中山大學 SUN YAT-SEN UNIVERSITY
国家超级计算广州中心 NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

Compilation Principle
编 译 原 理

第19讲：运行时环境(2)

张献伟

xianweiz.github.io

DCS290, 5/27/2021

Quiz

- Q1: 中间代码生成阶段的任务?

  将语法树转换为中间代码（e.g.，三地址码指令）

- Q2: TAC指令：A[j][k], type(A) = array(10, array(20, int))?

  Addr(A[j][k]) = base + j * 80 + k * 4

  t_1 = j * 80; t_2 = k * 4; t_3 = t_1 + t_2; t_4 = A[t_3]

- Q3: 对文法规则S -> if (B) S_1 else S_2的IR翻译而言，指令goto S.next放置在哪里?

  S_1.code {goto S.next} else: 执行完S_1代码段后跳过S_2

- Q4: 对布尔表达式E而言，E.true和E.false指代什么?

  E为真和假时分别要跳转到的位置标签

- Q5: 回填（Backpatching）的用途是什么?

  一遍的方式处理跳转标签（得到标签具体位置后向后回填）

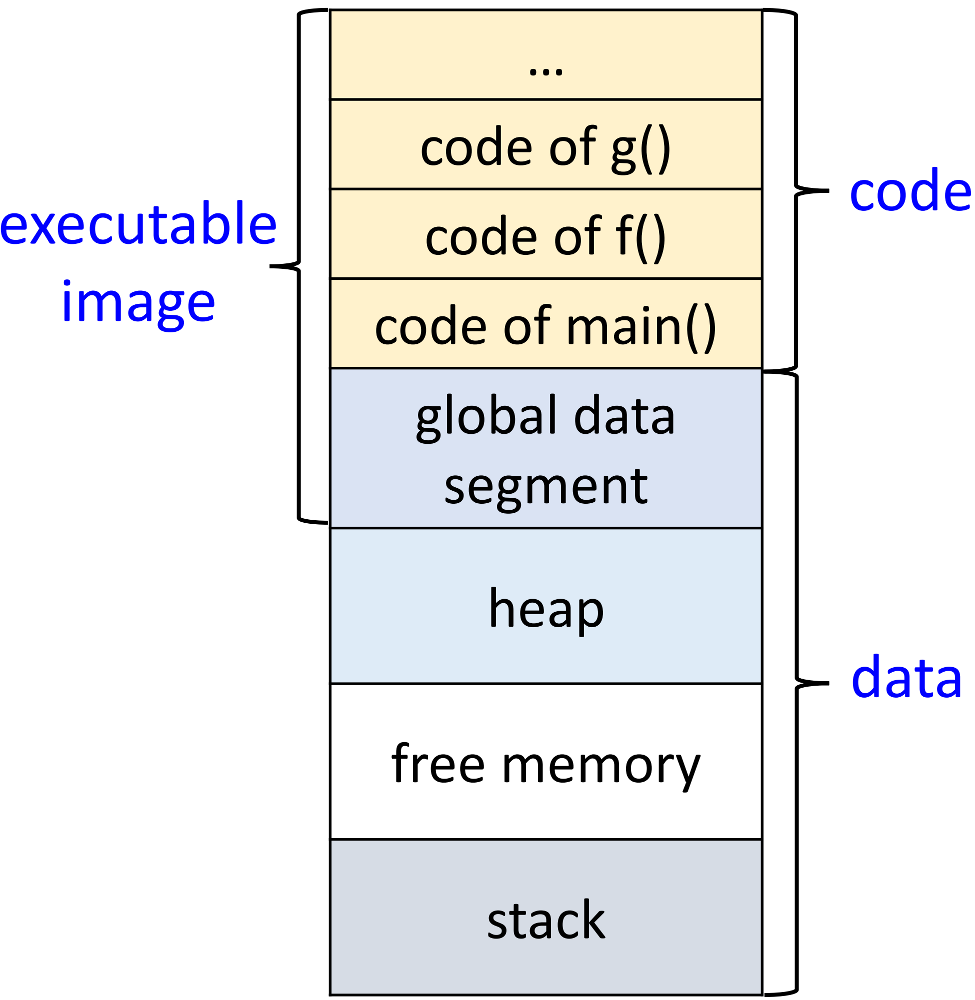
Run-Time Environments[运行时环境]

