惡補

惡性補習我失去的東西，一切關於著 Compiler。

這個是要答辯的 Lab。佔總成績 20% 的 Lab 5。考慮到 Lab 5 爆炸會連帶著炸掉 Lab 6，可以認為佔總成績的分數是 40%。

如果没能成功完成这些，整门课就不能通过。也就是说之前所做的事情都是白干。

前四个 Lab 加上期中考，一共仅仅占有 40% 而已。

我需要知道什麼？

• **src/tiger/frame/.*** Files related to function stack frame(chapter 6)

• **src/tiger/translate/.*** Files related to IR tree translation(chapter 7)

• **src/tiger/canon/.*** Files related to basic blocks & traces(chapter 8, this part has already been implemented)

• **src/tiger/codegen/.*** Files related to assembly code generation(chapter 9)

• **src/tiger/runtime/runtime.c** Tiger program runtime file(will be linked with code files generated by your compiler)

• **src/tiger/env/env.*** The EnvEntry classes which help compiler store the information of variables&functions

我該看哪些章節？

Before you start this lab, you should carefully read the chapter **6, 7, 8, 9, 12** of the textbook.

> 那我開始了。

Chapter 6: Activation Records

在實現 Tiger 的過程中，主要的難點是支持下面兩個東西：

- 局部變量

在一個語句塊（Scope）中，要可以訪問在*當前語句塊內*及*上層語句塊內*定義的變量。

- 函數嵌套

  函數像一個普通變量一樣可以嵌套聲明。內層函數體裏可以調用同層及外層函數（這裡的同層包括它自己：遞歸調用）。

  但是為了簡化起見，Tiger 不支持將函數作為返回值。ML 跟 Scheme 等語言都是支持的。

需要知道，所謂的這些局部變量、函數等等東西，都是基於一層基本的抽象：棧幀（Stack Frame）。

棧本身不是什麼高級東西；他是一種支持 Push 和 Pop 的數據結構；且總是保證先進後出。

要理解 Stack 本身，只需要想像一層疊放在一起的紙張就好了；總是只能從頂部放一張、或取一張。

棧位置

在聲明局部變量時，我們採用一套約定俗成的辦法：

> 在進入 Scope 時將該層級的局部變量成批壓入棧；退出該 Scope 時則按照相反順序成批彈出棧。

但是不像 C89 那種「聲明必須在 Scope 開頭！」的老古董語言，實際上我們可以在語句塊內的任何位置離散地聲明變量。

因此上面的論述可能要改一改了：

> 在進入 Scope 時將該層級的局部變量**所需的空間**成批壓入棧；
>
> 退出該 Scope 時則按照相反順序成批將**佔用的空間**彈出棧。

或者，用我們所熟悉的 `x86-64` 的黑話來講，

> 在進入 Scope 時將 %rsp（指定棧頂位置的指針）減少以分配空間；
>
> 在退出 Scope 時將 %rsp 還原以回收空間。

因此，局部變量的**初始化**是可以在任何部分發生的；

然而其內存的分配及內存位置的確定則是在進入 Scope 時就已知了的。

因此，對於編譯型語言 Tiger，一個 Scope 中局部變量所需的內存是在編譯期就能確定的。

> 題外話：在這個 Lab 中，我們不用任何寄存器，因此所有的局部變量都是上棧的。

也就是說，在生成實際代碼之前，Tiger Compiler 需要做到：

- 計算局部變量所佔空間；

  以便在 Scope 開頭結尾放置 `subq %rsp, SIZE` 跟 `addq %rsp, SIZE` 這樣的語句。
- 為**每個**（在這個 Lab 裏，就是每個）變量計算棧內存偏移；

為了實現藉由 `%rsp(OFFSET)` 樣式的訪存。

`SIZE` 跟 `OFFSET` 都是生成代碼前的定數。

> 題外話：由於歷史原因，棧是倒置的；
>
> 棧頂指針向下（減小），同時棧空間增大；棧頂指針向上（增大），同時棧空間減小。

棧幀構造

一般的棧幀應該長什麼樣子呢？

（從高地址位到低地址位）依次是：

（上一層棧幀）

- 實參 n
- 實參 n − 1
- ……
- 實參 1
- 靜態鏈（這個是什麼我們待會說） <− 幀指針在此處

（本層棧幀）

- 局部變量區域
- 返回地址
- 臨時變量
- 保護的寄存器
- 實參 m
- 實參 m − 1
- ……
- 實參 1
- 靜態鏈 <− 棧指針在此處

（下一層棧幀）

留意到之所以這裡把上層的棧幀給寫出來，是因為那是根據調用層級來看，那些是上層調用者放好，給這一層看的實際參數。

因此實際上要先放局部變量、返回地址、臨時變量、上棧要保護的寄存器，然後如果要再往深層調用，就要再往上堆下層實際參數和靜態鏈。

另外，在 32 位下，會有 %esp 指針指向棧頂，另有 %ebp 指針指向上一層幀之底。

但是，在 64 位下，%rsp 仍承擔 %esp 的職位；%rbp 則退化為普通寄存器，不再具有幀指針的作用。

當然實際上也沒什麼影響；幀的大小在編譯期（晚期）就能確定（除了 C 的 variable array 之類的奇怪玩意）。

因此 %esp + FRAME_SIZE 也就總是等於 %ebp。這大概就是 %ebp 退休的原因吧。

> 題外話：實際上我們仍然會討論幀指針 %ebp：為什麼？
>
> 因為在編譯過程中，要到比較晚的時候（掃描完整個 Scope）我們才能確定幀大小。
>
> 因此我們在掃描 Scope 的過程中，先使用 %ebp 作為一個標的，等到結束再填入。

寄存器

你問我寄存器好不好，當然好啊……讀寫速度超快，不用訪存。

就是太少了點。

我們按照早期人們的編程習慣，把寄存器分為兩類：

一類是調用者負責存儲的（Caller Saved）；一類是被調用者負責存儲的（Callee Saved）。

簡單說，對於一個剛出生的棧幀來說，被調用者負責存儲的寄存器你可以隨便用；

因為你的上層棧幀一定幫你保存好了，在你死掉的時候還原之；

但是調用者負責存儲的寄存器你要是想用，得先把他們上棧，退出的時候出棧，搞得好麻煩。

而當你想要 Invoke 一個新棧幀的時候，要記得把調用者負責存儲的寄存器給上棧，調用完了之後出棧，搞得好麻煩；

但是被調用者負責存儲的寄存器呢，不用特別處理；那是 Callee 的責任。

簡單說…大部分棧幀都是上有老下有小；保存寄存器的責任是對半開的。

上面的事情都是約定俗成；你可以不遵守，但是在跟其他程序混調的時候就可能產生問題。

Use as your own risk.

參數傳遞

傳參的方式。

遠古計算機（最早的那一批）是採用一塊固定內存空間來進行參數放置的；

而稍後的古代（20 世紀 70 年代上下）計算機則是將全部的參數都進 Stack 放置。

而考慮到大部分函數調用的參數實際上都不超過十個，因此現代計算機採用「寄存器」+「棧」的組合方式來傳遞參數。

> 優先使用高速寄存器來傳參；寄存器不夠用了再用棧（這個過程稱作 Spill）。

> 當然在這個 Lab 裏，我們始終 Spill 所有參數。按順序（從 n 到 1 倒序）在棧裏排開就好了。

返回地址

為了保證函數調用棧總能返回，我們需要在棧幀結構中記錄下「返回位置」這個重要信息。

> ret 指令所行之事就是從棧中彈出頂部的一個地址，並跳到那個位置。

所以，在 call 一個函數之前（假設此處的地址為 M），通常會將 M + L（L 是 call 指令的長度） 地址 push 入棧，以確保 ret 函數知道該從哪兒回來。如果不這麼做，就無法保證「棧平衡」。

然而現代計算機中，返回地址是被放在制定寄存器之中，而非在棧幀之中的。

這樣，如果一個函數需要 call 子函數，那麼他就有責任保存這個寄存器的內容到棧幀中，並在退出之前還原其值；反之如果該函數是葉子函數（不用再 call 子函數），那麼就無需保存該寄存器的內容。

棧幀中的變量

講完參數、返回地址，我們就該提一提「局部變量」了。

在一個函數調用的過程中，難免會需要局部變量所佔用的內存區塊。但並非所有的變量都必須放入棧幀，而須滿足如下條件之一：

- 需要對該變量進行取地址操作（如 C/C++ 中的 & 運算符）
- 該變量被嵌套在此 Procedure 內的 Procedure 引用了
- 該變量值佔用空間過大而無法放入單個寄存器中
- 該變量是一個數組，引用其內容元素需要對內存尋址
- 局部變量和臨時變量過多，寄存器不夠用而被「Spill」到棧幀中

如果一個變量被作為「引用」（Reference）傳遞給函數，也就是說傳遞的是指針的話，或者一個變量被取了地址，那麼稱這個變量為逃逸的（Escaped），即不可以被放入寄存器的。

實際上，寄存器分配並不是一件很簡單的事；一開始遇到一個變量的時候，並不能夠知道他在之後被引用的情況，也就是沒有足夠的信息來確認分配策略。

一般的編譯器是會先將變量分配道臨時位置，等完全掃描之後再去分配寄存器。

> 在這個 Lab 裏，我們全部往 Stack 上塞。

靜態鏈

棧幀中的最後一塊拼圖就是靜態鏈（Static Link）了。

問題在這裡：Tiger 語言中的嵌套函數，內層函數可以使用外層函數中聲明的變量。

> Pascal 跟 ML 都是這樣。

實際上有多種方式可以實現嵌套變量的訪問，主要下面三種：

- 靜態鏈：每當調用函數 f 的時候，同時傳給 f 一個指針，指向靜態包含 f 的那個函數。這個指針稱為

靜態鏈。如果想要找上層、上上層、……的嵌套函數的變量，那麼就必須走靜態鏈跳一次、跳一次、……。

- 全局數組：用一個全局的數組來記錄；數組的第 i 個位置放的是最後一次進入的棧幀中嵌套深度為 i 的棧幀位置。這個被稱作嵌套層次顯示表。
- λ 提升：在調用嵌套函數的時候，直接把自己有的局部變量一併發給內層函數，作為額外參數的形式。這個方法稱為 λ 提升法。

這裏我們用第一個方法（靜態鏈法）。

每次調用子函數的時候，都把一個指針傳遞給子函數，該指針指向的是程序正文中直接包含該函數的父親函數最近一次進入的活動紀錄。

Tiger 編譯器的棧幀

有必要留意到：不同的體系結構（Architecture）是有著不同的棧幀結構的。

最簡單的說，不同體系結構指針長度都不一樣。怎麼可能完全一致呢？

因此我們再插入一層抽象（呵呵），通過 `frame.h` 提供操作棧幀的抽象方法，而具體方法則分平台實現。

下面是 Frame 相關的接口：

```
typedef struct F_frame_ *F_frame;
typedef struct F_access_ *F_access;
typedef struct F_accessList_ *F_accessList;

struct F_accessList_ {
    F_access head;
    F_accessList tail;
};

F_frame F_newFrame(Temp_label name, U_boolList formals);
Temp_label F_name(F_frame f);
F_accessList F_formals(F_frame f);
F_access F_allocLocal(F_frame f, bool escape);
```

API 用法

如果我們要調用一個子函數，我們就得為他新開一個棧幀。

使用 `F_newFrame(f, l)` 函數來開。其中 l 是 k 個 Boolean 組成的表，True 表示該位置的參數是逃逸的（看上面的 @escaping 說法），False 則表示非逃逸的。f 則是我們的函數。

因此調用實例是：

```
F_newFrame(g, U_BoolList(true, U_BoolList(false, U_BoolList(false,
NULL)));
```

