

鲍辰的

编译原理笔记

目录

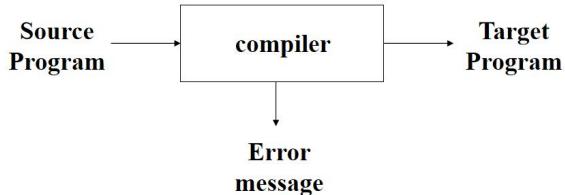
2021/9/14.....	5
2021/9/17.....	13
词法分析.....	18
Token 串的定义.....	19
用正则语言描述 Token.....	21
Maximal Munch (最大吞噬原则)	26
2021/9/24.....	28
用 DFA 判定属于哪个 token 集合.....	29
用描述 token 的正则表达式来构造 NFA.....	31
NFA 转化为 DFA.....	32
用查表来描述一个 DFA.....	33
压缩 DFA 的状态数量.....	33
让 DFA 支持 Maximal Munch 和 Priority Rule.....	35
Lex 工具.....	38
2021/9/26.....	40
语法分析.....	42
用上下文无关文法(CFG)来描述语法.....	43
CFG 的歧义.....	45
递归下降法构造 parse tree.....	48
2021/9/28.....	49
左递归和消除左递归.....	49
LL(1).....	50
LL(1)的 Parsing Table.....	52
LL(1) Parsing 算法.....	53
构造 LL(1) Parsing Table.....	54
First 集合和 Follow 集合.....	55
出错处理.....	57
LR Parser.....	59
2021/10/8.....	60
Shift 和 Reduce.....	61
LR(1).....	62
LR(1) item.....	66
2021/10/12.....	69
LALR(1):合并 LR(1)的 DFA 状态.....	73
语法分析工具 yacc.....	78
AST.....	80
2021/10/15.....	83
语法分析工具 Bisonc++.....	84
2021/10/19.....	95
Symbol Table.....	100
2021/10/22.....	103
Type Checking.....	110
2021/10/26.....	115

Tiger 中的递归声明.....	115
Tiger 中的递归函数调用.....	116
Activation Record (Frame)	119
2021/10/29.....	125
Displays.....	131
Access.....	134
Escape Analysis.....	138
2021/11/02.....	139
Tree Language.....	143
Expression.....	143
Statement.....	144
2021/11/5.....	149
数组和 Record 的翻译.....	151
2021/11/9.....	161
第八章： Basic Blocks and Traces.....	165
Canonical Form.....	168
2021/11/12.....	174
第九章： Instruction Selection.....	177
Maximal Munch (局部最优)	180
2021/11/16.....	181
2021/11/23.....	188
ProcEntryExit2.....	190
ProcEntryExit3.....	190
Global Optimization.....	191
变量活跃的条件.....	191
死码删除.....	192
Liveness Analysis： 数据流方程.....	195
常数传播.....	196
常数传播的八条规则.....	196
常数传播算法.....	197
常数传播算法的中止性.....	198
2021/11/26.....	198
Interference Graph.....	199
Kempe's Algorithm.....	199
Actual Spill (寄存器溢出)	201
Briggs 算法.....	205
George 算法.....	206
Constrained Move (受限的 Move 操作)	209
2021/11/30.....	214
在 Tiger 中寄存器分配的实现.....	221
Control Flow Graph.....	222
2021/12/03.....	228
在 Tree 上做寄存器分配.....	237
2021/12/07.....	239

Mark and Sweep.....	241
Mark And Sweep 的开销.....	242
Explicit Stack.....	243
Pointer Reversal.....	244
Reference Count.....	245
Defer reducing the reference count.....	246
RC 的性能开销高.....	246
RC 不能解决相互引用的问题.....	247
Copy 算法.....	247
Cheney's Algorithm.....	248
2021/12/10.....	250
Copy Collection 的开销.....	252
Generational Garbage Collection.....	252
Incremental Collection (增量回收)	258
Baker's Algorithm.....	263
2021/12/14.....	270
GC 的实现.....	270
Data Layout Description.....	272
Exact Root Description.....	276
Modern GC in Java.....	285
2021/12/17.....	289
Fun-tiger.....	291
闭包 (Closure)	293
PureFun-Tiger.....	296
Immutable Variables (不可变形)	296
Continuation-based I/O.....	298
2021/12/21.....	300
inline expansion.....	301
Unnesting Let.....	310
Closure Conversion.....	312
Efficient Tail Conversion.....	316
Lazy Tiger.....	321
2021/12/24.....	323
Call-by-need.....	326
Lazy 函数式程序的优化.....	329
单一继承.....	336
多重继承.....	339
2021/12/28.....	340
Global Graph Coloring.....	341
Membership Testing.....	347
2021/12/31.....	354
De-virtualizing.....	355
More C++.....	358
RAII (Resource Acquisition is Initialization)	359

2021/9/14

- There are 6 labs
 - The only homework
 - 60% total
 - The final due date is end of this semester
- There are two quizzes
 - 20% each
 - The first quiz contains the first five chapters
 - The second one contains the rest



编译核心来做就是拿到一个源程序(C/JAVA/...)编译成目标程序，当然编译可能不通过，会出现需要 fix 的 error/warning。

- Usually, high level programming languages
 - Fortran, Algo, Pascal, C
 - Lisp, ML
 - Prolog
 - Java, Python
- Sometimes, others
 - Verilog, lex languages, ...

源语言通常算是比较高级的语言，C 语言其实可以内联汇编，是比较低级的程序。ML 是这本书的作者偏爱的语言。这些语言是偏函数式的。Verilog(FPGA, 小板子编程)。

目标语言通常是机器的语言：汇编语言、二进制语言（可执行文件中的 0 和 1）。用 objdump 可以 dump 出来汇编代码，这两个是等价的。目标语言也可以是高级语言，这就是 source to source compiler。比如把其他语言翻译成 js，来放到 web 里去跑；或者把 web 代码部署到安卓上去运行。编译器的种类是很多的，和源语言、目标语言的多样性有关。

不同的编译器结构：

Single-pass 走一轮就结束了，现在比较少了，通常来说编译的源语言和机器码差别是很大的，各种各样的类型都不存在了，过程是比较复杂度。

multi-pass 市面上的大部分编译器

Load-and-go 像用 IDE，点一下 run 就执行完成了。把程序直接映射到内存里运行。把编译和执行的边界就模糊掉了。对内存要求比较高，因为要把生成的多个文件直接放进内存里。

Debugging：调试器也有编译的内容，比如单步执行可以模拟代码执行，这就需要知道代码在做什么

Optimizing: 需要分析代码，知道语义，再去做针对性的优化。也会用到编译的结构。

Basic 解释器，它是一行行读入解析的。读一行模拟执行一行。这就不像 **gcc** 中生成可执行文件再去执行。

编译是整体把程序编成了一个可执行的东西，再去执行。解释器和编译器差别比较大，但是在解释的时候也会用到一些编译的思路。像 **Python**，是翻译成了中间表达，再去解释执行。

