中山大學 SUN YAT-SEN UNIVERSITY 计算机学院（软件学院） SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Compilation Principle
编 译 原 理

第24讲：目标代码生成(3)

张献伟

xianweiz.github.io

DCS290, 6/27/2024

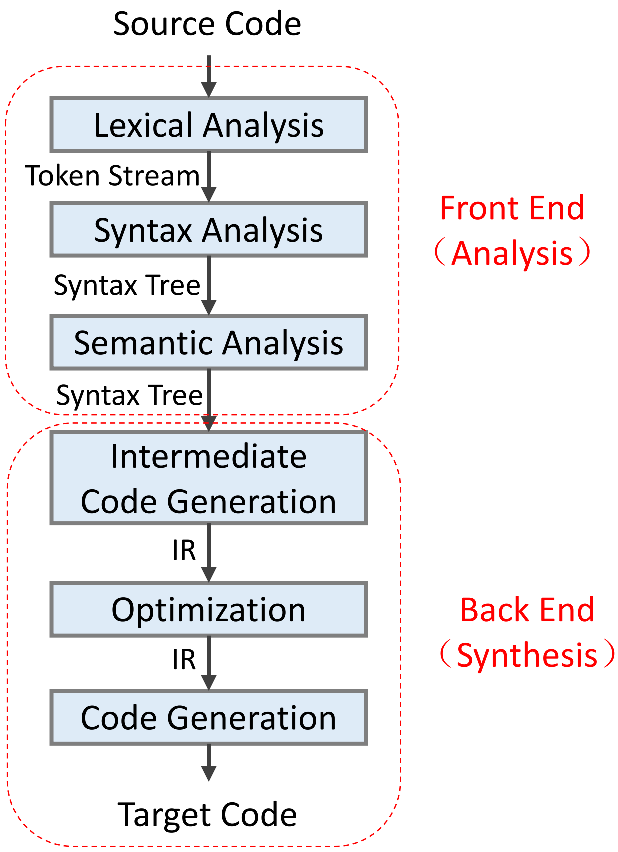
Brief Review

- The review is NOT intended to be complete and comprehensive:

- NOT 100% contents will be covered in exam
- NOT 100% contents in exam will be listed

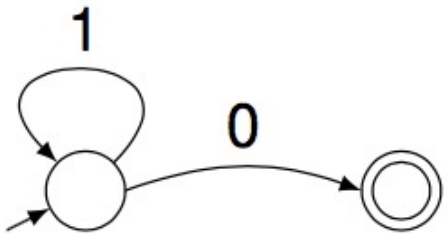
Compilation Phases[编译阶段]

- **Lexical**: source code → tokens
  - RE, NFA, DFA, …
- **Syntax**: tokens → AST or parse tree
  - CFG, LL(1), LALR(1), …
- **Semantic**: AST → AST +symbol table
  - SDD, SDT, typing, scoping, …
- **Int. Code Generation**: AST → TAC/IR
  - IR, offset, CodeGen, …
- **Optimization**: IR → (optimized) IR
  - BB, CFG, DAG, …
- **Code generation**: IR → Machine Insts
  - Instruction, register, stack, …



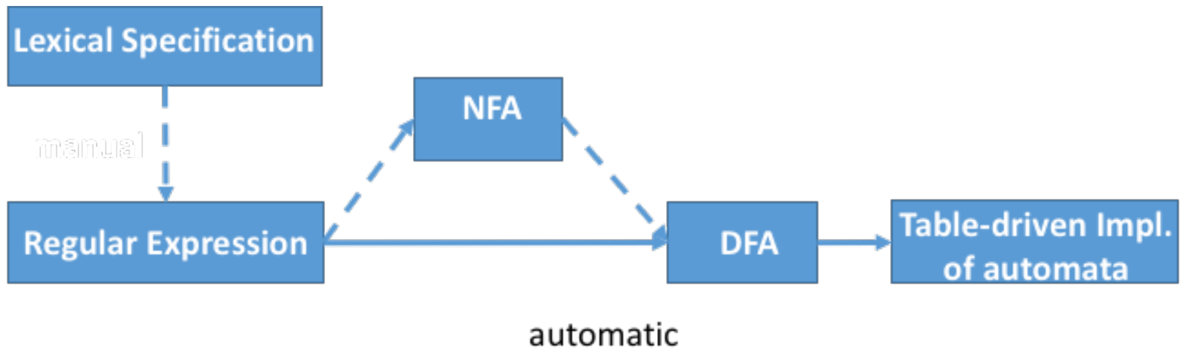
Lexical Analysis[词法分析]

- Characters --> tokens
  - 二元组：<class, lexeme>
- How to specify tokens?
  - Regular expression
    - Atomic, compound
- How to recognize tokens?
  - Transition diagram[转换图]
  - NFA, DFA, table



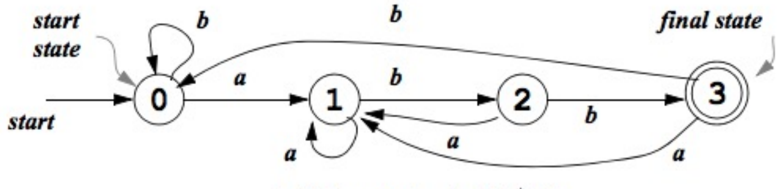
1*0

Any number of ‘1’s followed by a single ‘0’

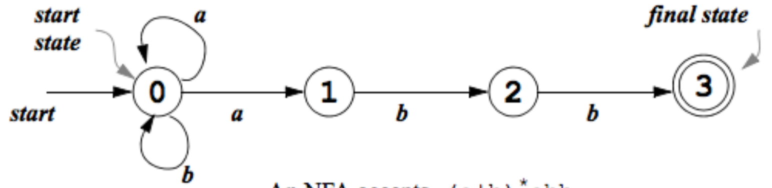


Lexical Analysis (cont.)

- Regular expression
  - 自然语言描述 <-> 正则表达式
  - 正则表达式 <-> NFA/DFA
    - M-Y-T算法
  - 局限性： RE vs. CFG
    - $L = \{a^n b^n \mid n \geq 1\}$ vs. $L = \{a^n b^n \mid 10 \geq n \geq 1\}$
- NFA, DFA
  - 状态和边的含义（ε-move）
    - 初始状态、终结状态
  - 形式上的区别
  - 意义上等价
    - NFA -> DFA: ε-闭包
    - 状态最小化



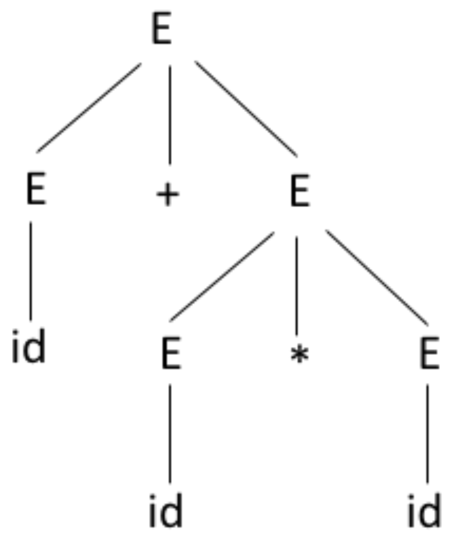
A DFA accepts (a|b)*abb



An NFA accepts (a|b)*abb

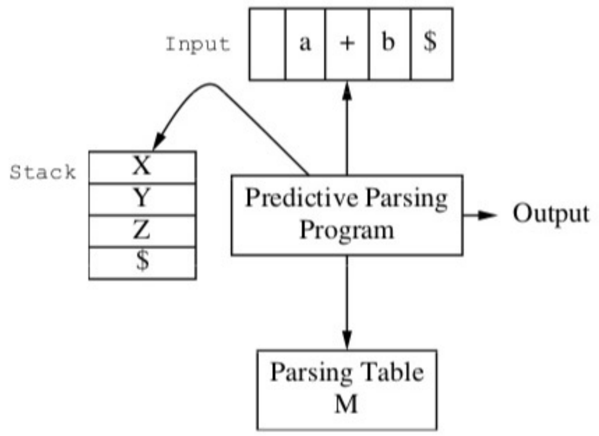
Syntax Analysis[语法分析]

