



编译原理

Compiler Principles

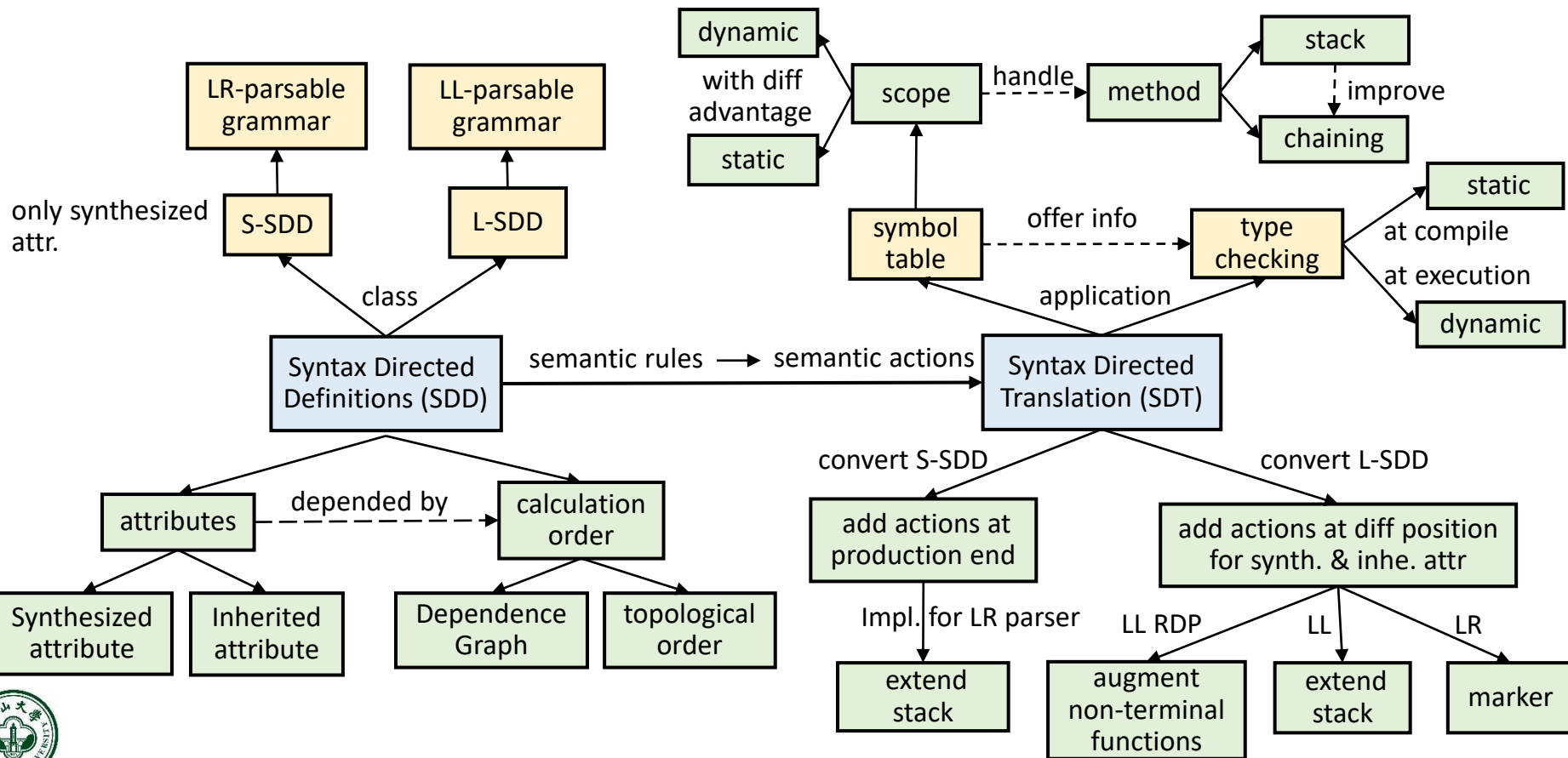
Lecture 8

Intermediate Code: Intro & IR

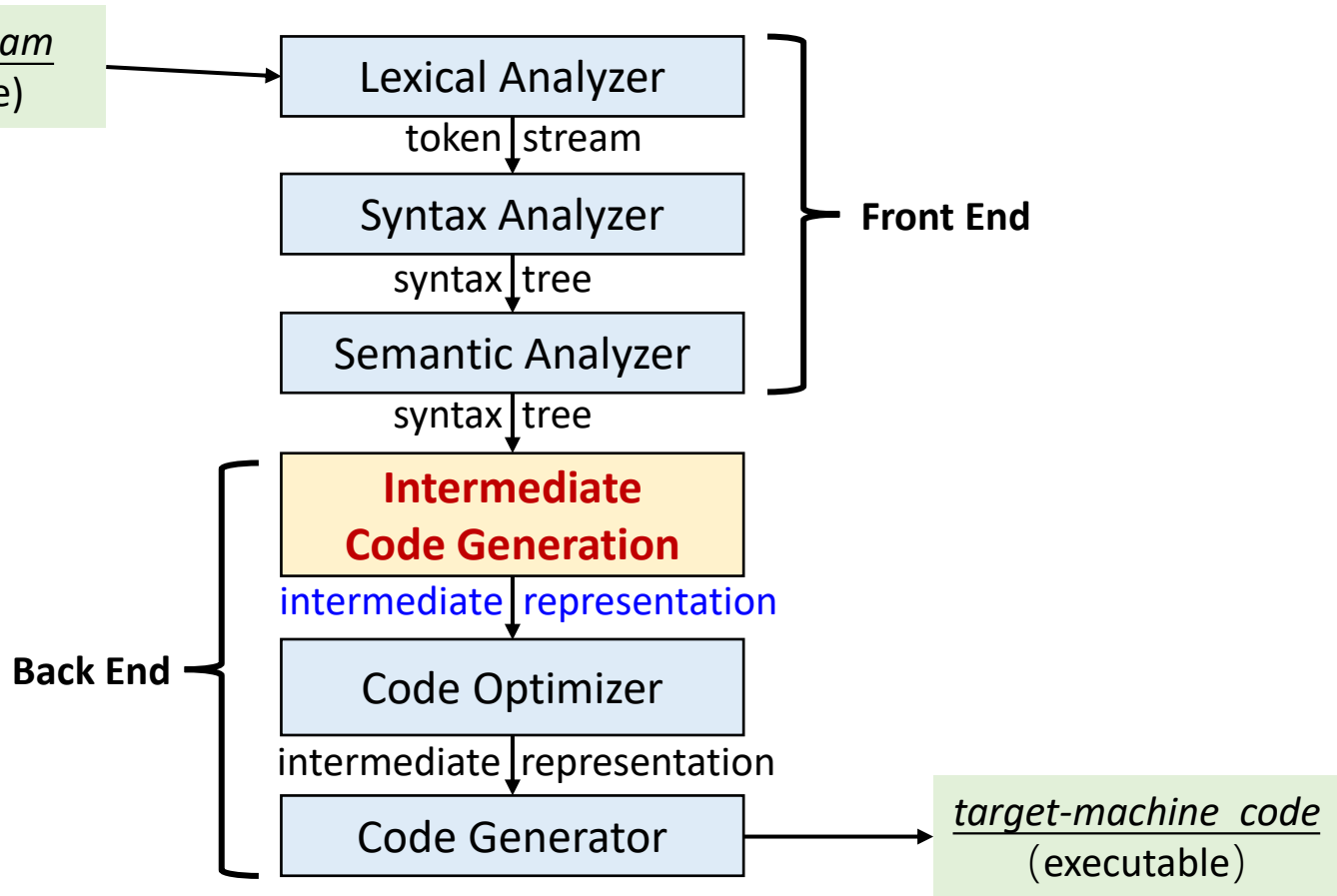
赵帅

计算机学院
中山大学

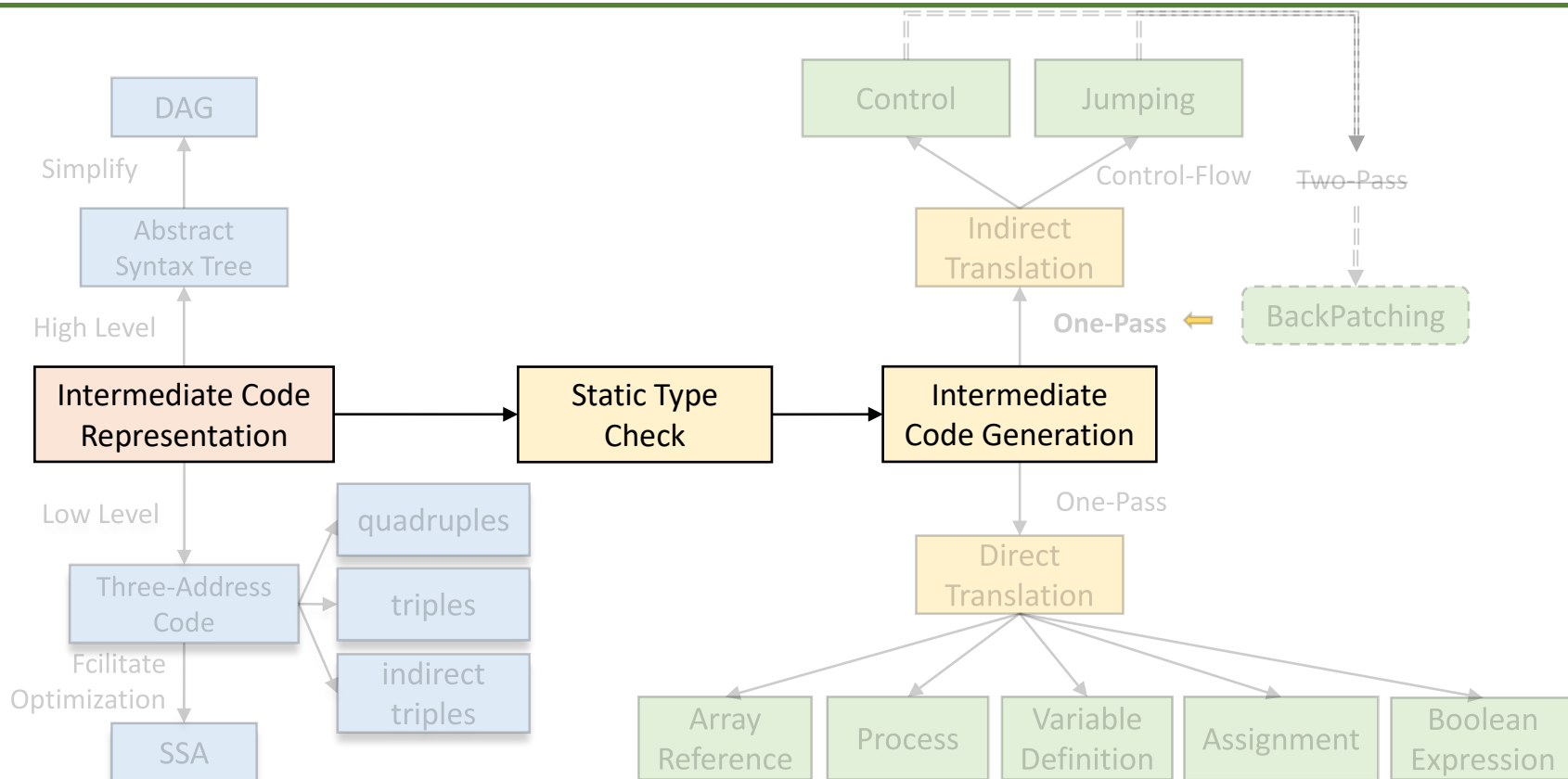
Revisit



Compilation Phases[编译阶段]