Compiler 的历史，第一个 **compiler** 是 1953 年……

我们用的是虎书，还有一个比较有名的是龙书。虎书的 **guided** 多一点。第一部分就是自己写一个编译器，第二部分就是内存管理、面向对象语言设计、函数式语言设计、后端优化的介绍。

Outline

- How to define source languages?
- How to represent programs internally?
- How to translate sources to internals?
- How to translate internals to targets?
- Tiger language

A simple language

- Straight line
 - Assign statements
 - Print statements
 - Sequence of statements
- Form of expressions
 - Identifiers, numbers
 - Expressions connected by operators such as
 - +, -, *, /
 - Expression sequence
 - (statement, expression)

这是一个 lab，我们定义了一个简单语言，没有分支，一条条执行了看，有赋值、输出、可以用分号隔开来执行，有 **expression** 包含加减乘除变量 etc。认为每个指令都是有值的。**Statement** 是没有返回值的，**expression** 是有返回值。**Expression sequence** 要求最后一个指令一定是 **expression**。

Example

```
a := 5 + 3 ; b := (print(a, a-1), 10*a); print (b)
```

Output

8, 7

80

因为 `expression` 或者一个代码块是有返回值的，可以作为一个返回值赋值给别人。

如何去定义这个语言呢？这里就需要用到词法和文法。在学英文的时候，词法就是字母表有哪些。计算机语言相对比较严谨，词法通常使用正则表达式。语法可以来定义哪些指令是允许的和不合规的指令。

Syntax Definition for the Source

- Alphabet
 - letter $\rightarrow A \mid B \mid \dots \mid Z \mid a \mid b \mid \dots \mid z$
 - digit $\rightarrow 0 \mid 1 \mid \dots \mid 9$
 - Vocabulary
 - id \rightarrow letter (letter | digit)*
 - num \rightarrow digit digit*
 - Delimiter
 - White space, newline, tab

对于这个语言，文法定义是比较简单的。竖线表示的是或。`Vocabulary` 对应的是变量和数字。`*` 表示 0 个或多个。在别的地方，`+` 表示 1 个或多个。

- **Statement**
 - $Stm \rightarrow Stm; Stm$ **(CompoundStm)**
 - $Stm \rightarrow id := Exp$ **(AssignStm)**
 - $Stm \rightarrow \text{print}(ExpList)$ **(PrintStm)**
 - **Exp**
 - $Exp \rightarrow id$ **(IdExp)**
 - $Exp \rightarrow num$ **(NumExp)**
 - $Exp \rightarrow Exp \text{ Binop } Exp$ **(OpExp)**
 - $Exp \rightarrow (Stm, Exp)$ **(EseqExp)**

语法的基本结构左边表示需要解析的一个符号，右边就是解析出来的结果。我们会给不同的推导命名。

CompoundStm(混合 statement)

`ExpList` 可以传入一个表达式的 list，变长 `print`。

Binop: 加減乘除

- $ExpList$
 - $ExpList \rightarrow Exp, ExpList$ (PairExpList)
 - $ExpList \rightarrow Exp$ (LastExp)
- $Binop$
 - $Binop \rightarrow +$ (Plus)
 - $Binop \rightarrow -$ (Minus)
 - $Binop \rightarrow *$ (Times)
 - $Binop \rightarrow /$ (Div)

我们简单给了一个上下文无关文法的定义，在第三章会详细结束。

Context-free Grammar

1. A set of tokens
 - terminal symbol
2. A set of nonterminals
3. A set of productions, where each production consists of
 - a nonterminal
 - the left side of the production
 - an arrow
 - a sequence of tokens and/or nonterminals
 - the right side of the production

Token (终结符, terminal symbol)：不能再继续推导的内容，比如加减乘除、括号

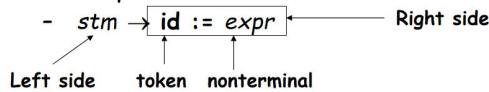
非终结符：可以继续推导的符号，比如 $ExpList$ 。

我们的语法就是包含了推导规则，左边是非终结符，右边数终结符和非终结符。并且要求给定一个开始推导的符号。

Eg:

4. A designation of one of the nonterminals as the start symbol

- Example



- Statement
 - $Stm \rightarrow Stm; Stm$ (CompoundStm)
 - | $id := Exp$ (AssignStm)
 - | $print(ExpList)$ (PrintStm)
- Exp
 - $Exp \rightarrow id$ (IdExp)
 - | num (NumExp)
 - | $Exp Binop Exp$ (OpExp)
 - | (Stm, Exp) (EseqExp)

更简洁的写法：

第二个问题：怎么内部表示一个程序

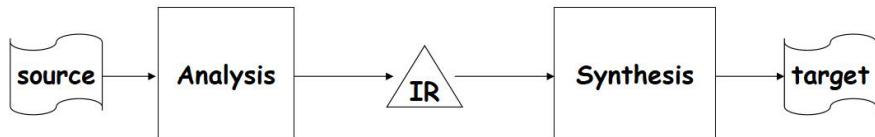
Basic Structures of Compilers

- **Analysis**

- From source programs to intermediate representations

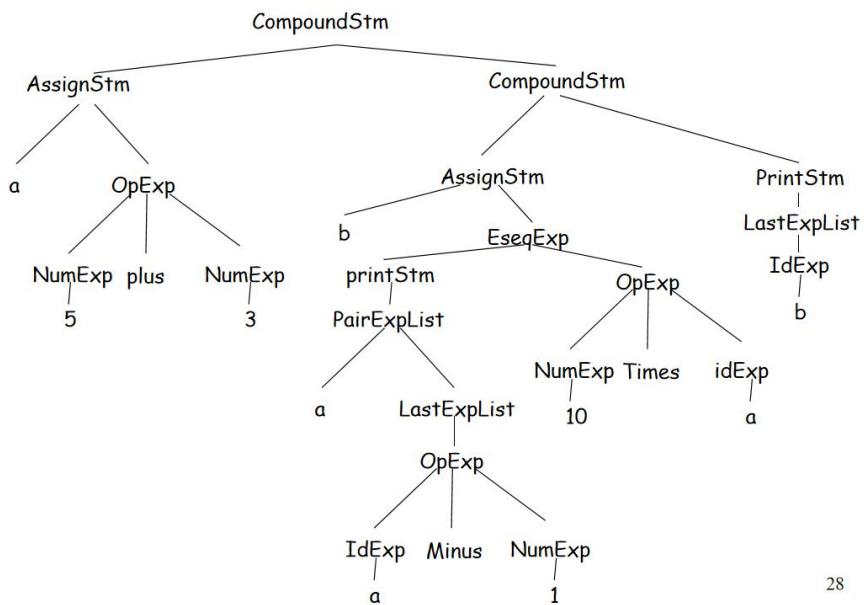
- **Synthesis**

- From intermediate representations to target programs



编译器的基本结构:大体也可以分成前端(分析, 把源代码编译成中间表达)和后端(合成)

a := 5 + 3 ; b := (print(a, a-1), 10*a); print (b)