> 用遞歸鏈表來保存數組，要麼是這個年代機能有限；要麼是作者腦子有坑。

這樣調用就能返回一個棧幀了。

F_access 則是用於訪問存放在棧或僅存起裏的形式參數和局部變量。他是抽象的數據類型，僅在 Frame 模塊中可用。

F_access_ 的定義如下：

```
struct F_access_ {
    enum {
        inFrame,
        inReg
    } kind;
    union {
        int offset;     /* if that's stored in frame */
        Temp_temp reg;  /* if that's stored in register */
    } u;
};

static F_access InFrame(int offset);
static F_access InReg(Temp_temp reg);
```

InFrame(X)會將一個變量放在相對指針偏移量為 X 的存儲位置（棧中存放該變量的位置）；

InReg(X) 會將該變量放在指定的寄存器位置。

重要留意：這些東西都是站在被調用者，也就是子函數的立場上看的。也就是說，按照子函數的幀指針（%rbp）偏移，或者說按照上層調用者的棧指針（%rsp）偏移。同時寄存器也是在子函數進入之後的寄存器分佈，不要按照外層調用者的立場來看。

留意到他們都是抽象的，因此在 Frame 模塊之外是不可以訪問他們的。

下一章會提到怎麼在外部 call 她們。

F_formals 接口函數抽取由 k 個「Call」組成的一張表。每次調用 InFrame 和 InReg 稱為一 Call。在完成 k 次 Call 之後，就可以調用 F_newFrame 了。

> 這種調用規則真是非常不容易出錯呢

局部變量

局部變量部分上棧，部分進寄存器。

我們怎麼聲明我們這裡有一個局部變量？

通過調用函數 F_allocLocal(f, BOOLEAN) 可以聲明我們需要一個局部變量。

BOOLEAN 值設定為 True，即是說明該局部變量是逃逸的，一定會被放置在棧空間上，並且該函數會返回棧地址。

BOOLEAN 值設定為 False，則該局部變量是非逃逸的，可能會被放置在寄存器中以加速。此時返回值可能是棧地址，也可能是寄存器號。

> 不必要一開棧幀，就調用 `allocLocal`。事實上除了在 C89 中以外，我們基本上可以在任何位置創建局部變量。甚至還可能會因嵌套地創建同名的臨時變量而帶來覆蓋問題。這些都是需要考慮的事情。

計算逃逸變量

非逃逸的局部變量可以被放在寄存器裏，也可以被放在棧幀裏。而逃逸變量則只能放在棧幀裏。

因此，我們關心的是有限制的「逃逸變量」。得想個辦法把他們找出來。確定一個變量是否逃逸，上面已經有明確的規則。

因此我們也有對應的方法來找出哪些逃逸的變量。

```
/* escape.h */
void Esc_findEscape(A_exp exp);

/* escape.c */
static void traverseExp(S_table env, int depth, A_exp e);
static void traverseDec(S_table env, int depth, A_dec d);
static void traverseVar(S_table env, int depth, A_var v);
```

留意到，這些函數都是相互遞歸（以及自遞归）的。在發現一個逃逸變量的時候，會怎麼做？

他會將 `VarDec` 的 `Escape` 設定為 True。如此，查看 Var 的這個 Field 就能了解他是否逃逸了。

具體的逃逸判斷策略，各語言自然都有區別。可以通過改變 `escape.c/h` 的內容來調整。

臨時變量 + 標號

什麼意思？

這裏的臨時變量要區別於前面的局部變量。

局部變量是在程序中明顯給出的變量，其作用域只到當前 Scope 退出為止。

臨時變量則不是在程序中出現的、有名字的變量，而是「虛擬寄存器」的代稱。

> 一定是腦子進水了才會起這麼一個名字。

因此在程序代碼裡，`Temp_temp` 指的就是它了。

在實際指明用哪個寄存器之前，我們可以先用一個抽象層「虛擬寄存器」來確定我們的非逃逸寄存器放哪裡。但虛擬寄存器有個嚴重的區別於寄存器：他不限制個數。有無限個虛擬寄存器可以使用。

然而有另一個東西稱為 Label。label 指的是準確地址還需要確定，但是其值在編譯後運行前就可以確定下來的。比如，某個函數的入口地址；或者之類的東西。

只要使用 `Temp_newtemp()` 函數，就能在無限的虛擬寄存器集合中拿出一個新的給我用。

`Temp_newlabel()` 從標號的無窮集合中拿出一個新的標號給我用。

`Temp_namedlabel(string)` 會返回一個匯編中標號為 string 的標號給我用。

留意到，不同作用域中的名字可能會重複。

兩層抽象

我們的 Tiger 編譯器實現過程中有兩層抽象。

Semant -> Translate -> Frame & Temp

Semant 是 Lab 4 裏寫的語義分析。

Translate 是這個 Lab 裏需要實現的指令翻譯。他做的更多的一件事就是處理嵌套作用域的表示。

Frame & Temp 是和實現機器無關的抽象。其實現跟不同的機器的棧幀和寄存器條件有關。

Translate

Translate 做什麼？他是基於 Semant 的，結構也非常相似。

不同之處在於，Semant 做類型檢查，Translate 做語意轉換。把他們混在一起也不是不可以，只是有點膨脹。

Tr 開頭的 API

```
typedef struct Tr_access_ *Tr_access;
/* ... 其他 typedef */

Tr_accessList Tr_AccessList(Tr_access head, Tr_accessList tail);
/* 深惡痛絕的鏈表結構 */

Tr_level Tr_outermost(void);
Tr_level Tr_newLevel(Tr_level parent, Temp_label name, U_boolList
formals);
Tr_accessList Tr_formals(Tr_level level);
Tr_accessList Tr_allocLocal(Tr_level level, bool escape);
```

現在，在 Tiger 的語意分析階段，我們不能再僅僅提供一個報錯了事，而是要處理好棧幀、寄存器分配之類的事情。

他可以使用的 API 已經很很顯然了：

`Tr_newLevel` 是創建棧幀的函數（調用的當然是 `F_newFrame`）。

嵌套層保存在哪裡呢？一個 FunEntry 結構中。這樣，遇到了 Function Call 的時候就可以將這個被調用者的嵌套層交給 Translate。另外，FunEntry 會請求得到函數機器碼入口位置的標號。

```
struct E_enventry {
    enum {
        E_varEntry,
        E_funEntry
    } kind;
    union {
        struct {
            Tr_access access;
            Ty_ty ty;
        } var;
        struct {
            Tr_level level;
            Temp_label label;
            Ty_tyList formals;
            Ty_ty result;
        } fun;
    } u;
};

E_enventry E_VarEntry(Tr_access access, Ty_ty ty);
E_enventry E_FunEntry(Tr_level level, Temp_label label,
                      Ty_tyList formals, Ty_ty result);
```

Semant 遇到一個 lev 層的局部變量聲明的時候，他就調用 `Tr_allocLocal(lev, esc)` 函數在 lev 對應的這一層創建變量。esc 參數指定其是否逃逸。

返回值是 `Tr_access`。`Tr_access` 比 `F_access` 要多出了靜態鏈相關的信息。

隨後，如果有一個該層或者是更深層的表達式引用了這個變量，那麼 Semant 就可以請教 Translate 來聲稱訪問到這個變量的機器代碼。同時，Semant 也在每個 VarEntry 裏記錄下這個訪問。

```
// inside translate.c
struct Tr_access_ {
    Tr_level level;
    F_access access;
}
```

抽象數據類型 `Tr_access` 記錄每個變量的層級 level 以及他的 `F_access` 組成的對。

`Tr_allocLocal` 要調用 `F_allocLocal` 來實現功能，與此同時，還需要記住這個變量生存在那個層次（離開這個層次，該變量就死亡）。最後，如果需要從不同層級訪問這個變量，就需要用到這個 Level 信息來計算靜態鏈。

靜態鏈管理

靜態鏈由誰負責？顯然不是 Frame。Frame 應該知道的只是跟語言無關的事情。反映到實際的代碼中，靜態鏈本身也只是棧上的一個數據。這就是 Frame 了解的信息。這件事情應該由 Translate 負責維護。

Translate 才知道關於靜態鏈的信息，每個棧幀有且僅有一個靜態鏈位，並且使用它來完成變量尋址之類的事宜。

考慮到靜態鏈跟變量傳入的參數非常類似，位置也非常類似（挨著的）。所以，我們將其作為形式參數對待為好。

我們直接用 `new_l = U_BoolList(TRUE, l)` 在原來的參數列表外再「套」一個逃逸的「參數」。

> 當然是逃逸的…否則靜態鏈放寄存器嗎？

然後，我們採用 `newFrame(label, new_L)` 開一個包含「偽參數」的新棧幀。

如果 f(x, y) 函數被套用在 g 函數之內（g 的層級為 `g_level`），那麼 `transDec` 就可以這麼調用：

```
1 Tr_newLevel(g_level, f,
2             U_BoolList(FALSE, U_BoolList(FALSE, NULL)));
3
4 // 假定了 x 跟 y 都是非逃逸的
```

來新開一個棧幀。

`Tr_newLevel` 則會自動地給形參表多加一個關於靜態鏈的參數，也就是調用 `F_newFrame(label, U_BoolList(TRUE, fmls))` 來創建新的（加入了靜態鏈的）棧幀。

返回值是一個棧幀 `F_Frame`；這個棧幀裏現在就已經有了三個 formals 了；

可以通過 `F_formals(frame)` 來獲取的位移 offset 值；第一個就是靜態鏈的 offset 位移；第二個就是參數 x、y 的 offset 位移。（此處的位移是在棧幀中，相對父親函數的棧指針（自己的幀指針）的偏移量，也就是在實參列表中的偏移量。）

然而在 Semant 調用 `Tr_formals(level)` 的時候，就已經只有兩個位移值了（因為 Translation 主管關於靜態鏈的東西，而且將其嚴格保留在自己以內，不將其擴散。）

追蹤層次信息

每次我們調用 `Tr_newLevel` 的時候，Semant 都會提交包圍層的 level（層級）。

但是有沒有想過，主函數（不包含在任何一個函數內的函數）的層次是如何創建的？

要創建這個主函數的層次，就需要獲取他的外層的層次。他的外層還有層次嗎？

> 沒有。

所以我們用 `Tr_outermost()` 函數來指代這個「main 函數的外層」的特殊層次。

> 這個層次裏還有一些實用的庫函數。不是我們這裡的討論範疇。

函數 `transDec` 對每個函數聲明都創建一個新層次。（一層一層的深入。）

`transExp`、`transVar` 也是這樣。而且，他們彼此裏還可能會包含彼此，甚至是自己。

因此，他們都需要一個 `level` 參數。

> End Of Chapter 6.