- Tokens --> parse tree
- Context-free grammar
  - 四元组：T, N, s, σ
    - 但通常只写σ
  - Production rule: LHS --> RHS
  - CFG→所定义语言的自然描述
- Derivation[推导]
  - Leftmost, rightmost
  - Parse tree: 推导的图形化表示
  - Sentential form[句型]、Sentence[句子]
  - Ambiguity[二义性]
    - 证明及消除（优先级、结合性）



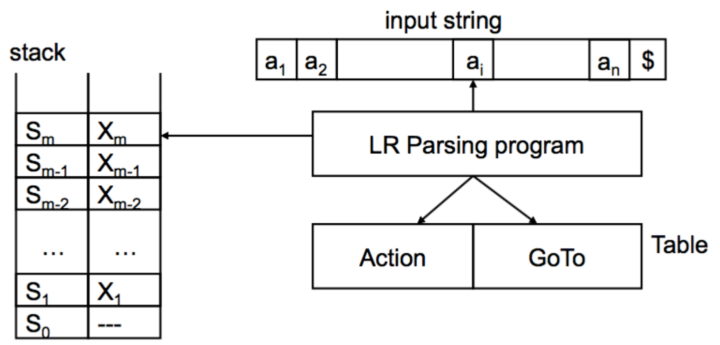
Syntax Analysis (cont.)

- Parser
  - Top-down: leftmost derivation
  - Bottom-up: reverse order of the rightmost derivation
- Top-down
  - Recursive descent, Predictive/LL(k)
  - Left recursion[左递归]: rewriting
  - Common prefix[共同前缀]: left factoring
- LL(1)
  - Build the parse table
    - FIRST, FOLLOW
  - Use the parse table: expand or match
    - 给定输入串的分析过程
  - Determine if G is LL(1)



Syntax Analysis (cont.)

- Bottom-up
  - Shift-reduce
  - Handle[句柄]、Viable Prefix[活前缀]、Phrase[短语]、Simple Phrase[直接短语]、 Leftmost Simple Phrase[最左直接短语]
    - 活前缀不能越过句柄：分析栈存放的都是活前缀，在等句柄出现；一旦出现就规约这个句柄
    - 句柄是一个直接短语
- LR: more powerful than LL
  - Parse table:
    - Action table: shift/reduce/accept/error
    - Goto table
  - LR分析器的工作过程实际上就是逐步产生规范句型的活前缀
    - 构造识别所有活前缀的DFA == 构造基于项目集的DFA
  - 给定parse table，对输入串进行分析



Syntax Analysis (cont.)

- LR(0): build parse table, construct a FA
  - Item/configuration: initial, reduce, accept
  - State/configuration set: closure()
  - Augmented grammar
  - DFA -> parse table
  - Conflicts: shift-reduce, reduce-reduce
- Other LRs
  - SLR: improve LR(0) using FOLLOW
  - LR(1)
    - LR(1) item: LR(0) item + lookahead symbols
    - Configuration set: closure()
  - LALR(1): YACC/Bison

Semantic Analysis[语义分析]

- For semantic analysis
  - Attributes: synthesized, inherited
  - Semantic rules or actions
- SDD vs SDT
  - Syntax directed definitions: attributes + semantic rules
    - S-attributed, L-attributed
  - Syntax directed translation scheme: attributes + semantic actions
    - An executable specification of the SDD
- Performed in parsing
  - Top-down: recursive descent, predictive
  - Bottom-up: marker, backpatching
- Annotated parse tree
  - With actions

Code Generation, Optimization[中后端]

- Intermediate representation
  - Three-address code
  - CodeGen: variable, array, control, …
- Runtime/Target code
  - Stack, AR, calling conventions
  - Memory: code, data (global/static, stack, heap)
  - Instruction selection, register allocation, instruction ordering
- Code optimizations
  - Concepts: basic block, flow graph, DAG
  - Optimization: metrics, techniques
    - Peephole, local, loop, global
    - Dead code elimination, common subexpression elimination
    - Strength reduction, constant folding, …

Compilation Principle
编 译 原 理

第24讲：目标代码生成(3)

张献伟

xianweiz.github.io

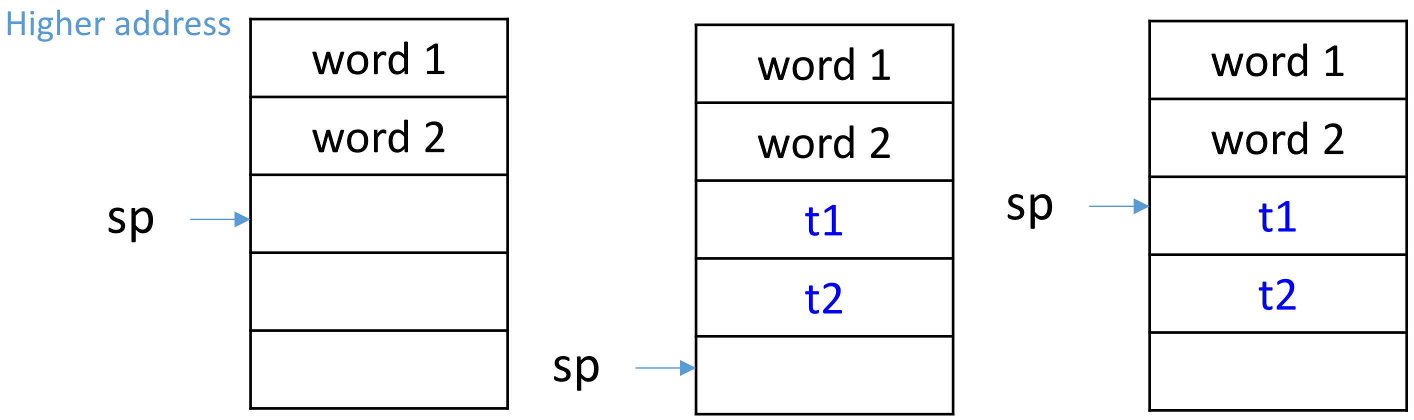
DCS290, 6/27/2024

Stack Operations[栈操作]

- To **push** elements onto the stack
  - To move stack pointer *sp* down to make room for the new data
  - Store the elements into the stack
- For example, to push registers *t1* and *t2* onto stack
- **Pop** elements simply by adjusting the *sp* upwards
  - Note that the popped data is still present in memory, but data past the stack pointer is considered invalid / undefined

```
sw t1, 0(sp)
sw t2, -4(sp)
sub sp, sp, 8
```

```
sub sp, sp, 8
sw t1, 8(sp)
sw t2, 4(sp)
```



Code Generation Strategy

- For each expression *e* we generate RISC-V code that:
  - Computes the value of *e* into *a0* (i.e., the accumulator)
  - Preserves *sp* and the contents of the stack
- We define a code generation function *cgen(e)*
  - Its result is the code generated for *e*