- Compiler responsibilities[编译器职责]
  - Accurately implement the semantics of the source program
  - Cooperate with OS and other systems to support the execution on the target machine
- Thus, compiler creates and manages a run-time environment in which it assumes its target programs are being executed[运行时环境]
  - How to layout and allocate storage locations
  - How to access variables
  - How to link different procedures?
  - How to interact with OS?
  - … …
- We'll focus on <u>memory management</u>

Three Types of Memory Management

- **Static** data management[静态]: static-lifetime data
  - Stores data at fixed locations laid out at compile-time
  - Laid out data forms an <u>executable image</u> file [可执行映像]
    - Contains program code, as well as initial values for vars
    - File copied verbatim to memory at program launch [逐字复制]
- **Stack** data management[栈]: scoped-lifetime data
  - Stores data in memory area managed like a stack
  - Data pushed/popped when scope entered/exited
  - Memory allocated only for the scope where data is valid
  - Compiler generates runtime code to manage stack
- **Heap** data management[堆]: arbitrary-lifetime data
  - Store data in area that allows on-demand allocation/free
  - Typically managed by memory management runtime library

Example Memory Layout



- Code
  - the size of the generated target code is fixed at compile time
- Global/static
  - the size of some program data objects, e.g., global constants, are known at compile time
- **Stack**
  - store dynamic data structures
- **Heap**
  - manage long-lived data

Activation[活动]

- Compiler typically allocates memory in the unit of procedure[以过程调用为单位]
- Each execution of a procedure is called as its **activation**[活动]
  - An execution of a procedure starts at the beginning of the procedure body
  - When the procedure is completed, it returns the control to the point immediately after the place where that procedure is called
- **Activation record** (AR) [活动记录] is used to manage the information needed by a single execution of a procedure
- **Stack** is to activation records that get generated during procedure calls

ARs in Stack Memory[在栈中管理]

- Manage ARs like a stack in memory[AR栈管理]
  - On function entry: AR instance allocated at top of stack
  - On function return: AR instance removed from top of stack
- Hardware support[硬件支持]
  - Stack pointer ($SP) register[栈指针]
    - $SP stores address of top of the stack
    - Allocation/de-allocation can be done by moving $SP
  - Frame pointer ($FP) register[帧指针]
    - $FP stores base address of current frame
    - **Frame**: another word for activation record (AR)
    - Variable addresses translated to an offset from $FP
  - $FP and $SP together delineate the bounds of current AR

Contents of ARs

- Example layout of a function AR

Temporaries	临时变量
Local variables	局部变量
Saved Caller/Callee Register Values	保存的寄存器值
Saved Caller's Instruction Pointer ($IP)	保存的调用者指令指针
Saved Caller's AR Frame Pointer ($FP)	保存的调用者帧指针
Parameters	参数
Return Value	返回值

- Registers such as $FP and $IP overwritten by callee → Must be saved to/restored from AR on call/return
  - Caller's $IP: where to execute next on function return (a.k.a. return address: instruction following function call)
  - Caller's $FP: where $FP should point to on function return
  - Saved Caller/Callee Registers: other registers (will discuss)