Chapter 7: Translation

翻譯成中間代碼

> 總覺得這書作者有點錯亂…每次做事情都只做一點，為這一點構造一層抽象，費很大力氣，然後把他丟到一邊。

中間表示呢是一種抽象的機器語言。這種語言是前端和後端的間隔；前端翻譯成中間語言（IR）；然後後端利用 IR 產生機器語言。

這樣，不同的前端語言和不同的機器 Architecture 就可以用一個共享的中介；這一種 IR 是抽離出所有的高級語言的最低共性，以及所有的機器語言的最高共同抽象。

就像 Figure 7-1 一樣的表示，這會減少編譯器的複雜性，使得編譯器可以變成可組合組件集合。

美好的願望！

（現實根本不可能這麼簡潔明瞭。）

中間表示樹

> 一棵樹？

為了表示這個，我們可是下了苦心。

通用的中間表示至少要有下面的特點：

- 方便 Semantor 生成
- 方便轉換為機器語言
- 結構需要足夠普適且簡單

> 天底下哪有這樣的好事？

於是作者就提供了 Figure 7-2 的樹中間表示的結構：（從略）

> 對於 Compiler 的 Labs 有了一種奇怪的感覺：所有的東西類型都已經被釘死了；一層層的釘死了。每一層結構該做什麼、有什麼標準都不可更改自定義。要做的事情只是在每一層裏做一些微小的改進。
>
> 有完美且唯一的一個路牌。沒有別的路被允許，就算最後的結果正確也不可以。
>
> 必須、一步、一步走，按照要求走。
>
> 就是在生活中總會體驗到的感覺。
>
> 雖然看起來這個體系（眼下）並不壞。雖然跟著走還能拿到分數。
>
> 不用思考。

树语言的基础表达式

有多么基础呢？我们一一来看。

T_exp

T_exp 是有返回值的表达式。

- `T_Const(i)` 代表值为 i 的整型常数。
- `T_Name(n)` 代表名为 n 的符号常数（也就是汇编里的标号）。
- `TEMP(t)` 代表虚拟寄存器 t。
- `BINOP(o, e1, e2)` 代表对 e1、e2 两个操作数施加 o 表示的操作。操作 o 可以是 PLUS、MINUS、MUL、DIV（加减乘除）、AND、OR、XOR（与、或、异或）、LSHIFT、RSHIFT（逻辑左右移位）、ARSHIFT（算数右移）。逻辑操作符本身在 Tiger 里是不存在的。这里提出来主要是在底层实现中可能会用到。
- `MEM(e)` 代表自内存地址 e 处开始的 wordSize 个字节的内容。

  > wordSize 是被定义在 Frame 模块中的一个常数。
  >
  > MEM 被放在等号左边时，代表对该地址的内容进行修改；在其他位置都代表读取该地址的值。

- `CALL(f, l)` 代表以参数列表 l 来调用函数 f。考虑到 f 也可能是一个表达式的情况下，优先计算 f 得到一个函数，随后从左到右地计算参数列表 f。
- `ESEQ(s, e)` 代表先计算语句 s 让其产生作用；然后计算 e 将其作为整个表达式的结果返回。

T_stm

T_stm 是没有返回值的语句。

> 留意：这门树语言的 MOVE 语句都是 `dst, src` 的形式。

- `MOVE(TEMP t, e)` 代表对表达式 e 进行计算，并将其放入虚拟寄存器 t 中。
- `MOVE(MEM(e1), e2)` 先计算 e1 表达式的结果，将其作为内存地址解读为地址 M；然后再计算表达式 e2，将其值放置在内存地址 M 中。
- `EXP(e)` 代表计算表表达式 e，但是忽略其结果。
- `JUMP(e, labs)` 代表将控制权转移给 e 处的代码。

    - e 可以是一个文字标号，labs 则是每个表号所对应的地址；
    - e 也可以是一个实际的地址；这样 labs 就不需要了。
- `CJUMP(o, e1, e2, t, f)` 代表首先依次计算 e1、e2，并使用操作 o 对其进行比较；如果结果为真，则跳到 t 所对应的位置；反之则跳到 f。

    > o 可以取到的值包括：LT、GT、LE、GE（以及所有前面带 U 前缀的无符号版本。）

- `SEQ(s1, s2)`：代表计算语句 s1 之后紧接着算 s2。
- `LABEL(n)` 把当前机器代码的地址（Program Counter）放入 n 中。

上面这些统统都是树语言的本身定义。看起来还蛮完备的。

那么我们该如何将我们之前的到的语法树给翻译（Translate）成这种 TreeIRLang 呢？

表达式的种类

之前我们的 AST 中的 `A_exp` 该用什么来表示？统统用 `T_exp` 吗？

Well，看起来应该这样没错，而且大部分其实都可以用 `T_exp` 来表示…除了其中一些特别的 `A_exp`。

例如，WhileExp 就很难直接用一个 `T_exp` 表示，考虑到他肯定会需要用到 JUMP / CJUMP。

因此，我们在 Translate 模块中给出三种表达式类型，来保证其尽可能简单的同时来覆盖 AST 中的所有组件。

- Tr_Ex
- Tr_Nx
- Tr_Cx

Tr_Ex 就是普通的、会产生简单返回值的 Expression；而 Tr_Nx 就是不产生结果的语句，即 Statement；而Tr_Cx 则比较复杂一点：它指的是可能转移控制的语句。只做一件事情：计算一个条件，并且从它所包含的 True / False 对应的两个位置中选择一个进行转移。不做别的事情了。

Example: `a > b | c < d` 可以被转化为这样的条件语句：

```
1  Temp_label z = Temp_newLabel();
2
3  T_stm s1 = T_Seq(T_Cjump(T_gt, a, b, NULL_t, z)), T_Seq(T_Label(z),
   T_Cjump(T_lt, c, d, NULL_t, NULL_f)));
```

z 是一个新的虚拟寄存器，我们用它来放置下一个判断所对应的内存地址。在这里，也就是原条件为 False 的时候，我们要去 OR 的右端继续进行二次判断（不可短路）的内存地址所在的虚拟寄存器名字。

我们在这里已经向其中写入了那个（右半段判断）内存地址，通过调用 `T_Label(z)` 来实现的。然而这里存在问题：最终结果不管为 TRUE 还是 FALSE，在目前我们都没办法了解到该到哪里去。

因此我们有两张有顺序的表，记录下了我们「仍然空着」的位置；这表的类型是 `patchList`。

用法：直接用 `*s1->u.SEQ.left->u.CJUMP.true`

来代表 Statement s1 的 SEQ 语句的左侧的 CJUMP 语句的 TRUE 分支仍然空着，在我们完成之后需要回填。

> 另外，`patchList` 的格式也是肮脏的 C 形式链表。

问题来了。`a > b | c < d` 在 IF 里会被解读成 Cx，然而在 `flag := a > b | c < d` 里就应该被解读成一个单纯的 Ex，返回一个 1 或 0 值。

那我们该怎么处理呢？

```
static T_exp unEx(Tr_exp e);
static T_stm unNx(Tr_exp e);
static struct Cx unCx(Tr_exp e);
```

可以将任何类型的 `Tr_exp` 转换成三类中的任何一类。

这样，赋值语句就可以直接写成 `MOVE(TEMP_flag, unEx(e))` 了。

虽然到头来我们构造的还是一个 Cx，但最终还是可以当作一个 Ex 处理。这就没问题了。

> 特别情况：拒绝将 Tr_nx 给 unwrap 成 Cx，这绝对不可能发生，也没有用。

三个 Unwrapper 的实现？看 Program 7-1，提供了一个 unEx。

现在我们要做的，就是把 AST 中所有的块给翻译成 IR Tree 表示。

翻译为树中间语言

简单变量

当我们在 AST 中提到一个简单变量的时候，我们到底要做什么？

这个 `transVar` 到底在哪里？

- 栈上

`MEM(BINOP(PLUS, TEMP fp, CONST k))`，

直接使用栈指针偏移 k 和 fp（Frame Pointer）算出内存位址，然后 MEM 拿下。

在以后的章节里，上面这句话简化为 `MEM(+(TEMP fp, CONST k))`。表达的意思一样。

- 外面

问题大了。这个变量开始不简单了——它放在了调用祖先函数的栈上。

那么我们要做的事情就变成了：追随静态链，不断地寻找上层调用函数的帧指针位置，并借此再次寻值。

一般形式：

```
MEM(+(CONST kn, MEM(+(CONST kn-1, ... MEM(+(CONST k1, TEMP FP)) ... ))))
```

k1 ~ kn 是各层嵌套函数的静态链相对于帧指针的 Offset 偏移量。

因此从里到外看，我们从最内层函数 f1 向外找，利用自己的 Frame Pointer 和 f1 内 FP 到静态链的偏移量寻值得到外层函数 f2 的 Frame Pointer 地址；然后再利用 f2 的 FP 以及 f2 内 FP 到静态链的偏移量找出 f3 的 Frame Pointer... 依次往後，直到得到了目标函数（存放着我们需要的变量的函数 Scope）的 FP 之后，不再找她的静态链，而是直接通过 FP 到目标变量的 Offset 找出这个值。Well Done。

因此为了读取使用层（simpleVar 的 Level 层级）以及定义层（位于变量的 Access，也就是访问记录）内的 Level 层之间 所有栈帧的静态链，simpleVar 需要在这两个层级之间生成一条由 MEM 和 + 组成的链。

数组

根据上面的内容，我们应该可以比较简单地学习到，分析数组主要由三部分组成：

- 处理数组的下标；
- 处理记录域；
- 处理各类型的表达式。

在 Tiger 语言里，数组变量的行为和指针类似。不仅如此，Record 类型也和指针行为类似。进行赋值只会改变内存指向而已。

左值

实际上这是一个隐含二义性的问题…但他实在是太自然，以至于大家都爱这么用。

左值是一个可以出现在赋值号左侧的值；

例如，普通变量 `x`、`p`、`y`；

或者是数组的一个位置，如 `a[i + 2]`（[] 内不要求是个左值）；

或者是 Record 的一个 Field，如 `rec.f`（. 后要求是 Record 的一个 Field）。

但类似于 `c + 1`、`f(x)` 等就无法对其进行赋值，所以我们将其称为右值。

左值可以取地址，而右值则不行。

但同时……痛苦的是左值也可以不出现在赋值号左边，这时他们隐含地表示取出这个地址的内容，也就是「值」。

下标和域选择

> 由于书翻译质量堪忧，Tiger 语言里的 Record 和 Field 翻译成「记录」和「域」很容易和其他的同义词产生混淆…因此在表示这两个意义的时候，直接用英文。

当我们要访问 `a[i]` 的时候，我们要做什么？

需要计算 a 的第 i 个元素的地址。

$(i-l)\times s+a$。

其中，l 是索引范围的下界（lbound），s 是数组每个元素的大小（单位：byte）。a 则是原数组的基地址。

域选择则更简单：对于 Record 中的每个 Field，我们都知道它在 Record 内存布局中的偏移量。如此，访问 `a.f` 就可以直接变成

`a_base + offset` 的计算。

考虑到我们之前的「左值」说辞，我们不可以在下表选择和域选择的阶段直接将其 MEM（取内存内容），否则它就失去了「左值」的精髓——可取地址性。

因此，我们会将其保留为一个内存字。

这个左值可能产生一个新左值：比如一个从 Array 中取出的 Record 是一个左值；她的每个 Field 也都是小左值。

这个左值也可以被隐含地当作右值用；在需要的上下文中，直接将其 MEM() 操作一下就可以了。

MEM() 之后就无法再变回原来哪个左值了。

对一个左值进行算术操作（就算是 + 0）也是不可逆的了。

Do it carefully.

