南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

CS323 Lab 13

Yepang Liu

liuyp1@sustech.edu.cn

Outline

- Project phase 3 tutorial

TAC Instructions

Instruction	Description
`LABEL x :`	define a label `x`
`FUNCTION f :`	define a function `f`
`x := y`	assign value of `y` to `x`
`x := y + z`	arithmetic addition
`x := y - z`	arithmetic subtraction
`x := y * z`	arithmetic multiplication
`x := y / z`	arithmetic division
`x := &y`	assign address of `y` to `x`
`x := *y`	assign value stored in address `y` to `x`
`*x := y`	copy value `y` to address `x`
`GOTO x`	unconditional jump to label `x`
`IF x [relop] y GOTO z`	if the condition (binary boolean) is true, jump to label `z`
`RETURN x`	exit the current function and return value `x`
`DEC x [size]`	allocate space pointed by `x`, `size` must be a multiple of 4
`PARAM x`	declare a function parameter
`ARG x`	pass argument `x`
`x := CALL f`	call a function, assign the return value to `x`
`READ x`	read `x` from console
`WRITE x`	print the value of `x` to console

READ and WRITE are designed for user interaction. In our IR simulator, READ statement can read an integer from the console, and WRITE prints an integer value to the console

Instruction Representation

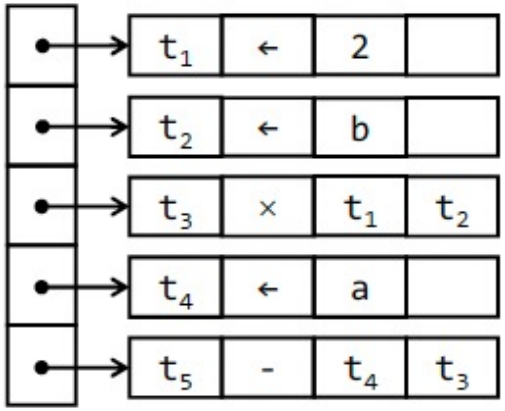
- The overall process of intermediate code generation:
  - Generate three address code during parsing (or after parsing, with a separate pass) and save it in memory
  - Perform possible optimizations
  - Output the intermediate code

Static Array Style (Quadruples)

Target	*Op*	*Arg$_1$*	*Arg$_2$*
t_1	←	2	
t_2	←	b	
t_3	×	t_1	t_2
t_4	←	a	
t_5	-	t_4	t_3

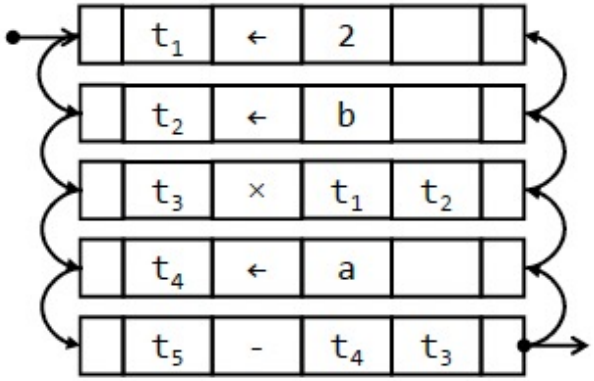
- The most straightforward implementation
- Disadvantages:
  - Low efficiency when moving instructions (imagine moving the first instruction to the end)
  - Code size is limited by the array's length

Pointer Array Style (Enhance Quadruples)



- Also, straightforward implementation
- Disadvantages:
  - Code size is still limited by pointer array's length
  - Better than static array style in that moving instructions only need to manipulate pointers (but still not very efficient)

Doubly Linked List Style



- Implementation is more complex
- Advantages:
  - Code size is only limited by memory capacity
  - Instruction insertion/replacement/movement are all very efficient

