CS323 Lab 13

Yepang Liu

liuyp1@sustech.edu.cn

Outline

- Project 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

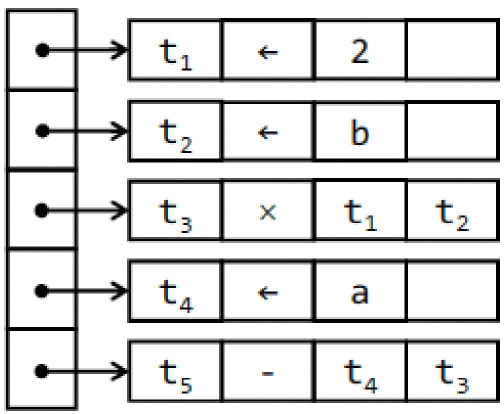
- The overall process of intermediate code generation:
  - Generate three address code during parsing (or after parsing, with a separate pass) and save it in memory
  - Perform possible optimizations
  - Output the intermediate code

Static Array Style (Quadruples)

Target	*Op*	Arg_1	Arg_2
t_1	$\leftarrow$	2	
t_2	$\leftarrow$	b	
t_3	$\times$	t_1	t_2
t_4	$\leftarrow$	a	
t_5	-	t_4	t_3

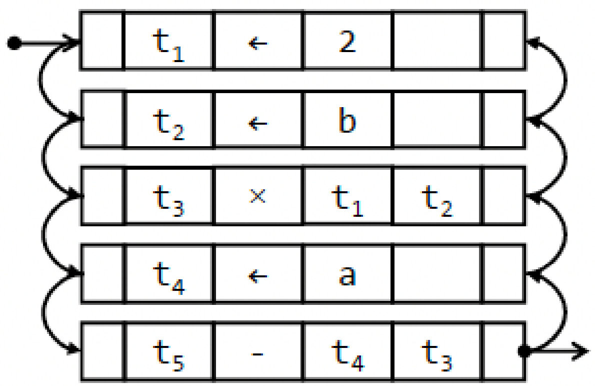
- The most straightforward implementation
- Disadvantages:
  - Low efficiency when moving instructions (imagine moving the first instruction to the end)
  - Code size is limited by the array's length

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

$\texttt{Exp}_1$ PLUS $\texttt{Exp}_2$

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

- t1, t2 = new_place(): Create temporary addresses
- code1, code2: Translate subexpressions*
- code3: Store evaluation result
- return: 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

$\texttt{Exp}_1$ ASSIGN $\texttt{Exp}_2$

```
variable = symtab_lookup(Exp₁.ID)
tp = new_place()
code1 = translate_Exp(Exp₂, tp)
code2 = [variable.name := tp]
code3 = [place := variable.name]
return code1 + code2 + code3
```

Find symbol information

Translate the right-hand side

Value copies (optimizations?)

Suppose $\texttt{Exp}_1$ is a simple case: an identifier

Conditional Statements

```
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
```

Create labels (jump targets)

Mark the beginning of body

Translate conditional expressions

Translate body code

Mark the end of body

Conditional Expressions

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

Exp₁ EQ Exp₂

```
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
```

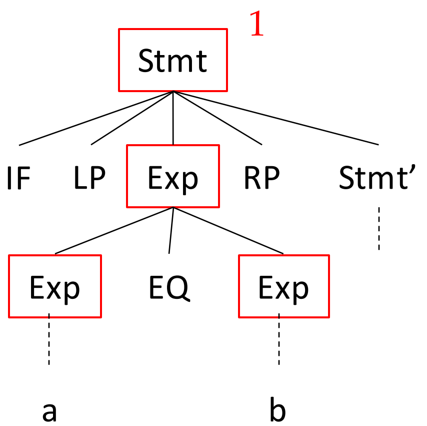
Create temporary addresses to hold subexpression values

Translate subexpressions

Generate jumps

Example

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



`translate_Stmt(Stmt):`

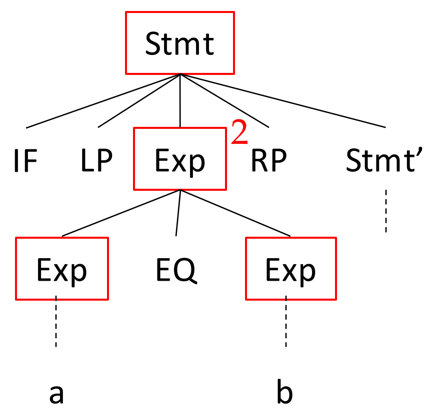
`IF LP Exp RP Stmt`	`lb1 = new_label()`
	`lb2 = new_label()`
	`code1 = translate_cond_Exp(Exp, lb1, lb2) + [LABEL lb1]`
	`code2 = translate_Stmt(Stmt) + [LABEL lb2]`
	`return code1 + code2`