算术操作

这并不难做。一个 Absyn 操作符号就是一个 Tree 操作符号。

形如 `- i` 这样的一元操作符是不被 Tiger 支持的。它将被转化为 `0 - i`，implicitly。

这么做是正确的，因为 Tiger 根本不支持浮点数。

条件表达式

大问题…

- 比较语句

比较操作符号的结果自然是一个 Cx，在直接用于赋值的情况下也可以被 Force-unwrapped 成 Ex。

基本结构如下：

trues = {t}

falses = {f}

stm = CJUMP(LT, x, CONST(5), ?t, ?f)

留意到 ? 对应的部分是等待被回填的。

- IF-THEN-ELSE 一般语句

if e1 then e2 else e3

我们一般将 e1 视为 Cx，e2 和 e3 视为 Ex。这样，结果总是可以解决的。

- 无返回值的 IF-THEN-ELSE 语句

这种时候，虽然像一般情况下操作也是可以做的，只是需要将返回的 Nx 给强制 unEx。这不太好，但勉强还能用。

- 返回值是 Cx 的 IF-THEN-ELSE 语句

对 Cx 做 unEx 会产生比较大的问题，因此有必要对这种情况分别处理。

e.g. `if x < 5 then a > b else 0`

就是一个这样的例子。

将其翻译为

`x < 5` => `Cx(s1)`、`a > b` => `Cx(s2)`、

`SEQ(s1(z, f), SEQ(LABEL z, s2(t, f)))`

会更好。

字符串比较

我们不需要自己实现。很幸运的，Tiger Standard Library^{TM} 帮我们实现了这一点。调用 `stringEqual` 即可。我们不必要额外写代码了。

字符串

Tiger 的字符串本质上是一个指针；这个指针指向了一个字，这个字将被解读成一个整数 K；K 就是字符串的长度。紧接着後面的 K 个 8 位字就是字符串本体了。因为不像 C 使用 `'\0'` 作为结束标识，K 里可以包含任何内容。

创建 Record 和 Array

Tiger 语言中的 Record 和 Array 都必须分配在 Heap 里（原因显而易见。）而我们的 Tiger 语言并没有提供任何垃圾回收的机制（原因也是显然。），因此所有的 Record 和 Array 都永远不会被回收。

创建一个记录的方法：调用一个外部存储分配函数（Allocator），将得到的内存地址保存在虚拟寄存器里；然后，随后的每个 Record Field Access 都可以变成这个 Address + Offset 的形式了。

参见 Figure 7-3。

> 谁给你的勇气把 Record Allocation 翻译成记录分配的…你自己能看懂吗…

Array 更简单一些；有一个系统函数 `initArray` 可以帮我们处理好「用相同的初始值来初始化一个 Array」这件事。

调用 System Runtime API

上面我们只是说「调用 `initArray`」，但是如何发起这个调用？

这个小程序不可能用 Tiger 来写，因为 Tiger 根本没有开辟内存的方法。

所以我们要进行的是「External Call」。也就是，调用非 Tiger 语言生成的方法，并在最後对其进行链接。

```
CALL(NAME(Temp_namedlabel("initArray")), T_ExpList(a, T_ExpList(b, NULL)))
```

就相当于调用了 `initArray(a, b)`。

跟系统程序的交互由 `T_externalCall` 来打点。她需要处理 Label 标号的标准化、静态链的存废，等等。

While 循环

While 的一般形式是：

```
test:
    if not(condition) goto done
    body
    goto test
done:
```

看起来很美好；然而别忘了有一个叫做「BREAK」的东西。BREAK 会强制从最靠近当前 Scope 的循环里跳出，无论是 While 循环还是 For 循环。

为了让 `transExp` 能正确转译 Break 语句，我们需要添加一个新的形式参数 Break，这里放的是直接包含这条 BREAK 语句的循环（While 或 For）所对应的 done 位置的标号。

`transExp` 被递归调用时，假如那层包含的东西不是循环，那么大可以直接将自己上层的 break 发给孩子；如果当前层就是 For 或者 While，那么自己就是最内层直接包含孩子的循环体了；应该更新 break 参数再传下去。就是这样。

For 循环

For 循环的一般形式为：

```
for i := lo to hi
    do body
```

直接将其展开为

```
let var i := lo
    var limit := hi
in while i <= limit
    do (body; i := i + 1)
end
```

但是要留意 limit = maxint 的情形。这种情况下这样展开会死循环的。

> （但是我们的测试用例里似乎没有这个情形？）

函数调用

相当简单；因为 IRlang 里面也有 CALL 函数。

不过留意：Translator 肩负着保护静态链的任务；我们需要将当前层的 Frame Pointer 作为第一个隐含参数传递给孩子。

`CALL(NAME l_f, [sl, e1, e2, ..., en])`

翻译声明

翻译声明比较玄妙：声明本身不会出现在 IRlang 中；但是，会对生成的 IRlang 产生多方面的影响。

一点一点说。

变量定义

调用 `transDec` 会对 Frame 产生副作用：会迫使 Frame 在栈帧中预留更多的空间给变量。

但是，变量值的初始化需要被转换成一个表达式。

let 表达式的「变量声明」部分是需要对值给出初始值的。因此，我们在 let 的函数体之前放置初始化所需要的 IRlang。

因此 `transDec` 应该返回一个 `Tr_Exp`，这个 Exp 应该完成给值的所有操作。这样我们就可以把这个 Expression 放在 let 的函数体之前。

如果我们对函数和类型声明调用 `transDec`，应该返回 `Ex(CONST(0))` 之类的无意义值。

函数定义

每一个 Tiger 函数都会被翻译成三部分：

- 入口处理代码（Prologue）
- 函数体（Body）
- 出口处理代码（Epilogue）

Body 就是一个大表达式；函数体的翻译也就是翻译这个表达式。

入口处理代码需要做这些事情：

1. 部分汇编语言需要一条特殊的指令来标记一个函数的开始；
2. 函数名字的标号定义；
3. 调整栈指针，为局部变量开栈内存；
4. 将静态链和逃逸的参数保存到栈帧，将非逃逸的参数保存到临时寄存器；
5. 保存 Callee-Saved 寄存器。（当然还是上栈。）

在此之后就是函数体（Body）：

6. 该函数的函数体。一个大 Exp。

出口处理代码需要做这些事情：

7. 将返回值（函数的结果）传送到用于返回结果的寄存器（%rax）。
8. 恢复 Callee-Saved 寄存器的值。（当然还是从栈里拿。）
9. 调整栈指针，回收局部变量占用的栈内存。
10. 一条 ret 指令。（从栈上弹出 Return Address。）
11. 部分汇编语言需要一条特殊的指令来标记一个函数的结束。

3、9 步是需要在了解到栈帧的大小之后才能确定的。而了解栈帧的大小又不得不在所有局部变量都分析完之后才能确定。

这些指令会在非常晚的时候才会生成；参见 FRAME 模块的 `procEntryExit3` 函数。

第 7 条很容易，直接生成一条 `MOV(RV, body_exp)` 就好了。

第 4、5、8 条跟视角有关，是 6.2.1 节中的视角移位的一部分。参见 FRAME 模块中的 `procEntryExit1` 函数实现。

第 12 章会解读具体的实现。

片段

片段？什么片段？

给出了一个嵌套层级 Level 和一个翻译好了的函数体表达式组成的 Tiger 函数的定义，Translator 应该可以生成一个关于这个函数的描述符。包含下面一些必要信息：

- 栈帧

  > 包含了有关局部变量和参数的信息。

- 函数体

  > 也就是 `procEntryExit1` 函数的返回值。

这两样信息并称为「片段」（Fragment）。

片段仍需要被翻译为汇编语言。

frag 数据类型专门用来描述片段。

片段分两类：一类是 String Fragment（字符串字面值片段）；另一类是 Function Fragment（函数体片段）。

本质上，他们有一定的共性：他们都是游离于主程序之外的片段；都需要通过一个 Label 才能访问它们。

都只是当作一个工具使用；访问一个字符串，或者是调用一个工具函数。用完就会回去。

不同之处？StringFrag 的关键信息是 label 和 string（标签和字符；而 FuncFrag 的关键信息是 T_stm body 和 F_frame frame（函数体和栈帧）。

> 啥？要问函数名字在哪里？
>
> 在 `F_procEntryExit1` 的返回值中，第二条就是函数名的标号定义。

最後，调用 `Tr_getResult` 就可以得到全部的 Fragments 了。在生成代码的时候，把它们放在边缘就好了。

Chapter 8: Basic Blocks & Trace

基本塊和軌跡……那是什麼意思？

本章的關鍵字是 Ca-non-i-cal。正規的，被儘可能簡化到最簡單或最清楚的。

好好回憶一下上一章（Translation）幹了什麼。我們把所有語言翻譯成了 Tree 語言：那一堆基本 Ex（Cx、Ex、Nx）的組合。然而，這種 Tree 語言本身並非和機器語言一一對應的。

主要問題是：一個表達式的子表達式計算順序（有時候）是敏感的。也就是說，Tree 語言的部分 Exp 子句含有副作用。例如，ESEQ 和 CALL 兩個字句可能含有賦值且執行輸入輸出。這就是說，按照不同的順序計算他們有可能會產生不同的結果。

反過來說，假如一個節點所有的子樹都不包含 ESEQ 和 CALL 節點，那麼我們就可以按照任何順序對他們進行計算。

> 簡而言之，就是在一個節點的子樹中沒有 ESEQ（他強行要求了賦值和返回值的順序）和 CALL（這就是直接開了一個新棧幀了）的話，我們就可以任意交換子樹求值的順序。

另外，TreeIR 和機器語言還存在這麼一些區別。

- CJUMP 的跳轉位置為兩個（一真一假）；然而很遺憾實際機器語言在條件為假的時候默認跳轉到下一條指令。

  > 其實也還是能理解的…畢竟 Treelang 這個等級還沒有實際出現「上一條下一條」逐條指令的概念；結果只能是用樹來實現。因此同時指定 t 和 f 是必須的。並非故意如此增大難度。

- CALL 函數的參數列表中存在 CALL 的時候，會產生寄存器混淆的問題。
- ESEQ 節點出現在表達式中不太好；他會引起不同計算順序帶來不同結果。
- CALL 節點也是這樣的；計算順序相關。

為了處理 ESEQ 和 CALL 的問題，我們決定把 ESEQ 去除。

換一句話說，我們就是要把原树改写成这样：

1. SEQ 的父亲只可能是 SEQ。
2. 根据 1.，要么这棵树没有 SEQ（which is very unlikely），要么树的根一定是 SEQ。
3. 根据 1.、2.，所有的 SEQ 都会聚集在树的根部。

因此，前面那一串 SEQ 实际上并没有什么意义；他们只是一串要求顺序的 Statement 而已。

所以我们何不干脆直接把所有的 SEQ 拿走，换成一串 Statement？

这就是我们要做的：在所有的 SEQ 都被提前之后，清除所有无用的 SEQ 并将其使用 C::StmList 代替。

大体步骤

正规化步骤分为以下步骤：

第一步，将树的 ESEQ 都转化成聚集在树根部的 SEQ。

第二步，清除根部的 SEQ，用 StmList 代替它们。

第三步，将这个数分拆成不包含转移和标号的基本块集合。

第四步，对基本块进行排序形成一组轨迹，而且保证每个 CJUMP 後面紧跟着的都是她的 False 标号。

> 总的来说，每一步说的都不清不楚。
>
> 就当作者和译者不会说人话好了。心平气和。

书也好心提供了 C 的 API。很可惜，天才助教们把 Lab 改写成了 C++，因此只能连蒙带猜了。

```
typedef struct C_stmListList_ *C_stmListList;
struct C_block {
    C_stmListList stmLists;
    Temp_label label;
```

```
};

struct C_stmListList_ {
    T_stmList head;
    C_stmListList tail;
}

T_stmList C_linearize(T_stm stm);
struct C_block C_basicBlocks(T_stmList stmList);
T_stmList C_traceSchedule(struct C_block b);
```

留意 stmList 是一般的 Statement List；而为了保存每个块里的（每棵树开头的那一串 stmList），我们得用 stmListList 来存储这些 stmList……

> 更加想死了。

Linearize 函数做的事情是对每一棵树，删除其中的 ESEQ 并且将 CALL 移动到顶层。

BasicBlocks 把语句分成一组组的直线型代码序列。

traceSchedule 对基本块进行排序形成一组轨迹，而且保证 False 标号紧贴。

规范树

上面都是领导安排工作。下面就是苦力进行实现。

领导这一章的目标是：

- 不包含 SEQ 或 ESEQ。
- 每一个 CALL 的父亲要么是 EXP（），要么是 MOVE（TEMP t，）。

> EXP 意为调用函数但是放弃其值。
>
> MOVE t, 意为将函数返回值放在临时寄存器里。

要做转化，我们只有下面一些等式可以用：

等式 1

ESEQ(s1, ESEQ(s2, e)) => ESEQ(SEQ(s1, s2), e)