- Code generation for constants
  - The code to evaluate a constant simply copies it into the register: *cgen(i) = li a0 i*
    - Note that this also preserves the stack, as required

Code Generation for ALU

- Default

```
cgen(e1 + e2):
          # stores result in a0
          cgen(e1)
          # pushes a0 on stack
          addiu sp sp -4
          sw a0 4(sp)
          # overwrites result in a0
          cgen(e2)
          # pops value of e1 to t1
          lw t1 4(sp)
          addiu sp sp 4
          # performs addition
          add a0 t1 a0
```

→

```
cgen(e1 + e2):
          # stores result in a0
          cgen(e1)
          # copy result of a0 to t1
          mv t1 a0
          # stores result in a0
          cgen(e2)
          # performs addition
          add a0 t1 a0
```

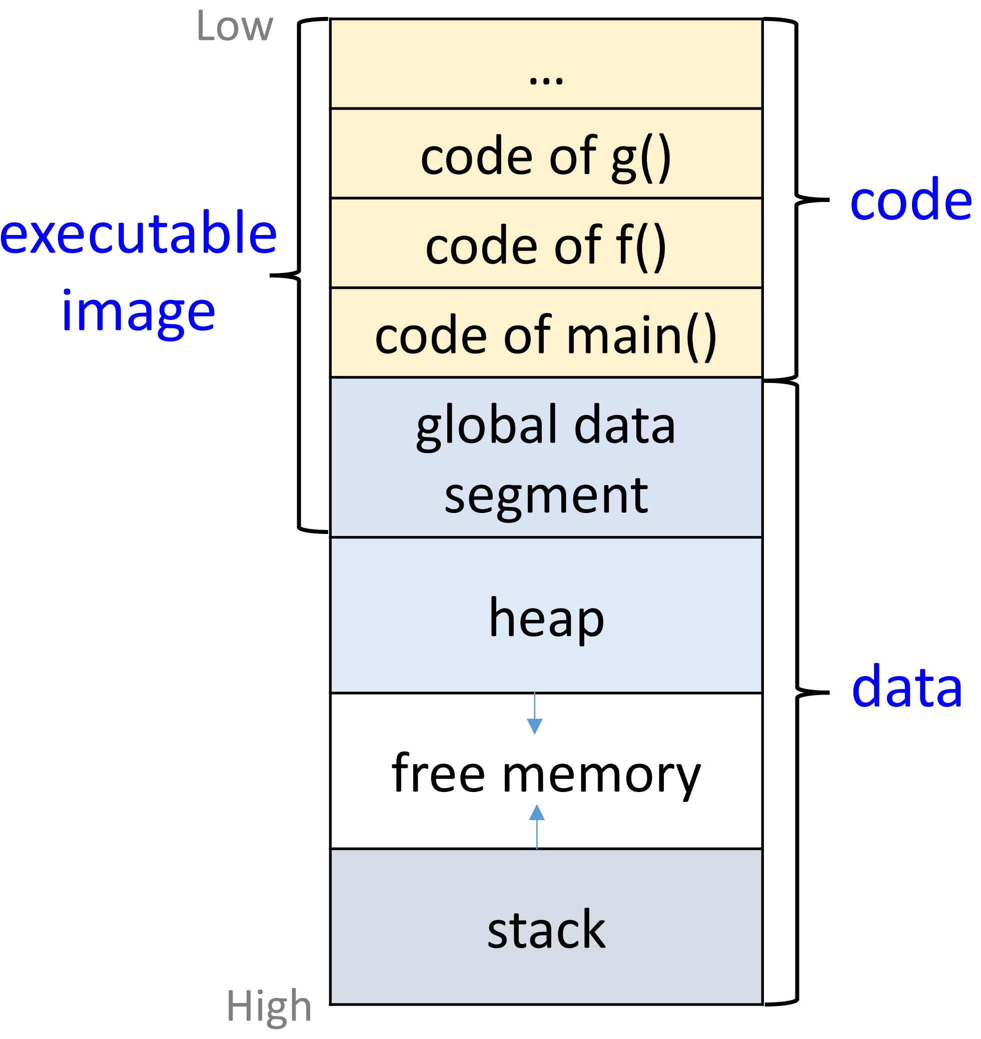
- Possible optimization: put the result of *e1* directly in register *t1*? What if 3 + (7 + 5)?

Code Generation for Conditional

- We need flow control instructions
- New instruction: *beq reg1 reg2 label*
  - Branch to label if *reg1 == reg2*
    - Ow, does nothing and moves on to the next command
- New instruction: *b label*
  - Unconditional jump to *label*

```
cgen(if e1 == e2 then e3 else e4):
        cgen(e1)
        # pushes a0 on stack
        addiu sp sp -4
        sw a0 4(sp)
        # overwrites a0
        cgen(e2)
        # pops value of e1 to t1
        lw t1 4(sp)
        addiu sp sp 4
         # performs comparison
        beq a0 t1 true_branch
    false_branch:
        cgen(e4)
        b end_if
    true_branch:
        cgen(e3)
    end_if:
```

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 hold 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*
      - Always points to the top of current AR as long as invocation is active
  - *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 Return Address (ra)	保存的调用者返回地址
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 *ra* overwritten by callee → Must be saved to/restored from AR on call/return
  - Caller's *ra*: where to execute next on function return (a.k.a. instruction pointer: instruction following function call)
  - Caller's *FP*: where *FP* should point to on function return
  - Saved Caller/Callee Registers: other registers (will discuss)