28

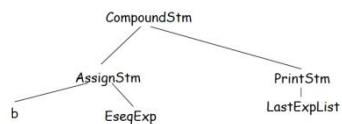
一个递归向下的树的结构。

我们可以把树的结构和语言对应起来。

```

class CompoundStm : public Stm {
public:
    CompoundStm(Stm *stm1, Stm *stm2) : stm1(stm1), stm2(stm2) {}
private:
    Stm *stm1, *stm2;
};

namespace A {
class Stm;
class Exp;
enum BinOp { PLUS = 0, MINUS, TIMES, DIV };
}
  
```



使用 c++写 lab, 比如说 compound 是有一个构造器。

```

class IdExp : public Exp {
public:
    explicit IdExp(std::string id) : id(id) {}
private:
    std::string id;
};

class NumExp : public Exp {
public:
    explicit NumExp(int num) : num(num) {}
private:
    int num;
};

```

```

graph TD
    Exp[Exp] --> IdExp[IdExp]
    Exp --> NumExp[NumExp]
    subgraph IdExp_inherit [ ]
        direction TB
        IdExp[id]
        IdExp --- id
    end
    subgraph NumExp_inherit [ ]
        direction TB
        NumExp[num]
        NumExp --- num
    end

```

Assignment

1. Write a function `int Maxargs()`

- Tells the maximum number of arguments of any `print` statement within any subexpression of a given statement
- Remember `print` statements can contain expressions that contain other statements

2. Write a function `void Interp()`

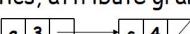
- Interprets a program in this language
- Make two mutually recursive method `Interp()` in `Stm` and `Exp`

两个工作：1.写一个 `maxargs` `print` 最多包含了多少个 `argument`, 比如 `print(explist)`, 注意在 `expressionlist` 可能会分裂成更多的 `expression`, 需要递归判定

2. 解释, 把程序通过解释的方法去执行一下。对不同的结构需要写不同的 `interp` 函数。

实现 lab1 需要的东西, `symbol table`, 对应的一个是映射, 变量 `id` 到数字(`int`)的映射。已经实现好了这样的一个表, 有一个 `tail` 把它们连起来。核心函数就是 `lookup` (拿到 `key` 找 `value`) 和 `update` (放到 `table` 里去)

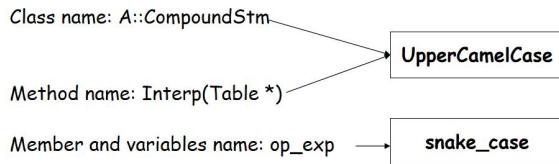
Straight-line program interpreter

- Input
 - Write hardcoded programs by directly invoking constructors
- No side effect
 - Denotational semantics, attribute grammar
 - $\{a \mapsto 3, c \mapsto 4\}$, 
 - Changing `c` to 7
 - $\{a \mapsto 3, c \mapsto 7\}$, 

在解释的时候用到这个 `table`, 只需要拿这个 `statement` 做出相应的解释。我们不希望有 `side-effect`, 也就是我们不会去修改已有的值。也就是 `straight-line program interpreter` 不依赖硬件去执行, 比如我们赋值 `c=7`, 那么在所有硬件和操作系统, 都应该 `c=7`。

- Execution
 - Traverse the tree
 - evaluate each subtrees and then the tree itself
- ```
Table *Stm::Interp(Table *)
 - The result is a table
 • final values of each identifier
struct IntAndTable { int i; Table *t; };
IntAndTable Exp::interp(Table *);
 - The result is a table
 • For side effects
 - and the value of the expression itself
```

给定一个树，每个子树有我们定义的虚函数执行一下，最终执行根节点。值得一提的是 interp 的返回值是一个 table 和一个 int（存的是返回值），我们维护的就是这么一个环境，表达的是变量和值之间的映射。

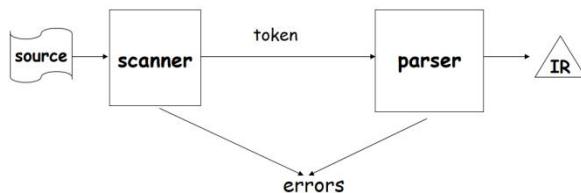


Each module (header file) shall have a namespace unique to that module

类型和方法：首字母大写，上驼峰；变量：下划线。

第三部分：怎么把源语言转化为 Internal.

刚才我们看到的树（AST），有了这个以后我们怎么做转化呢？涉及到 scanner（词法分析）和 parser（文法分析）



Source 传过来只是一个个字符，通过 scanner 就识别出了其中的 token，给到 parser 就会根据我们定义的语法规则，生成中间表达，也就是我们的抽象语法树。

## Front end

---

- Responsibilities:
  - recognize legal procedure
  - report errors
  - produce IR
  - preliminary storage map
  - shape the code for the back end
- Much of front end construction can be automated

为什么是 preliminary，是比较原始的。如果 a 只有可能是 0 和 1，并且后面有一句 if (a > 2)，那么后端检测到以后就可以去掉，而在前端里面只是最简单的存储。

比如 a:=a-1，分解成 a 赋值再一个 a-1，scanner 可以得到 a 的类型。换句话说 scanner

的责任就是把字符映射到终结符(token)，做完词法分析以后，对语法分析来说没什么差别。因为格式被整理过了。

- Parser
  - recognize context-free syntax
  - guide context-sensitive analysis
  - construct IR
  - produce meaningful error messages
  - attempt error correction
- Parser generators mechanize much of the work

很多 c++ 的错误是语法分析的错误，还有一些是语义分析的错误（类型、强制转换）。语法分析器还会尝试做一些 error correction，为的不是通过编译，而是为了找到后面的 bug，尽可能一次性把所有的语法分析都报告出来。这里要提的是 parser generator，现在的 parser 是自动生成的，需要给的就是语法规则。

## Back End

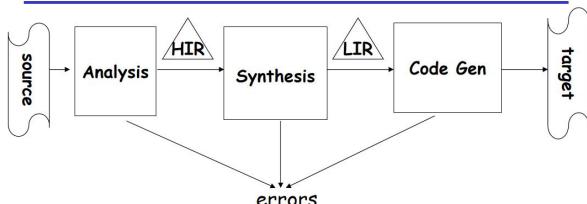
---

- Responsibilities
  - translate IR into target machine code
  - choose instructions for each IR operation
  - decide what to keep in registers at each point
  - ensure conformance with system interfaces
- Automation has been less successful here

后端职责：拿到前端的中间表达转成机器码，需要给每个中间表达选择 instruction。我们希望尽可能把所有的东西都存到寄存器里，这就是寄存器分配，后端编译器优化比较重要的方面。最后一个就是要和 system interface 兼容，有些平台无关的语言（java）就要为不同的硬件去适配接口，需要后端完成。所以目前后端还是很少有在做自动化的东西。