留意 SEQ 的语义是：翻译 s1，然后翻译 s2。

等式 2

BINOP(op, ESEQ(s, e1), e2) => ESEQ(s, BINOP(op, e1, e2))

副作用 s 在 e1 之前产生作用就可以了。所以把他从 BINOP 里提升出来当然没问题。

MEM(ESEQ(s, e)) => ESEQ(s, MEM(e))

JUMP(ESEQ(s, e)) => ESEQ(s, JUMP(e))

CJUMP(op, ESEQ(s, e1), e2, l1, l2) => SEQ(s, CJUMP(op, e1, e2, l1, l2))

也都是当然。

等式 3

针对一个指令有两个 Expression 的情况下，副作用在第二求值位的时候：

BINOP(op, e1, ESEQ(s, e2)) 比较复杂一点：因为 s 副作用应该在 e1 之后、e2 之前产生。

所以我们必须先行计算 e1，然后执行副作用，最后抽离出 ESEQ。

BINOP(op, e1, ESEQ(s, e2)) => ESEQ(MOVE(TEMP t, e1), ESEQ(s, BINOP(op, TEMP t, e2)))

同理，

CJUMP(op, e1, ESEQ(s, e2), l1, l2) => SEQ(MOVE(TEMP t, e1), SEQ(s, CJUMP(op, TEMP t, e2, l1, l2))

本质上就是先计算无副作用的一号位表达式，然后把副作用表达式从基础语句中抽离。

消耗一个新的临时变量。而已。

等式 4（特别情况）

是等式 3 的特殊情况。在 s 不对 e1 的计算造成影响的情况下，我们可以直接抽离副作用表达式，而不用提前保存 e1。

也就是，

BINOP(op, e1, ESEQ(s, e2)) =?> ESEQ(s, BINOP(op, e1, e2))

（CJUMP 懒得写了）

但是这个等式一定要 s & e1 is commute 作为条件。怎么判断 s 不对 e1 产生副作用啊？这个判断一定要很保守，因为如果犯错，结果就是错误的。

> 我们说 Stm & Exp is commute，也就是 s 跟 e 可交换，也就是说他们在最终的汇编中的顺序可以交换出现。也就是 Statement 的副作用不会影响到 Expression。

因此我们的特殊情况是：if e1 == CONST，则直接将其抽离。否则，一律不抽离。

保守一点总归不容易犯错。

一般重写规则

上面这些东西给人来做或许能勉强改写完。那对于一台计算机，它该遵循什么规则来*利用*这些规则等式呢？

一般化的规则，还是基于要求的求值顺序和副作用所产生的时间点。

一般化地，在

Op[e1, e2, ESEQ(s, e3)] 之中，我们的目的是将其化为 ESEQ(s, Op(e1', e2', e3')) 的样子。

但是，留意到原来的执行顺序是 e1 -> e2 -> s -> e3。因此如果我们直接暴力将其抽出，求值顺序就变成了 s -> e1 -> e2 -> e3。因此，这件事情跟 e3 无关，我们要考虑的事情是 s 的提前求值会不会对 e1 和 e2 的求值产生影响（未预料的副作用）。

假如 e1 和 s 不可交换（也就是 s 的副作用会影响到 e1 的计算），那么就需要加一句

SEQ(MOVE(t1, e1), s);

然后用 t1 假借 e1（副作用产生前）来做最终计算。e2 同理。

因此整件事情都是可以规范化的。

> 根据我们之前保守的算法，只要 e 不是常数（Constant），我们就都认为它跟 s 是不可交换的。

Reorderer

Reorder Master 可以接受一个表达式表，并且返回（语句，表达式表）构成的偶对。

「语句」表示的就是在表达式表之前必须发生的事情，根据我们之前的讨论，那基本上就是表达式的求值并放到虚拟寄存器里。

> 注意：表达式的求值之间（例如上面的 e1 和 e2）也可能会产生副作用…所以我们仍然保留原有的顺序，并使用一个大 T_stm 来存储他们。
>
> 抽离出来这些所有的副作用後，事情就变得简单了；留下来的那个 Exp 就已经是求值顺序无关的了。

算法

Reorder 的算法是这样的：

```
typedef struct expRefList_ *expRefList;
struct expRefList_ {
    T_exp *head;
    expRefList tail;
};

struct stmExp {
    T_stm s;
    T_exp e;
}

static T_stm reorder(expRefList rlist);

static T_stm do_stm(T_stm stm);
static struct stmExp do_exp(T_exp exp);
```

我们要传递给 Reorderer 的是一个 expRefList，也就是一串 T_exp 而已。

这一串 T_exp 就是目标节点（需要净化的节点）所有子表达式的指针。

以 BINOP(CONST, MEM) 为例子，我们将其标记为

e2(e1, e3)，其中 e2 对应 BINOP；e1 对应 CONST，e3 对应 MEM。

那么传入 reorder 的链表实际上就是 (&e1, &e3)。

Reorder 会抽取其中所有的 ESEQ，并且把他们合并成一个 T_stm（顺序需要保留，当然）。

为了达到这个目的，我们对每个传入的子表达式（这里是 e1 和 e3）依次调用 do_exp 函数。这个函数干什么呢？观察我们可以得到，它的返回值是（T_stm, T_exp）偶对。

T_exp 会仅仅包含类似于 MEM(TEMP a) 这样的从寄存器 / 栈帧中取值的语句；而原来的 e1/e3 会出现在 T_stm 里；包括了将其放入内存的 Statement。

对于不包含 ESEQ 的语句，我们完全无需进行任何处理；只需要构造出它的子表达式链表，并且递归地调用 reorder。而在遇到 ESEQ 的时候，我们必须抽出其中全部的 ESEQ。

为了实现找出 ESEQ 的功能，我们又加入了一个 do_stm……

> 或许编译器本身就不是人可以写出来的代码。
>
> 要有一些不写代码的人来构造这个大而无当的架构，然后由写代码的人一点点实现它。
>
> 当然，这个 Tiger Compiler Lab 是垃圾到不能称之为编译器的了。

MOVE 左边的操作数，也就是 MOVE 的 Destination 我们需要特别说明：我们不把它看作一个子表达式，因为那是这个语句的目的操作数，而这个语句是不使用它的。

但是，假如左边的操作数是一个 Memory Location…

> 很难想象中文书把 Memory Location 翻译成了前不着村后不着店的「存储单元」……惊讶到我去翻了前言发现这章是赵克佳翻译的。这位互联网上找不到任何信息的人的作品，跟陈昊鹏有的一拼。

…也就是一个内存地址的话，我们就该把它当作一个「计算所得」了。

这很好理解：你不可能动态计算出一个寄存器编号来作为 MOVE 的 destination，但有可能动态计算出一个内存地址来作为 destination。事实上，这挺常见的。

因此，我们有必要对这两种情况分开处理。

……

书上给出了 reorder 的 C 实现。所以不出意外，我们要做的也就仅仅是 Translate 成 C++ 而已。

事实上，已经提供了实现好的 canon 模块。

有理由相信他们是不会出错的。假若真出错，那么可以认为是你 Translate 错了。

Chapter 9: Instruction Selection

在 Lab 5 之中，它对应的就是 `./codegen` 目录下的代码。

经过了我们的 Translation（章 7）和 Canonicalization（章 8），我们终于可以生成实际的代码了。

作者再次插入了一层抽象：Jouette Arch。

> 连指令集都要自己制造一套…

然而这个指令集太弱鸡了。甚至没法做到 JUMP。

我们可以用的 Jouette 指令集只有下面的一点点：

- ADD，$r_i <= r_j + r_k$
- MUL，$r_i <= r_j - r_k$
- SUB，$r_i <= r_j \times r_k$
- DIV，$r_i <= r_j \div r_k$
- ADDI，$r_i <= r_j + c$
- SUBI，$r_i <= r_j - r_k$
- LOAD，$r_i <= M[r_j + c]$
- STORE，$M[r_j + c] <= r_i$
- MOVEM，$M[r_j] <= M[r_i]$

这个指令集只是为了说明「瓦片覆盖」算法制造出来的东西。

我们实际上还是要在 CISC 下面进行 codegen。

CISC versus RISC

我们在这里针对的现代 CISC 体系计算机具有下面的特征：

- 为数不多的寄存器（16、8、或 6 个。）
- 寄存器分为不同的类型。有一些操作只能在某类特定寄存器上进行。
- 算数运算可以通过不同的「寻址模式」来访问寄存器或存储器。
- 指令一般只有两个参数，形如 `r1 <- r1 OP r2`。
- 有许多种不同的寻址模式。
- 指令是变长的、不等长的。
- 存在「副作用」指令，例如「自增」。

为了处理 CISC 体系结构下的代码生成问题，我们决定用下面「快刀斩乱麻」的方式来处理问题：

1. 寄存器少，我们就无限地生成 TEMP 节点，假定 Register Allocator 可以帮我们决定寄存器的使用。
2. 寄存器分类：这个问题在我们要生成的 AMD64 体系里已经不存在了。所以我们也不说啥了。
3. 两地址指令：在 RISC 里，我们可以实现 `t1 <- t2 OP t3`。但是在 CISC 里只能有两个地址。所以我们的实现方法是将 `t1 <- t2 OP t3` 实现为 `t1 <- t2`、以及 `t1 <- t1 OP t3`。多一条 MOV 指令而已。
4. 算数运算可以访问寄存器，也可以访问内存地址。不像 RISC，算数运算只能对寄存器作用。

换句话说，`MOV %eax, [%ebp - 8]; add %eax, %ecx; %move [%ebp - 8], %eax` 和 `add[%ebp - 8], %ecx` 具有相同的效果。

但是奇怪的是，执行它们所消耗的时间是完全一致的！这都多亏了高度流水线化的 CPU，加速了前面三条指令的执行速度。

5. 若干中寻址方式。不像 RISC，要访问内存地址，只能使用 M[reg+const] 形式。虽然理论上，不同的寻址方式并不会对最终执行结果产生很大的影响。
6. 变长指令。这和编译器本身没有关系，应该是 Assembler 要做的事情。
7. 副作用指令。有一条指令产生的作用是：`r2 <- M[r1]`，同时 `r1 <- r1 + 4`。

   对于这种指令的设计我一句话也不想说。这或许在某中程序上是一点小聪明，但最终绝对没什么好处。

   这种指令的设计思路跟 Wintel 联盟一脉相承。

指令选择

用 C/C++ 实现树的瓦片覆盖是很简单的，参见程序 Figure 9-1。然而这只是说明了对于某一段树应该使用哪一类「指令瓦片」对其进行覆盖。然而，该用哪个 / 哪些寄存器作为中介连接这些瓦片，我们仍然不能确定。

AWA 和 MG 又开始了……它们引入另一层结构抽象（建议把他们俩吸收入党），称之为 `As_instr`：这个数据类型表示「不指定寄存器的汇编语言指令」。

主要结构包括了：

- OPER：包含了汇编语言指令 `assem`、操作数寄存器表 `src` 和结果寄存器表 `dst`。同时，还包括了用于指定跳转分支的 `jump`。

  BTW，我们可以设定 `jump` 为 NULL，这样它总会下落到下一条指令。但如果不是这种情况（例如可能跳转到下一条、也可能跳转到其他位置），就必须显式地指定所有可能跳转目标的标号。它们存放在数据结构 `AS_targets` 里。
- LABEL：则是程序中可能会被 JUMP 到的位置。包含两个成员：`assem`、`label`。`assem` 指明了汇编语言中标号的形式，而后者则指明了表示标号的符号。
- MOVE 和 OPER 类似，但是是专门进行 mov 的指令。单独把它拿出来的目的主要是如果 mov 的 src 和 dst 被分配到同一个寄存器里，就可以直接删除这条指令（原地 tp 是没有意义的）。