Intermediate Code[中间代码生成]



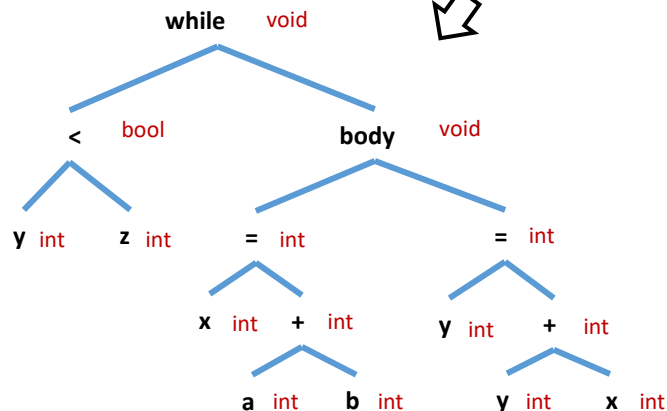
Compilation Phases[编译阶段]



```
while (y<z){  
  int x = a + b;  
  y += x;  
}
```

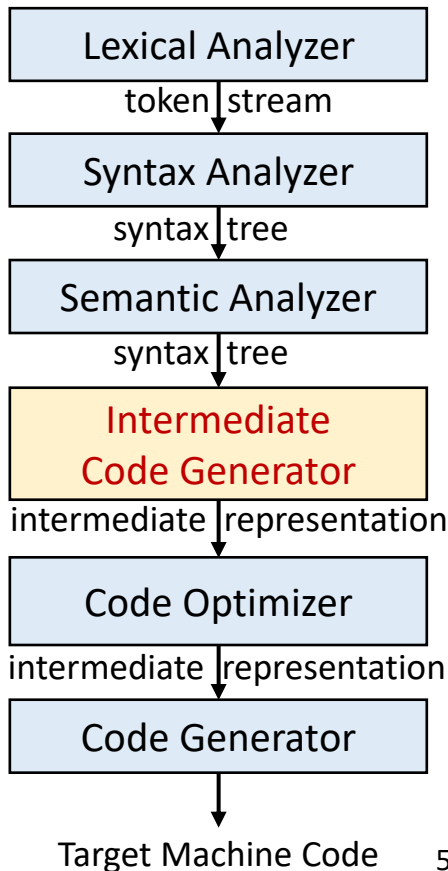


(keyword, while)	(id, b)
(id, y)	(sym, ;)
(sym, <)	(id, y)
(id, z)	(sym, +=)
(id, x)	(id, x)
(id, a)	(sym, ;)
(sym, +)	



```
goto L1  
L2:  
  t1 := a + b  
  x := t1  
  t2 := y + x  
  y := t2  
L1:  
  if y < z goto L2
```

Annotated AST/Decorated AST
[带标注的抽象语法树]



Multiple IR Levels [不同层级的中间表示]



- IR provides advantages [中间表示的优势]
 - ◆ Increased abstraction and cleaner separation
- A compiler may construct a sequence of intermediate representations.



- Modern compilers use different IRs at different stages.
- **High-level IR** are close to the source code [接近源语言]
 - ◆ Example: Parse Tree, Abstract Syntax Tree [抽象语法树]
 - ◆ Language dependent (a high-level IR for each language)
 - ◆ Purpose: semantic analysis of program

Multiple Level IR[不同层级的中间表示]



- **Low-level IR** are close to assembly [接近汇编]
 - ◆ E.g., three address code (TAC) [三地址码], static single assignment [静态单赋值]
 - ◆ Essentially an instruction set [指令集] for an abstract machine
 - ◆ Language and Machine **independent** (one common IR)
 - ◆ Purpose: **compiler optimizations** to make code efficient
 - All optimizations written in this IR is automatically **applicable to all languages and machines**
- **Machine-Level IR** [机器层级]
 - ◆ Example: x86 IR, ARM IR, ...
 - ◆ Actual instructions for a concrete machine ISA [指令集架构]
 - ◆ **Machine dependent** (a machine-level IR for each ISA)
 - ◆ Purpose: code generation, CPU register allocation, etc

Multiple Level IR[不同层级的中间表示]



- Possible to have only one IR (AST) — some compilers follows this
 - ◆ Generate **machine code from AST** after semantic analysis [AST直接到机器码]
 - ◆ Makes sense if compilation time is the primary concern (e.g., JIT)
 - Skip the IR generation step
- **Why multiple IRs?**
 1. Better to have an appropriate IR **for the task at hand** [针对性]
 - Semantic analysis much easier with high-level IR (AST)
 - Compiler optimizations much easier with low-level IR (TAC)
 - Register allocation only possible with machine-level IR (ISA)

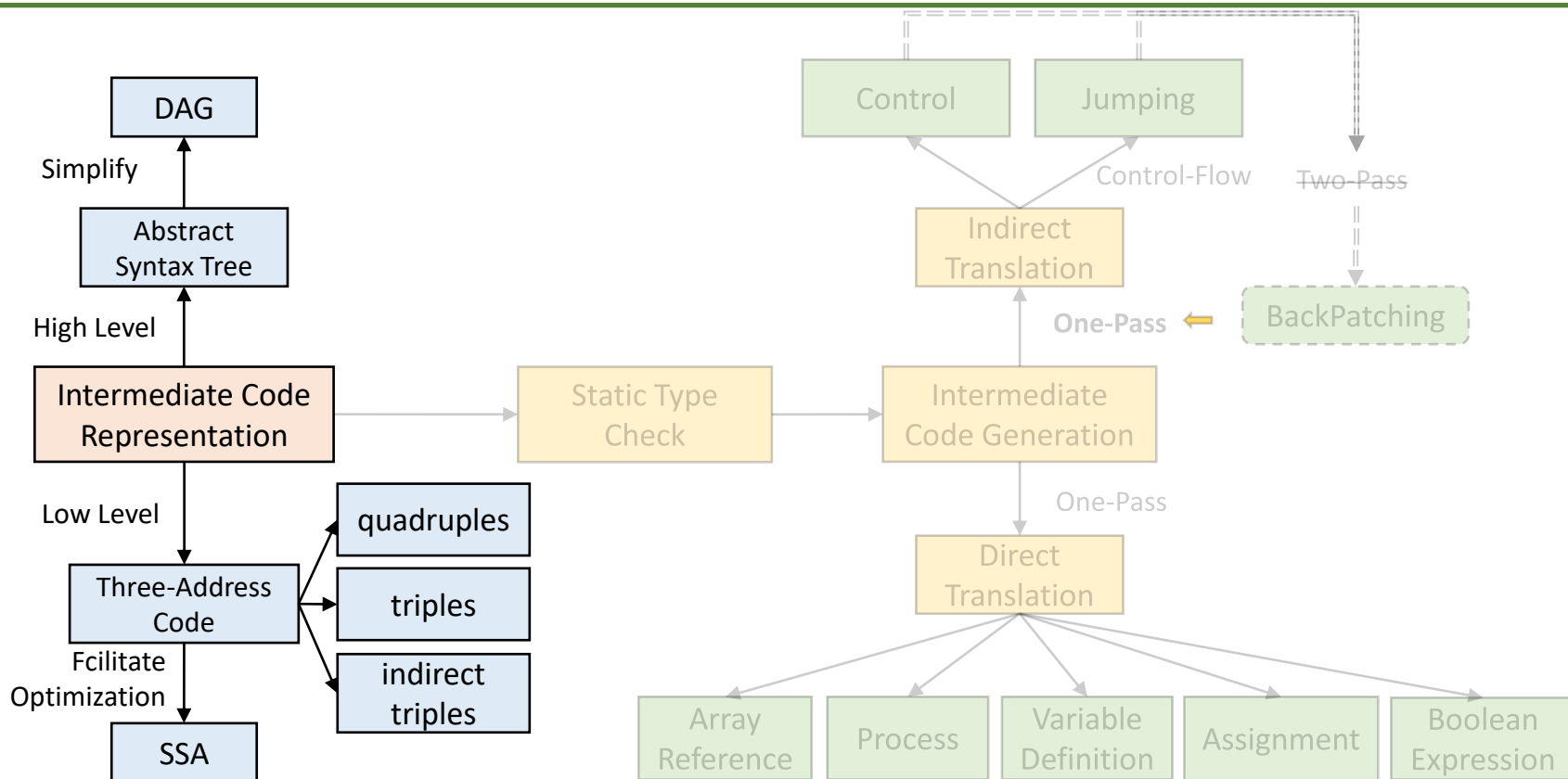
Multiple Level IR[不同层级的中间表示]



- Why multiple IRs?

2. Easier to add a new front-end (language) or back-end (ISA) [易于扩展]
 - Front-end: a new AST → low-level IR converter
 - Back-end: a new low-level IR → machine IR converter
 - Low-level IR acts as a bridge between multiple front-ends and back-ends, such that they can be reused
- If one IR (AST), and adding a new front-end...
 - ◆ Reimplement all compiler optimizations for new AST
 - ◆ A new AST → machine code converter for each ISA
 - ◆ Same goes for adding a new back-end

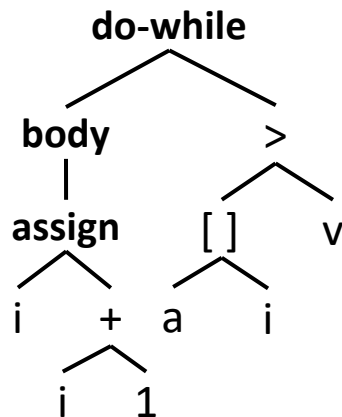
Intermediate Code[中间代码生成]



Intermediate Representation[中间表示]

- **Two Most important IR:**

- ◆ **Trees** [树形结构], including parse trees and (abstract) syntax trees [语法分析树和抽象语法树]
- ◆ **Directed Acyclic Graph (DAG)** [有向无环图]
- ◆ **Linear representations** [线性表示形式], especially “**three-address code**” [三地址代码]



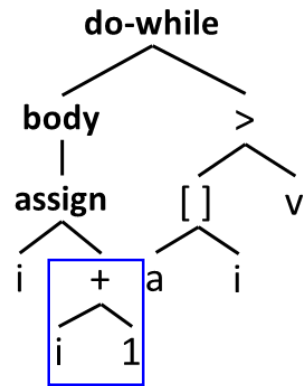
1: $i = i + 1$
2: $t_1 = a[i]$
3: if $t_1 < v$ goto 1

Fig. Two forms of intermediate code for “do $i=i+1$; while($a[i]<v$);”

Three-Address Code[三地址代码]



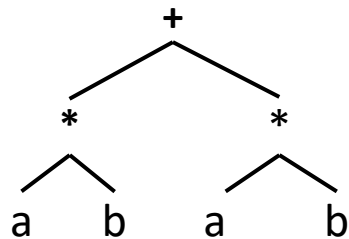
- At most one operator on the right side of an instruction in three-address code, e.g., $x + y * z$ translated into $t_1 = y * z$ $t_2 = x + t_1$
- Generic form is $X = Y \text{ op } Z$ [最多3个操作数]
 - ◆ where X, Y, Z can be variables, constants, or compiler-generated temporaries holding intermediate values.
- Characteristics [特性]
 - ◆ a linearized representation of a syntax tree or a DAG.
 - ◆ Assembly code for an “abstract machine”
 - ◆ Long expressions are converted to multiple instructions
 - ◆ Control flow statements are converted to jumps [控制流->跳转]
 - ◆ Machine independent
- Design goal: for easier machine-independent optimization



Three-Address Code[三地址代码]



- Example: $a * b + a * b$ is translated to
$$\begin{aligned}t_1 &= a * b \\t_2 &= a * b \\t_3 &= t_1 + t_2\end{aligned}$$



- ◆ t_1, t_2, t_3 are temporary variables
- ◆ Can be generated through a **depth-first traversal of AST**, and internal nodes in AST are translated to temporary variables
- The repetition of $a * b$ can be eliminated by a compiler optimization called **common subexpression elimination** (CSE)[通用子表达式消除]

$$\begin{aligned}t_1 &= a * b \\t_3 &= t_1 + t_1\end{aligned}$$

- Using 3-address code rather than AST makes it:
 - Easier to **spot optimization opportunities**
 - Easier to **manipulate IR**.