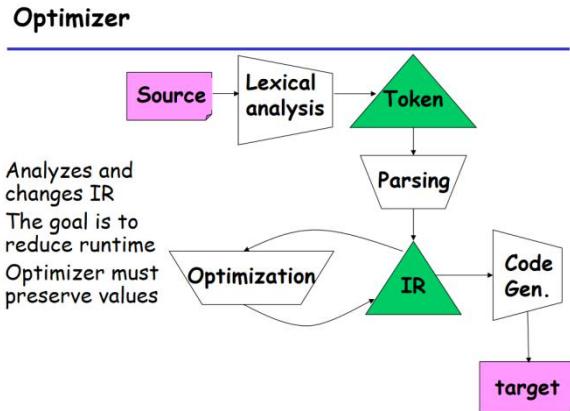
## Three pass compilers

---

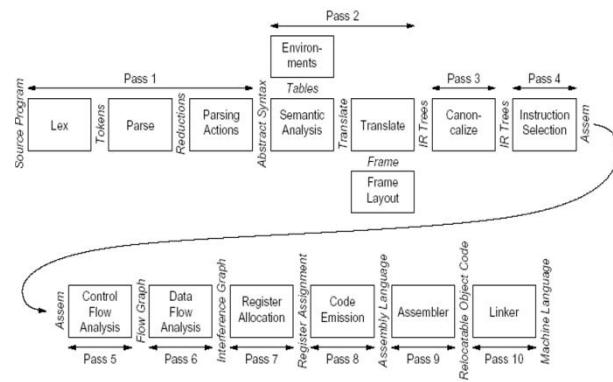


- Analyzes and generates HIR close to source
- translates HIR to LIR close to target
- Emits target from LIR

Source 到 target 差距实在是太远了，所以拆成两个 IR，把 High-level 的 IR（接近源代码，还原了 source 中的语义）转化为 low-level 的 IR（接近 target，里面会有 move, load 之类的指令）。最终我们通过 code gen 转化为机器码。实际的编译器可能更加复杂，就有多层 IR，随着结构复杂，对 IR 的需求也会不一样。



这里给了一个优化器的介绍，我们之前都在讲前端的词法分析和文法分析。优化器的目标就是分析 IR 和改变 IR，就是在树上不断地做遍历，找到一些匹配的 pattern 去做优化。优化本质就是效率导向的模块。这里的目标是优化不能违反原本程序的语义，需要 **preserve** 原先的 **value**。



这个是课程的 flow，元编译器通过 parse 后面有语义分析和中间检查。……

## 2021/9/17

今天讲第二章，我们先要讲一下虎书

编译有三本书，第一本是龙书，是两个图灵奖写的，是用的比较多的一本编译书，但是读完以后写不来编译器。

我们这本书的作者自己设计了一个 tiger language，

# Tiger Language(queens.tig)

```
let
 var N := 8
 type intArray = array of int ← Buiding type
 var row := intArray [N] of 0 ← array
 var col := intArray [N] of 0
 var diag1 := intArray [N+N-1] of 0
 var diag2 := intArray [N+N-1] of 0

 function printboard() =
 (for i:= 0 to N-1
 do (for j:= 0 to N-1
 do print (if col[i] = j then "0" else ".");
 Standard library print("\n"));
 print("\n"))
```

```
function try(c: int) =
 if c=N
 then printboard()
 else for r := 0 to N-1
 do if row[r] = 0 & diag1[r+c] = 0 & diag2[r+7-c] = 0
 then (row[r] := 1; diag1[r+c] := 1; diag2[r+7-c] :=1 ;
 col[c] :=r ;
 try(c+1) ;
 row[r] :=0; diag1[r+c] :=0; diag2[r+7-c] :=0)
 in try(0)
end
```

很多东西都是一个表达式，函数式的语言。前边这部分就是一个定义，定义完了以后去执行 try。主体比较简单，大头在前面的定义部分，var 是变量的定义，type 是类型的定义，还定义了两个函数 printboard 和 try。这个例子在我们树上的例子有，也就是用 tiger language 来模拟八皇后问题。里面有控制语句也有循环结构。:=赋值，=判等。在 tiger 中只有两个 build-in type，int 和 string，因为这是一个教学用的语言，所以为了保持简单性，没有浮点数

及其计算。intArray 实际上就是整数数组，row 和 col 在 tiger 中都是一维数组。print 函数和 C 的概念差不多，属于标准库。

## Tiger Language(Merge.tig)

```
let
 type any = {any : int} /*type declaration */
 var buffer := getchar() /*variable declaration */
 function readint(any: any) : int = /*function declaration */
 let
 var i := 0
 function isdigit(s: string) : int =
 ord(buffer)>=ord("0") & ord(buffer)<=ord("9")
 in
 while buffer==" " | buffer=="\n" do buffer := getchar()
 any.any := isdigit(buffer);
 while isdigit(buffer)
 do (i := i*10+ord(buffer)-ord("0"); buffer := getchar())
 i
 end

 type list = {first: int, rest: list} record

 function printint(i: int) =
 let function f(i: int) = if i > 0
 then (f(i/10); print(chr(i - i/10*10+ord("0"))))
 in if i < 0 then (print("-"); f(-i))
 else if (i > 0) then f(i)
 else print("0")
 end

 function printlist(l: list) =
 if l=nil then print("\n")
 else (printint((l.first); print(" ")); printlist(l.rest))

/* BODY OF MAIN PROGRAM */
in
 printlist(merge(readlist(), readlist()))
end
```

**Nested function**: Points to the nested function `function isdigit(s: string) : int =`.

**Buiding type**: Points to the type declaration `type any = {any : int}`.

**Standard library**: Points to the standard library function `getchar()`.

**record**: Points to the record type `type list = {first: int, rest: list}`.

第二个例子是 merge sort 这个例子，里面有更复杂的类型。之前看到的数组和整数，这里我们看到结构这样的类型。Getchar()是标准库中得到一个字符，就有字符串变量 buffer。如果有返回值，返回值是写在函数名字后面的:int，并且函数可以嵌套，里面的函数的作用域只在里面有效。在 ICS 中，i 要么应该是栈上的局部变量，i 要么是全局变量。这里 i 也应该在栈上的，理论上另一个函数应该是没有办法访问到的。在 C 中是没有碰到过的，而在 tiger 中会作为一个问题。

List 就是一个 record 结构类型，l 是类型为 list 的变量。nil 就是一个空指针。在 tiger 中没有显式的 pointer 变量声明，而在此 array 变量和 record 都是指针。我们在 C 中如果说一个变量是一个 structre，array 代表了这个首地址，而 structure 代表了内存中的一块东西。这里就是判定一个 array 或者一个 record 有没有初始化。Tiger 语言复杂的都在 let 这部分。一有 let 就可以在 let 中定义一些东西，再把函数体写在 in 这部分。这本书附录对 tiger 的描述需要大家多多去翻。

