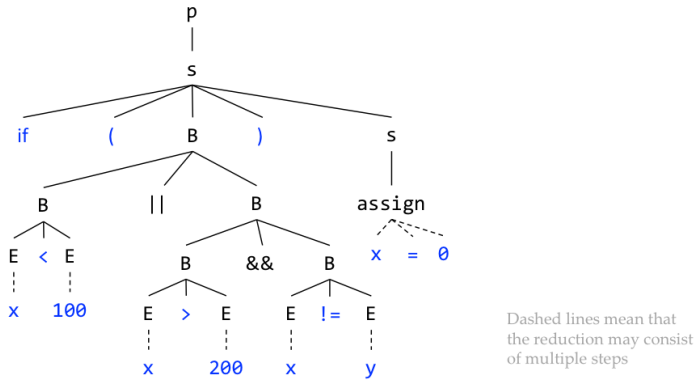


Generating Three-Address Code for Boolean

Production	Semantic Rules
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ // short-circuiting $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1 \,\&\&\, B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ // short-circuiting $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$
$B \rightarrow \,!\, B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1$ **rel** E_2	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen('\texttt{if}'\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ '\texttt{goto}'\ B.true)$ $\mid\mid gen('\texttt{goto}'\ B.false)$
$B \rightarrow$ **true**	$B.code = gen('\texttt{goto}'\ B.true)$
$B \rightarrow$ **false**	$B.code = gen('\texttt{goto}'\ B.false)$

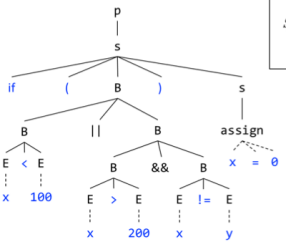
Example

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



Dashed lines mean that the reduction may consist of multiple steps

Example

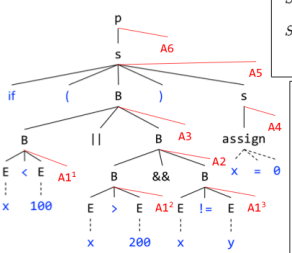


This SDD is L-attributed, not S-attributed. The grammar is not LL. There is no way to implement the SDD directly during parsing.

Production	Semantic Rules
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$
$S \rightarrow$ **assign**	$S.code = \textbf{assign}.code$
$S \rightarrow$ **if** (B) S_1	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1 \,\&\&\, B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$
$B \rightarrow \,!\, B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1$ **rel** E_2	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen('\texttt{if}'\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ '\texttt{goto}'\ B.true)$ $\mid\mid gen('\texttt{goto}'\ B.false)$

Traversing the parse tree to evaluate the attributes helps generate intermidate code

Example



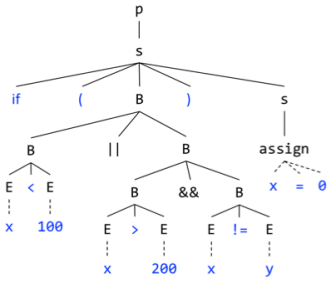
Virtual nodes are in red color

Application order of actions (preorder traversal of the tree):

$A1^1$ $A1^2$ $A1^3$ A2 A3 A4 A5 A6

Production	Semantic Rules
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$ A6
$S \rightarrow$ **assign**	$S.code = \textbf{assign}.code$ A4
$S \rightarrow$ **if** (B) S_1	$B.true = newlabel()$ $B.false = S_1.next = S.next$ A5 $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ A3 $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1 \,\&\&\, B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ A2 $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$
$B \rightarrow \,!\, B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1$ **rel** E_2	$B.code = E_1.code \mid\mid E_2.code$ A1 $\mid\mid gen('\texttt{if}'\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ '\texttt{goto}'\ B.true)$ $\mid\mid gen('\texttt{goto}'\ B.false)$