Addresses in three-address code[地址]

- An address can be one of the following:
 - ◆ **A name**[名字]. For convenience, we allow source-program names to appear as address in three-address code.
 - ◆ **A constant**[常量]. In practice, a compiler must deal with many different types of constants and variables.
 - ◆ **A compiler-generated temporary**[编译器生成的临时变量]. Creating a distinct name each time a temporary is needed [在每次需要临时变量时产生一个新名字是必要的], especially in optimizing compilers.
 - These temporaries **can be combined, if possible**, when registers are allocated to variables.

Three-Address Instruction Form[三地址指令形式]

1. Assignment instructions [二元赋值]

◆ $x = y \text{ op } z$, op is a **binary arithmetic**[双目算术符] or **logical operation** [逻辑运算符]

2. Assignment instructions [一元赋值]

◆ $x = \text{op } y$, where op is a **unary operation**[单目运算符]. Essential unary operations include unary minus, logical negation, shift operators, and conversion operators.

3. Copy instructions [复制]

◆ $x = y$, where x is assigned the value of y [把 y 的值赋给 x].

4. Unconditional Jump instructions [无条件转移指令]

◆ **goto L**: the three-address instruction with **label L** is the next to be executed.

5. Conditional Jump instructions [条件转移指令]

◆ **if x goto L if False x goto L if (x relop y) goto L**

◆ where **relop** is a relational operator such as $=$, $>$, $<$, etc.

Three-Address Instruction Form[三地址指令形式]

6. Procedure calls [程序调用]

- ◆ *param x* for parameters [参数传递];
- ◆ *call p,n* for procedure call – *p*: the procedure, *n*: the number of params.

<i>param x₁</i>	<i>param x1</i>
<i>param x₂</i>	<i>param x2</i>
...	<i>param x3 call</i>
<i>param x_n</i>	<i>foo, 3</i>
<i>call p,n</i>	

- ◆ Part of a call of the procedure $p(x_1, x_2, \dots, x_n)$.

7. Procedure calls return statement [程序调用返回]

- ◆ *return y*, *y* representing a returned value, is optional.

Three-Address Instruction Form[三地址指令形式]

8. Indexed copy instructions [带下标的复制指令]

- ◆ $x = y[i]$ $x[i] = y$
- ◆ $x = y[i]$ sets x to the value in the location i memory units beyond location y .
- ◆ $x[i] = y$ sets the contents of the location i units beyond x to the value of y .

9. Address and pointer assignments instructions. [地址及指针赋值指令]

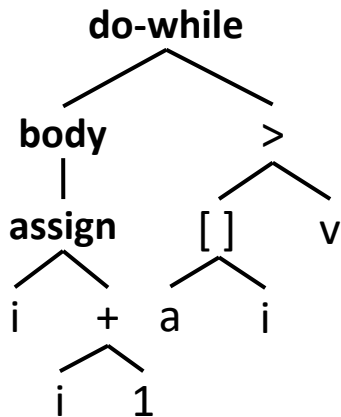
- ◆ $x = \&y$ $a\ pointer\ x\ is\ set\ to\ address\ of\ y$ [取址]
- ◆ $x = *y$ $x\ is\ set\ to\ the\ value\ of\ location\ pointed\ to\ by\ pointer\ y$ [y 地址指向的值赋给 x]
- ◆ $*x = y$ $location\ pointed\ to\ by\ x\ is\ assigned\ y$ [y 的值赋给 x 地址指向的位置]

Example



```
do {  
    i = i + 1;  
} while(a[i] < v)
```

Source program



Syntax tree

L: $t_1 = i + 1$

$i = t_1$

$t_2 = \&a$

$t_3 = \text{sizeof}(\text{int})$

$t_4 = t_3 * i$

$t_5 = t_2 + t_4$

$t_6 = *t_5$

if $t_6 < v$ goto L

$a[i]$

Three-address code

Implementation of Three-address Code[实现]

- Three representations. (and more)
 - ◆ **quadruples.** [四元式]
 - ◆ **triples.** [三元式]
 - ◆ **indirect triples.** [间接三元式]
- Trade-offs between **space**, **speed**, **ease of manipulation**.
- **Quadruples.** [四元式]
 - ◆ A quadruple has four fields, which we call **op**, **arg₁**, **arg₂**, **result**.
 - ◆ Examples & some exceptions:
 - $x = y + z \Rightarrow (+, y, z, x)$
 - Note that for a copy statement like **x = y**, **op is =**, while for most other operations, the assignment operator is implied. [隐含表示的]

- **Quadruples.** [四元式]

- ◆ Examples & some exceptions:

- $x = \text{minus } y \Rightarrow (\text{minus}, y, , x)$

- Instructions with unary[一元] operators like $x = \text{minus } y$ or $x = y$ do not use arg_2 .

- Operators like `param` use neither arg_2 nor `result`.

- $(\text{param}, x1, ,)$

- $\text{goto } L \Rightarrow (\text{goto}, , , L)$

- Conditional and Unconditional jumps put the target label in `result`.

- [条件或非条件转移指令将目标标号放入result字段]

Quadruples[四元式]



- **Example:** $a = b * (-c) + b * (-c)$

- ◆ The special operator `minus` is used to distinguish the unary minus operator, as in “-c”, from the binary minus operator, as in “b-c”.

$t_1 = \text{minus } c$

$t_2 = b * t_1$

$t_3 = \text{minus } c$

$t_4 = b * t_3$

$t_5 = t_2 + t_4$

$a = t_5$

	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

Three-address code

Quadruples

Triples[三元式]



- A triple has only three fields, which we call **op**, **arg₁**, **arg₂**.
 - ◆ Quadruple without the result field.
 - $x = y + z \Rightarrow (+, y, z)$
 - the assignment operator (**x=**) is implied
 - ◆ Result field is implicitly **index** of instruction.
 - ◆ Result referred to **by index** of instructions computing it.
 - See example in the next slide

Triples[三元式]



• **Example:** $a = b * (-c) + b * (-c)$

- ◆ The copy statement $a=t_5$ is encoded in the triple representation by placing a in the arg_1 field and (4) in the arg_2 field.

	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

Quadruples

	op	arg ₁	arg ₂
(0)	minus	c	
(1)	*	b	(0)
(2)	minus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	=	a	(4)

Triples

More About Triples[三元式]



- How can the following statements be expressed in triple?

- ◆ Array location (e.g. $x[i] = y$)
- ◆ Pointer location (e.g. $*(x+i) = y$)
- ◆ Struct field location (e.g. $x.i = y$)

- Example: $x[i] = y$

- ◆ Requires **two entries** in the triple structure.

- ◆ is translated to:

	op	arg ₁	arg ₂
(0)	[]	x	i
(1)	=	(0)	y

Complex LHS may require more triples to compute address

// Compute address of $x[i]$ location

// Assign y to that location

Problems About Triples[三元式]



- Problem with triples

- ◆ In code optimization, instructions are often moved around.
- ◆ With triples, the result of an operation is referred to by its position, so **moving an instruction may require us to change all references to that result**.

	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) (2)	+	t ₂	t₄ t ₂	t ₅
(5) (3)	=	t ₅		a

	op	arg ₁	arg ₂
(0)	minus	c	
(1)	*	b	(0)
(2)	minus	c	
(3)	*	b	(2)
(4) (2)	+	(1)	(3) (1)
(5) (3)	=	a	(4)

t₁ = a * b

~~t₂ = a * b~~

t₃ = t₁ + t₂

t₁ = a * b

t₃ = t₁ + t₁

CSE

Problems About Triples[三元式]



- Problem with triples

- ◆ In code optimization, instructions are often moved around.
- ◆ With triples, the result of an operation is referred to by its position, so **moving an instruction may require us to change all references to that result**.

	op	arg ₁	arg ₂	result
(0)	minus	c		t ₁
(1)	*	b	t ₁	t ₂
(2)	+	t ₂	t ₂	T ₅
(3)	=	t ₅		a

	op	arg ₁	arg ₂
(0)	minus	c	
(1)	*	b	(0)
(2)	+	(1)	(1)
(3)	=	a	(4)✗

Instruction (3) refers to (4) which is no longer there.

Three-Address Code[三地址代码] (Recap)



- Generic form is $X = Y \text{ op } Z$ [最多3个操作数]
- Three representations. (and more)

◆ quadruples [四元式]

□ $x = y + z \Rightarrow (+, y, z, x)$

◆ triples [三元式]

□ $x = y + z \Rightarrow (1) (+, y, z)$, use index for the result

◆ indirect triples [间接三元式]

□ use an [index list](#) for TAC execution

step	instruction
0	(0)
1	(1)
2	(2)
3	(0)
4	(3)
5	(4)

<i>a triple 'database'</i>			
index	op	arg ₁	arg ₂
(0)	+	a	b
(1)	*	(0)	c
(2)	=	x	(1)
(3)	/	d	(0)
(4)	=	y	(3)

• Single Static Assignment

- Every variable is assigned exactly once statically[仅一次]

Indirect Triples[间接三元式]



- The problem does not occur with **indirect triples**.
- Indirect triples consist of a listing of pointers to triples, rather than a listing of triples themselves.

