Compilation Principle
编 译 原 理

第21讲：目标代码生成(2)

张献伟

xianweiz.github.io

DCS290, 06/10/2021

Quiz

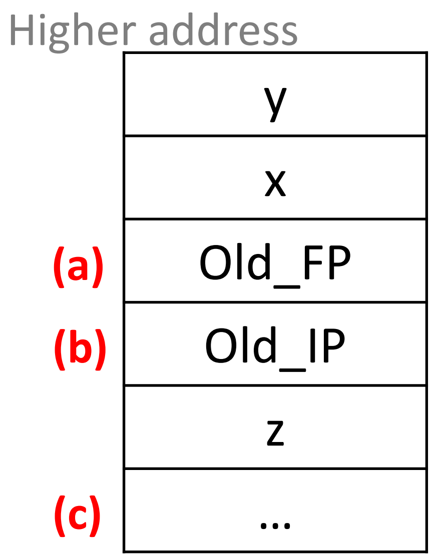
右图是函数调用过程中的栈空间：

- Q1: 确定$fp和$sp的位置。从(a)(b)(c)中选择。
- Q2: 在右图x/y/z三者中，哪些是函数参数？哪些是局部变量?
- Q3: 栈元素Old_IP存放什么信息？
- Q4: 以下目标代码在做什么?

```
add $sp $sp -4; sw $t0 0($sp)
```

- Q5: 以下指令是否正确？请简述理由。（假设针对MIPS架构）

```
add 0($sp) $t0
```



Quiz Solutions

右图是函数调用产生的栈空间：

- Q1: 确定$fp和$sp的位置。从(a)(b)(c)中选择。

  $fp: (b), $sp: (c)

- Q2: 在右图x/y/z三者中，哪些是函数参数？哪些是局部变量?

  参数：x, y　　局部变量：z

- Q3: 栈元素Old_IP存放什么信息？

  返回地址，也即函数调用处的下一条指令的地址

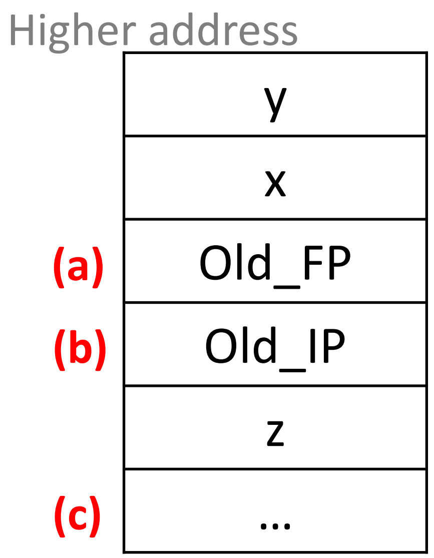
- Q4: 以下目标代码在做什么?

  ```
  add $sp $sp -4; sw $t0 0($sp)
  ```

  分配4字节栈空间，然后将t0寄存器中的值放入栈顶

- Q5: 以下指令是否正确？请简述理由。（假设针对MIPS架构）错误：只有load/store可以操作内存，其他指令只能从寄存器中取操作数

  ```
  add 0($sp) $t0
  ```



Code Generation for OO

- Objects are like structures in C
  - Objects are laid out in contiguous memory
  - Each member variable is stored at a fixed offset in object
- Unlike structures, 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
    ```

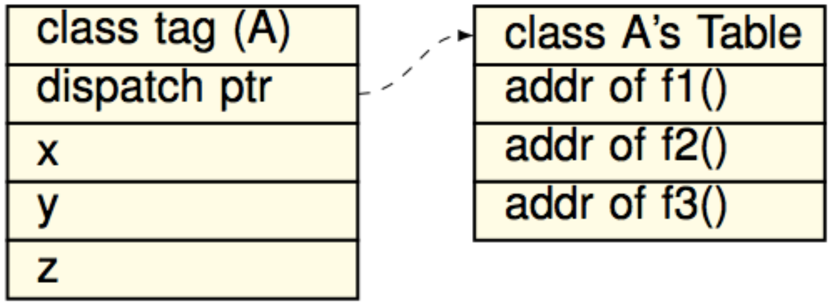
    Method called depends on (runtime) type of object