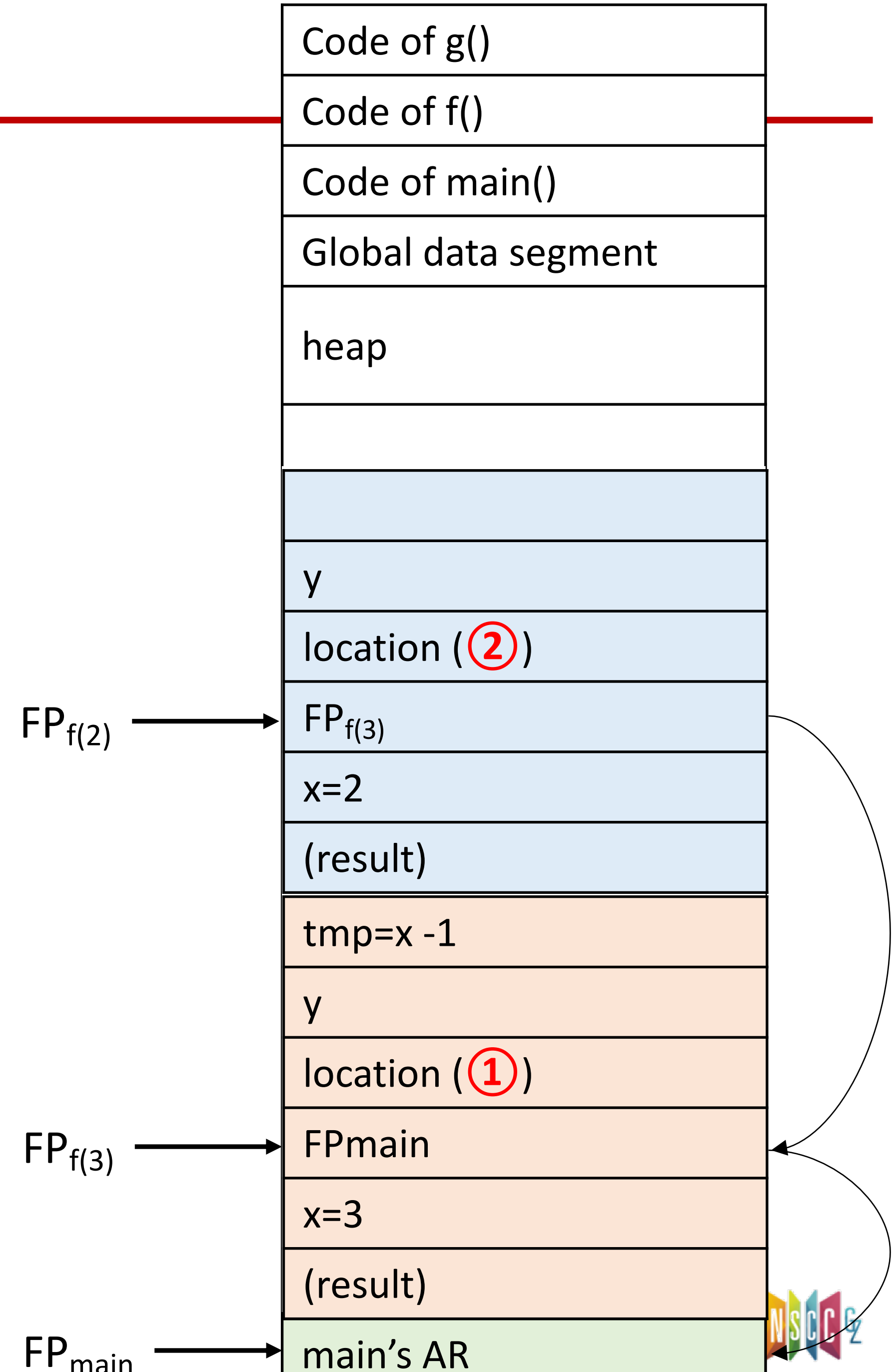
Example

Temporaries
Local variables
Saved Caller/Callee Register Values
Saved Caller's Instruction Pointer ($IP)
Saved Caller's AR Frame Pointer ($FP)
Parameters
Return Value

```
int g() {
  return 1;
}

int f(int x) {
  int y;
  if (x==2)
    y = 1;
  else
    y = x + f(x-1);
   ② ...
   return y;
}

int main() {
  f(3);
  ① ...
}
```



Calling Convention[调用规范]

- Calling convention: rules on how caller/callee interact

- All interactions happen through AR (relevant parts in bold):

Temporaries
Local variables
Saved Caller/Callee Register Values
Saved Caller's Instruction Pointer ($IP)
Saved Caller's AR Frame Pointer ($FP)
Parameters
Return Value

Calling Convention (cont.)

- Caller's responsibility[调用者]

Before call	Evaluate **parameters** and save them in callee's AR
	Save **$FP** in callee's AR; update $FP to base of callee's AR
	Save **$IP** in callee's AR; jump to callee (updating $IP)
	Save **caller-saved registers** in caller's AR
After call	Restore **caller-saved registers** in caller's AR

- Callee's responsibility[被调用者]

At begin	Save **callee-saved registers** in callee's AR
At end	Evaluate **return value** and save it in callee's AR
	Restore caller's **$FP** into $FP
	Restore caller's **$IP** into $IP (jumping to return address)
	Restore **callee-saved registers** in callee's AR

- Why separate caller-saved and callee-saved registers?