`AS_print(f, i, m)` 函数用用于把一条汇编指令输出为字符串形式，并输出到 `FILE *f` 处。`m` 为临时变量映射，给出了每一个临时变量的寄存器分配策略。在这里，我们会将所有可能的变量都上栈，而不拘泥于具体的寄存器分配策略。那是 Lab 6 的事情。

`m` 的实现在 `temp.h` 里（这个模块并不在本次 Lab 的关注范围里。）。本质上只是一张保存着 `std::pair<TEMP::Temp, std::string>` 的链表。但留意，特别的，Entry 之间可以相互覆盖。越靠链表头的节点越优先。

为了方便调试，我们直接用虚拟寄存器的名字作为 `TEMP_name` 来加入映射表。

另外，信不信，这套东西居然还是「机器无关」的。

生成汇编

现在，我们就可以做汇编生成的事情了。我们依然还是采用自底向上的 MaxMunch 算法。存在两个函数 `munchStm` 和 `munchExp`。`munchStm` 无需任何返回值，直接写汇编就可以了；而 `munchExp` 则有必要返回一个 `TEMP::Temp`；因为 Exp 的返回值我们是在意的。

汇编的语句最终是通过调用 `emit` 函数来写出的。`emit` 函数会将指令登记在指令表中，最后再被写成 Assembly。

调用

我们怎么处理 Function Call 和 Procedure Call？

过程调用采用 `EXP(CALL(f, args))` 来表示；

而函数调用则是 `MOVE(TEMP t, CALL(f, args))`。

`munchArgs` 负责将所有参数传递到正确位置（寄存器或内存地址）。这个函数是递归的，通过递归调用自己来处理所有的参数。

`munchArgs` 的返回值是一张表，其中包含了所有要传递给 CALL 的所有虚拟寄存器。它们不会在实际的汇编中出现，然而我们有必要保存它们，以便进行活跃分析。

CALL 指令可能会造成某些 Caller-saved 寄存器和 %rax 寄存器的破坏。

因此我们将其列在表 `calldefs` 中，以防止它们被破坏。

去掉帧指针（%rbp）

帧指针多好啊…有了它 + offset，就可以非常轻易地寻找出栈帧中的值。然而 %rsp 跟 %rbp 之间的差值在同一个栈帧里是固定的——就是栈帧的大小。因此既然 %rsp 我们没法去掉（得用它 push pop 啊），我们干脆直接去掉 %rbp 了。

虚拟的帧指针总是等于栈指针加上栈帧的大小。但很可惜，在 Translate 阶段我们做不到确定栈帧大小，到了 Instruction Selection 我们还是确认不了。因为还没进行寄存器分配，不知道究竟有多少变量会被放置在栈帧里。

> 寄存器真是愚蠢。还不如给栈帧加几级快点的缓存来的实在。

因此我们只好先创建一个汇编常数 Lxx_framesize，用来指定 Lxx 这个 Scope 的栈帧大小。在函数的入口代码 F_procEntryExit3 里生成汇编语言常数 Lxx_framesize，这样才能访问到栈帧里的实参。

> 书里就说了：「有真实的帧指针的实现不需要对代码进行这种修改，并且可以忽略给 codegen 的 frame 参数。」
>
> 「但是，既然使用真实帧指针的实现浪费时间与空间，为什么还要用这种有帧指针的实现呢？」
>
> 前提不成立。显然访问真实 %rbp 比起间接计算耗时更短，因此不存在浪费时间；
>
> 另外，你新增了一段 Lxx_framesize，加了间接寻址的代码，你怎么敢说「浪费时间与空间」？

> 另外，这种做法不支持 C 的变长数组。差劲。

Semi-Fin

那么，可以说，Lab 5 所需要的东西到这里就结束了。

而三天之后、这周四的期末考试，需要再看 Chapter 10 和 Chapter 11 的内容。

这还远算不上 Fin，无论是复杂的 Compilers 还是坎坷的人生。

Chapter 10: Liveliness Analysis

活跃分析：分析什么？

在之前的章节（6～9 章）中，我们奢侈地使用着无限量的虚拟寄存器（临时变量）。但在真实的机器之中，我们在同一时刻可用的寄存器数量实际上很有限——就算往多了说也不会超过 32 个。

虽说如果寄存器不够用我们还可以放入栈帧，然而我们希望尽可能地用寄存器存取来代替内存存取，鉴于前者比后者快得多。

事实上，优化的关键在于并不是所有的虚拟寄存器都在同时活跃：在某一段代码中，被频繁利用的虚拟寄存器实际上不多：大部分情况下能够被放入物理寄存器里。

为了对当前活跃的虚拟寄存器进行分析，我们需要对程序的中间表示（IR）进行分析，以便确定哪些临时变量在同时被使用到。对某一个变量的出现来说，如果一个变量的值在此之后还会被使用，则把这个变量称为「活跃的」。因此这种分析被称为「活跃分析」。

Control Flow Graph

生成程序的控制流图：通常来说这对活跃分析很有帮助。

流图实际上很好画：按照生成的每条语句按顺序排列下来；连续的指令用箭头由前一条指向後一条。如果出现了 JUMP，则需要用箭头指出跳转终点；CJUMP，则需要指出跳转的条件以及两个可能的跳转方向。

Data Flow Graph

基于上面的控制流图，我们就每个变量画出其对应的「数据流图」。

给控制流图中的每一个「语句块」编号；用「连续的块之间的连线」作为一个数据流；我们要了解的是某一个变量的活跃范围。

```
1: a := 0       => 2
2: b := a + 1   => 3
3: c := c + b   => 4
4: a := b * 2   => 5
5: a < N        => 2, 6
6: return C
```

谨记「活跃」的定义：假如一个变量的当前值在将来还需要被用到，就说这个变量（在当前）是活跃的。

因此我们会采用回溯的方法来判断活跃性：即，假如在 N 语句处用到了 b 的值，那么往回找，直到那个最初的赋值语句这一串的 b 都是活跃的。

举个例子：变量 b 在语句 4 中被使用，往回找，3 中也被使用，2 中也被使用；但 2 语句是给 b 赋值的语句，因此其活跃范围是：{2 => 3, 3 => 4}。

活跃范围可能是不连续的。

以 a 为例子：他在 1=>2 活跃、在 4=>5=>2 之间再次活跃；但是这次的值是重新定义的。因此我们认为 2=>3=>4 之间是不活跃的。尽管在 3 处 a 是存在值的（继承自 2），但是在 4 中 a 被重新赋值前我们并不需要用到这个值。

至于变量 c，我们无法回溯到其定义的点；要么她是未定义的变量；要么她在进入这段程序之前就活跃。

所以总结一下：

在这段程序里，

a 的数据流为：{1->2, 4->5->2}。

b 的数据流为：{2->3, 3->4}。

c 的数据流为：{1->2, 2->3, 3-> 4, 4->5, 5-> 6, 5->2}（全程活跃）。

术语

In & Out

流图里面，每个结点都有一条引向后继节点的「出边」，以及一条从前驱节点流入的「入边」。

对结点 n 来说，其所有的前驱节点位于集合 pred[n] 中；所有的后继节点位于集合 succ[n] 中。

Define & Use

对于变量的赋值（变量出现在赋值号左边）称之为 Define（定值）；变量出现在赋值符号右边或其他表达式中的情况称之为 Use（使用）。

对于每个结点，use[n] 集合代表这个语句块中所 Use 的变量；def[n] 集合代表这个语句块中所 Define 的变量。

> Example：对于上面的例子来说，def[3] = {c}，use[3] = {b, c}。

Liveliness

我们说一个变量在一条边上「活跃」，到底是什么意思？

存在一条从这条边通向该变量的一个 use 的有向路径，且这条路径不经过该变量的任何 def。

Extremely Active

如果一个变量在某个结点的所有入边上全都活跃，那么就说**这个变量在这个结点**是入口活跃的。

出口活跃不用说了。

计算

怎么进行活跃分析？有下面的几条规则可以参考：

- 假如一个变量属于 use(n)，那么该变量在结点 n 是入口活跃的。
- 假如一个变量在结点 n 是入口活跃的，那么她在所有属于 pred[n] 的结点 m 中都是出口活跃的。

  > 人话：如果……，那么她的所有前驱节点都是出口活跃的。

- 假如一个变量在节点 n 是出口活跃的，而且不属于 def[n]（即不是当场定义当场出口活跃的），则该变量在节点 n 是入口活跃的。（它的活跃性一定是从上面继承下来的。）

就是这么简单的三条、环环相扣的规则。

可以简化为两个简单的方程：

> in[n] = use[n] ∪ (out[n] - def[n])
>
> out[n] = ∪(any s ∈ succ[n]) in[s]

算法也是很直接：对于所有的 n，依次更新 in[n] 和 out[n]，直到到达一个不动点为止。

优化

上面的算法总是按照流向的顺序进行更新；然而遗憾的是，我们的活跃分析所以来的回溯法顺序恰好相反！这就导致我们会需要很多无意义的运算。

如果我们根据「基本块」的思想，把那些只有一个前驱、只有一个后继、没有任何赋值的块合并在一起，就能大大简化计算。

又如果，我们每次只更新一个变量，利用深度优先搜索找到所有她被使用的块，并只更新跟他相关的那些结点，那么速度会快很多。

> 尽管直观上来看速度很慢，但是实际上每个变量的引用链都不会很长。反而是计算这么做会很快。

集合的表示

现在我们计算完成了；该怎么表示我们得到的数据流方程呢？

我们假设我们的程序片段里有 N 个变量、K 个块。

如果我们采用 Bit Array 的话，那么我们一共需要 N × K 位来保存所有的集合。

好处在于求集合的交、并集可以直接使用位操作来实现。缺点是可能会产生位浪费。

集合也可以用链表来表示；缺点是要求并集则必须用「合并链表」这样的操作，时间复杂度很大。优点是不会额外浪费空间。

Summary：假如集合很稀疏，那么用链表表示会比较快；假如集合很稠密，那么位数组表述会比较快。

运算时间复杂度

大小为 N 的程序中最多只有 N 个结点，也最多只有 N 个变量。出入口活跃集合也最多只有 N 个开关位。

对于每个结点的活跃集合数组，进行并运算的时间复杂度总是 O(N)。

> 无论用 Bit Array 还是链表都是这样。然而事实上链表比起 Bit Array 慢了不知道多少……

而在我们的每一次遍历之中，都需要对所有的结点进行并运算，因此每次 for 循环的时间复杂度是 $O(N^2)$。

然而一次 for 不会完；我们需要反复进行 for，直到我们的结果稳定（不变）。这可怎么分析好呢？

考虑到每次 repeat 迭代只可能往 in、out 里增加元素，而不可能将其变小。因此最差的情况下，for 会执行 $2\times N^2$ 次（in 的最大元素个数是 N^2；out 亦然。）；所以最终的最差时间复杂度是 $O(n^4)$。

> 实际上并不会这么差劲。经过 DFS 优化之后 repeat 的迭代次数一般在 2 到 3 次之间；实际的结果在 O(n) 到 $O(n^2)$ 之间。

最小不动点

在我们仔细分析活跃解之後，我们发现一件事情：活跃解并不是唯一的！往解中加入其它无用变量仍然可以满足原方程。为什么？

我们需要理解，数据流的方程都只是一个保守的近似解。假如一个变量会在一个结点后续被使用到，那么他一定会出现在该节点的出口活跃数组里。然而反过来并不是：你并不能保证该结点的所有出口活跃数组里的变量在后面都被用到。