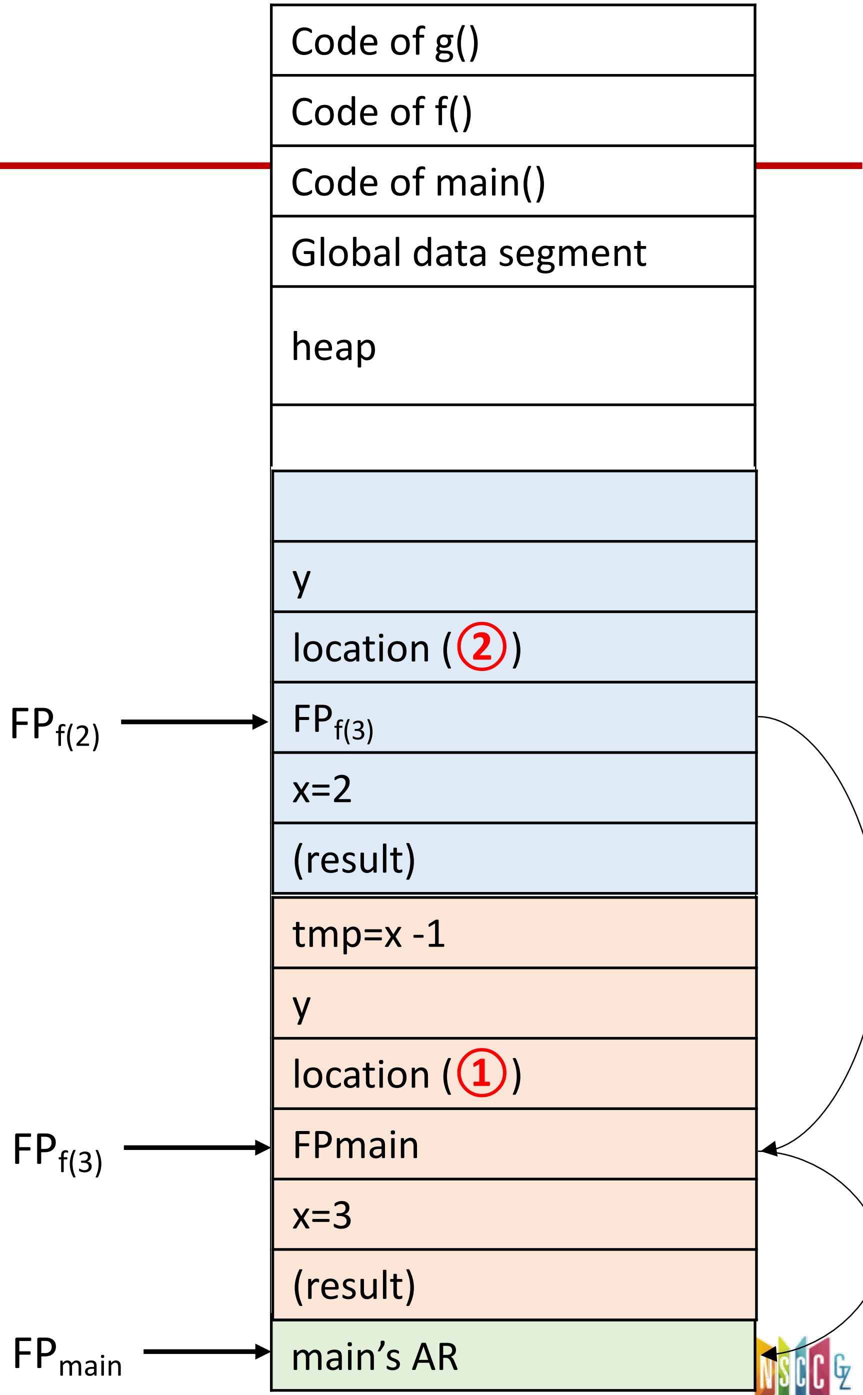
Example

Temporaries
Local variables
Saved Caller/Callee Register Values
Saved Caller's Return Address (ra)
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 ...
}
```



https://drive.google.com/file/d/1qe7it1bz7Ioaa8UBduaAv08XU8AFzpbt/view

Caller/Callee Conventions[规范]

- Important registers should be saved across function calls
  - Otherwise, values might be overwritten

- Caller and callee should cooperate
  - Caller: the function making the call
  - Callee: the function that is being called
- But, who should take the responsibility?
  - The <u>caller</u> knows which registers are important to it and should be saved[调用者知道哪些重要]
  - The <u>callee</u> knows exactly which registers it will use and potentially overwrite[被调用者知道哪些会被覆写]
  - However, in the typical “block box” programming, caller and callee don’t know anything about each other’s implementation

Caller/Callee Conventions (cont.)

- Potential solutions
  - **Sol1**: caller to save any important registers that it needs before calling a func, and to restore them after (but not all will be overwritten)[调用者保存任何重要寄存器，但并非所有都会覆写]
  - **Sol2**: callee saves and restores any registers it might overwrite (but not all are important to caller)[被调用者保存并恢复任意可能覆写，但并非所有都重要]

- Caller and callee should cooperate
  - Caller: the function making the call
  - Callee: the function that is being called

Caller/Callee Conventions (cont.)

- Caller and callee should cooperate

- Callee-saved registers (**preserved registers**): the registers that a function promises to leave unchanged[预留寄存器]
  - The caller may assume these registers are not changed by the callee

- Caller-saved registers (**non-preserved registers**): the registers that a function does not promise to leave unchanged[非预留寄存器]
  - The callee may freely modify these registers, under the assumption that the caller already saved them

RISC-V Calling Conventions

- Caller: save and restore any of the following caller-saved registers that it cares about

  t0-t6 a0-a7 ra

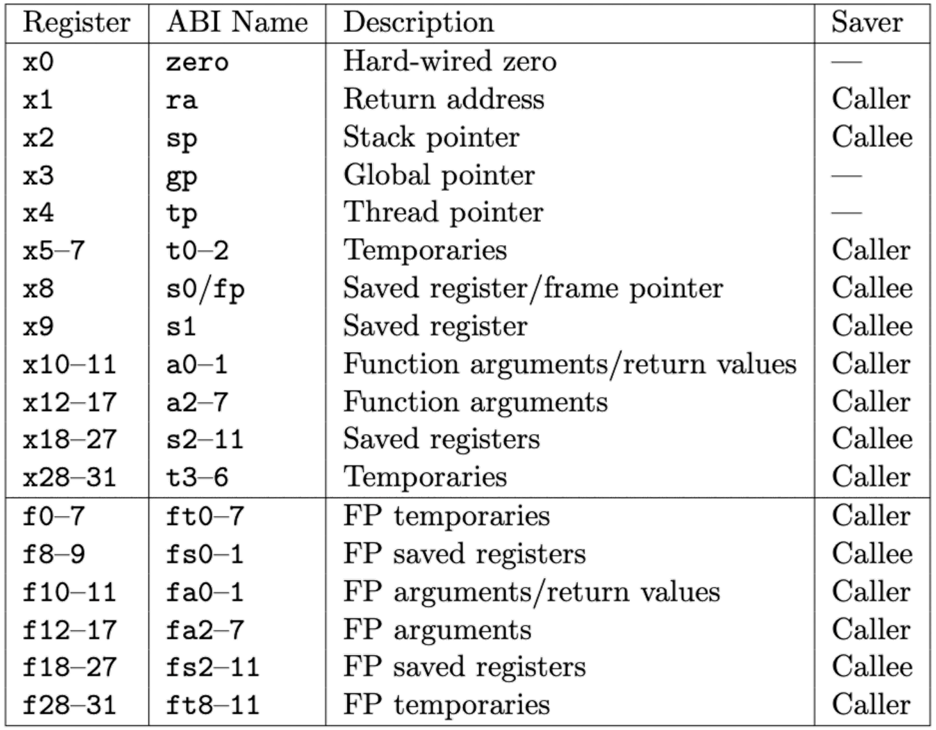
  a0-a7 for function arguments, a0-a1 for return values

- Callee: save and restore any of the following callee-saved registers that it uses

  s0-s11 sp

  s0 is fp

a0 - a7 (x10 - x17): eight argument registers to pass parameters and two return values (a0-a1)



Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

The Caller Perspective

Caller-saved: t0-t6   a0-a7   ra
Callee-saved: s0-s11 sp

- We we call a function, that function promises to not modify any of the preserved registers[调用者：这些预留寄存器不会被改动]
  - I.e., when the function returns, we can be sure that the preserved registers have not changed
    - The called function may modify across the calling, but finally restores
- However, that function is allowed to freely modify any of the non-preserved registers[调用者：这些非预留寄存器会被随意改动]
  - I.e., after calling a function and the function returns, every non-preserved register now contains garbage
    - Garbage refers to unknown values, even if the values in non-preserved remain unchanged across the function call (just assume changed)

```
addi s0, x0, 5    # s0 contains 5
jal ra, func.     # call a function
addi s0, s0, 0    # s0 still contains 5 here!
```

```
addi t0, x0, 5    # t0 contains 5
jal ra, func      # call a function
addi t0, t0, 0    # t0 contains garbage!
```

The Callee Perspective

Caller-saved: t0-t6 a0-a7 ra
Callee-saved: s0-s11 sp

- We we write a function, we are allowed to freely change any of the non-preserved registers
  - I.e., those non-preserved ones are supposed to be saved by the caller
- However, we must promise to not change any of the preserved ones
  - I.e., if to use the preserved registers during the function, we must save the values on the stack at the function start, then restore at the function end

```
# Prologue
addi sp, sp, -12 # decrement stack
sw ra, 4(sp) # store ra value on the stack
sw s0, 8(sp) # store s0 value on the stack
sw s1, 12(sp) # store s1 value on the stack