    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 <u>class</u>, containing call targets for all virtual methods of that class
  - At runtime, each <u>object</u> 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

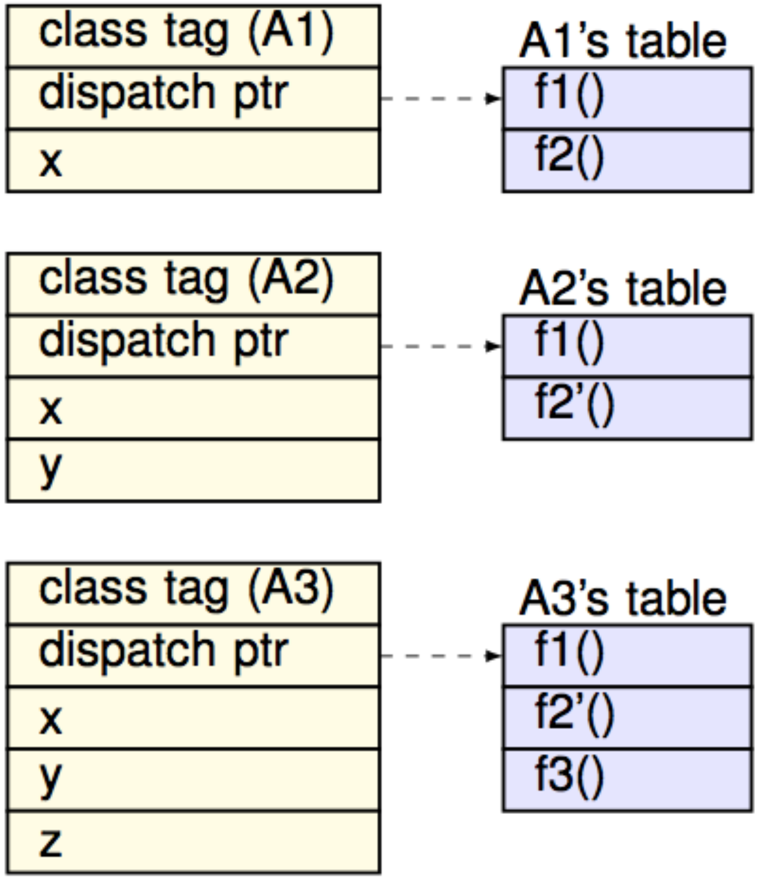
```
if(...) obj = new Parent()
else  obj = new Child();
obj.x = 10;          // move 10, x_offset(obj)
obj.f2();            // call f2_offset(obj.dispatch_ptr)
```