## Tiger Language

---

### Mutually recursive types

declared by a consecutive sequence of type declarations. Each recursion cycle must pass through a record or array type.

```
type tree = {key: int, children: treelist}
type treelist = {hd: tree, tl: treelist}
```

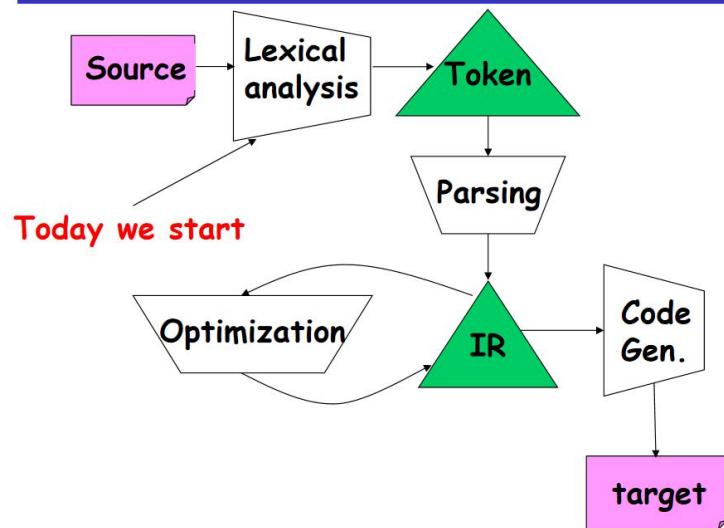
### Mutually recursive functions

declared by a sequence of consecutive function declarations

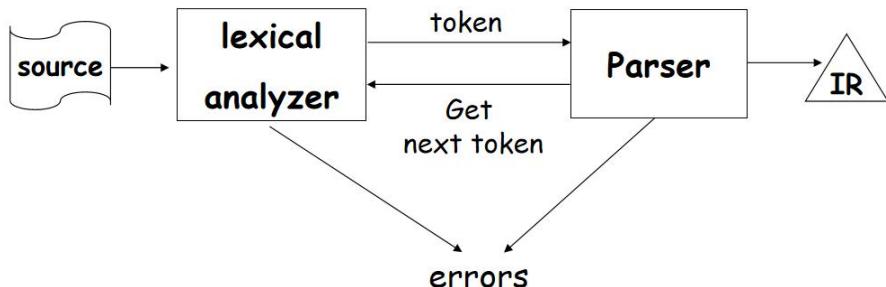
```
function treeLeaves(t : tree) : int =
 if t = nil then 1
 else treelistLeaves(t.children)
function treelistLeaves(L : treelist) : int =
 if L = nil then 0
 else treeLeaves(L.hd) + treelistLeaves(L.tl)
```

Tiger 语法上的情况，比如 tiger 中支持递归的数据类型和支持的函数调用。比如我们要定义一个二叉树或者定义一个链表，我们经常会使用递归的方式去定义。在 tiger 中定义，必须是两个东西挨着的。Tree 的儿子的节点构成了一个链表。这边的递归我们可以看到，自己和自己递归，在定义 tree 的时候用到了 treelist，在定义 treelist 的时候用到了 tree，tiger 要求这两个是互相挨着的，我们之后会学到这样要求会使得处理起来相对简单一点。同样的，函数递归调用。Treeleaves 调用 treelistleaves，而 treelistleaves 调用 treeleaves，这也要求两个函数挨在一起。

## The Structure of a Compiler



上节课我们讲过，给一个 source program，出来一个汇编。前端：把 source 变成计算机更加容易处理的数据结构。比如 lab1 中变成了一个 tree 和一个 table，这就在计算机上非常容易做了。我们从第二章~第五章，我们要做的事情就是把程序变成树和表，让程序自动来做。在自动做的时候，又分成两步。一步是词法分析，现在我们的 source program 是一个字符串的序列，然后经过词法分析要变成一个 token 序列（变量、函数都是 token），把没有意义的字符串序列变成有意义的单词序列，再把词法分析出来的单词接口变成段落文章，这部分就是语法分析。



在这个过程中，经常拼错或者少一个分号之类的，所以会报错会有一个出错处理的机制，碰到错误以后希望能够容忍这个错误继续编译下去找到更多的错误报告给用户。

# 词法分析

## Lexical Analysis

---

- What do we want to do? Example:

```
if (i == j)
 z = 0;
else
 z = 1;
```

- The input is just a sequence of characters:

```
\tif (i == j)\n\t\tz = 0;\n\telse\n\t\tz = 1;
```

- Goal:

- Partition input string into substrings (lexeme)
- classify them according to their role (tokens)

词法分析要把字符串序列变成 token 序列，我们下面的词法分析就是两件事情

1. 划分，tab 是没有用的，if 是一个对象，== 又是一个 pattern，我们要把这样的序列做一个切分，然后做一个分类。

2. 分类，比如 tab、空格、回车都是没用的东西，i、j、z 都属于变量，if、else 都属于关键字。

所以我们词法分析好以后，把原来的一串字符串变成了一个 token 串，它相当于是一个类型，和自然语言中的名词、动词、形容词，然后就可以分出主谓宾、定语状语之类的。在程序设计语言中，比如有 identifier, keyword, integer, relation(==), whitespace(tab, space, enter)。我们把字符串变成 token 串以后，对于我们的语法分析就很重要。我们的语法分析是基于这个 token 串来做的。

# Token 串的定义

## Tokens

A token corresponds to a set of strings:

- Identifier: strings of letters or digits, starting with a letter
- Keyword: "else" or "if" or "begin" or ...
- Integer: a non-empty string of digits
- Relation: <, <=, =, >, >=
- LeftPar: (
- Whitespace: a non-empty sequence of blanks, newlines, and tabs

... ...

### Lexeme

- A member (string) of the set (token) such as "else", "if"
- An instance of the token

### Nontoken

- Comments, Macro

什么叫做 token 呢？ Token 是字符串的集合。Tiger 中的变量（identifier）一定要是字母开头的， keyword 是 tiger 中自己规定的。每个集合就是一个 token

Lexeme 是集合当中的一个元素，所以我们前面看到的第一步划分出来的时候， i、j、== 都是集合中的一个元素，这就是 lexeme。分类以后就是 token。所以词法分析出来得到的就是一个 token 和 lexeme。

Non-token 就是我们的注解的一些宏定义，就需要用预处理去处理掉。在 tiger 中， comments 是可以嵌套的，允许/\* /\* \*/ \*/的嵌套匹配。

下面我们要来讲怎么实现这件事情。

An implementation must do two things:

1. Recognize substrings corresponding to tokens
  - Lexeme (substring) and token
2. Return the pair of token and lexeme just recognized

划分出 substring，以及 lexeme 属于的那一类，然后把这个返回给 parser。我们再以刚才的字符串为例

Recall:

```
\t if (i == j)\n\t\tz = 0;\n\telse\n\t\tz = 1;
```

Token-lexeme pairs returned by the lexer:

- (Whitespace, "\t")
- (Keyword, "if")
- (LeftPar, "(")
- (Identifier, "i")
- (Relation, "==")
- (Identifier, "j")
- ...