Intermediate Code Example

```
int main() {
    int n;
    n = read();
    if (n > 0) write(1);
    else if (n < 0) write (-1);
    else write(0);
    return 0;
}
```

```
FUNCTION main :
READ t1
v1 := t1
t2 := #0
IF v1 > t2 GOTO label1
GOTO label2
LABEL label1 :
t3 := #1
WRITE t3
GOTO label3
LABEL label2 :
t4 := #0
IF v1 < t4 GOTO label4
GOTO label5
LABEL label4 :
t5 := #1
t6 := #0 - t5
WRITE t6
GOTO label6
LABEL label5 :
t7 := #0
WRITE t7
LABEL label6 :
LABEL label3 :
t8 := #0
RETURN t
```

Immediate value 0

Our sample output adopts the naming convention that variable names follow the pattern *tn* or *vn*, and *labeln* for label names. However, this is not the only way. Your compiler can generate any valid names as you wish.

Two Translation Strategies

1. Augment the semantic actions in project phase 2 and generate intermediate code while doing semantic analysis
   - Advantage: efficiency (only one pass)
   - Disadvantage: fragmented code, lack of modularity, difficult to implement
2. Write a separate module for translation. The module traverses the parse tree (in preorder) to generate code
   - Advantage: better modularity
   - Disadvantage: slower, requires two passes

Arithmetic Expressions

For each non-terminal, we will need to implement such a function

`translate_Exp(Exp, place)`: returns three-address code for the node Exp and its children nodes; `place` is the address that stores the evaluation result of the expression

Exp_1 PLUS Exp_2

```
t1 = new_place()
t2 = new_place()
code1 = translate_Exp(Exp1, t1)
code2 = translate_Exp(Exp2, t2)
code3 = [place := t1 + t2]
return code1 + code2 + code3
```

Create temporary addresses

Translate subexpressions*

Store evaluation result

Concatenate code

* The translation order of the two subexpressions does not matter. Here, we follow a typical left-to-right order.

Assignment

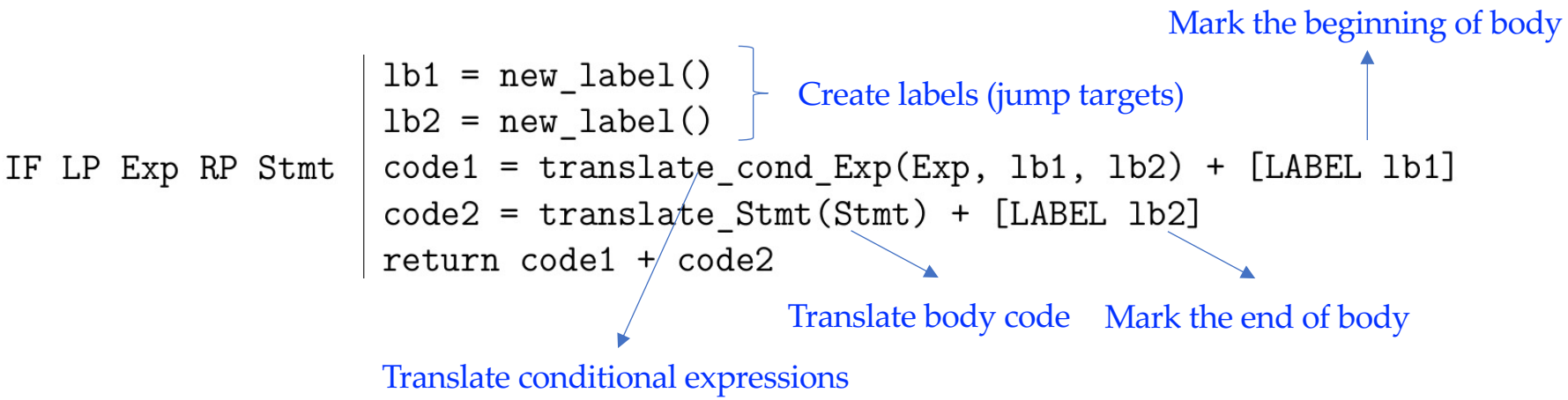
For each non-terminal, we will need to implement such a function

`translate_Exp(Exp, place)`: returns three-address code for the node Exp and its children nodes; `place` is the address that stores the evaluation result of the expression