# do stuff in the function

# Epilogue
lw ra, 4(sp) # restore ra value from the stack
lw s0, 8(sp) # restore s0 value from the stack
lw s1, 12(sp) # restore s1 value from the
stack
addi sp, sp, 12 # increment stack

# finish up any loose ends

ret # return from function
```

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

Detailed Calling Steps

- The **caller** sets up for the call via these steps[调用者]
  - 1) Make space on stack for and save any caller-saved registers
  - 2) Pass arguments by pushing them on the stack, one by one, right to left[传参数]
  - 3) Execute a jump to the function (saves the next inst in *ra*)
- The **callee** takes over and does the following[被调用者]
  - 4) Make space on stack for and save values of fp and ra
  - 5) Configure frame pointer by setting fp to base of frame
  - 6) Allocate space for stack frame (total space required for all local and temporary variables)
  - 7) Execute function body, code can access params at positive offset from *fp*, locals/temps at negative offsets from *fp*

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

Detailed Calling Steps (cont.)

- When ready to exit, the **callee** does following[调用退出]
  - 8) Assign the return value (if any) to a0[返回值]
  - 9) Pop stack frame off the stack (locals/temps/saved regs)
  - 10) Restore the value of fp and ra
  - 11) Jump to the address saved in ra

- When control returns to the **caller**, it cleans up from the call with the steps[调用返回]
  - 12) Pop the parameters from the stack
  - 13) Restore value of any caller-saved registers, pops spill space from stack

Code Generation for Function Call

- The calling sequence is the instructions (of both caller and callee) to set up a function invocation

- New instruction: *jal label*
  - Jump to label, after saving address of next instruction in *ra*
  - Actually, *jal ra label*
    - Store PC+4 in *ra*
    - Similar to *jal x0 label* for jumping inside a loop

```
cgen(f(e1, …, en)):
        # pushes arguments (reverse order)
        cgen(en)
        addiu sp sp -4
        sw a0 4(sp)
        …
        cgen(e1)
        addiu sp sp -4
        sw a0 4(sp)
        # saves FP
        addiu sp sp -4
        sw fp 4(sp)
        # pushes return address
        addiu sp, sp, -4
        sw ra, 4(sp)
        # begins new AR in stack
        mv fp, sp
        # jumps to func entry (update ra)
        jal f_entry
```

Code Generation for Function Definition

- New instruction: *jr reg*
  - Jump to address in register reg
  - Acutally, *jalr ra rd rmm*, jump to *rd + imm*
    - Set the PC to *rd + imm*

```
cgen(def f(x1,…,xn) = e):
f_entry:  # save registers ra and si
          cgen(e)
           # removes AR from stack
           mv sp fp
           # pops return address
           sw ra 4(sp)
           addiu sp sp 4
           # pops old FP
           lw fp 4(sp)
           addiu sp sp 4
           # jumps to return address
           jr ra
```

RISC-V uses *jal* to invoke a function and *jr* to return from a function

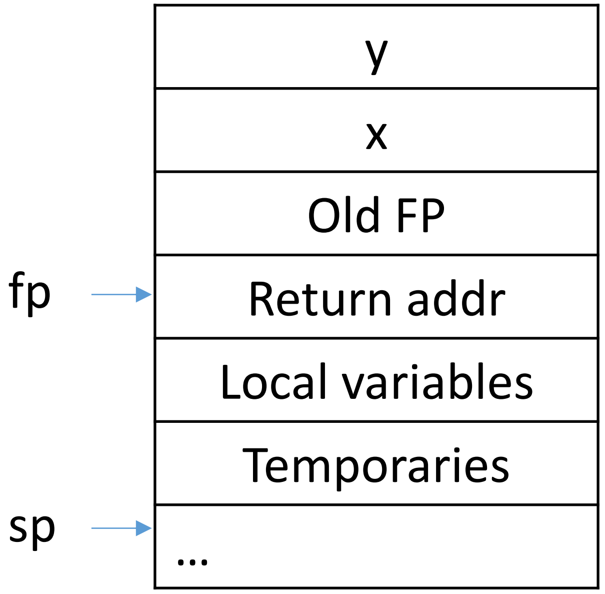
https://drive.google.com/file/d/1qe7it1bz7Ioaa8UBduaAv08XU8AFzpbt/view

Code Generation for Variables

- The “variables” of a function are just its ‘parameters’
  - They are all in the AR
  - Pushed by the caller
- **Problem**: because the stack grows when intermediate results are saved, the variables are not at a fixed offset from *sp*
  - Thus, access to locations in the stack frame cannot use *sp*-relative addressing
- **Solution**: use the frame pointer *fp* instead
  - Always points to the return address on the stack
  - Since it does not move, it can be used to find the variables

Example

- Local variables are referenced from an offset from *fp*
  - *fp* is pointing to *ra* (return address)
- For a function *def f(x,y) = e* the activation and frame pointer are set up as follows:



x: +8(fp)

y: +12(fp)

First local variable: -4(fp)

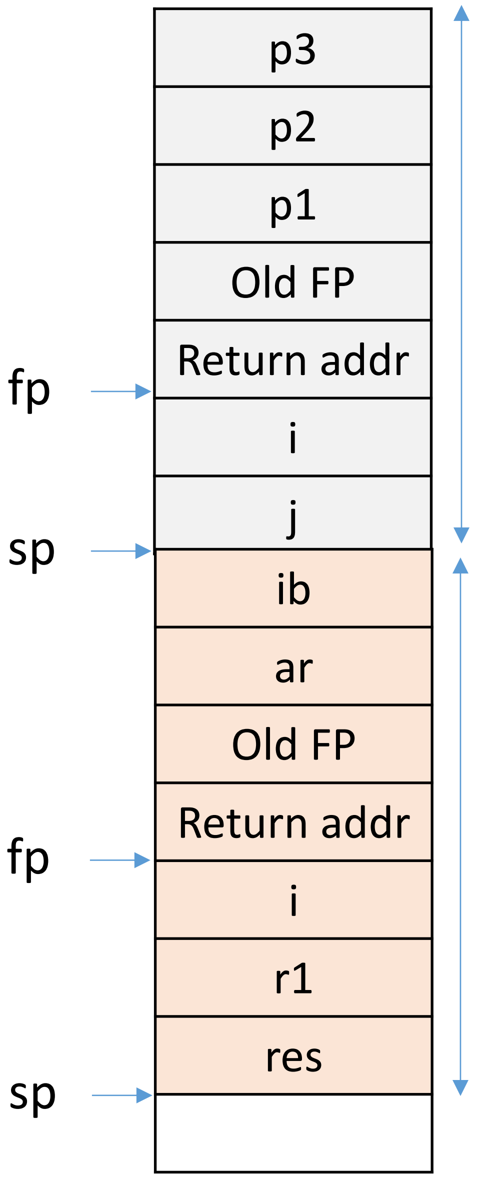
The parameters are pushed right to left by the caller

The locals are pushed left to right by the callee

Example

```
double fun1(int p1, double p2, int p3) {
    int i, j;
    res = fun2(p1*p2, j);
    return res;
}
```