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):`

$\texttt{Exp}_1$ EQ $\texttt{Exp}_2$	t1 = new_place() t2 = new_place() code1 = translate_Exp($\texttt{Exp}_1$, t1) code2 = translate_Exp($\texttt{Exp}_2$, 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`

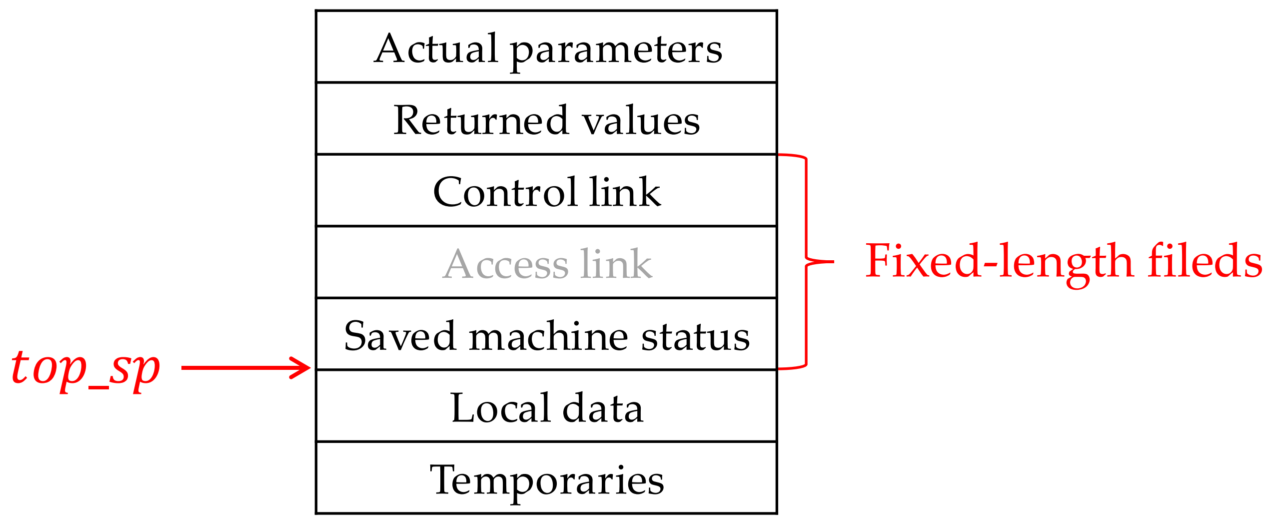
Example

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

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

Function Invocation

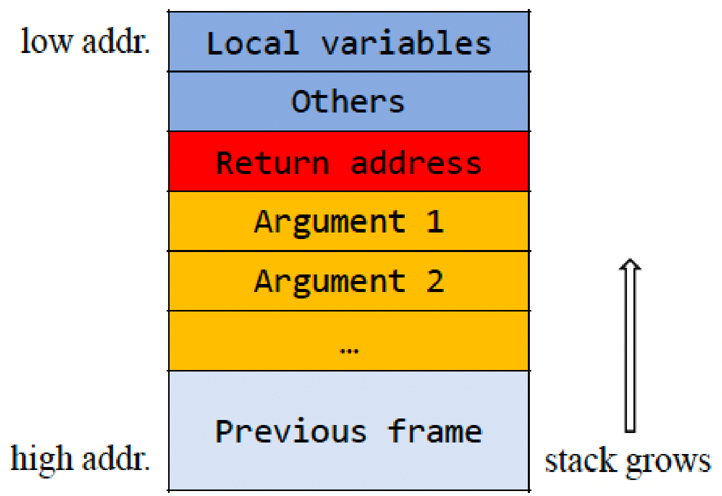
- Each active function has its own activation record, which stores the key information related to the function invocation
  - Actual parameters, local variables, saved register values, return address, etc.



Theoretical model

Function Invocation

- In most architectures, the activation records are managed using a stack. For this reason, activation records are often called stack frames.



A runtime stack of a Linux-x86 process

Main Tasks in Project Phase 3

- We do not need to manage the stack frames, which is machine-dependent (our IR simulator will do the job)
- What we need to do mainly includes the following steps:
  - Prepare the arguments
  - Pass arguments using the `ARG` instruction (e.g., `ARG x`)
  - Invoke the function using the `CALL` instruction (e.g., `x := CALL f`)

Argument Passing

C++ supports both approaches:

- `void foo(type arg)`, `arg` is passed by value regardless of whether type is a simple type, a pointer type or a class type
- `void foo(type& arg)`, `arg` is passed by reference

- **Two main approaches:**
  - **Pass by value**: A copy of the actual argument's value is made in memory, i.e., the caller and callee have two independent copies. If the callee modifies the parameter variable, the effect is not visible to the caller.
    - Typical languages: C, Java
  - **Pass by reference (a.k.a., pass by address)**: Pass the reference of the actual argument in the caller to the corresponding formal parameter of the callee so that the parameter variable becomes an alias of the argument variable (it cannot be alias of other variables, which is different from pass-by-value for reference types in Java). If the callee modifies the parameter variable, the effect is visible to the caller.
    - Languages: C++ (using the `&` operator, see above example), C# (using the `ref` keyword), etc.