- 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() { ... }
}
```



Inheritance and Subclasses (cont.)

- Member variable access
  - Generate code using offset for reference type (class)
  - Object may be of child type, but will still have same offset

- Member method call
  - Generate code to load call target from dispatch table using offset for reference type
  - Again, object may be of child type, but still same offset

- No inheritance in our project
  - No dynamic dispatching
  - Statically bind a function call to its address

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++

Compilation Principle
编 译 原 理

第21讲：代码优化(1)

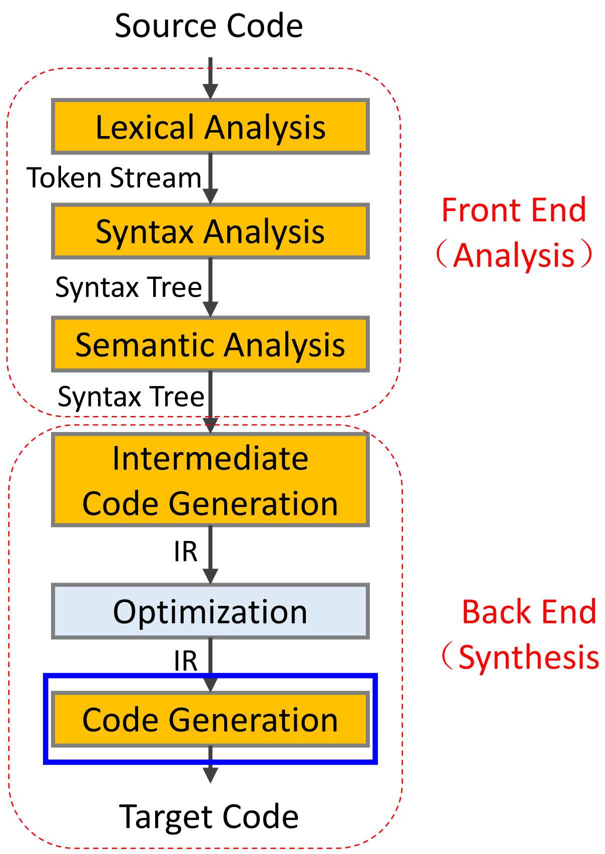
张献伟

xianweiz.github.io

DCS290, 06/10/2021

Optimization [代码优化]

- What we have now
  - IR of the source program (+symbol table)
- Goal of optimization[优化目标]
  - Improve the IR generated by the previous step to take better advantage of resources
- A very active area of research[研究热点]
  - Front end phases are well understood
  - Unoptimized code generation is relatively straightforward
  - Many optimizations are NP-complete
    - Thus usually rely on heuristics and approximations



To Optimize: Who, When, Where?

- Manual: source code
  - Select appropriate algorithms and data structures
  - Write code that the compiler can effectively optimize
    - Need to understand the capabilities and limitations of compiler opts.
- **Compiler**: intermediate representation Focus
  - To generate more efficient TAC instructions
- **Compiler**: final code generation
  - E.g., selecting effective instructions to emit, allocating registers in a better way
- Assembler/Linker: after final code generation
  - Attempting to re-work the assembly code itself into something more efficient (e.g., link-time optimization)

Example

```
int find_min(const int* array, const int len) {
  int min = a[0];
  for (int i = 1; i < len; i++) {
    if (a[i] < min) { min = a[i]; }
  }
  return min;
}
int find_max(const int* array, const int len) {
   int max = a[0];
   for (int i = 1; i < len; i++) {
     if (a[i] > max) { max = a[i]; }
   }
   return min;
}
void main() {
  int* array, len, min, max;
  initialize_array(array, &len);
  min = find_min(array, len);
  max = find_max(array, len);
  ...
}
```

Inline

Loop merge