```
double fun2(double ar, int ib) {
    int i, r1;
    double res;
    …
    return res;
}
```



Code Generation for OO

- Objects are like structs in C
  - Objects are laid out in contiguous memory
  - Each member variable is stored at a fixed offset in object
- Unlike structs, objects have member methods
- Two types of member methods:
  - **Nonvirtual** member methods: cannot be overridden

    ```
    Parent obj = new Child();
    obj.nonvirtual(); // Parent::nonvirtual() called
    ```

    Method called depends on (static) reference type

    Compiler can decide call targets statically
  - **Virtual** member methods: can be overridden by child class

    ```
    Parent obj = new Child();
    obj.virtual(); // Child::virtual() called
    ```

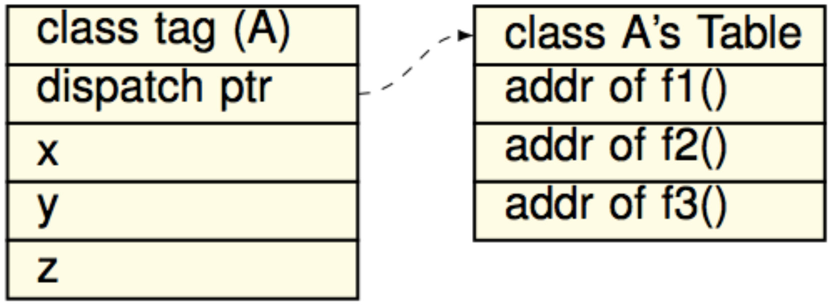
    Method called depends on (runtime) type of object

    Need to call different targets depending on runtime type

Static and Dynamic Dispatch

- **Dispatch**: to send to a particular place for a purpose
  - I.e., to jump to a (particular) function
- **Static Dispatch**: selects call target at compile time
  - Nonvirtual methods implemented using static dispatch
  - Implication for code generation:
    - Can hard code function address into binary
- **Dynamic Dispatch**: selects call target at runtime
  - Virtual methods implemented using dynamic dispatch
  - Implication for code generation:
    - Must generate code to select correct call target
- How?
  - At compile time, generate a **dispatch table** for each class, containing call targets for all virtual methods of that class
  - At runtime, each object has a pointer to its dispatch table, which is indexed into to find call target for its runtime type

Typical Object Layout



- Class tag is used for dynamic type checking
- Dispatch ptr is a pointer to the dispatch table
- Compiler translates member accesses to offset accesses

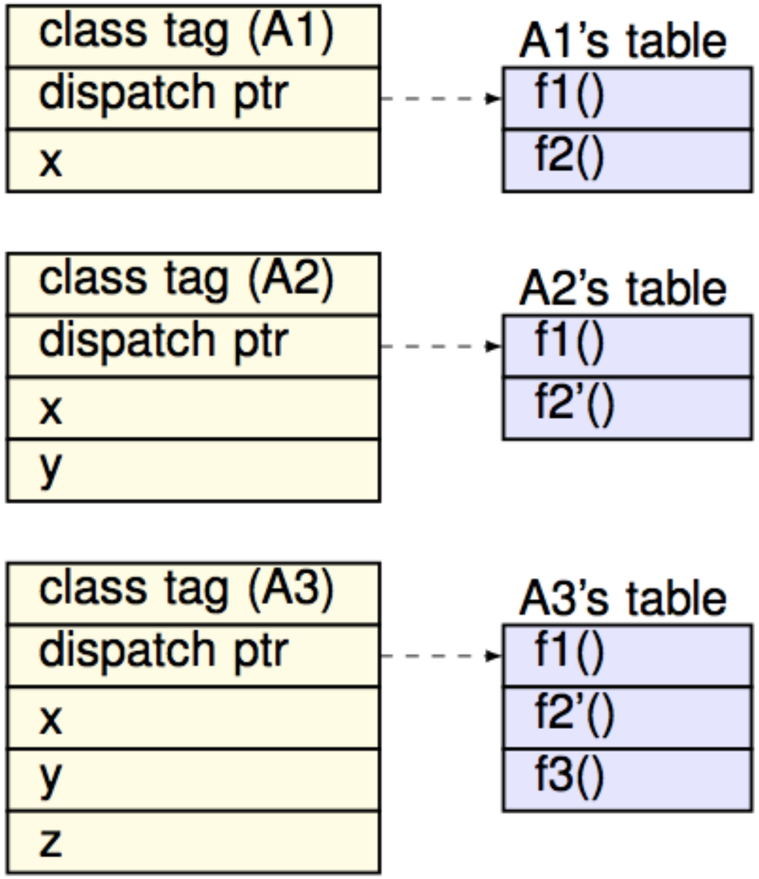
```
if(...) obj = new Parent()
else  obj = new Child();
obj.x = 10;             // move 10, x_offset(obj)
obj.f2();               // call f2_offset(obj.dispatch_ptr)
```

- Offsets must remain identical regardless of object type
  - How to layout object and dispatch table to make it so?

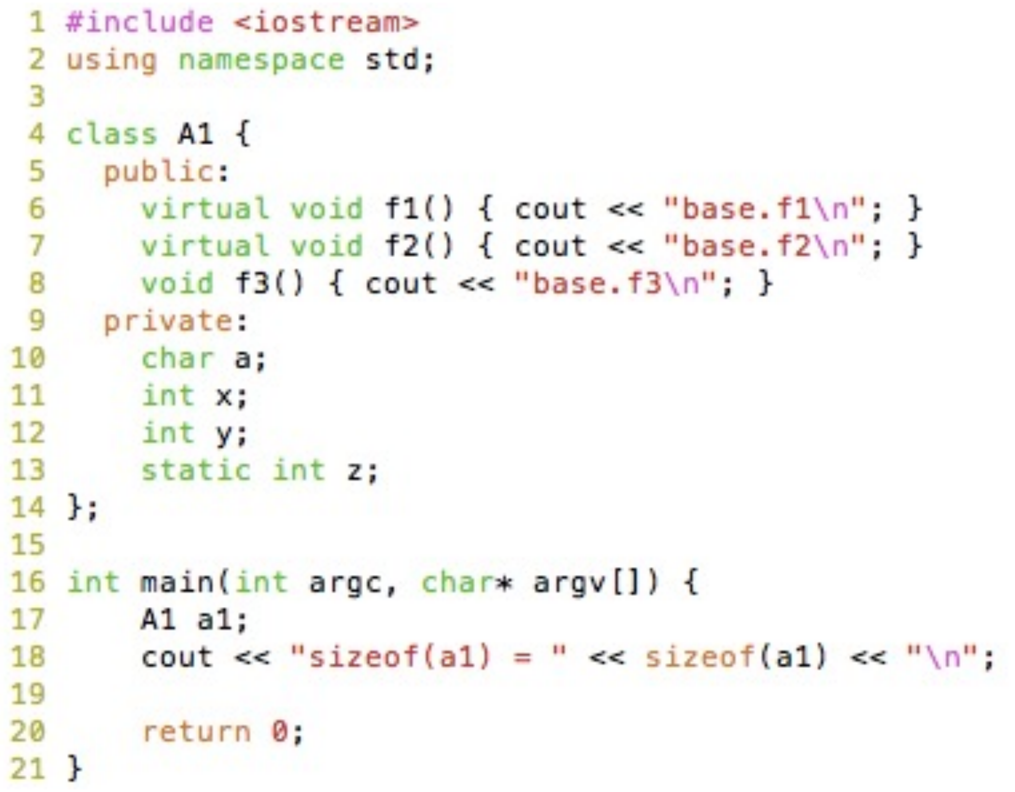
Inheritance and Subclasses

- Invariant: the offset of a member variable or member method is the same in a class and all of its subclasses

```
class A1 {
   int x;
   virtual void f1() { ... }
   virtual void f2() { ... }
}
class A2 inherits A1 {
   int y;
   virtual void f2() { ... }
}
class A3 inherits A2 {
   int z;
   virtual void f3() { ... }
}
```



A Question ...