Triples are stored in a triple '**database**'

step	instruction
0	(0)
1	(1)
2	(2)
3	(3)
4	(4)
5	(5)

index	op	arg ₁	arg ₂
(0)	minus	c	
(1)	*	b	(0)
(2)	minus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	=	a	(4)

Indirect Triples[间接三元式]



- After CSE, empty entries in database can be reused
 - ◆ Code in triple database becomes **non-contiguous** over time
 - ◆ That's fine since the listing is the code, not the database

Triples are stored in a triple **'database'**

<i>step</i>	<i>instruction</i>
0	(0)
1	(1)
2	(4)
3	(5)

index	op	arg ₁	arg ₂
(0)	minus	c	
(1)	*	b	(0)
(2)	empty		
(3)	empty		
(4)	+	(1)	(1)
(5)	=	a	(4)

Indirect Triples[间接三元式]



- Another Example: $x = (a+b)*c$; $y = d/(a+b)$

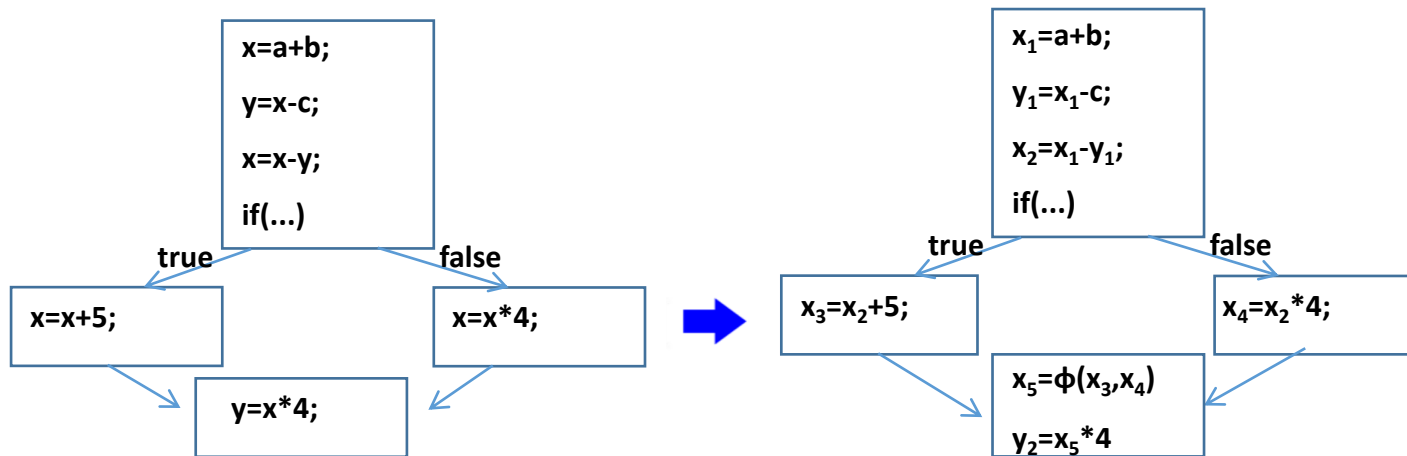
<i>step</i>	<i>instruction</i>
0	(0)
1	(1)
2	(2)
3	(0)
4	(3)
5	(4)

index	op	arg ₁	arg ₂
(0)	+	a	b
(1)	*	(0)	c
(2)	=	x	(1)
(3)	/	d	(0)
(4)	=	y	(3)

- With indirect triples, an optimizing compiler can **move an instruction by reordering the instruction list**, without affecting the triples themselves.

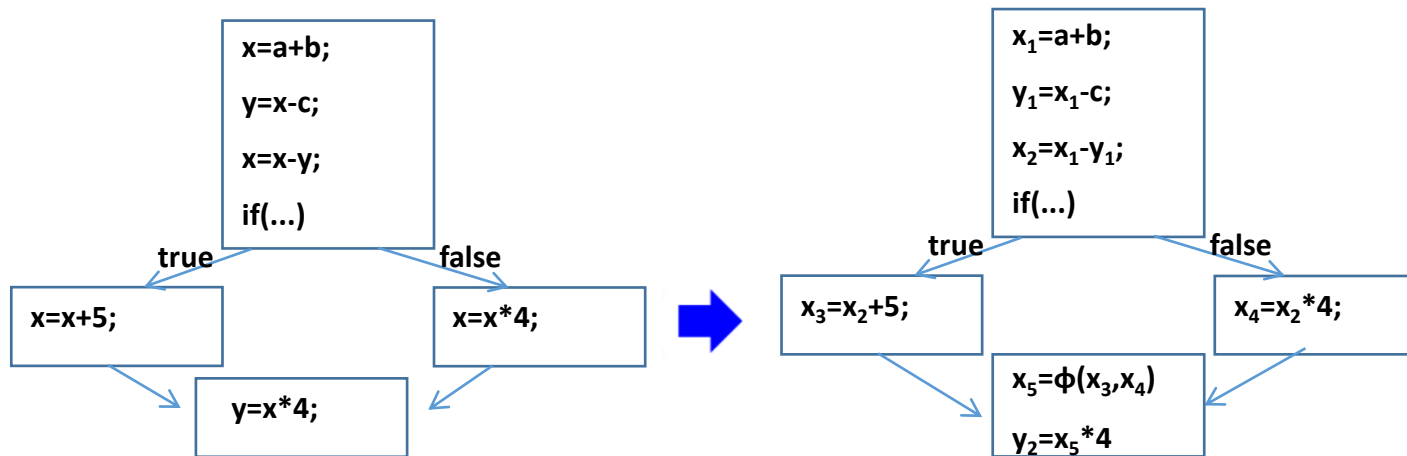
Single Static Assignment[静态单赋值]

- Every variable is assigned exactly once statically[仅一次]
 - ◆ Give variable a **different version name** on every assignment
 - e.g. $x \rightarrow x_1, x_2, \dots, x_5$ for each static assignment of x
 - ◆ Now value of each variable guaranteed **not to change**
 - ◆ On a control flow merge, ϕ -function combines two versions
 - e.g. $x_5 = \phi(x_3, x_4)$: means x_5 is either x_3 or x_4



Benefits of SSA

- **SSA** is an IR that facilitates code optimization
 - ◆ SSA tells you **when an optimization should not happen**
 - ◆ Suppose compiler performs CSE on previous example:
 - Without SSA, (incorrectly) tempted to eliminate second $x * 4$
 $x = x * 4; \quad y = x * 4; \quad \rightarrow \quad x = x * 4; \quad y = x;$
 - With SSA, $x_2 * 4$ and $x_5 * 4$ are clearly different values



Benefits of SSA (cont.)



- SSA is an IR that facilitates code optimizations
 - ◆ SSA tells you when an optimization should happen
 - ◆ Suppose compiler performs dead code elimination (DCE): (DCE removes code that computes dead values)
 - ◆ Without SSA, not very clear whether there are dead values
 - ◆ With SSA, x_1 is never used and clearly a dead value

```
x = a + b;  
x = c - d;  
y = x * b;
```



```
x1 = a + b;  
x2 = c - d;  
y1 = x2 * b;
```

- Why does SSA work so well with compiler optimizations?
 - ◆ SSA makes flow of values explicit in the IR
 - ◆ Without SSA, need a separate dataflow graph
 - ◆ Will discuss more in **Compiler Optimization** section



Syntax Directed Translation[语法制导翻译]

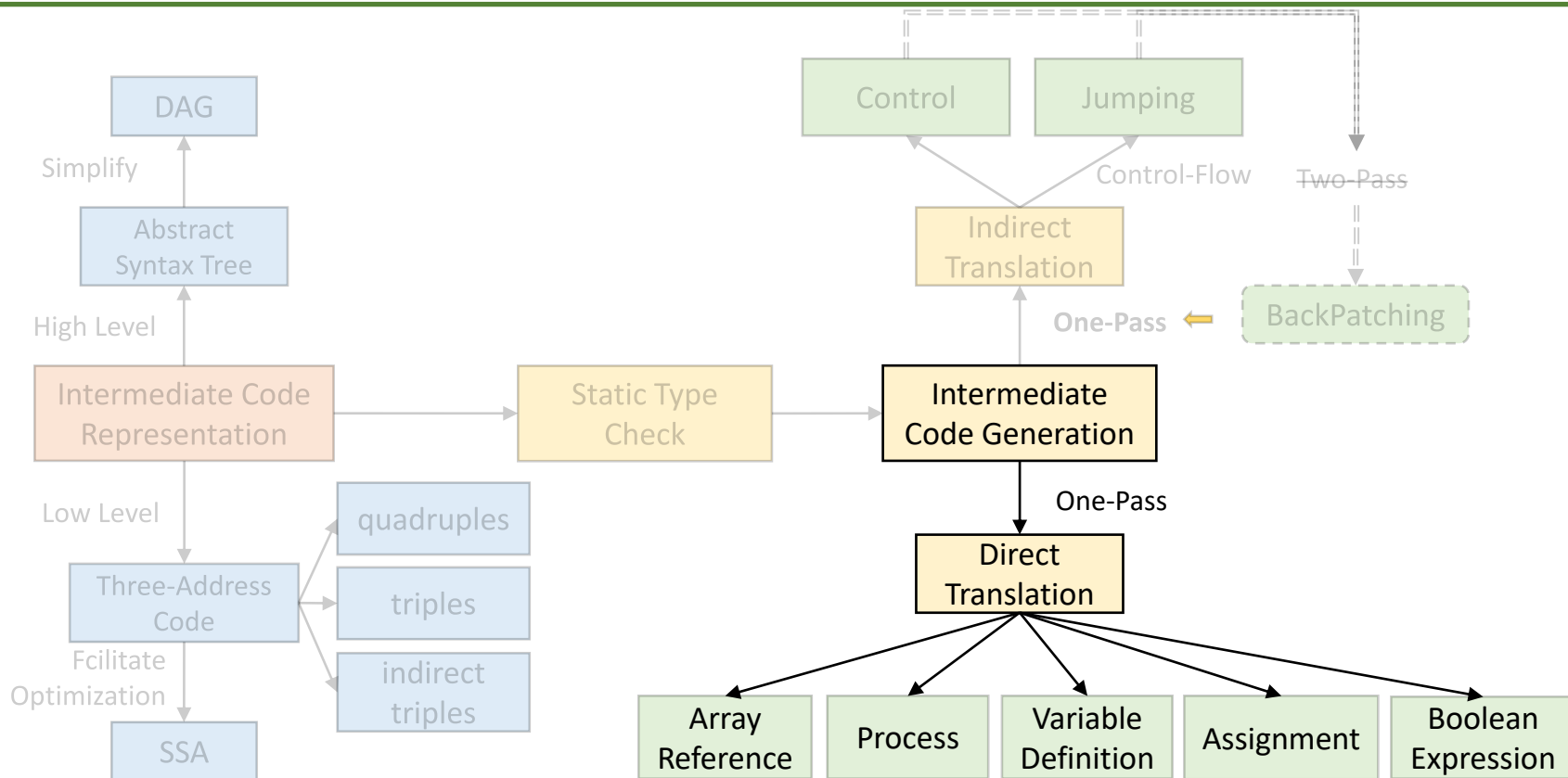


中山大學
SUN YAT-SEN UNIVERSITY

- **Syntax directed translation** can be used again for **code generation** [代码生成]
 - ◆ Code generation is **dependent on syntax/AST**
 - ◆ Code generation is to translate the **syntactic structures**
- What language structures do we need to translate?[翻译]
 - ◆ **Definitions** (variables, functions, ...)
 - ◆ **Assignment** statements
 - ◆ **Array** references
 - ◆ **Boolean** expressions
 - ◆ **Control flow** statements (if-then-else, for, etc)...
- We are going to use the following strategy:
 - ◆ Specify **SDD semantic rules** (without ordering)
 - ◆ Convert SDD rules to **SDT actions** (with ordering)



Intermediate Code[中间代码生成]



Code Generation Overview[代码生成]



- Program code is a collection of functions
 - ◆ By now, all functions are listed in symbol table
- Goal is to generate code for each function in that list
- Generating code for a function involves two steps:
 - ◆ Processing variable definitions[变量定义] -> Laying out variables in memory
 - ◆ Processing statements[语句] -> Generating instructions for statements
 - Assignments, array references, boolean expressions, control-flow statements
- We will start with variable definitions



Processing Variable Definitions[变量定义]



- To lay out a variable, both **location** and **width** are needed
 - ◆ **Location**: where variable is located in memory
 - ◆ **Width**: how much space variable takes up in memory
- Attributes for variable definition:
 - ◆ **T V**, e.g., `int x`;
 - ◆ **T**: non-terminal for **type** name
 - **T.type**: type (int, float, ...)
 - **T.width**: width of type in bytes (e.g., 4 for int)
 - ◆ **V**: non-terminal for **variable** name
 - **V.type**: type (int, float, ...)
 - **V.width**: width of variable according to type
 - **V.offset**: offset of variable in memory
- ◆ But **offset from what**...?