```
void main() {
  int* array, len, min, max;
  initialize_array(array, &len);
  min = a[0]; max = a[0];
  for (int i = 0; i < len; i++) {
    if (a[i] < min) { min = a[i]; }
    if (a[i] > max) { max = a[i]; }
  }
  ...
}
```

Overview of Optimizations

- Goal of optimization is to generate **better** code[更好的代码]
  - Impossible to generate **optimal** code (so, it is <u>improvement</u>, actually)
    - Factors beyond control of compiler (user input, OS design, HW design) all affect what is optimal
    - Even discounting above, it’s still an NP-complete problem
- Better one or more of the following (in the average case)
  - **Execution time** [运行时间]
  - **Memory usage** [内存使用]
  - Energy consumption [能耗]
    - To reduce energy bill in a data center
    - To improve the lifetime of battery powered devices
  - Binary executable size [可执行文件大小]
    - If binary needs to be sent over the network
    - If binary must fit inside small device with limited storage
  - Other criteria [其他]
- Should <u>never</u> change program semantics[正确性是前提]

Types of Optimizations

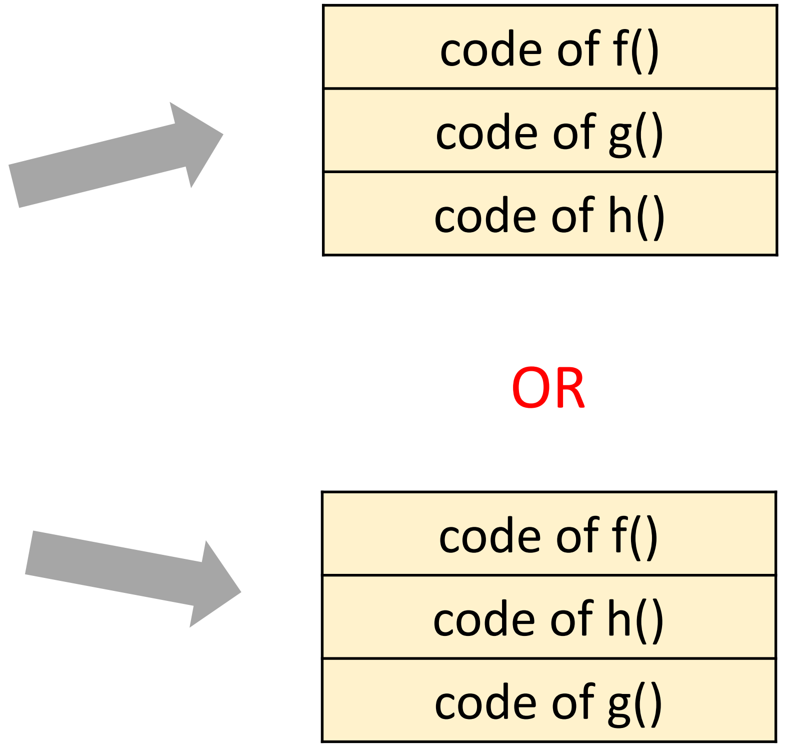
- Compiler optimization is essentially a transformation[转换]
  - Delete / Add / Move / Modify something
- **Layout-related** transformations[布局相关]
  - Optimizes *where* in memory code and data is placed
  - Goal: maximize **spatial locality** [空间局部性]
    - Spatial locality: on an access, likelihood that nearby locations will also be accessed soon
    - Increases likelihood subsequent accesses will be faster
      - E.g. If access fetches cache line, later access can reuse
      - E.g. If access page faults, later access can reuse page
- **Code-related** transformations[代码相关]
  - Optimizes *what* code is generated
  - Goal: execute least number of most costly instructions

Focus

Layout-Related Opt.: Code

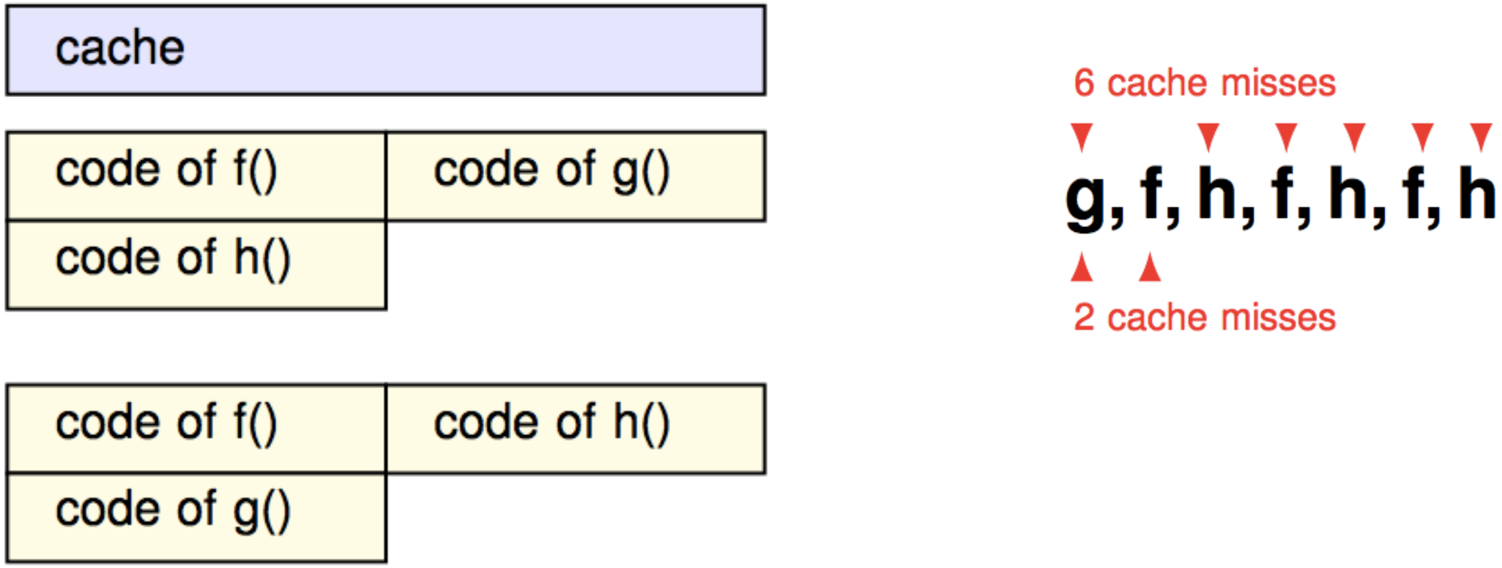
- Two ways to layout code for the below example