这个就是我们第二个 lab，给大家这样的一个字符串序列，大家要做的事情就是把这样一个 token-lexeme 串生成出来，大家还要处理 nontoken 之类的东西。

在我们识别出来了这些东西以后，为了改善我们程序的可读性，我们认出的东西如果会直接扔掉。只会传递下去有意义的东西。我们能不能先做一些预处理，比如先把 comments 和 whitespace 先处理掉呢？答案：有些程序是可以的，有些程序是不行的。比如在 C 中，一个字符串中放了一个\\*, 在另一个字符串里头放了一个\*//，如果我们在预处理的时候把中间的都去掉了，也不是那么容易去掉的。

### C Program:

```
main()
{
 int i=1, j=2, ij=0;
 i j=i+j;
 printf("%d\n", ij);
}
```

### Fortran Program:

```
program main
integer i,j,ij
i=1
j=2
ij=0
i j=i+j
print *, ij
end
```

比如上例的空格如果被处理掉了，本来应该出错的地方就编译通过了。Fortran 里头是，有没有这个空格没事，大概会被处理掉。

```
DO 5 I = 1
DO 5 I = 1.25
DO 5 I = 1, 25
```

```
if (then .gt. else) then
 then = else
else
 else = then
endif
```

Do 就是一个 for, 5 就是一个 label, 到了 label 这个地方就结束了。因为空格会自动去除, 如果把逗号写成了句号, 就会把循环写成一个 DO5I 的变量赋值

下边我们要来分割字符串, 然后分类分成 token, token 就是一个字符串的集合, 这个字符串要么是一个有限的集合, 要么是字母开始的变量, 要么数字开头的非空数字串。我们用这样的自然语言来描述 token, 我们希望用形式化的方法来描述 token, 也就是正则语言、上下文无关文法。

## 用正则语言描述 Token

我们先用 regular language 来描述 token。Regular language 很简单也很有用, 易于理解和实现。

**Def.** Let  $\Sigma$  be a set of characters ( $\Sigma$  is called the alphabet). A language over  $\Sigma$  is a set of strings of characters drawn from  $\Sigma$ .

Tiger 语言所有允许的字符都会出现在这个字母表中。

## Examples of Languages

- |                                                                   |                                                               |
|-------------------------------------------------------------------|---------------------------------------------------------------|
| • Alphabet =<br>English characters                                | • Alphabet =<br>ASCII                                         |
| • Language =<br>English words                                     | • Language =<br>C constructs                                  |
| • Not every string of<br>English characters is an<br>English word | • Not every string of<br>ASCII characters is a C<br>construct |

Note: ASCII character set is  
different from English character set

英语中字典中的单词是合法的，其他单词是不合法的。程序设计中也是这样。

在正则语言当中，语言首先是字符串的集合。下面对于我们词法分析比较重要，首先这个字符串是不是属于一个合法的字符串，如果合法那它属于哪一类。把合法的字符串怎么识别怎么分类，就用到了 regular expression 来描述这样一个东西。正则表达式是正则语言的一个 notation，A 是一个表达式，它所描述出来的那些字符串，就是一个语言。正则表达式和正则语言相当简单，

### Atomic Regular Expressions

- Single character: 'c'  
 $L('c') = \{ "c" \}$  (for any ' $c \in \Sigma$ )
- Concatenation: AB (where A and B are reg. exp.)  
 $L(AB) = \{ ab \mid a \in L(A) \text{ and } b \in L(B) \}$
- Example:  $L('i' 'f') = \{ "if" \}$   
(we will abbreviate 'i' 'f' as 'if')

第一类正则表达式就是 single character，有两个正则表达式，其实的就是 single character。Concatenation: A 和 B 就是，A 中的字符串和 B 中的字符串连在一起，也是我们的正则语言。

- Union:  $A|B$   
 $L(A \mid B) = \{ s \mid s \in L(A) \text{ or } s \in L(B) \}$
- Examples:  
 $'if' \mid 'then' \mid 'else' = \{ "if", "then", "else" \}$   
 $'0' \mid '1' \mid \dots \mid '9' = \{ "0", "1", \dots, "9" \}$   
 (note the ... are just an abbreviation)
- Another example:  
 $('0' \mid '1') ('0' \mid '1') = \{ "00", "01", "10", "11" \}$

第三个就是 Union,  $A \mid B$  也是一个正则语言。这样我们就可以把之前构造出来的 keyword 用 union 放在一起变成了一个 keyword 的集合。

### More Compound Regular Expressions

---

- So far we do not have a notation for infinite languages
- Repetition:  $A^*$   
 $L(A^*) = \{ "" \} \mid L(A) \mid L(AA) \mid L(AAA) \mid \dots$
- Examples:  
 $'0'^* = \{ "", "0", "00", "000", \dots \}$   
 $'1' '0'^* = \{ \text{strings starting with 1 and followed by 0's} \}$
- Epsilon:  $\epsilon$   
 $L(\epsilon) = \{ "" \}$

刚才表达出来的表达式都是有限的，要定义一个无限的，就是用 repetition（星号）规则。

### Example: Keyword

---

- Keyword: "else" or "if" or "begin" or ...

$'else' \mid 'if' \mid 'begin' \mid \dots$

(Recall: 'else' abbreviates 'e' 'l' 's' 'e')

这样就可以写出一个 keyword 这样的一个 regular expression。

## Example: Integers

Integer: a non-empty string of digits

digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'  
number = digit digit\*

Abbreviation:  $A^+ = A \ A^*$

number = digit<sup>+</sup>

整数是一个非空的数字串，可以空的数字串就是 digit\*，要保证非空就要在前头加一个 digit。  
后面可以跟任意多的数字。我们可以用 A+ 来替代 AA\*

## Example: Identifier

Identifier: strings of letters or digits,  
starting with a letter

letter = 'A' | ... | 'Z' | 'a' | ... | 'z'  
identifier = letter (letter | digit) \*

Is (letter<sup>\*</sup> | digit<sup>\*</sup>) equal to (letter | digit)<sup>\*</sup>?