Variable Location from Offset



- **Naive method:** reserve a **big memory** section for all data
 - ◆ Size data section to be large enough to contain all variables
 - ◆ Location = **var offset** + **base of data section**
- Naive method **wastes a lot of memory**
 - ◆ Vars with limited scope **only live briefly in memory**
 - E.g., function variables last only for duration of call
- **Solution:** allocate memory for each scope[域内]
 - ◆ **Allocate** when **entering** scope, **free** when **exiting** scope
 - ◆ Variables in the same scope are allocated / freed together
 - ◆ Location = **var offset** + **base of scope memory** section
 - ◆ Will discuss more later in **Runtime Management**



Storage Layout of Variables in a Function



- When there are **multiple variables** defined in a function
 - ◆ Compiler **lays out variables** in memory **sequentially**
 - ◆ Current offset used to place variable **x** in memory
 - ▣ $\text{address}(x) \leftarrow \text{offset}$
 - ▣ $\text{offset} += \text{sizeof}(x.\text{type})$

```
void foo() {  
    int a;  
    int b;  
    long long c;  
    int d;  
}
```

Address		
0x0000	a	Offset = 0 Addr(a) \leftarrow 0
0x0004	b	Offset = 4 Addr(b) \leftarrow 4
0x0008	c	Offset = 8 Addr(c) \leftarrow 8
0x000c	c	
0x0010	d	Offset = 16 Addr(d) \leftarrow 16
		Offset = 20



More about Storage Layout



- Allocation alignment[对齐]

- ◆ Enforce $\text{addr}(x) \% \text{sizeof}(x.\text{type}) == 0$
- ◆ Most machine architectures are designed such that computation is most efficient at $\text{sizeof}(x.\text{type})$ boundaries
 - E.g. most machines are designed to load integer values at integer word boundaries
 - If not on word boundary, need to load two words with shift & concatenate → inefficient

```
void foo() {  
    char a;      // addr(a) = 0  
    int b;       // addr(b) = 1  
    int c;       // addr(c) = 5  
    long long d; // addr(d) = 9  
}
```



```
void foo() {  
    char a;      // addr(a) = 0  
    int b;       // addr(b) = 4  
    int c;       // addr(c) = 8  
    long long d; // addr(d) = 16  
}
```

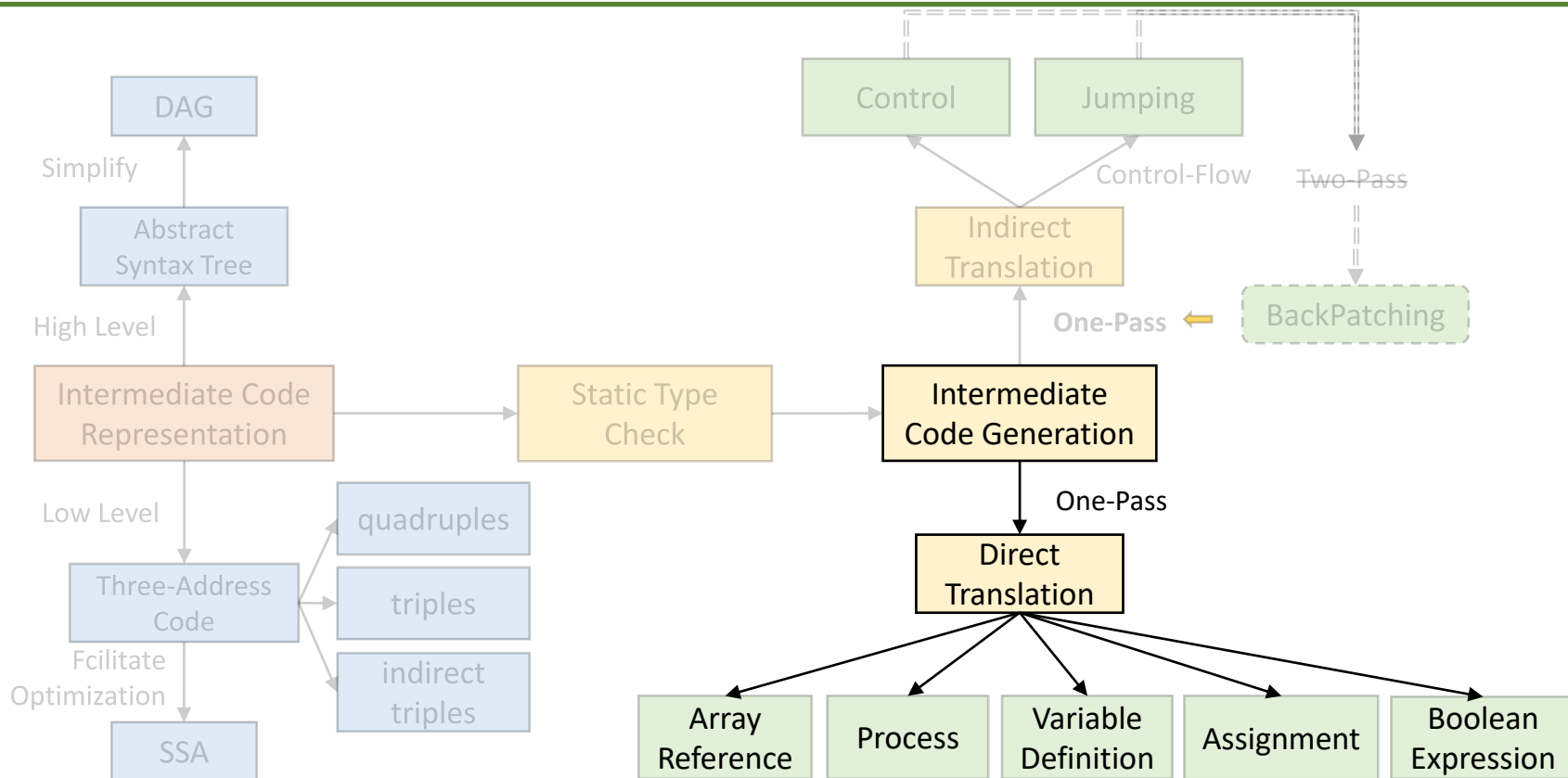
```
#include<stdio.h>  
  
struct{  
    char x; int y;  
}Test;  
  
int main() {  
    printf("%d\n",sizeof(Test));  
    return 0;  
}
```

What is the output?
Not aligned: 5, Aligned: 8



- We will use the **syntax-directed formalisms** to specify translation
 - ◆ Variable definitions[变量定义] -> Recall semantic analysis & symbol table
 - ◆ Assignment[赋值]
 - ◆ Array references[数组引用]
 - ◆ Boolean expressions[布尔表达式]
 - ◆ Control-flow statements[控制流语句]
- To generate three-address codes (TACs)
 - ◆ Lay out variables in memory
 - ◆ Generate TAC for any subexpressions or substatements
 - ◆ Using the result, generate TAC for the overall expression

Intermediate Code[中间代码生成]



- SSA (Single Static Assignment)
- We will use the **syntax-directed formalisms** to specify translation
 - ◆ Variable definitions[变量定义] -> Recall semantic analysis & symbol table
 - ◆ Assignment[赋值]
 - ◆ Array references[数组引用]
 - ◆ Boolean expressions[布尔表达式]
 - ◆ Control-flow statements[控制流语句]
- To generate three-address codes (TACs)
 - ◆ Lay out variables in memory
 - ◆ Generate TAC for any subexpressions or substatements
 - ◆ Using the result, generate TAC for the overall expression

CodeGen: Assignment Statement



- Translate into three-address code[赋值语句]
 - ◆ An expression with **more than one operator** will be translated into instructions with **at most one operator per instruction**
- Helper functions in translation
 - ◆ **lookup(id)**: search **id** in symbol table, return **null** if none
 - ◆ **emit()/gen()**: generate three-address IR
 - ◆ **newtemp()**: get a new temporary location

- ① $S \rightarrow id = E;$
- ② $E \rightarrow E_1 + E_2;$
- ③ $E \rightarrow - E_1$
- ④ $E \rightarrow (E_1)$
- ⑤ $E \rightarrow id$

Assignment statement:

$a = b + (-c)$

Three-address code:

$t_1 = \text{minus } c$

$t_2 = b + t_1$

$a = t_2$



SDT Translation of Assignment



- Attributes code and addr

- ◆ **S.code** and **E.code** denote the **TAC** for **S** and **E**, respectively
- ◆ **E.addr** denotes the **address** that will hold the value of **E** (can be a name, constant, or a compiler-generated temporary)

① $S \rightarrow id = E;$

② $E \rightarrow E_1 + E_2;$

③ $E \rightarrow - E_1$

④ $E \rightarrow (E_1)$

⑤ $E \rightarrow id$

Assignment statement:

$a = b + (-c)$

Three-address code:

$t_1 = \text{minus } c$

$t_2 = b + t_1$

$a = t_2$

① $S \rightarrow id = E; \{ p = \text{lookup}(id.\text{lexeme}); \text{if } !p \text{ then error; } S.\text{code} = E.\text{code} \mid \mid \text{gen}(p \text{ '=' } E.\text{addr}); \}$

② $E \rightarrow E_1 + E_2; \{ E.\text{addr} = \text{newtemp}(); E.\text{code} = E_1.\text{code} \mid \mid E_2.\text{code} \mid \mid \text{gen}(E.\text{addr} \text{ '=' } E_1.\text{addr} \text{ '+' } E_2.\text{addr}); \}$

③ $E \rightarrow - E_1 \{ E.\text{addr} = \text{newtemp}(); E.\text{code} = E_1.\text{code} \mid \mid \text{gen}(E.\text{addr} \text{ '=' 'minus' } E_1.\text{addr}); \}$

④ $E \rightarrow (E_1) \{ E.\text{addr} = E_1.\text{addr}; E.\text{code} = E_1.\text{code}; \}$

⑤ $E \rightarrow id \{ E.\text{addr} = \text{lookup}(id.\text{lexeme}); \text{if } !E.\text{addr} \text{ then error; } E.\text{code} = \text{' '}; \}$



Incremental Translation[增量翻译]



- Generate only the new three-address instructions
 - ◆ `gen()` **not only constructs** a three-address inst, it **appends the inst** to the sequence of insts generated so far

- ① $S \rightarrow id = E;$
- ② $E \rightarrow E_1 + E_2;$
- ③ $E \rightarrow - E_1$
- ④ $E \rightarrow (E_1)$
- ⑤ $E \rightarrow id$

$S \rightarrow id = E; \{ p = \text{lookup}(id.\text{lexeme}); \text{if } !p \text{ then error; } S.\text{code} = E.\text{code} \parallel \text{gen}(p \text{ '=' } E.\text{addr}); \}$

② $E \rightarrow E_1 + E_2; \{ E.\text{addr} = \text{newtemp}(); E.\text{code} = E_1.\text{code} \parallel E_2.\text{code} \parallel \text{gen}(E.\text{addr} \text{ '=' } E_1.\text{addr} \text{ '+' } E_2.\text{addr}); \}$

③ $E \rightarrow - E_1 \{ E.\text{addr} = \text{newtemp}(); E.\text{code} = E_1.\text{code} \parallel \text{gen}(E.\text{addr} \text{ '=' 'minus' } E_1.\text{addr}); \}$

④ $E \rightarrow (E_1) \{ E.\text{addr} = E_1.\text{addr}; E.\text{code} = E_1.\text{code}; \}$

⑤ $E \rightarrow id \{ E.\text{addr} = \text{lookup}(id.\text{lexeme}); \text{if } !E.\text{addr} \text{ then error; } E.\text{code} = ''; \}$