```
f() {
   …
   h();
   …
}
g() {
   …
}
h() {
  …
}
```



Layout-Related Opt.: Code (cont.)

- Which code layout is better?
- Assume
  - data cache has one *N*-word line
  - the size of each function is *N/2*-word long
  - access sequence is "**g, f, h, f, h, f, h**"



Layout-Related Opt.: Data

- Change the variable declaration order

```
struct S {
   int x1;
   int x2[200];
   int x3;
} obj[100];

for(...) {
   ... = obj[i].x1 + obj[i].x3;
}
```

```
struct S {
   int x1;
   int x3;
   int x2[200];
} obj[100];

for(...) {
   ... = obj[i].x1 + obj[i].x3;
}
```

- Improved spatial locality
  - Now x1 and x3 likely reside in same cache line
  - Access to x3 will always hit in the cache

Layout-Related Opt.: Data (cont.)

- Change AOS (array of structs) to SOA (struct of arrays)

```
struct S {
   int x;
   int y;
} points[100];

for(...) {
   ... = points[i].x * 2;
}
for(...) {
   ... = points[i].y * 2;
}
```

```
struct S {
   int x[100];
   int y[100];
} points;

for(...) {
   ... = points.x[i] * 2;
}
for(...) {
   ... = points.y[i] * 2;
}
```

- Improved spatial locality for accesses to ‘x’s and ‘y’s

Code-Related Optimizations

- Modifying code e.g. **strength reduction**

  A=2*a; ≡ A=a«1;

- Deleting code e.g. **dead code elimination**

  A=2; A=y; ≡ A=y;

- Moving code e.g. **code scheduling**

  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;)

- Inserting code e.g. **data prefetching**[数据预取]