Exp_1 ASSIGN Exp_2	`variable = symtab_lookup(`Exp_1`.ID)` → Find symbol information
	`tp = new_place()`
	`code1 = translate_Exp(`Exp_2`, tp)` } Translate the right-hand side
	`code2 = [variable.name := tp]`
	`code3 = [place := variable.name]` } Value copies (optimizations?)
	`return code1 + code2 + code3`

Suppose Exp_1 is a simple case: an identifier

Conditional Statements



Conditional Expressions

```
translate_Cond_Exp(Exp, lb_t, lb_f):
```

Exp_1 EQ Exp_2

```
t1 = new_place()
t2 = new_place()
code1 = translate_Exp(Exp1, t1)
code2 = translate_Exp(Exp2, t2)
code3 = [IF t1 == t2 GOTO lb_t] + [GOTO lb_f]
return code1 + code2 + code3
```

Create temporary addresses to hold subexpression values

Translate subexpressions

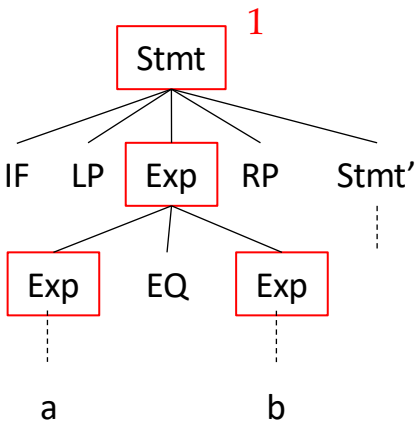
Generate jumps

Example

```
if(a == b) { }
```

`translate_Stmt(Stmt)`:

IF LP Exp RP Stmt	lb1 = new_label() lb2 = new_label() code1 = translate_cond_Exp(Exp, lb1, lb2) + [LABEL lb1] code2 = translate_Stmt(Stmt) + [LABEL lb2] return code1 + code2

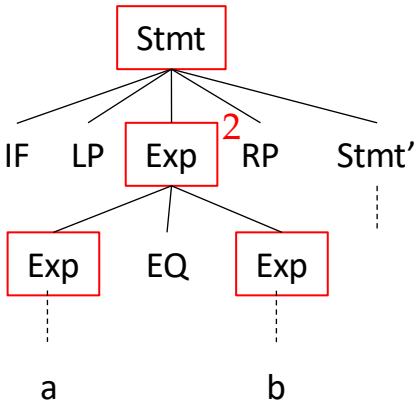


Step 1: invoke `translate_stmt()` function

To be generated when visiting the `Exp` children node of `Stmt`
`LABEL lb1`
To be generated when visiting the `Stmt'` children node of `Stmt`
`LABEL lb2`

Example

```
if(a == b) { }
```



`translate_Cond_Exp(Exp, lb_t, lb_f)`:

Exp_1 EQ Exp_2	t1 = new_place() t2 = new_place() code1 = translate_Exp(Exp_1, t1) code2 = translate_Exp(Exp_2, t2) code3 = [IF t1 == t2 GOTO lb_t] + [GOTO lb_f] return code1 + code2 + code3

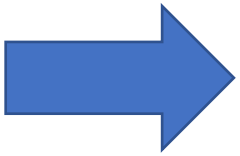
Step 2: invoke `translate_cond_Exp` function

To be generated when visiting the Exp children node of Stmt
LABEL lb1
To be generated when visiting the Stmt' children node of Stmt
LABEL lb2

t1 := a
t2 := b
IF t1 == t2 GOTO lb1
GOTO lb2

Example

```
if(a == b) { }
```



```
t1 := a
t2 := b
IF t1 == t2
GOTO lb1
GOTO lb2
LABEL lb1
… body code
LABEL lb2
```

Arrangement of Presentations

- Time: Lab session of week 15 (Dec. 25)
- Project presentations
  - By invitation only
- Research presentation
  - Send us an email before Dec. 20 to reserve a time slot
  - First come first served
- Presenters will receive some bonus points