- We will use the **syntax-directed formalisms** to specify translation
 - ◆ Variable definitions[变量定义] -> Recall semantic analysis & symbol table
 - ◆ Assignment[赋值]
 - ◆ Array references[数组引用]
 - ◆ Boolean expressions[布尔表达式]
 - ◆ Control-flow statements[控制流语句]
- To generate three-address codes (TACs)
 - ◆ Lay out variables in memory
 - ◆ Generate TAC for any subexpressions or substatements
 - ◆ Using the result, generate TAC for the overall expression

CodeGen: Array Reference[数组引用]



- Primary problem in generating code for array references is to determine the address of element

- 1D array:

```
int A[N];
```

```
A[i] ++;
```

◆ **base**: address of the first element

◆ **width**: width of each element

□ $i * \text{width}$ is the offset



- Addressing an array element

◆ $\text{addr}(A[i]) = \text{base} + i \times \text{width}$



N-dimensional Array



- Laying out 2D array in 1D memory

```
int A[N1][N2]; /* int A[0..N1][0..N2] */  
A[i1][i2] ++;
```

- Organization by **row-major** or column-major

- ◆ C language uses row major (i.e., **row by row**)

- ◆ $\text{addr}(A[i_1, i_2]) = \text{base} + (i_1 \times \underbrace{N_2}_{W_1} \times \text{width} + i_2 \times \underbrace{\text{width}}_{W_2})$

第 i_1 行 第 i_2 列

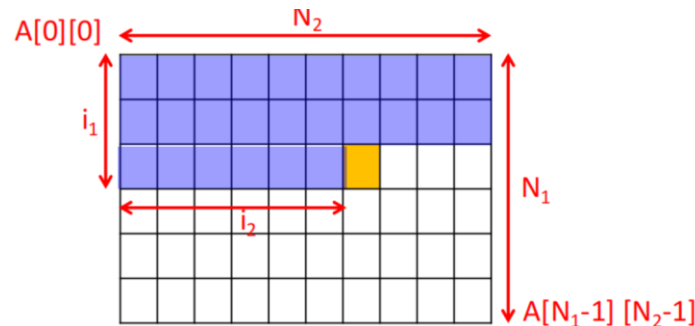
W_1

W_2

N_2 : 列数

- k-dimensional array

- ◆ $\text{addr}(A[i_1][i_2] \dots [i_k]) = \text{base} + i_1 \times W_1 + i_2 \times W_2 + \dots + i_k \times W_k$



Translation of Array References



- `Type(a) = array(10, int)`

◆ `c = a[i];`

$$\text{addr}(a[i]) = \text{base} + i * 4$$

$$t_1 = i * 4$$

$$t_2 = a[t_1]$$

$$c = t_2$$

3行5列

- `Type(a) = array(3, array(5, int))`

◆ `c = a[i1][i2];`

$$\text{addr}(A[i_1, i_2]) = \text{base} + (i_1 \times N_2 * \text{width} + i_2 \times \text{width})$$

$$\text{addr}(a[i_1][i_2]) = \text{base} + i_1 * 20 + i_2 * 4$$

$$t_1 = i_1 * 20$$

$$t_2 = i_2 * 4$$

$$t_3 = t_1 + t_2$$

$$t_4 = a[t_3]$$

$$c = t_4$$

3个5行8列

- `Type(a) = array(3, array(5, array(8, int)))`

◆ `c = a[i1][i2][i3]`

$$\text{addr}(a[i_1][i_2][i_3]) = \text{base} + i_1 * w_1 + i_2 * w_2 + i_3 * w_3$$

$$= \text{base} + i_1 * 160 + i_2 * 32 + i_3 * 4$$

160: a complete 2D array



Translation of Array References (cont.)



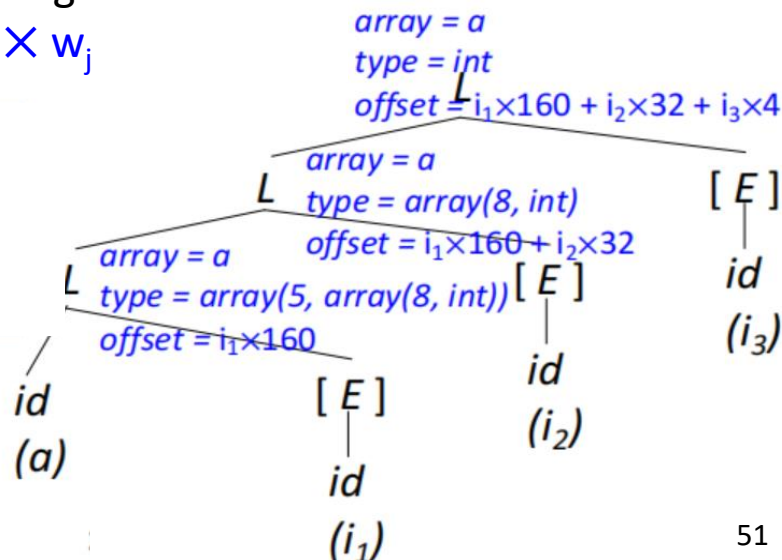
- $A[i_1][i_2][i_3]$, $\text{type}(a) = \text{array}(3, \text{array}(5, \text{array}(8, \text{int})))$
 - ◆ $L.\text{array}$: a pointer to the symbol-table entry for the array name
 - ▢ $L.\text{array}.\text{base}$ gives the array's base address
 - ◆ $L.\text{type}$: the type of the subarray generated by L
 - ◆ $L.\text{addr}$: a temporary that is used while computing the offset for the array referenced by summing the terms $i_j \times w_j$

① $S \rightarrow \text{id} = E; \mid L = E;$

② $E \rightarrow E_1 + E_2 \mid - E_1 \mid (E_1) \mid \text{id} \mid L$

③ $L \rightarrow \text{id}[E] \mid L_1[E]$

$\text{base} + i_1 \times w_1 + i_2 \times w_2 + \dots + i_k \times w_k$



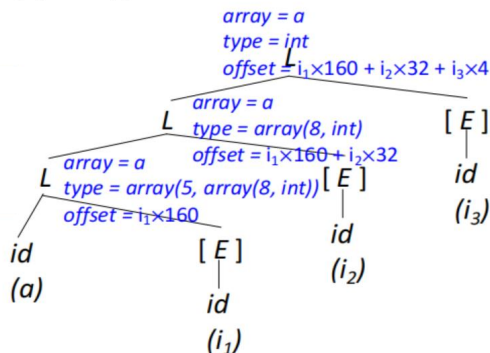
Translation of Array References (cont.)



- `A[i1][i2][i3]`, `type(a) = array(3, array(5, array(8, int)))`

- ① `S -> id = E; | L = E; { gen(L.array.base['L.addr'] '=' E.addr); }`
- ② `E -> E1 + E2 | - E1 | (E1) | id | L { E.addr = newtemp(); gen(E.addr '=' L.array.base['L.addr']); }`
- ③ `L -> L1 [E] { L.array = L1.array; L.type = L1.type.elem; t = newtemp(); gen(t '=' E.addr '*' L.type.width); L.addr = newtemp(); gen(L.addr '=' L1.addr '+' t); } |`

`id [E] { L.array = lookup(id.lexeme); if !L.array then error; L.type = L.array.type.elem; L.offset = newtemp(); gen(L.addr '=' E.addr '*' L.type.width); }`



$$t_1 = i_1 * 160$$

$$t_4 = i_3 * 4$$

$$t_2 = i_2 * 32$$

$$t_5 = t_3 + t_4$$

$$t_3 = t_1 + t_2$$

$$c = a[t_5]$$

- We will use the **syntax-directed formalisms** to specify translation
 - ◆ Variable definitions[变量定义] -> Recall semantic analysis & symbol table
 - ◆ Assignment[赋值]
 - ◆ Array references[数组引用]
 - ◆ Boolean expressions[布尔表达式]
 - ◆ Control-flow statements[控制流语句]
- To generate three-address codes (TACs)
 - ◆ Lay out variables in memory
 - ◆ Generate TAC for any subexpressions or substatements
 - ◆ Using the result, generate TAC for the overall expression

CodeGen: Boolean Expressions



- Boolean expression: **a op b**
 - ◆ where op can be `<`, `<=`, `!=`, `>`, `>=`, `&&`, `||`, `==`, ...
- Short-circuit evaluation[短路计算]: to skip evaluation of the rest of a boolean expression once a boolean value is known
 - ◆ Given following C code: `if (flag || foo()) { bar(); };`
 - If `flag` is true, `foo()` never executes
 - Equivalent to: `if (flag) { bar(); } else if (foo()) { bar(); };`
 - ◆ Given following C code: `if (flag && foo()) { bar(); };`
 - If `flag` is false, `foo()` never executes
 - Equivalent to: `if (!flag) { } else if (foo()) { bar(); };`
- For control flow, boolean operators is translated to jump statements



Boolean Expressions



- Computed just like any other arithmetic expression

$E \rightarrow (a < b) \text{ or } (c < d \text{ and } e < f)$

$t_1 = a < b$

$t_2 = c < d$

$t_3 = e < f$

$t_4 = t_2 \ \&\& \ t_3$

$t_5 = t_1 \ || \ t_4$

- Then, used in control-flow statements

◆ **S.next**: label for code generated after S

$S \rightarrow \text{if } E \ S_1$

$\text{if } (!t_5) \text{ goto } S.\text{next}$

$S_1.\text{code}$

$S.\text{next}: \dots$



Boolean Expressions



- Implemented via a series of **jumps**[利用跳转]
 - ◆ converted to **two gotos** (true and false)
 - ◆ Remaining evaluation skipped when result known in middle
- Example
 - ◆ **E.true**: label for code to execute when E is 'true'
 - ◆ **E.false**: label for code to execute when E is 'false'
 - ◆ E.g. if above is condition for a do-while loop
 - **E.true** would be label at **beginning of loop body**
 - **E.false** would be label for **code after the loop**

$E \rightarrow (a < b) \text{ or } (c < d \text{ and } e < f)$

if $(a < b)$ goto **E.true**

goto L_1

L_1 : if $(c < d)$ goto L_2

goto **E.false**

L_2 : if $(e < f)$ goto **E.true**

goto **E.false**

E为真: 只要a < b真

a < b假: 继续评估

a < b假、c < d真: 继续评估

E为假: a < b假, c < d假

E为真: a < b假, c < d真, e < f真

E为假: a < b假, c < d真, e < f假



Boolean Expressions



- **Boolean expressions** are composed of
 - ◆ **Boolean operators** ($=$, $\&\&$, $||$) applied to elements that are **boolean variables** or **relational expressions** ($E1 \text{ relop } E2$)
- Computed just like any other arithmetic expression

$E \rightarrow (a < b) \text{ or } (c < d \text{ and } e < f)$

$t_1 = a < b$

$t_2 = c < d$

$t_3 = e < f$

$t_4 = t_2 \&\& t_3$

$t_5 = t_1 || t_4$

- Then, used in control-flow statements
 - ◆ **S.next**: label for code generated after **S**

$S \rightarrow \text{if } E \text{ } S_1$

if ($!t_5$) goto S.next

S_1 .code

S.next: ...