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



Generated 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:
```

Backpatching (回填)

- A **key problem** when generating code for boolean expressions and flow-of-control statements is to match a jump instruction with the jump target
- **Example: if (*B*) *S***
  - According to the short-circuit translation, *B*'s code contains a jump to the instruction following the code for *S* (executed when *B* is false)
  - However, *B* must be translated before *S*. The jump target is unknown when translating *B*
  - Earlier, we address the problem by passing labels as inherited attributes (*S.next*), but this requires another separate pass (traversing the parse tree) after parsing

How to address the problem in one pass?

B比S先翻译，但翻译B时需要知道S.next
因此不知道B.false应该是什么.

One-Pass Code Generation Using Backpatching

- **Basic idea of backpatching (基本思想):**
  - When a jump is generated, its target is temporarily left unspecified.
  - Incomplete jumps are grouped into lists. All jumps on a list have the same target.
  - Fill in the labels for incomplete jumps when the targets become known.
- **The technique (技术细节):**
  - For a nonterminal *B* that represents a boolean expression, we define two synthesized attributes: *truelist* and *falselist*
  - *truelist*: a list of jump instructions whose target is the label to which the control goes when *B* is true
  - *falselist*: a list of jump instructions whose target is the label to which the control goes when *B* is false
- **The technique (技术细节) Cont.:**
  - *makelist*(i): create a new list containing only i, the index of a jump instruction, and return the pointer to the list
  - *merge*(p_1, p_2): concatenate the lists pointed by p_1 and p_2, and return a pointer to the concatenated list
  - *backpatch*(p, i): insert i as the target for each of the jump instructions on the list pointed by p

生成跳转指令时不指定该指令的目标，
并将这些指令放入由跳转指令组成的列表中
等到能确定时再填充.

Backpatching for Boolean Expressions

- An SDT suitable for generating code for boolean expressions during bottom-up parsing
- Grammar:
  - $B \to B_1 \,\|\, M B_2 \mid B_1 \,\&\&\, M B_2 \mid !B_1 \mid (B_1) \mid E_1 \textbf{ rel } E_2 \mid \textbf{true} \mid \textbf{false}$
  - $M \to \epsilon$

Keep this question in mind: Why do we introduce M before B_2?

M在B_2前是为了读下一条指令的序号.

	1)	$B \to B_1 \mid\mid M\ B_2$	{ $backpatch(B_1.falselist, M.instr)$; $B.truelist = merge(B_1.truelist, B_2.truelist)$; $B.falselist = B_2.falselist$; }
(2)	2)	$B \to B_1$ && $M\ B_2$	{ $backpatch(B_1.truelist, M.instr)$; $B.truelist = B_2.truelist$; $B.falselist = merge(B_1.falselist, B_2.falselist)$; }
	3)	$B \to\ !\ B_1$	{ $B.truelist = B_1.falselist$; $B.falselist = B_1.truelist$; }
	4)	$B \to (\ B_1\ )$	{ $B.truelist = B_1.truelist$; $B.falselist = B_1.falselist$; }
(1)	5)	$B \to E_1$ **rel** E_2	{ $B.truelist = makelist(nextinstr)$; $B.falselist = makelist(nextinstr + 1)$; $gen('\texttt{if}'\ E_1.addr$ **rel**$.op\ E_2.addr\ '\texttt{goto}\ \_')$; $gen('\texttt{goto}\ \_')$; }
	6)	$B \to$ **true**	{ $B.truelist = makelist(nextinstr)$; $gen('\texttt{goto}\ \_')$; }
	7)	$B \to$ **false**	{ $B.falselist = makelist(nextinstr)$; $gen('\texttt{goto}\ \_')$; }
	8)	$M \to \epsilon$	{ $M.instr = nextinstr$; }

Tip: understand 1 and 2 at a high level first and then revisit this slide after you understand the later examples.

Backpatching vs. Non-Backpatching

(1) Non-backpatching SDD with inherited attributes:

$B \to E_1$ **rel** E_2	$B.code = E_1.code \mid\mid E_2.code \mid\mid gen('\texttt{if}'\ E_1.addr$ **rel**$.op\ E_2.addr\ '\texttt{goto}'\ B.true) $\mid\mid$ $gen('\texttt{goto}'\ B.false)$

(2) Backpatching scheme:

$B \to E_1$ **rel** E_2	{ $B.truelist = makelist(nextinstr)$; $B.falselist = makelist(nextinstr + 1)$; $gen('\texttt{if}'\ E_1.addr$ **rel**$.op\ E_2.addr\ '\texttt{goto}\ \_')$; $gen('\texttt{goto}\ \_')$; }

Comparison:

- In (2), incomplete instructions (指令坯) are added to corresponding lists
- The instruction jumping to $B.true$ in (1) is added to $B.truelist$ in (2)
- The instruction jumping to $B.false$ in (1) is added to $B.falselist$ in (2)

(1) Non-backpatching SDD with inherited attributes:

$B \to B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$

(2) Backpatching scheme:

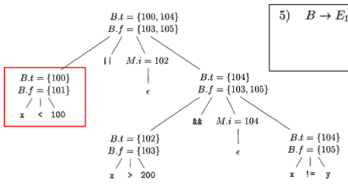
$B \to B_1 \mid\mid M\ B_2$	{ $backpatch(B_1.falselist, M.instr)$; $B.truelist = merge(B_1.truelist, B_2.truelist)$; $B.falselist = B_2.falselist$; }

Comparison:

- The assignments to $true/false$ attributes in (1) correspond to the assignments to $truelist/falselist$ or $merge$ in (2)

Example – Boolean Expressions

- The earlier SDT is a postfix SDT. The semantic actions can be performed during a bottom-up parse.
- Boolean expression: $x < 100 \parallel x > 200$ && $x != y$
- **Step 1:** reduce $x < 100$ to B by production (5)

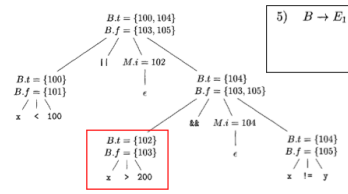


5)	$B \to E_1$ **rel** E_2	{ $B.truelist = makelist(nextinstr)$; $B.falselist = makelist(nextinstr + 1)$; $gen('\texttt{if}'\ E_1.addr$ **rel**$.op\ E_2.addr\ '\texttt{goto}\ \_')$; $gen('\texttt{goto}\ \_')$; }	

Generated code:

```
100:  if x < 100 goto _
101:  goto _
```

- **Step 3:** reduce $x > 200$ to B by production (5)

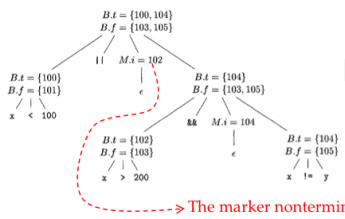


5)	$B \to E_1$ **rel** E_2	{ $B.truelist = makelist(nextinstr)$; $B.falselist = makelist(nextinstr + 1)$; $gen('\texttt{if}'\ E_1.addr$ **rel**$.op\ E_2.addr\ '\texttt{goto}\ \_')$; $gen('\texttt{goto}\ \_')$; }	

Generated code:

```
102:  if x > 200 goto _
103:  goto _
```

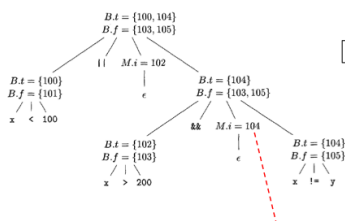
- **Step 2:** reduce ϵ to M by production (8)



8)	$M \to \epsilon$	{ $M.instr = nextinstr$; }	

The marker nonterminal records the value of *nextinstr*, 102

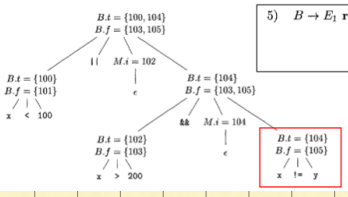
- **Step 4:** reduce ϵ to M by production (8)



8)	$M \to \epsilon$	{ $M.instr = nextinstr$; }	

The marker nonterminal records the value of *nextinstr*, 104

- **Step 5:** reduce $x! = y$ to B by production (5)

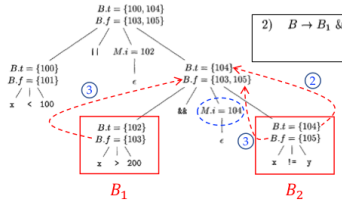


5) $B \to E_1$ **rel** E_2 { $B.truelist = makelist(nextinstr)$;
$B.falselist = makelist(nextinstr + 1)$;
$gen('\texttt{if}'\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ '\texttt{goto}\ \_')$;
$gen('\texttt{goto}\ \_')$; }

Generated code:

```
104:  if x != y goto _
105:  goto _
```

- **Step 6:** reduce B_1 && MB_2 to B by production (2)

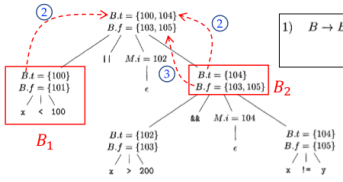


2) $B \to B_1$ && M B_2 { $backpatch(B_1.truelist, M.instr)$; ①
$B.truelist = B_2.truelist$; ②
$B.falselist = merge(B_1.falselist, B_2.falselist)$; ③

Backpatch instruction 102: ①

```
100:  if x < 100 goto _
101:  goto _
102:  if x > 200 goto 104
103:  goto _
104:  if x != y goto _
105:  goto _
```

- **Step 7:** reduce B_1 || MB_2 to B by production (1)



1) $B \to B_1$ || M B_2 { $backpatch(B_1.falselist, M.instr)$; ①
$B.truelist = merge(B_1.truelist, B_2.truelist)$; ②
$B.falselist = B_2.falselist$; } ③

Backpatch instruction 101: ①

```
100:  if x < 100 goto _
101:  goto 102
102:  if x > 200 goto 104
103:  goto _
104:  if x != y goto _
105:  goto _
```

The remaining jump targets will be filled in later parsing steps