Argument Passing in SPL

- For primitive types, arguments are passed by value
  - The callee's stack frame will contain copies of these values

- For derived types, your compiler should make sure that the callee gets the starting address of each argument (like Java's treatment*)
  - To pass a `struct` variable `s1` as an argument to a called function, we should push the argument onto stack using `ARG &S1` rather than `ARG S1`

* In C, we will explicitly pass a struct pointer to avoid copying the whole structure.

Translation Schemes

- First, we should add two pre-defined functions that simulate I/O to the symbol table
  - `read`: takes no parameter and returns an integer value
  - `write`: accepts an integer argument and outputs it

translate_Exp(Exp, place) = case Exp of	
read LP RP	return [READ place]
write LP Exp RP	tp = new_place() return translate_Exp(Exp, tp) + [WRITE tp]

Translated into `read` and `write` instructions

* place is the address to store the evaluation result of the expression

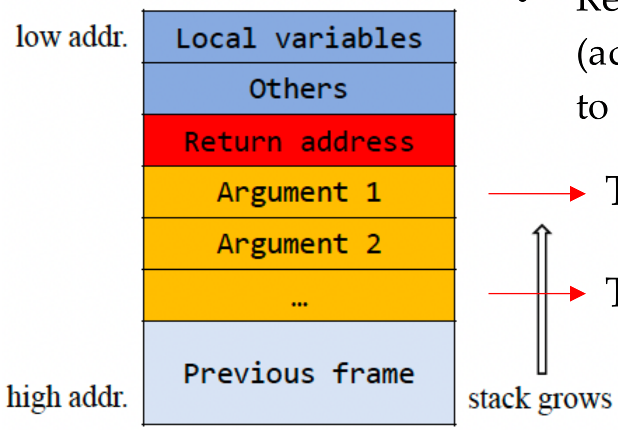
Translation Schemes

- Invoking functions without parameters

translate_Exp(Exp, place) = case Exp of	
ID LP RP	function = symtab_lookup(ID) return [place := CALL function.name]

Translation Schemes

- Invoking functions with parameters



Arguments should be pused in reverse order of declaration

- Reverse-order pushing makes the argument fetching easier (adding offsets to the stack top pointer, which typically points to the end of fixed-length fields)

The last pushed argument (using the `ARG` instruction)

The first pushed argument (using the `ARG` instruction)

Translation Scheme

translate_Exp(Exp, place) = case Exp of	
ID LP Args RP	function = symtab_lookup(ID) arg_list = EMPTY_LIST 1: Create an empty list to hold arguments code1 = translate_Args(Args, arg_list) code2 = EMPTY_CODE for i = 1 to arg_list.length: code2 = code2 + [ARG arg_list[i]] 3: Traverse the list and generate ARG instructions return code1 + code2 + [place := CALL function.name] 4: Generate CALL instruction
translate_Args(Args, arg_list) = case Args of	
Single parameter: Exp	tp = new_place() code = translate_Exp(Exp, tp) arg_list = tp + arg_list 2: Adding each argument to the list head return code
Multiple parameters: Exp COMMA Args	tp = new_place() code1 = translate_Exp(Exp, tp) arg_list = tp + arg_list code2 = translate_Args(Args, arg_list) Handling the remaining parameters return code1 + code2

Example

```
int fact(int n)
{
    if (n == 1)
        return n;
    else
        return (n * fact(n - 1));
}
int main()
{
    int m, result;
    m = read();
    if (m > 1)
        result = fact(m);
    else
        result = 1;
    write(result);
    return 0;
}
```

```
FUNCTION fact :
PARAM v1
IF v1 == #1 GOTO label1
GOTO label2
LABEL label1 :
RETURN v1
LABEL label2 :
t1 := v1 - #1
ARG t1
t2 := CALL fact
t3 := v1 * t2
RETURN t3

FUNCTION main :
READ t4
v2 := t4
IF v2 > #1 GOTO label3
GOTO label4
LABEL label3 :
ARG v2
t5 := CALL fact
v3 := t5
GOTO label5
LABEL label4 :
v3 := #1
LABEL label5 :
WRITE v3
RETURN #0
```