```cpp
#include <iostream>
using namespace std;

class A1 {
  public:
    virtual void f1() { cout << "base.f1\n"; }
    virtual void f2() { cout << "base.f2\n"; }
    void f3() { cout << "base.f3\n"; }
  private:
    char a;
    int x;
    int y;
    static int z;
};

int main(int argc, char* argv[]) {
    A1 a1;
    cout << "sizeof(a1) = " << sizeof(a1) << "\n";

    return 0;
}
```

- What is the output?
  - **24** (on my 64-bit MBA)
- How come?
  - Fields (12B)
    - char a: 1 --> 4
    - int x: 4
    - int y: 4
  - Functions (8B)
    - virtual: 8B
  - Alignment
    - 12+8 --> 24

[1] Determining the Size of a Class Object
[2] sizeof class in C++

Heap Memory Management

- 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

Heap Memory Management (cont.)

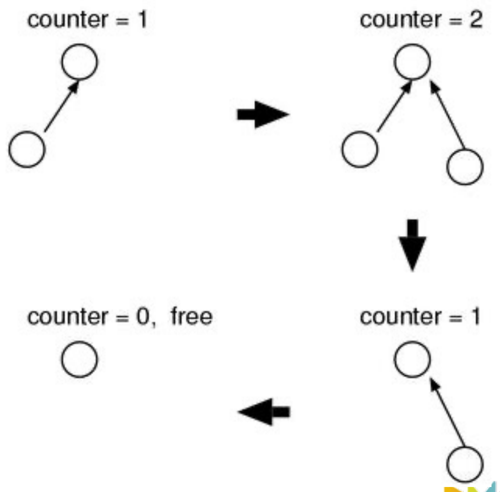
- Manual memory management is typically more efficient
  - Programmers know when data is no longer needed
    - With automatic management, runtime must somehow detect when data is no longer needed and recycle it, incurring overheads
- Automatic management leads to fewer bugs
  - Disallowing programmer free() calls is essential for security
- 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

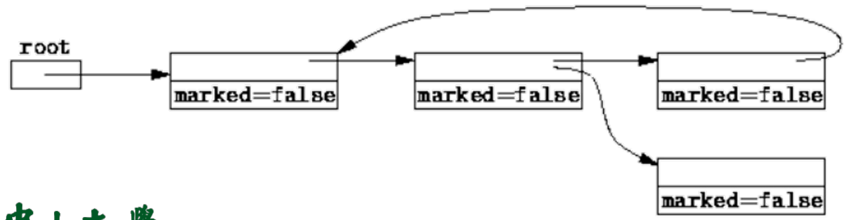
Heap Memory Management (cont.)

- Detection: determining an object will no longer be used
  - In general, impossible for compiler to tell exactly
    - Requires knowledge of program beyond what compiler has
  - But compiler can tell when it can no longer be used
- An object x is **reachable** iff
  - A named object contains a reference to x, or
  - A reachable object y contains a reference to x
- An unreachable object is referred to as **garbage**
  - Garbage can no longer be used and its memory can be reclaimed
  - This reclamation is process is called **garbage collection**

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/标记清除]
  - When the heap runs out of memory to allocate:
    - 1. Pause the program
    - 2. Trace through all reachable objects
    - 3. Garbage collect remaining objects
    - 4. Restart the program





https://drive.google.com/file/d/1qe7it1bz7Ioaa8UBduaAv08XU8AFzpbt/view

Machine Optimizations[机器相关优化]

- After performing IR optimizations
  - We need to further convert the optimized IR into the target language (e.g. assembly, machine code)
- Specific machines features are taken into account to produce code optimized for the particular architecture[考虑特定的架构特性]
  - E.g., specialized instructions, hardware pipeline abilities, register details
- Typical machine optimizations[典型的优化方案]
  - **Instruction selection and scheduling**: select and reorder insts to implement the operators in IR
  - **Register allocation**: map values to registers and manage
  - **Peephole optimization**: locally improve the target code

Instruction Selection[指令选取]

- To find an efficient mapping from the IR of a program to a target-specific assembly listing[IR到汇编的映射]
- Instruction selection is particularly important when targeting architectures with CISC (e.g., x86)
  - In these architectures there are typically several possible implementations of the same IR operation, each with different properties
  - e.g., on x86 an addition of one can be implemented by an *inc*, *add*, or *lea* instruction

x = y + z

```
MOV y,R0
ADD z,R0
MOV R0,x
```

a = a + 1

```
MOV a,R0
ADD #1,R0
MOV R0,a
```

$\Rightarrow$

```
MOV a,R0
INC R0
MOV R0,a
```

Instruction Cost[指令成本]

- Instruction cost = 1 + cost(source-mode) + cost(destination-mode)

Mode	Form	Address	Added Cost
Absolute	**M**	**M**	1
Register	**R**	**R**	0
Indexed	*c*(**R**)	*c*+*contents*(**R**)	1
Indirect register	***R**	*contents*(**R**)	0
Indirect indexed	****c*(**R**)	*contents*(*c*+*contents*(**R**))	1
Literal	**#***c*	N/A	1

- Examples

Instruction	Operation	Cost
`MOV R0,R1`	Store *content*(**R0**) into register **R1**	1
`MOV R0,M`	Store *content*(**R0**) into memory location **M**	2
`MOV M,R0`	Store *content*(**M**) into register **R0**	2
`MOV 4(R0),M`	Store *contents*(4+*contents*(**R0**)) into **M**	3
`MOV *4(R0),M`	Store *contents*(*contents*(4+*contents*(**R0**))) into **M**	3
`MOV #1,R0`	Store 1 into **R0**	2
`ADD 4(R0),*12(R1)`	Add *contents*(4+*contents*(**R0**)) to *contents*(12+*contents*(**R1**))	3

Instruction Cost (cont.)

- Suppose we translate TAC x:=y+z to:
  - MOV *y*, R0
  - ADD *z*, R0
  - MOV R0, *x*

Mode	Form	Address	Added Cost
Absolute	**M**	**M**	1
Register	**R**	**R**	0
Indexed	*c*(**R**)	*c*+*contents*(**R**)	1
Indirect register	***R**	*contents*(**R**)	0
Indirect indexed	***c*(**R**)	*contents*(*c*+*contents*(**R**))	1
Literal	#*c*	N/A	1

- a := b + c

```
MOV b, R0
ADD c, R0
MOV R0, a
```

cost = 6

```
MOV b, a
ADD c, a
```

cost = 6

```
MOV *R1, *R0
ADD *R2, *R0
```

Assuming R0, R1 and R2 contain the addresses of a, b, and c

cost = 2

- a := a + 1

```
MOV a, R0
ADD #1, R0
MOV R0, a
```

cost = 6

```
ADD #1, a
```

cost = 3

```
INC a
```

cost = 2

Instruction Scheduling[指令调度]

- Some facts
  - Instructions take clock cycles to execute (latency)
  - Modern machines issue several operations per cycle (Out-of-Order execution)
  - Cannot use results until ready, can do something else
  - Execution time is order-dependent
- Goal: reorder the operations to minimize execution time
  - Minimize wasted cycles
  - Avoid spilling registers
  - Improve locality