Caller-/Callee-saved Registers

- Convention
  - Allows caller to use callee-saved registers w/o save/restore
  - Allows callee to use caller-saved registers w/o save/restore
- Assume R1 is caller-saved and R2 is callee-saved:

```
void foo() {
    R2 = R2 + 1; // no need to save/restore R2
    bar();
}
void bar() {
    R1 = R1 + 1; // no need to save/restore R1
}
```

- W/o convention, foo() must save R2, bar() must save R1
  - Since no guarantee bar() will not overwrite R2
  - Since no guarantee foo() will not use R1
- Especially important if foo(), bar() compiled separately
  - E.g. foo() and bar() maybe in different libraries
  - foo(), bar() cannot look into each other to decide

Calling Convention

- AR layout is also part of calling convention
  - Designed for **easy access**
    - Parts of callee's AR written by caller ($FP, $IP, parameters)
    - ⇒ Place them at bottom of AR where caller can find them easily
      - (If at top, location will differ depending on number of variables and temporaries in callee's AR, something compiler generating caller doesn't necessarily know)
  - Designed for **execution speed**
    - E.g. first 4 arguments in MIPS typically passed in registers (register accesses are faster than stack accesses)
- Who decides on the calling convention?
  - Entirely up to the compiler writer
  - When linking modules generated by different compilers, care must be taken that same conventions are followed (e.g. Java Native Interface allows calls from Java to C)

Heap Memory Management[堆管理]

- Heap data
  - Lives beyond the lifetime of the procedure that creates it

```
TreeNode* createTREE() { {
TreeNode* p = (TreeNode*)malloc(sizeof(TreeNode));
return p; }
```

  - Cannot reclaim memory automatically using a stack
- Problem: when and how do we reclaim that memory?
- Two approaches
  - **Manual** memory management
    - Programmer inserts deallocation calls. E.g. “free(p)”
  - **Automatic** memory management
    - Runtime code automatically reclaims memory when it determines that data is no longer needed

Why Manual?[人为管理]

- Manual memory management is typically more efficient
  - Programmers know when data is no longer needed
- With automatic management, runtime must somehow detect when data is no longer needed and recycle it
  - Performance overhead
    - Detection code significantly impacts program performance
  - Memory overhead
    - Detection can be done every so often (Typically only when program runs out of memory)
    - Runtime may keep around data longer than necessary
    - Results in larger memory footprint
  - Poor response time
    - Program must be paused during detection phase
    - Program will be unresponsive during that time

Why Automatic?[自动化管理]

- Fewer bugs
  - With manual management, programmers may
    - forget to free unused memory -> memory leak
    - free memory too early -> dangling pointer access
    - free memory twice (double free)
  - Memory bugs are extremely hard to find and fix
    - While there are tools (e.g., valgrind), but the tools have limitations and may involve much overhead

- More secure system
  - Disallowing programmer free() calls is essential for security
  - Automatic management prevents all memory corruption

Implementation: Automatic & Manual

- Common functionality in both automatic and manual
  - Runtime code maintains used/unused spaces in heap
    - e.g. linked together in the form of a list
  - malloc(int size)
    - move size bytes from unused to used
  - free(void *p)
    - move given memory from used to unused
- Only in automatic memory management
  - Routines to perform detection of unused memory
- We will focus on automatic memory management
  - Because detection often requires involvement of compiler

Reachable Objects and Garbage

- Named objects
  - Can be global variables in global data segment
  - Can be local variables in stack memory or registers
  - Also called **root objects**
    - They form the root of the tree of reachable objects
- An object *x* is **reachable** iff
  - A named object contains a reference to *x*, or
  - A reachable object *y* contains a reference to *x*
- **Garbage** refers to the data that cannot be referenced
  - Garbage can no longer be used and its memory can be reclaimed
  - This reclamation is process is called **garbage collection**

Two Garbage Collection Schemes

- Reference counting[引用计数]
  - Maintain a **reference counter** inside each object
    - Counts the number of references to object
  - When counter becomes 0, the object is no longer usable
    - Garbage collect unreachable object