Identifier: 字母开头，后面可以跟任意多的字母和数字。

## Example: Whitespace

Whitespace: a non-empty sequence of blanks,  
newlines, and tabs

(' ' | '\t' | '\n')<sup>+</sup>

Whitespace 包含空格、tab 和回车

## Example: Phone Numbers

- Regular expressions are all around you!
- Consider (021) 3420 - 7408

$\Sigma = \{ 0, 1, 2, 3, \dots, 9, (,), - \}$   
area = digit<sup>3</sup>  
exchange = digit<sup>4</sup>  
phone = digit<sup>4</sup>  
number = (' area ') exchange '-' phone

用 regular expression 可以描述我们日常生活中比较简单的东西，比如电话号码。这只是形式上合法，并不是任意三个组合都是一个有效的电话号码。

我们下面要知道词法分析怎么做。

### Regular Expressions => Lexical Spec. (1)

Lexical analyzer return token-lexeme pair

1. Select a set of tokens
  - Number, Keyword, Identifier, ...
2. Write a R.E. for the lexemes of each token
  - Number = digit<sup>\*</sup>
  - Keyword = 'if' | 'else' | ...
  - Identifier = letter (letter | digit)\*
  - LeftPar = '('
  - ...
3. Construct R, matching all lexemes for all tokens

$$R = \text{Keyword} | \text{Identifier} | \text{Number} | \dots \\ = R_1 | R_2 | R_3 | \dots$$

Facts: If  $s \in L(R)$  then  $s$  is a lexeme

- Furthermore  $s \in L(R_i)$  for some " $R_i$ "
- This " $R_i$ " determines the token that is reported

4. Let the input be  $x_1 \dots x_n$   
 $(x_1 \dots x_n$  are characters in the language alphabet)  
 For  $1 \leq i \leq n$  check  $x_1 \dots x_i \in L(R)$  ?
5. It must be that  $x_1 \dots x_i \in L(R_j)$  for some  $i$  and  $j$
6. Remove  $x_1 \dots x_i$  from input and go to (4)
  1. 先要定义 number、keyword、identifier，也就是 tiger 语言中有哪些词法成分。
  2. 用 regular expression 定义 token。
  3. 然后整个语言合法的子串写了有限个，这些合到一起都是我们合法的 substring。下面还要做一件事情。一个字串是不是合法的会有方法来判定。如果  $s$  属于  $L(R)$ ，就判断这个  $s$  是合法的字串。编译器里一定要落实到哪一个子类  $R_i$ 。
  4. 比如我们输入一个串  $x_1, \dots, x_n$ ，如果  $x_1, \dots, x_i \in L(R)$ ，一定  $x_1, \dots, x_i \in L(R_j)$ ，一边做一边 remove 掉这些东西。

## Maximal Munch (最大吞噬原则)

有了这些过程，我们来看以下这个例子

### Lexing Example

$R = \text{Whitespace} \mid \text{Integer} \mid \text{Identifier} \mid +'$

- Parse "f+3 +g"
  - "f" matches R, more precisely Identifier
  - "+" matches R, more precisely '+'
  - ...
  - The token-lexeme pairs are  
 (Identifier, "f"), ('+', "+"), (Integer, "3")  
 (Whitespace, " "), ('+', "+"), (Identifier, "g")
- We would like to drop the Whitespace tokens
  - after matching Whitespace, continue matching

我们碰到 F 一定是和 identifier match 的，一个个匹配完 remove 掉。最后我们就识别好了。是不是真的这么简单呢？

我们尝试换一个输入

- There are ambiguities in the algorithm
- Example:  
 $R = \text{Whitespace} \mid \text{Integer} \mid \text{Identifier} \mid +$
- Parse "foo+3"
  - "f" matches R, more precisely Identifier
  - But also "fo" matches R, and "foo", but not "foo+"
- How much input is used? What if
  - $x_1 \dots x_i \in L(R)$  and also  $x_1 \dots x_k \in L(R)$
  - "Maximal munch" rule: Pick the longest possible substring that matches R

如果 f 匹配了直接 remove，就会出问题，我们发现到了 foo 还是，还是 foo+ 不是了。我们还要尽量往后看，这就不是正则语言能干的事情了。Maximal munch 规则，也就是如果都属于  $L(R)$  的话，我们要选择更长的。

## More Ambiguities

---

$R = \text{Whitespace} \mid \text{'new'} \mid \text{Integer} \mid \text{Identifier}$

- Parse "new foo"
  - "new" matches R, more precisely 'new'
  - but also Identifier, which one do we pick?
- In general, if  $x_1 \dots x_i \in L(R_j)$  and  $x_1 \dots x_i \in L(R_k)$ 
  - Rule: use rule listed first (j if  $j < k$ )
- We must list 'new' before Identifier

第二件事情，我们要把这个语言改一改，加了一个 keyword 叫做 new，new 既满足 keyword 还满足 identifier。当同时满足的时候，我们选择前边的这个，也就是优先级规则，需要在规则中制定好优先级。

## Error Handling

---

$R = \text{Whitespace} \mid \text{Integer} \mid \text{Identifier} \mid +$

- Parse "=56"
  - No prefix matches R: not "=", nor "=5", nor "=56"
- Problem: Can't just get stuck ...
- Solution:
  - Add a rule matching all "bad" strings; and put it last
- Lexer tools allow the writing of:  
 $R = R_1 \mid \dots \mid R_n \mid \text{Error}$ 
  - Token Error matches if nothing else matches

最后是处理出错的情况，出错的字符串属于哪里呢？我们要在规则中加一个 Error token，不在前头的东西就属于 error。

## Summary

---

- Regular expressions provide a concise notation for string patterns
- Use in lexical analysis requires small extensions
  - To resolve ambiguities
  - To handle errors
- How to implement a lexical analyzer ?
  - Go to next

我们要做词法分析的时候，首先要用正则表达式来把 token 精确地描述出来，描述完了以后要做一个 extension，我们要知道词法规则是若干个 union 出来的。分类的时候就会有一些二义性，需要用到 maximal munch 和 priority rule。这样的话把我们的词法的描述，以及在识别的过程中碰到问题怎么处理讲清楚了。

接下来，给你一个字符串怎么判断属于这个集合。判定这件事情，我们用的就是有限自动机。

## Finite Automata

---

- Regular expressions = specification
- Finite automata = implementation
- A finite automaton consists of
  - An input alphabet  $\Sigma$
  - A set of states  $S$
  - A start state  $n$
  - A set of accepting states  $F \subseteq S$
  - A set of transitions  $state \rightarrow^{input} state$

自动机首先有一个字母表、状态集合、初始状态、一个终止状态集合、状态的转移。

2021/9/24

上节课我们说可以用 regular expression 来标识我们的词法，我们下面的问题转换成了：

## 用 DFA 判定属于哪个 token 集合

给定一个 regular expression  $L(R)$ , 我们要判定这个字符串是不是属于这个 regular expression 的定义中。这时候我们就要引入自动机（automaton）。接下来我们会讨论怎么通过这个 regular expression 自动化生成代码，会介绍自动化工具。今年得图灵奖的两位 aho 和 ullman, 当年在自动化这个方面做了很多的工作。然后就是 maximal munch 和 priority rule, 我们需要加入这些 extension rule。

## DFA 的定义

自动机在这里就是一个五元组：一个字母表，一个状态，开始状态，接收状态的集合  $F \subseteq S$  和 transition。

- Transition

$$s_1 \xrightarrow{a} s_2$$

- Is read

In state  $s_1$  on input "a" go to state  $s_2$

- If end of input (or no transition possible)

- If in accepting state => accept

- Otherwise => reject

系统在状态  $s_1$  的时候接收到字符 a, 就转换到了状态  $s_2$ 。输入是一串字符串走到最后，如果进入了一个 accept state，这个输入就被这个自动机接收了，那么字符串就属于这个自动机对应的 regular language 了。