  ```
  while (p!=NULL)
  { process(p); p=p->next; }
  ≡
  while (p!=NULL)
  { prefetch(p->next); process(p); p=p->next; }
  ```

  (Now access to p->next is likely to hit in cache)

Control-Flow Analysis[控制流分析]

- The compiling process has done lots of analysis
  - Lexical
  - Syntax
  - Semantic
  - IR
- But, it still doesn't really know how the program does what it does
- **Control-flow analysis** helps compiler to figure out more info about how the program does its work
  - First construct a control-flow graph, which is a graph of the different possible paths program flow could take through a function
    - To build the graph, we first divide the code into basic blocks

Basic Block[基本块]

- A **basic block** is a maximal sequence of instructions that
  - Except the first instruction, there are no other labels
  - Except the last instruction, there are no jumps
- Therefore, [进出口唯一]
  - Can only jump into the beginning of a block
  - Can only jump out at the end of a block
- Are units of control flow that cannot be divided further
  - All instructions in basic block execute or none at all

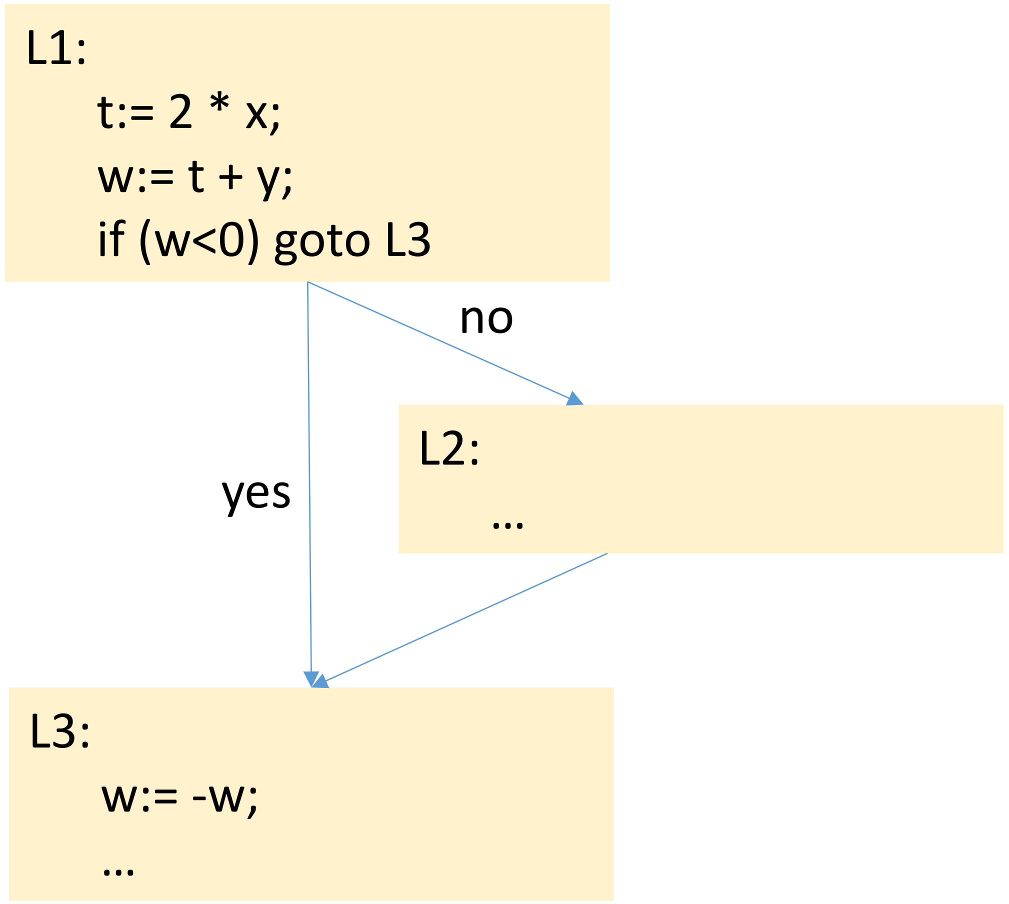
- <u>Local optimizations</u> are limited to scope of a basic block
- <u>Global optimizations</u> are across basic blocks

Control Flow Graph[控制流图]

- A **control flow graph** is a directed graph in which
  - Nodes are basic blocks
  - Edges represent flow of execution between basic blocks
    - Flow from end of one basic block to beginning of another
    - Flow can be result of a control flow divergence
    - Flow can be result of a control flow merge
  - Control statements introduce control flow edges
    - e.g. if-then-else, for-loop, while-loop, ...
- CFG is widely used to represent a function
- CFG is widely used for program analysis, especially for global analysis/optimization

Example

```
L1:
    t:= 2 * x;
    w:= t + y;
    if (w<0) goto L3
L2:
    ...
L3:
    w:= -w
     ...
```



Construct CFG

- Step 1: partition code into basic blocks[分解为基本块]
  - Identify **leader** instructions that are
    - the first instruction of a program, or
    - target instructions of jump instructions, or
    - instructions immediately following jump instructions
  - A basic block consists of a leader instruction and subsequent instructions before the next leader
- Step 2: add an edge between basic blocks B1 and B2 if[连接基本块]
  - B2 follows B1, and B1 may “fall through” to B2[相邻]
    - B1 ends with a conditional jump to another basic block[若条件假，到达B2]
    - B1 ends with a non-jump instruction (B2 is a target of a jump)[无跳转，B1顺序执行到达B2]
    - Note: if B1 ends in an unconditional jump, cannot fall through[B1无条件跳转，会绕开B2]
  - B2 doesn’t follow B1, but B1 ends with a jump to B2 [不相邻，但B2是B1的跳转目标]

Example

- Partition code into basic blocks
  - Identify leader instructions
- Add edges between basic blocks

```
01:       A=4
02:       T1=A*B
03.  L1:  T2=T1/C
04:       if (T2<W) goto L2
05:       M=T1*K
06:       T3=M+1
07:  L2:  H=I
08:       M=T3-H
09:       if (T3>0) goto L3
10:       goto L1
11:  L3:  halt
```