- Tracing[追踪]
  - When the heap runs out of memory to allocate:
    - Pause the program
    - Trace through all reachable objects
    - Garbage collect remaining objects
    - Restart the program

Reference Counting[引用计数]

- Idea: when reference counter (RC) == 0, collect object
  - If collected object has references to other objects
    - may trigger recursive collection of other objects
- Implementation
  - Compiler generates code to maintain reference counters
  - Whenever program modifies a reference
    - For object losing reference, decrement RC
    - For object gaining reference, increment RC

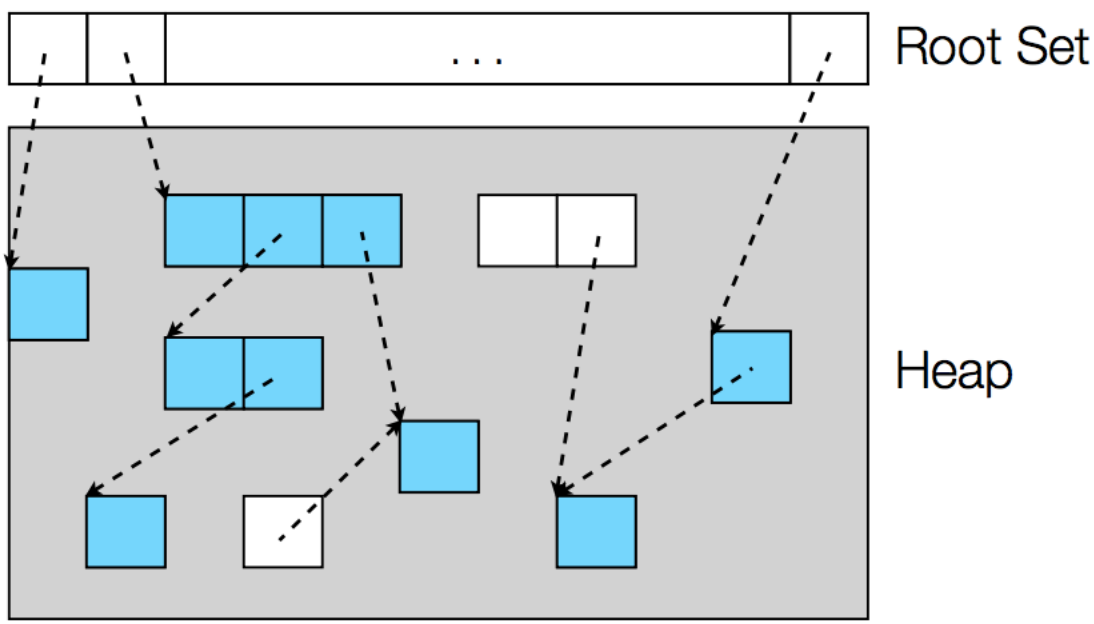
```
Object x = new Foo(), y = new Bar();
// Now, RC of Foo == 1, RC of Bar == 1
x = y;
// Now, RC of Foo == 0, RC of Bar == 2
```

Reference Counting (cont.)

- Advantages
  - Relatively easy to implement
    - Compiler only needs to insert RC manipulation code at reference assignments
  - Good response time
    - Garbage is collected whenever there is opportunity
    - No need to pause program for long time ! responsive (Unless freeing a long chain of objects!)
- Disadvantages
  - Cannot collect circular data structures (Must rely on tracing GC for these corner cases)
  - Bad performance
    - Manipulating RCs at each assignment is high overhead
    - RC must be synchronized with multithreading ! even slower

Tracing[追踪]

- Start from named objects (also called root objects)
  - If object is value: no further action
  - If object is reference: follow reference
  - If object is struct: go through each field
- Mark all traversed objects as live objects
- All remaining objects can be collected as garbage



Tracing (cont.)