```
A = x * y;
B = A + 1;
C = y;
```

→

```
A = x * y;
C = y;
B = A + 1;
```

(Now C=y; can execute while waiting for A=x*y;)

Register Allocation[寄存器分配]

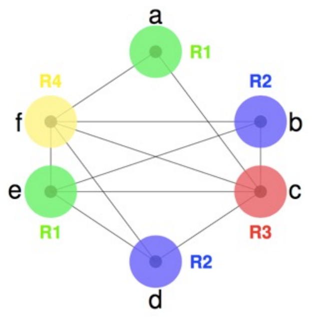
- In TAC, there are an unlimited number of variables
  - On a physical machine there are a small number of registers
- **Register allocation** is the process of assigning variables to registers and managing data transfer in and out of registers
  - How to assign variables to finitely many registers?
  - What to do when it can't be done?
  - How to do so efficiently?
- Using registers intelligently is a critical step in any compiler
  - Accesses to memory are costly, even with caches
  - A good register allocator can generate code orders of magnitude better than a bad register allocator

Register Allocation (cont.)

- Goals of register allocation
  - Keep frequently accessed variables in registers
  - Keep variables in registers only as long as they are live
- Local register allocation[局部]
  - Allocate registers basic block by basic block
  - Makes decisions on a per-block basis (hence ‘local’)
- Global register allocation[全局]
  - Makes global decisions about register allocation such that
    - Var to reg mappings remain consistent across blocks
    - Structure of CFG is taken into account on decisions
- Three well-known register allocation algorithms
  - Graph coloring allocator[图着色]
  - Linear scan allocator[线性扫描]
  - LP (Integer Linear Programming) allocator[整数线性规划]

Graph Coloring[图着色]

- Register interference graph (RIG)[相交图]
  - Each node represents a variable
  - An edge between two nodes V_1 and V_2 represents an interference in live ranges[活跃期/生存期]



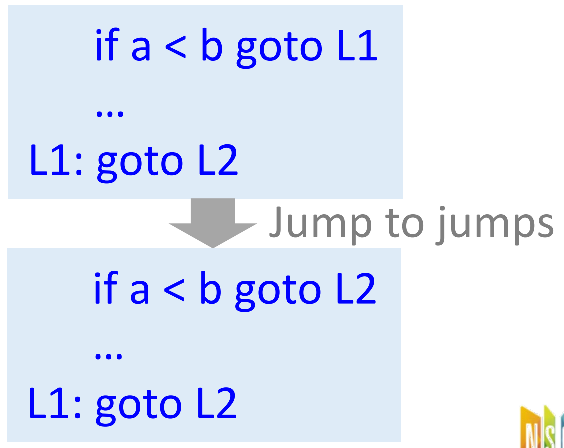
- Based on RIG,
  - Two variables can be allocated in the same register if there is no edge between them[若无边相连，可使用同一寄存器]
  - Otherwise, they cannot be allocated in the same register
- Problem of register allocation maps to graph coloring
  - Once solved, *k* colors can be mapped back to *k* registers
  - If the graph is *k*-colorable, it’s *k*-register-allocatable

Register Spilling[寄存器溢出]

- Determining whether a graph is *k*-colorable is NP-complete
  - Therefore, problem of *k*-register allocation is NP-complete
  - In practice: use heuristic polynomial algorithm that gives close to optimal allocations most of the time
  - Chaitin's graph coloring is a popular heuristic algorithm
    - E.g. most backends of GCC use Chaitin's algorithm
- What if *k*-register allocation does not exist?
  - Spill a variable to memory to reduce RIG and try again
  - Spilled var stays in memory and is not allocated a reg
- Spilling is slow
  - Placed into memory, loaded into register when needed, and written back to memory when no longer used

Peephole Optimization[窥孔优化]

- Optimization ways
  - Usual: produce good code through careful inst selection and register alloca
  - Alternative: generate naïve target code and then improve
- A simple but effective technique for locally improving the target code[很局部的优化，但可能带来性能的极大提升]
  - Done by examining a sliding window of target instructions (called **peephole**) and replacing instruction sequences within the peephole by a shorter or faster sequence, whenever psbl
  - Can also be applied directly after IR generation to improve IR
- Example transformations
  - Redundant-instruction elimination
  - Flow-of-control optimizations
  - Algebraic simplifications
  - Use of machine idioms


```
    if a < b goto L1
    …
L1: goto L2
```

```
    if a < b goto L2
    …
L1: goto L2
```

LLVM

- llc: LLVM static compiler
  - Input: *.ll* or *.bc*
  - Output: assembly language for a specified architecture
- End-user options

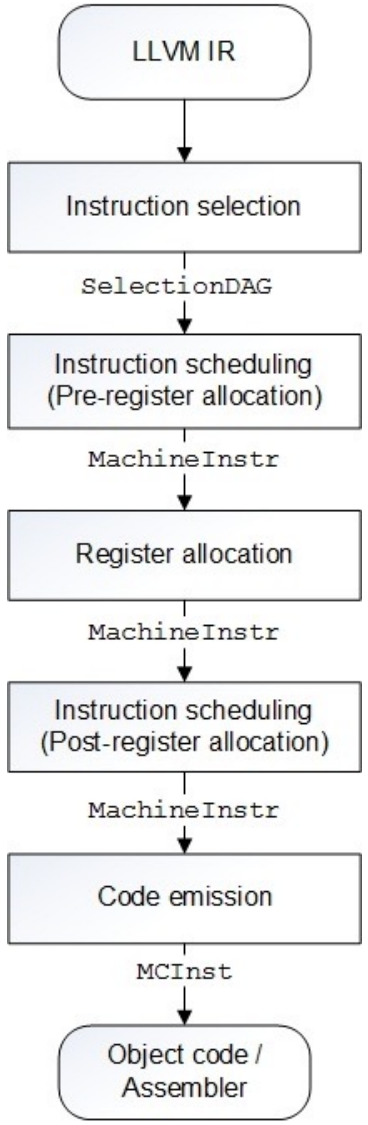
  -march=<arch>: e.g., x86

  -mcpu=<cpuname>: e.g., corei7-avx
- Tuning/Configuration Options

  --print-after-isel: print generated machine code after instruction selection (useful for debugging)

  --regalloc=<allocator>: specify the register allocator to use, basic/fast/greedy/pdqp

  --spiller=<spiller>: simple/local



https://www.llvm.org/docs/CommandGuide/llc.html

Optimizations[总结]

- Code can be optimized at different levels with various techniques
  - Peephole, local, loop, global
  - IR: local, global, common subexpression elimination, constant folding and propagation, …
  - Target: instruction, register, peephole, …
- Interactions between the various optimization techniques
  - Some transformations may expose possibilities for others
  - One opt. may obscure or remove possibilities for others
- Affect of compiler opts are intertwined and hard to separate
  - Finding optimal opt combinations is in itself research
  - Compilers package opts that typically go together into levels (e.g -O1, -O2, -O3)

Thanks and good luck!

张献伟

副教授@计算机学院/国家超算广州中心(NSCC-GZ)

研究方向：计算机体系结构、编程及编译优化、高性能及智能计算

顾宇浩

博士生'22@NSCC-GZ
研究方向：高性能计算、编译系统

郑中淳

硕士生'23@NSCC-GZ
研究方向：编译系统、计算机体系结构

潘文轩

硕士生'23@NSCC-GZ
研究方向：GPU系统、编译优化设计

单招文

硕士生'22@NSCC-GZ
研究方向：AI应用、高性能计算

郑腾扬

本科生'20@NSCC-GZ
研究方向：高性能计算、编译系统

孙高锦

本科生'20@NSCC-GZ
研究方向：编译系统、计算机体系结构

席梦悦

硕士生'23@NSCC-GZ
研究方向：GPU系统、编译优化设计

黄瀚

硕士生'23@NSCC-GZ
研究方向：高性能计算、编译优化设计

陈淏泉

本科生'23@NSCC-GZ
研究方向：编译系统、高性能计算