SDT Translation of Booleans[布尔表达式]



- $B \rightarrow B_1 \parallel B_2$

- ◆ $B_1.\text{true}$ is same as $B.\text{true}$, B_2 must be evaluated if B_1 is false[B1假才评估B2]
- ◆ The true and false exits of B_2 are the same as B [B2与B同真假]

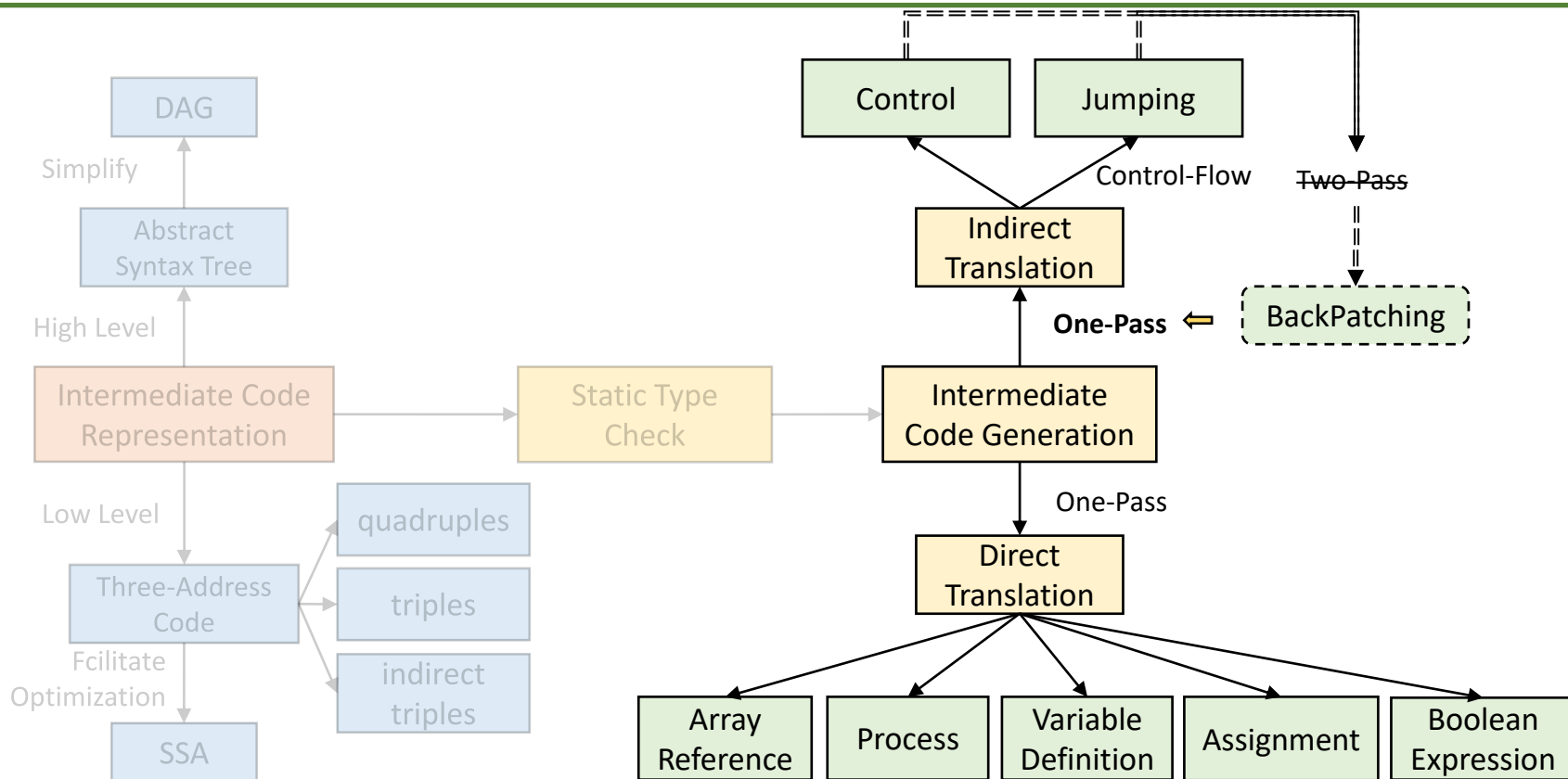
- $B \rightarrow E_1 \text{ relop } E_2$

- ◆ Translated directly into a comparison TAC inst with jumps

- ① $B \rightarrow \{ B_1.\text{true} = B.\text{true}; B_1.\text{false} = \text{newlabel}(); \} B_1 \parallel \{ \text{label}(B_1.\text{false}); B_2.\text{true} = B.\text{true}; B_2.\text{false} = B.\text{false}; \} B_2$
- ② $B \rightarrow \{ B_1.\text{true} = \text{newlabel}(); B_1.\text{false} = B.\text{false}; \} B_1 \&\& \{ \text{label}(B_1.\text{true}); B_2.\text{true} = B.\text{true}; B_2.\text{false} = B.\text{false}; \} B_2$
- ③ $B \rightarrow E_1 \text{ relop } E_2 \{ \text{gen}(\text{'if' } E_1.\text{addr relop } E_2.\text{addr 'goto' } B.\text{true}); \text{gen}(\text{'goto' } B.\text{false}; \}$
- ④ $B \rightarrow ! \{ B_1.\text{true} = B.\text{false}; B_1.\text{false} = B.\text{true}; \} B_1$
- ⑤ $B \rightarrow \text{true} \{ \text{gen}(\text{'goto' } B.\text{true}; \}$
- ⑥ $B \rightarrow \text{false} \{ \text{gen}(\text{'goto' } B.\text{false}; \}$



Intermediate Code[中间代码生成]



- We will use the **syntax-directed formalisms** to specify translation
 - ◆ Variable definitions[变量定义] -> Recall semantic analysis & symbol table
 - ◆ Assignment[赋值]
 - ◆ Array references[数组引用]
 - ◆ Boolean expressions[布尔表达式]
 - ◆ Control-flow statements[控制流语句]
- To generate three-address codes (TACs)
 - ◆ Lay out variables in memory
 - ◆ Generate TAC for any subexpressions or substatements
 - ◆ Using the result, generate TAC for the overall expression

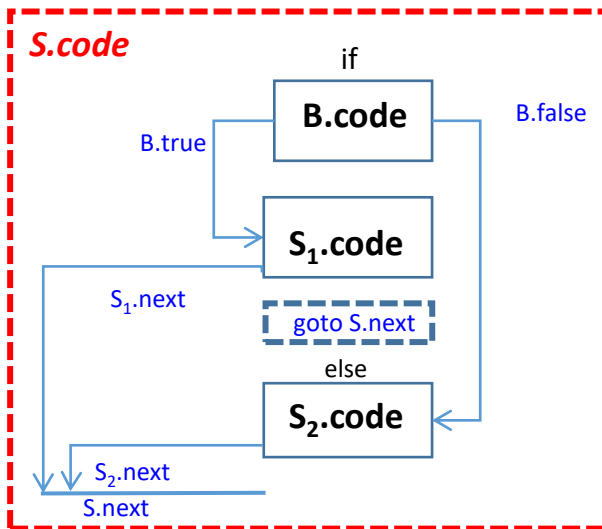
- Inherited attributes[继承属性]

- ◆ **B.true**: the label to which control flows if B is true (依赖于S1)
- ◆ **B.false**: the label to which control flows if B is false (依赖于S2)
- ◆ **S.next**: a label for the instruction immediately after the code of S

① $S \rightarrow \text{if} (B) S_1$

② $S \rightarrow \text{if} (B) S_1 \text{ else } S_2$

③ $S \rightarrow \text{while} (B) S_1$



Translation of Controls



- **Helper functions**[辅助函数]

- ◆ **newlabel()**: creates a new label

- ◆ **label(L)**: attaches label L to the next three address inst to be generated

① $S \rightarrow \text{if} (B) S_1$

② $S \rightarrow \text{if} (B) S_1 \text{ else } S_2$

③ $S \rightarrow \text{while} (B) S_1$

```
S -> if { B.true = newlabel();
```

```
  B.false = newlabel(); }
```

```
  ( B ) { label(B.true); S1.next = S.next; }
```

```
  S1 { gen('goto' S.next); }
```

```
  else { label(B.false); S2.next = S.next; } S2
```

If false B goto B.false

B.true:

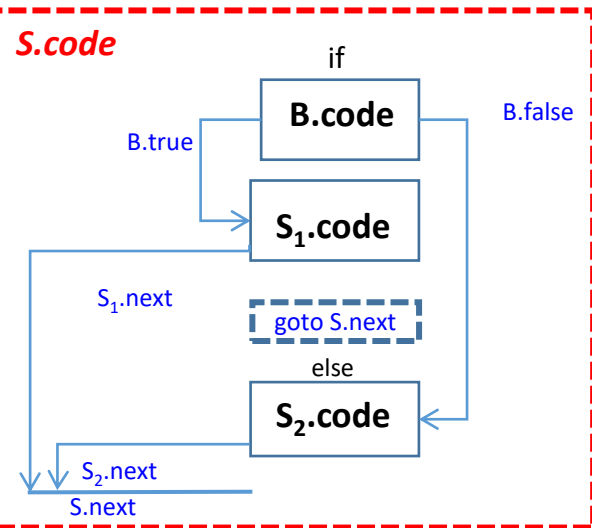
S₁.code

goto S.next

B.false:

S₂.code

S.next:



Translation of Controls (cont.)



① $S \rightarrow \text{if } (B) S_1$

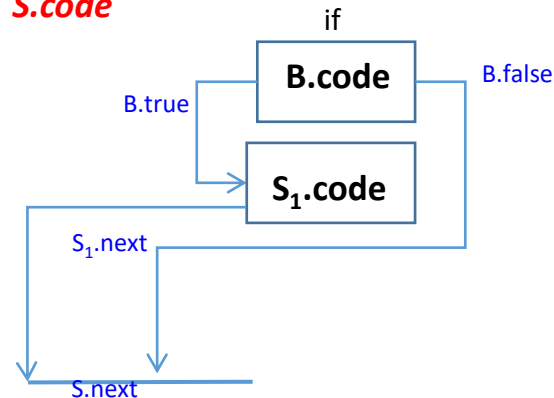
② $S \rightarrow \text{if } (B) S_1 \text{ else } S_2$

③ $S \rightarrow \text{while } (B) S_1$

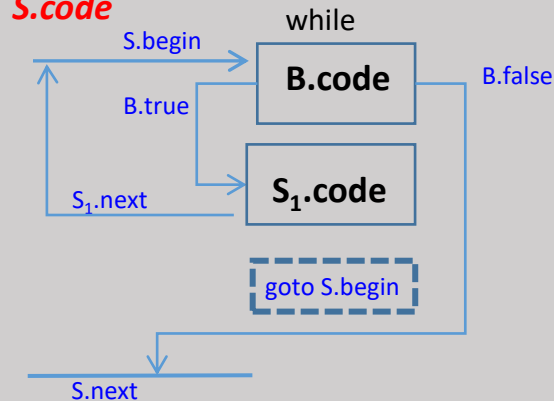
```
S -> if { B.true = newlabel(); B.false = S.next; }  
      (B) { label(B.true); S1.next = S.next; }  
      S1
```

```
S -> while { S.begin = newlabel();  
            label(S.begin);  
            B.true = newlabel();  
            B.false = S.next; }  
      (B) { label(B.true); S1.next = S.begin; }  
      S1 { gen('goto' S.begin); }
```

S.code



S.code



Jumping Labels[跳转标签]



- Key of generating code for Boolean and flow-control: **matching** a **jump** inst with the **target of jump**[跳转指令匹配到跳转目标]
 - ◆ Forward jump: a jump to an instruction in below
 - ◆ Label for jump target has not yet been generated

```
B -> { B1.true = newlabel(); B1.false = B.false; } B1 && { label(B1.true); B2.true = B.true; B2.false = B.false; } B2  
S -> if { B.true = newlabel(); B.false = S.next; } ( B ) { label(B.true); S1.next = S.next; } S1
```