究其根本，我们要做的事情是保守的。假如我们误将一个後面不会用到的变量加入了 out，无非只是浪费一点资源，生成一些无意义的代码，但对正确性不会有什么影响；而假如一个後面会用到的变量没能被加入 out，那问题就大了：很可能会导致执行程序出错。

因此，这也是一个 Conservative Problem。

动态活跃

上面我们提到的都是「静态活跃」；即将程序任何的执行流都纳入考虑，而不管他是否可达。

动态活跃则相对精明一些：他会考虑程序可能前往的执行流，并忽略那些不可达的执行流。

事实上这个优化略有些激进了，而且确实有可能犯错。我们不多谈。

冲突图

Interference Graph，是啥？

举例：我们有一大组临时变量 a、b、c、……。我们需要将它们分配到我们有限的寄存器 r_1、r_2、……、r_k 中去。但是寄存器总是不够用的；我们不得不将有一些临时变量放到同一个寄存器里。

并不可以随便混合放置！

- 如果在某一点处，a 和 b 同时活跃（活跃范围重叠），那么他们就不能进入同一个寄存器。
- 如果需要对临时变量 a 进行寻址，那么可以认为 a 和所有变量都冲突（也就是不让他进入寄存器）。

表示

矩阵可以用来表示：画一个 N × N 的矩阵，横、纵索引依次写上 a、b、c、……。对角线上留空（你总不会跟自己冲突？），假如 a 同 c 冲突，那么在 (a, c) 和 (c, a) 处都打上 ×。

> （所以你看到了，有一半的矩阵加上对角线都是无效值。）

也可以用图来表示：把那些冲突的变量用边连接。

特别处理

MOVE 指令需要特别处理。问题在于有一些冲突可能是人为造成的，实际上并不是必要的。

例如：

```
s <- t          (Define)
x <- ... s ...
y <- ... t ...
```

看起来在 Define 之后，s 跟 t 都是活跃的；那么我们是不是应该创建冲突边 (s, t)，说他们不能放在一个寄存器里呢？不一定！在这里的情况下，s 跟 t 仍然携带相同的值，我们并没有必要在这里就声明冲突。

因此我们采用一个更弱一点的条件来声明冲突：

- 对于任何对变量 a 定值的非传送指令（非 MOVE），以及在该指令出口处活跃的变量 b1、b2、……，分别添加冲突边 (a, b1)、(a, b2)、…。

  > 如果不是 MOVE，照常进行。

- 对于传送指令 a <= c，对每个在该指令出口处活跃的变量 b1、b2、……，对每个不同于 c 的 bi 添加冲突边。

这个，就是我们需要知道的，对于 MOVE 指令的特别处理了。

Chapter 11: Register Allocation

分配吧，寄存器。

现在问题已经很明了了：所有的虚拟寄存器（临时变量）组成了一个图；有些变量之间存在冲突，用无向边连接起来；然后，用 K 种颜色给这张图上色（K = 可用寄存器的个数），相连的边不准许上同一种颜色。

于是这就变成了一个着色问题。

> 留意到，有可能是无法着色的——寄存器是可能不够用的。这种情况下，部分的变量就需要进栈存储——这个学名唤做 Spill（溢出）。

着色

就是四步骤：构造、简化、溢出、选择。

构造

构造，就是构造冲突图。利用数据流分析方法，计算每个程序点处同时活跃的变量列表，并据此构造出冲突图。

为了方便我们得到的上色，我们采用无向图方式来表示冲突图。

简化

何谓简化？一个简单的启发式着色。技术在于：给出一个临结点度数小于 K 的结点，将其删除之后再分析剩余的图；假如剩余部分的图仍然可以 K 着色，那么原图一定也可以 K 着色。因为结点数量变小了，我们的分析也可以更简单。

同时，我们把简化掉的结点保存在一个简化栈里。

溢出

在简化过程中，某一个图 G 仅仅包含了高度数的结点（每一个结点的度都大于 K），那么就无法做简化了。这时我们就标记某个结点为「需要溢出的结点」，即在图中选择一个高度数结点，并认定他将被存储在栈帧中而非寄存器中。这样，我们就可以将此结点从图中删除，并对剩余图在进行着色。

选择

将 K 种颜色指派给图中的结点，使得所有存在边相连的结点都不同色。

但我们并不是采用「着色」的策略，而是选择从简化栈里弹出结点，通过一个个结点重建整张图，并在重建的过程中指派颜色。这样，我们总是在取出结点的过程中指定可用的颜色。但请留意，并不是总能完成选择；颜色并不一定总是够用。

重新开始

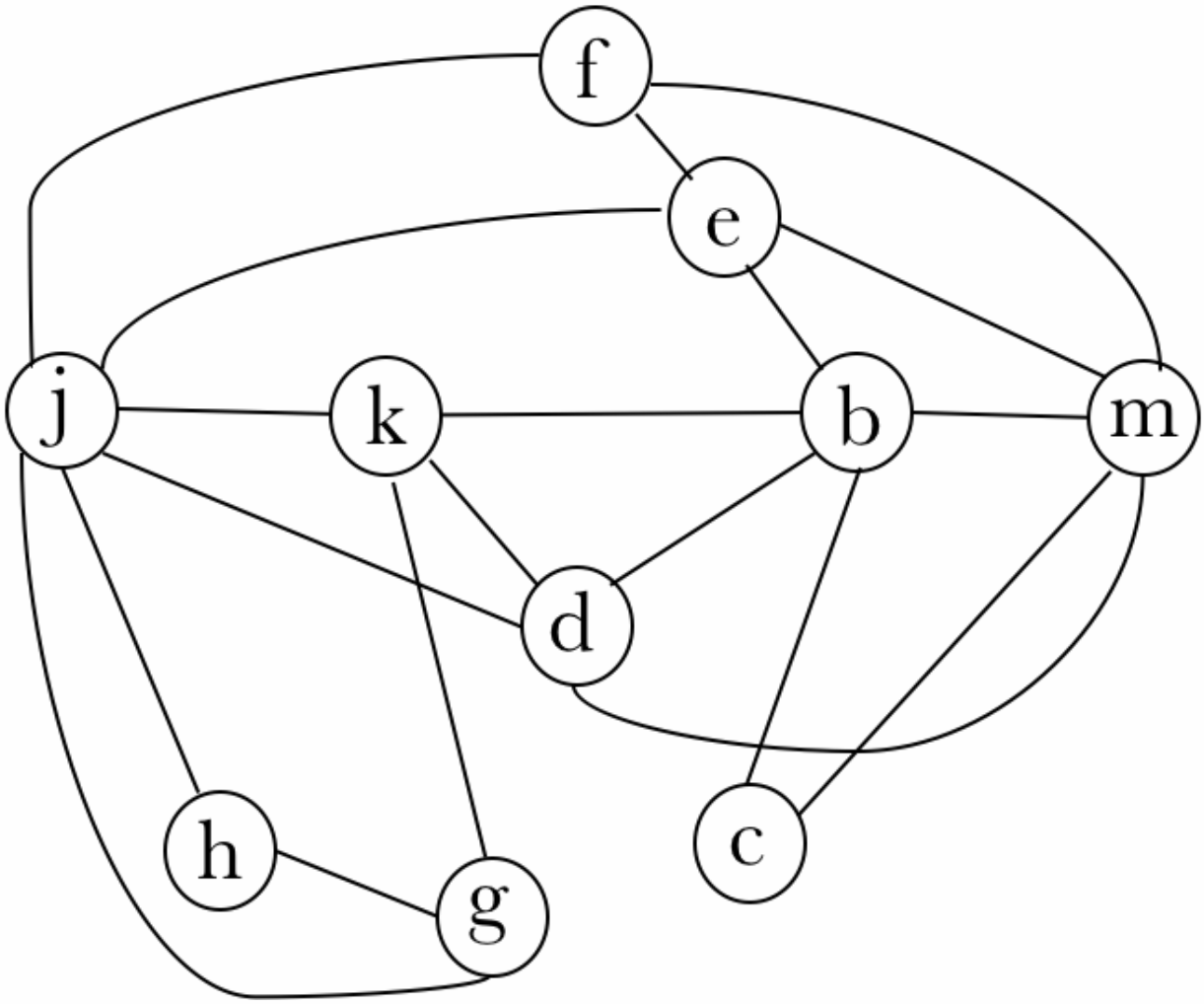
选择可能会失败。如果在选择的过程中出现了「颜色不够用」的问题，那么我们就不得不重新进行迭代操作、溢出一些变量，直到着色成功。

栗子

我们需要一个例子来使自己清醒。

```
[live in: k j]
g := mem[j + 12]
h := k - 1
f := g * h
e := mem[j + 8]
m := mem[j + 16]
b := mem[f]
c := e + 8
d := c
k := m + 4
j := b
[live out: d j k]
```

我们对这个程序进行冲突分析，同时留意一般的冲突使用实线连接，而 MOVE 则用虚线连接。



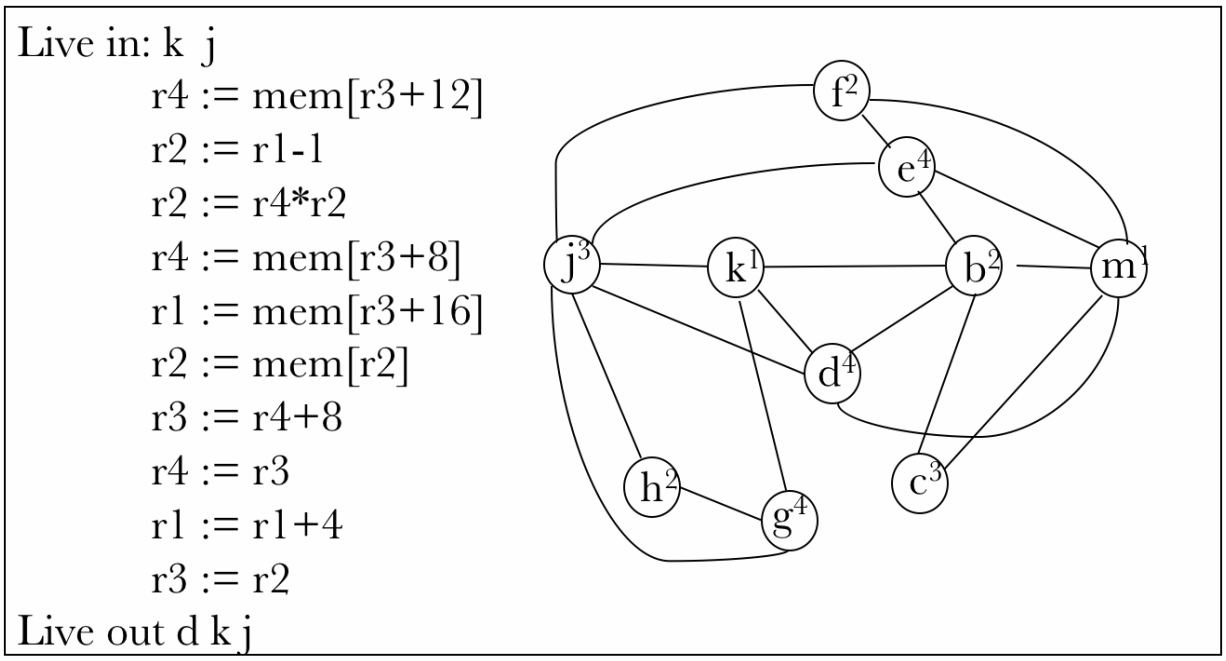
我们一个个地做简化（从图中删除度小于 K 的结点），直到删除干净——我们得到的简化栈是这样的。

```
Stack

=== stack top ===
m
c
b
e
d
j
f
k
h
g
=== stack bottom ===
```