- Advantages
  - Is guaranteed to collect even cyclic references
  - Good performance
    - Overhead proportional to traced (live) objects Garbage (dead) objects do not incur any overhead!
    - Most objects have short lifetimes: dead by the time tracing GC runs
- Disadvantages
  - Bad response time: requires pausing program
  - Prone to heap thrashing
    - Thrashing: frequent GCs to collect small amounts of garbage
    - If heap does not have extra ‘headroom’ beyond working set
      - GC becomes very frequent
      - Most objects are now live (bad performance)

Flavors of Tracing Collection

- To move or not to move objects?
  - Garbage collection can leave ‘holes’ inside heap
  - Objects can be moved during GC to "compress" holes
  - **Mark-and-Sweep**: example of non-moving GC
  - **Semispace**: example of moving GC
- To collect at once or incrementally?
  - Tracing entire heap can lead to long pause times
  - Possible to collect only a part of the heap at a time
  - **All-at-once**: naive GC with no partitions
  - **Incremental**: divides heap into multiple partitions
  - **Generational**: divides heap into generations
- The two choices are orthogonal to each other

Compilation Principle
编 译 原 理

第19讲：目标代码生成(1)

张献伟

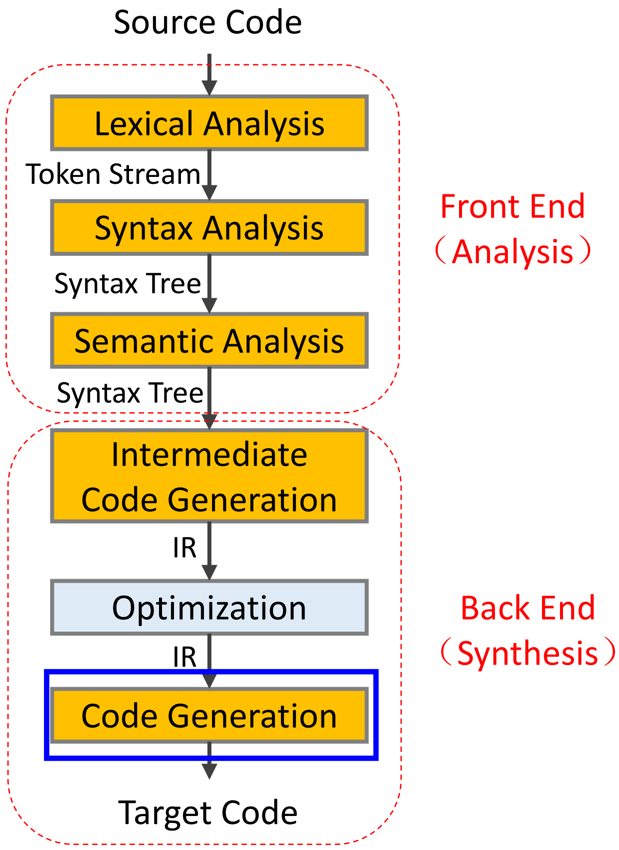
xianweiz.github.io

DCS290, 5/27/2021

Target Code Generation[目标代码生成]

- What we have now
  - IR of the source program
  - Symbol table

- Goals of target code generation
  - <u>Correctness</u>: the target program must preserve the semantic meaning of the source program
  - <u>High-quality</u>: the target program must make effective use of the available resources of the target machine
  - <u>Fast</u>: the code generator itself must runs efficiently



Example

- An example on real machine (*x86_64*)
  - Symbols have to be translated to memory addresses

```c
int x = 1;
int y = 2;
int z = 3;

void main() {
  x = y + z;
}
```

gcc -O0 -S test.c

```
movl    _y(%rip), %eax
addl    _z(%rip), %eax
movl    %eax, _x(%rip)
```

- A simplified representation

x = y + z

```
LD R0, y          // R0 = y (load y into register R0)
ADD R0, R0, z     // R0 = R0 + z (add z to R0)
ST x, R0          // x = R0 (store R0 into x)
```

Translating IR to Machine Code

- Machine code generation is machine ISA dependent*
  - Complex instruction set computer (CISC): x86
  - Reduced instruction set computer (RISC): ARM, MIPS, RISC-V

- Three primary tasks
  - Instruction selection[指令选取]
    - Choose appropriate target-machine instructions to implement the IR statements
  - Register allocation and assignment[寄存器分配]
    - Decide what values to keep in which registers
  - Instruction ordering[指令排序]
    - Decide in what order to schedule the execution of instructions

* CPU及指令集演进 (漫画 | 20多年了，为什么国产CPU还是不行？)