Handle Jumping Labels



- Idea: generate code using dummy labels first, then **patch** them with addresses later after labels are generated
- **Two-pass** approach: requires two scans of code
 - ◆ Pass 1:
 - Generate code creating **dummy labels** for forward jumps. (Insert label into a **hashtable**)
 - When label emitted, **record address** in hashtable
 - ◆ Pass 2:
 - **Replace dummy labels** with target addresses (Use previously built hashtable for mapping)
- **One-pass** approach
 - ◆ Generate **holes** when forward jumping to an un-generated label
 - ◆ **Maintain a list of holes** for that label
 - ◆ **Fill in holes with addresses** when label generated later on



One-Pass Code Generation[单遍生成]



- One Pass Generation takes less time along with LR parser
- However, given the example below, we need to know the address of **E2.label** to insert jumps in **E1**
 - ◆ E.g. **E1.false = E2.label** in $E \rightarrow E1 \mid E2$
- Solution: **Backpatching**[回填]
 - ◆ Leave **holes** in IR in place of forward jump addresses
 - ◆ Record indices of jump instructions in **a hole list**
 - ◆ When target address of label for jump is eventually known, **backpatch holes** using the hole list for that particular label



- Synthesized attributes[综合属性]. $S \rightarrow \text{if } (B) S1$
 - ◆ **B.truelist**: a list of jump or conditional jump insts into which we must insert the label to which control goes if B is true[B为真时控制流应该转向的指令的标号]
 - ◆ **B.falselist**: a list of insts that eventually get the label to which control goes when B is false[B为假时控制流应该转向的指令的标号]
 - ◆ **S.nextlist**: a list of jumps to the inst immediately following the code for S[紧跟在S代码之后的指令的标号]
- Helper functions to implement backpatching
 - ◆ **makelist(i)**: creates a new list out of statement index i
 - ◆ **merge(p1, p2)**: returns merged list of $p1$ and $p2$
 - ◆ **backpatch(p, i)**: fill holes in list p with statement index i

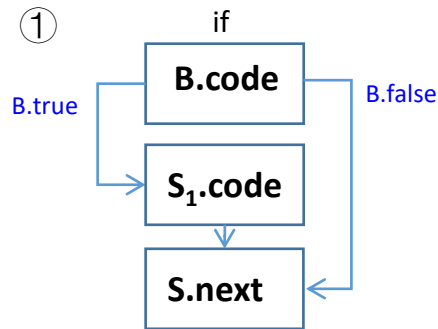
Backpatching of Control-Flow



- Slightly modify the grammar

```
① S -> if (B) M S1 { backpatch(B.truelist, M.inst);  
                        S.nextlist = merge(B.falselist, S1.nextlist); }  
  
② S -> if (B) M1 S1 N else M2 S2 { backpatch(B.truelist, M1.inst);  
                                         backpatch(B.falselist, M2.inst);  
                                         temp = merge(S1.nextlist, N.nextlist);  
                                         S.nextlist = merge(temp, S2.nextlist); }  
  
③ S -> while M1 (B) M2 S1 {backpatch(B.truelist, M2.inst);  
                               backpatch(S1.nextlist, M1.inst);  
                               S.nextlist = B.falselist);  
                               gen('goto' M1.inst); }  
  
④ M -> ε { M.inst = nextinst; }  
⑤ N -> ε { N.nextlist = makelist(nextinst); gen('goto _'); }
```

- makelist(i)**: creates a new list out of statement index i
- merge(p1, p2)**: returns merged list of p1 and p2
- backpatch(p, i)**: fill holes in list p with statement index i



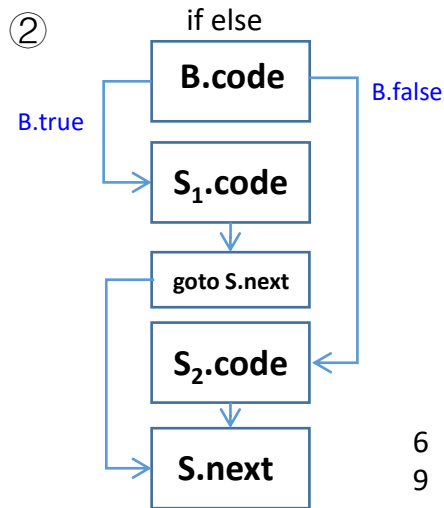
Backpatching of Control-Flow



- Slightly modify the grammar

```
① S -> if (B) M S1 { backpatch(B.truelist, M.inst);  
                        S.nextlist = merge(B.falselist, S1.nextlist); }  
  
② S -> if (B) M1 S1 N else M2 S2 { backpatch(B.truelist, M1.inst);  
                                         backpatch(B.falselist, M2.inst);  
                                         temp = merge(S1.nextlist, N.nextlist);  
                                         S.nextlist = merge(temp, S2.nextlist); }  
  
③ S -> while M1 (B) M2 S1 { backpatch(B.truelist, M2.inst);  
                               backpatch(S1.nextlist, M1.inst);  
                               S.nextlist = B.falselist);  
                               gen('goto' M1.inst); }  
  
④ M -> ε { M.inst = nextinst; }  
⑤ N -> ε { N.nextlist = makelist(nextinst); gen('goto _'); }
```

- makelist(i)**: creates a new list out of statement index i
- merge(p1, p2)**: returns merged list of p1 and p2
- backpatch(p, i)**: fill holes in list p with statement index i



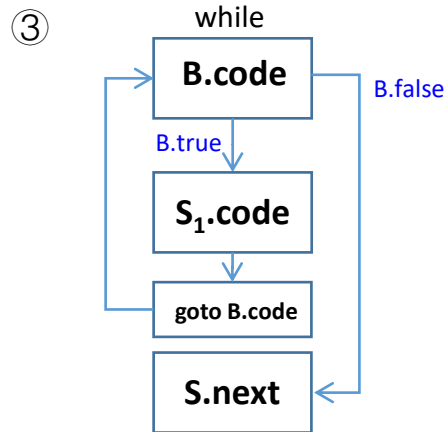
Backpatching of Control-Flow



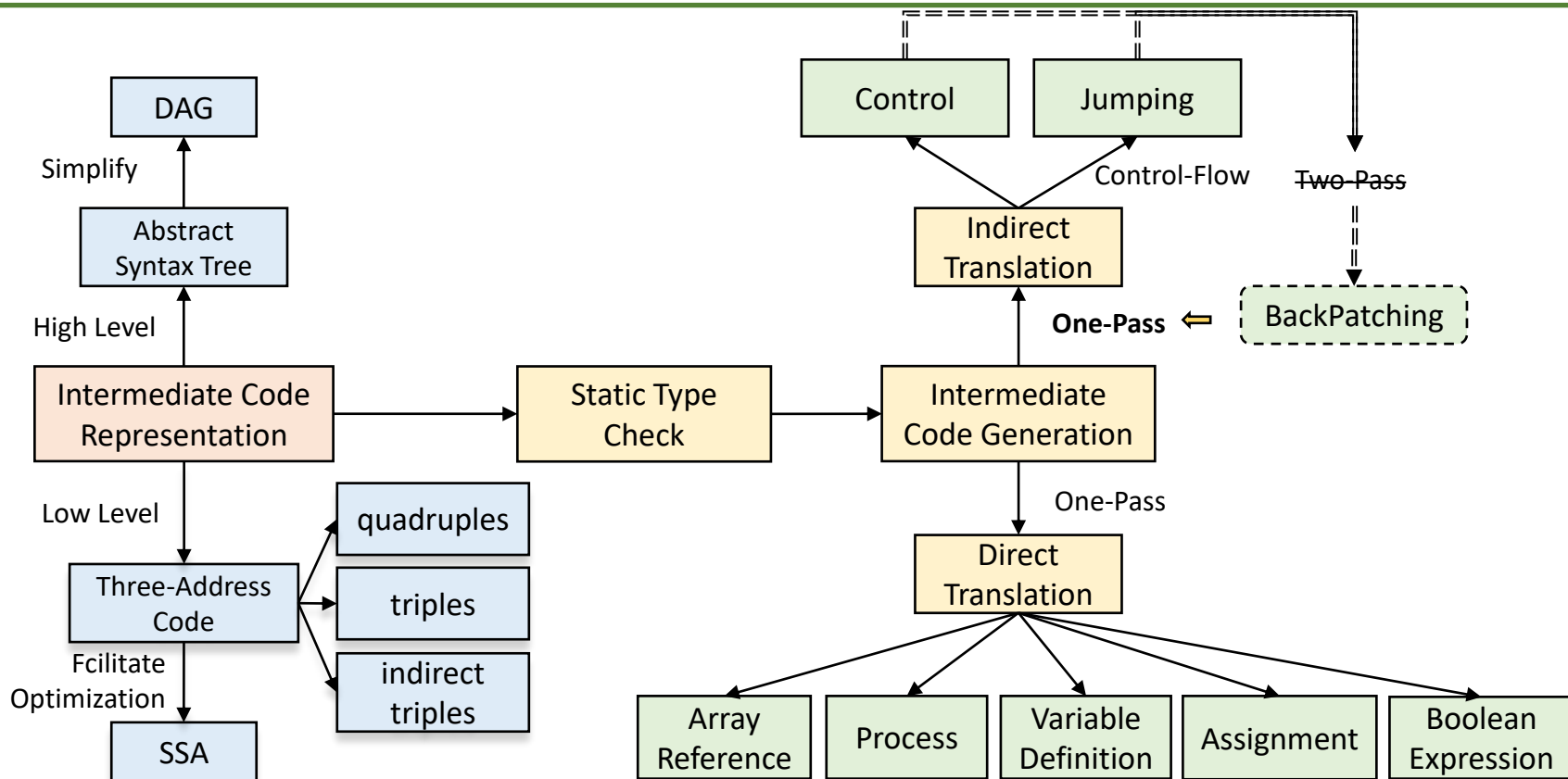
- Slightly modify the grammar

```
① S -> if (B) M S1 { backpatch(B.truelist, M.inst);  
                        S.nextlist = merge(B.falselist, S1.nextlist); }  
  
② S -> if (B) M1 S1 N else M2 S2 { backpatch(B.truelist, M1.inst);  
                                        backpatch(B.falselist, M2.inst);  
                                        temp = merge(S1.nextlist, N.nextlist);  
                                        S.nextlist = merge(temp, S2.nextlist); }  
  
③ S -> while M1 (B) M2 S1 {backpatch(B.truelist, M2.inst);  
                                backpatch(S1.nextlist, M1.inst);  
                                S.nextlist = B.falselist);  
                                gen('goto' M1.inst); }  
  
④ M -> ε { M.inst = nextinst; }  
⑤ N -> ε { N.nextlist = makelist(nextinst); gen('goto _'); }
```

- makelist(i)**: creates a new list out of statement index i
- merge(p1, p2)**: returns merged list of p1 and p2
- backpatch(p, i)**: fill holes in list p with statement index i



Intermediate Code[中间代码生成]



Summary



- Three-Address Code: $X = Y \text{ op } Z$
 - ◆ Three representations
 - quadruples [四元式]
 - triples [三元式]
 - indirect triples [间接三元式]
- Single Static Assignment
- Code generation: TAC instructions using syntax directed translation
 - ◆ Variable definitions[变量定义]
 - ◆ Expressions and statements
 - Assignment[赋值]
 - Array references[数组引用]
 - Boolean expressions[布尔表达式]
 - Control-flow[控制流]



Further Reading



- Dragon Book, 2nd Edition

- ◆ 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.