然后，我们根据这个栈来重建同时着色。

Example(K=4)



最后，成功着色。

但是我们留意到两条 MOVE 指令：r4 := r3 和 r3 := r2，这是没什么意义的。

我们希望可以通过「合并」来减少这种无聊的指令。

合并 (Coalesce)

利用冲突图我们可以很轻松的删除冗余的传送指令。

规则：假如在冲突图中，一条传送指令的源操作数和目的操作数对应的结点之间不存在边，那么就可以删除这条传送指令，同时将它们合并为一个新结点。表现在结果上，就是这两个变量共用同一个寄存器。

存在下面两种安全的策略，所谓安全的测试是指他一定不犯错。

Briggs 策略

假如结点 a、b 合并产生的结点 ab 的高度数临结点的个数少于 K，则它们可以被合并。

> 人话就是：把 a、b 合并之后，观察他们所有的临界点；取出其中度数高于 K 的结点，计数；如果这种结点的个数少于 K，则它们可以被合并；反之不行。

这个规则的目的在于保证一个可 K 着色的图在合并之后不会变成一个不可 K 着色的。

> 因为合并 a、b 实际上是增加了限制：要求 a、b 一定是同一种颜色。这种限制有可能使得 K 着色不可能；Briggs 策略就是为了规避这种风险。

George 策略

结点 a 和 b 可以合并，除非下面的条件满足：

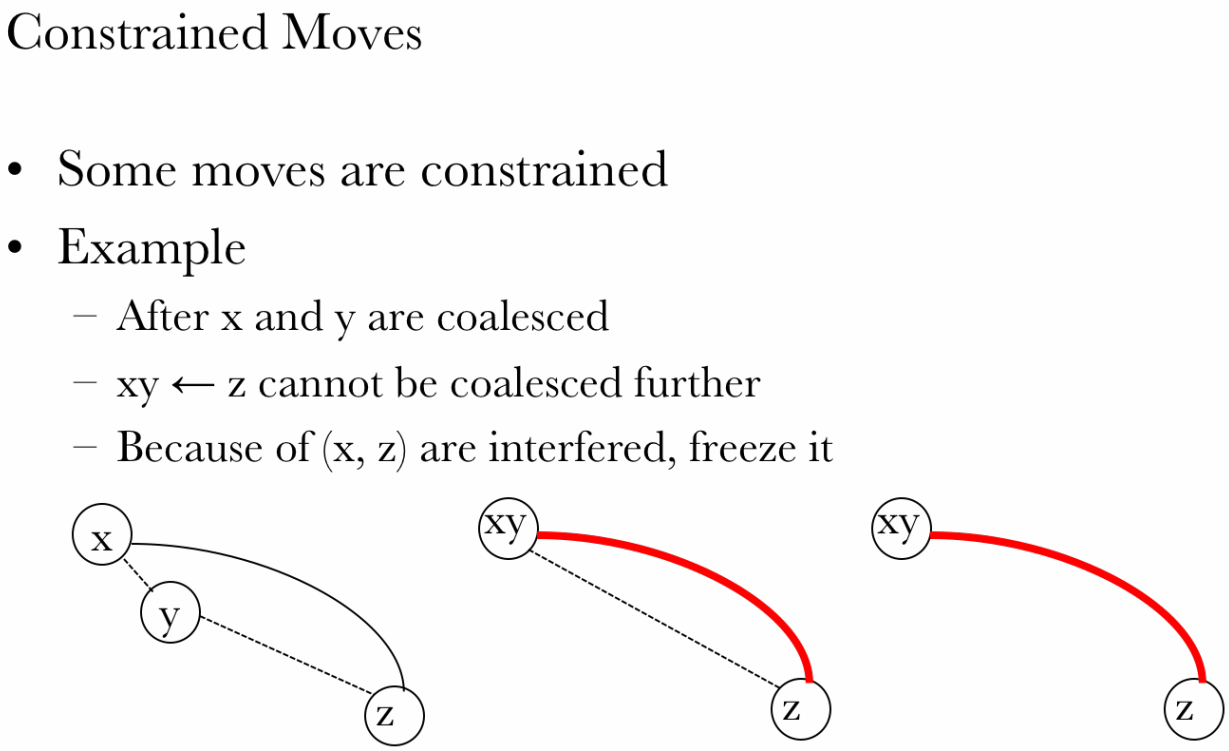
- 对 a 的任何一个邻结点 t，或者 t 是低度数（度数小于 K）的结点，或者 t 与 b 已经有冲突。

> 这个规则的目的在于：保证在合并前能够完成简化为空的图，在合并之后也总是能简化为空图。
>
> 又因为合并後的图是合并前的图的子图，所以它至少比原图更容易着色才对。

这两种策略是很保守的；也就是说，他可能在某些可以合并的情况下选择不合并，从而造成一些冗余的传送；但这总比激进地合并，最後不得不溢出来的好。

冻结

这是一个很要紧的点：并不是满足上面条件的 MOVE 都是可以 coalesce 的。看这个例子：



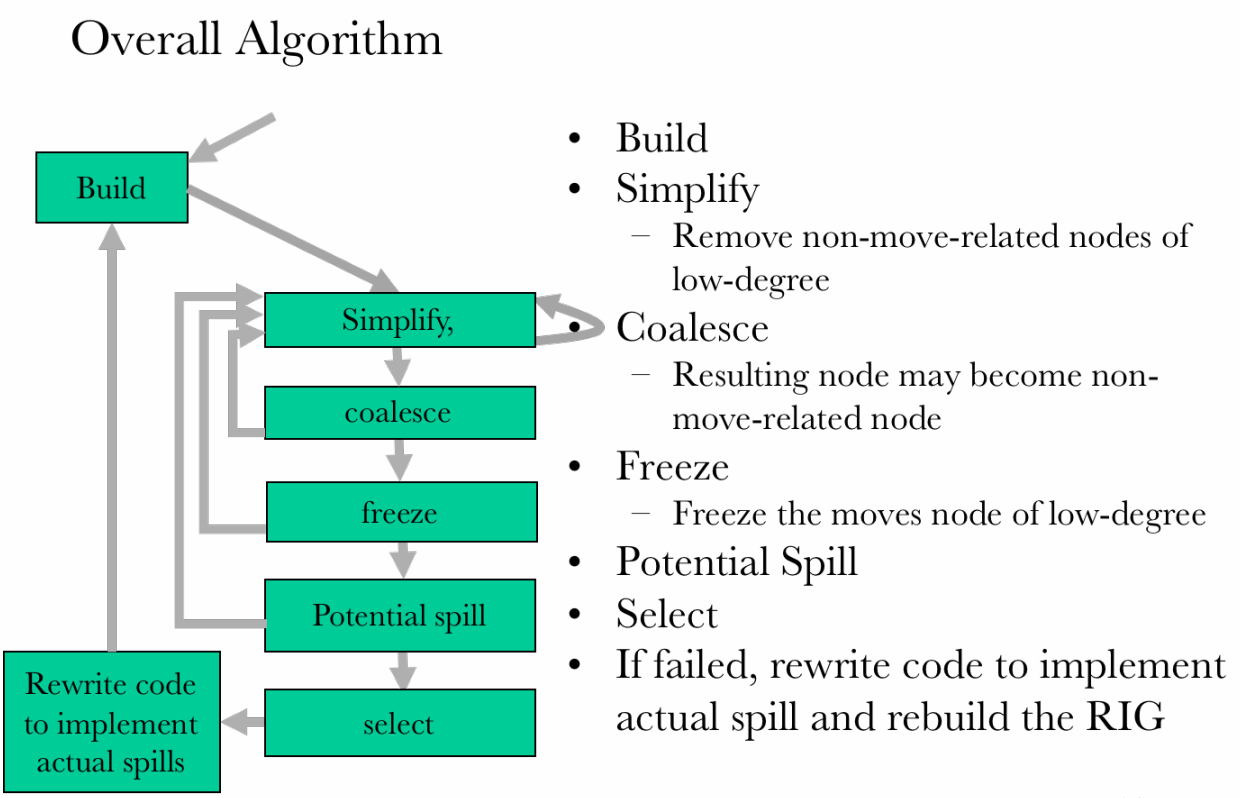
虽然 xy <= z 是 MOVE，但是同时他们还存在冲突（是 x 跟 z 的冲突继承下来的）。所以这时我们不能够合并它们二位，而是选择直接将它们画成实心线，也就是放弃对他们进行合并的希望，这个唤作 FREEZE。

步骤

因此，最终的步骤可以归结为下面的流：

- build
  - if success, go to simplify.
  - it never fails.
- simplify

    - if the graph isn't yet empty, go to simplify itself.
    - if the graph is empty, go to coalesce.
- coalesce
    - after the coalesce, if the node isn't a move, we can go further simplify.
    - If the simplify doesn't work, we can go to freeze, which means giving up doing any coalesce any more.
- freeze
    - that means more node can be regarded as move-innocent, and then we could do more simplify.
    - if there's no more simplify available, go to potential spill.
- (potential) spill
    - if there are no low-degree nodes, we will pick a high-degree node and mark it should be put into stack. go to select.
- select
    - now we will pop out the whole stack, and assign color to each node.
    - if there are nothing conflicts here, we can just pop out the output.
    - or if it can't be done, we will get that there are some actual spill. We will remove them, rewrite our code and rebuild the graph. go to build again.



预着色的结点

好麻烦 好麻烦

有一些机器寄存器是预先着色的；我们不能给他指派任意颜色。例如，frame pointer（帧指针）所在的寄存器、第一、二个参数所使用的标准寄存器等。这些变量要求特定颜色的寄存器，并且不能为其他变量所用。

简单说，对于任意一个 Pre-colored 的机器寄存器，应该只存在一个使用这种颜色的预着色结点。同一块语句中，不能指定两个变量呆在同一个预指派寄存器里。

然而，选择和合并操作实际上可以在不造成冲突的情况下随意使用这些预着色的的寄存器。

> Keep in mind: 预着色寄存器只是规定了特定位置的变量（如传参变量、返回值变量）必须有特定的颜色，而且每种颜色只会出现一次限定——但这并不妨碍你在使用的过程中随意取用、改变他的值，甚至和其他临时变量合并。这也是为什么我们称之为「预着色寄存器」而非「专用寄存器」。

但是，我们不可以对一个预着色的结点进行「简化」。因为把它抽离出来、再重新指派颜色之后，就无法保证他具有原来指定的颜色了。同时，也不可能将这个预着色的结点「溢出」，因为他们已经注定要呆在寄存器里了。

> 为了方便考虑，我们定义这类变量的度为无限大。这样我们就不可能把它们溢出了。

预着色结点下的寄存器分配

着色算法本身依赖于不停地进行简化、合并、和溢出。这一操作要持续进行到只剩下预着色结点为止。

预着色结点不溢出，因此前端必须很小心地让他们的活跃范围尽可能小——例如，用一个额外寄存器来存储那些预着色的寄存器，以便规避预着色的影响，最后再将其恢复。

调用者保护寄存器和被调用者保护寄存器

寄存器分配器应该做到将跨调用的活跃变量分配到被调用者保护的寄存器里；而局部变量和跨过程调用不活跃的变量分配到调用者保护的寄存器里。这样，就能尽可能减少 Push 跟 Pop 这类无意义的语句。

溢出优先级

在我们遇到一个非溢出不可的情况之下（无法简化、亦无法冻结），该怎么决定溢出谁？

- 优先级为：(Use + Def) ÷ Degree。
- 假如结点出现在循环里，那么用（外层循环的 Use+Def + 内层循环的 10 × Use+Def）÷ Degree。
- 优先级越低，越优先溢出他。
- 永远不溢出预着色结点。