### Finite Automata State Graphs

- A state



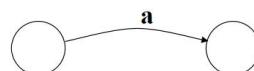
- The start state



- An accepting state



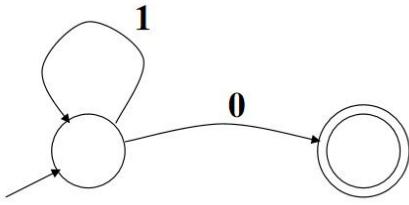
- A transition



Eg: 一个只接受“1”的正则语言对应的自动机



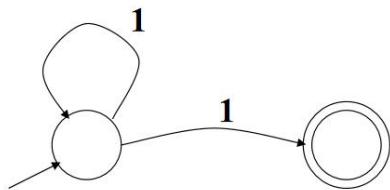
Eg: 只接受一串 1 之后跟了一个 0 的有限自动机



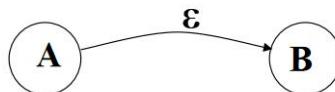
比如接收 1110，而 11101 就会 reject 0 后的 1

## NFA 的定义

一个有限自动机接收的是从 `start_state` 出发一个个状态走，最终走到 `accept state` 的字符串。刚才这些例子，在一个状态上，我们看到一个字符以后，`transition` 是确定的。我们看如下的这样一个自动机。这时候就会有不一样了，在 `start_state` 的时候看到 1 有两种 `transition`。

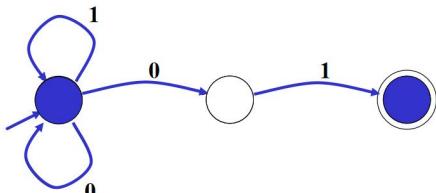


这也是一类自动机，但是和之前的 `transition` 确定的情况是不一样的。而且我们在两个状态之间加上一个  $\epsilon$  转移。说明可以不接收字符就可以无条件从 A 转移到 B。



上面这两种就是我们的 NFA (nondeterministic finite automata)。不管是 DFA 还是 NFA，都是一个非常简单的计算模型，我们在实现的时候，只需要当前状态和 input 就可以决定我们的动作，和之前无关，同时状态的数量也是有限的。

- An NFA can get into multiple states



- Input: 1 0 1

- Rule: NFA accepts if it can get in a final state

来了 0 之后，我们就会有多个 `transition`，可以同时处于这两个状态。再来一个 1，如果在左边状态，那么就继续停在原地，而中间状态的状态接收到 1 了以后就可以到 `accept state`。所以 101 就是这个 NFA 可以接收的字符串。

49

结论 1: NFA 和 DFA 能够识别的东西是相同的。

结论 2: DFA 写出来的代码是很容易的, 只需要  $n * n$  的一张 transition 表就可以了。

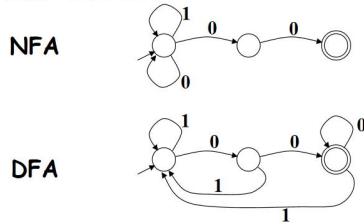
|       | a1 | ..... | am |
|-------|----|-------|----|
| S1    |    |       |    |
| ..... |    | S'    |    |
| Sn    |    |       |    |

也就是  $S_i$  和  $a_j$  可以转化到  $S'$  状态, 一张表查表就可以了。

查表的程序是一样的, 我们只需要自动生成这张表就可以了。我们声明的 regular expression, 直接构造 DFA 会比较困难, 我们可以先用 regular expression 来构造 NFA。

Eg: 识别同一个语言的 NFA 和 DFA 比较, DFA 的 transition 会比较复杂一些。

- For a given language the NFA can be simpler than the DFA



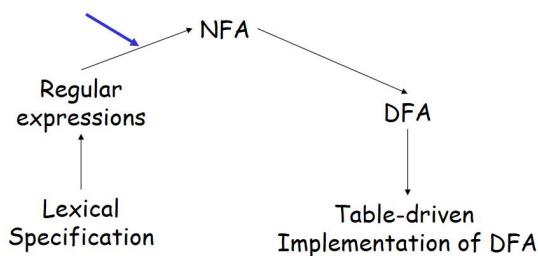
- DFA can be exponentially larger than NFA

同一个 NFA 变成 DFA 的时候, 有可能其状态会指数增长。

## 用描述 token 的正则表达式来构造 NFA

下边我们来看一下, 我们用 regular expression 来描述词法, 描述好词法以后, 我们希望用这个 regular expression 来构造出 NFA, 再转化成 DFA, 然后就变成一个查表判定的过程。

### High-level sketch



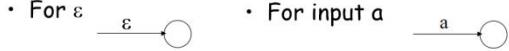
Regular expression 也就是 5 个构造的法则。

1.  $\epsilon$  法则
2. single character

图上一根线代表进去的地方, 后面有一个状态。我们假设 regular expression A 的图是这样, B 是对应下图的那样。Cat 法则就是结束节点连一根线到下一个的开始状态。

### Regular Expressions to NFA (1)

- For  $\epsilon$
- For input  $a$
- For  $A \mid B$

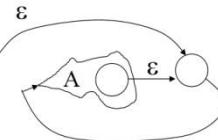
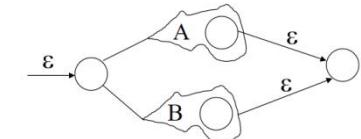


- For each kind of rexp, define an NFA
- Notation: NFA for rexp A

- For  $AB$



- For  $A^*$

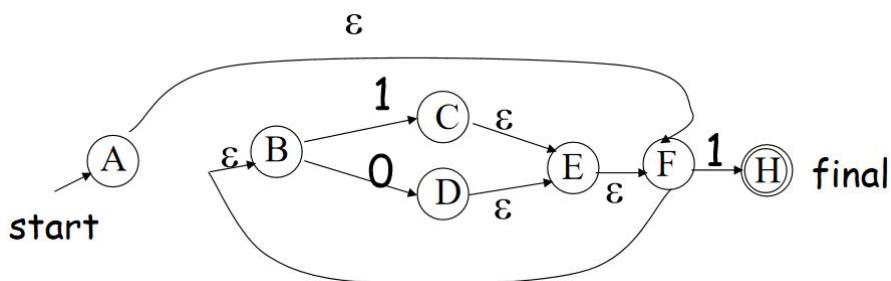


Union 我们就使用  $\epsilon$  连起来。

$A^*$  的话就是在 accept state 能够非确定性地转移到开始，当然空字符串也可以识别。所以要再加一根  $\epsilon$  的线。

这样我们就可以把一个 regular expression 画出一个 NFA 的状态转移图。

考虑正则表达式  $(1|0)^*1$



最后把图画完以后，我们再把 start 和 final 加进去，有了这两个才真正把图变成了状态转移图。这就是我们把一个 regular expression 变成一个 NFA 的过程。

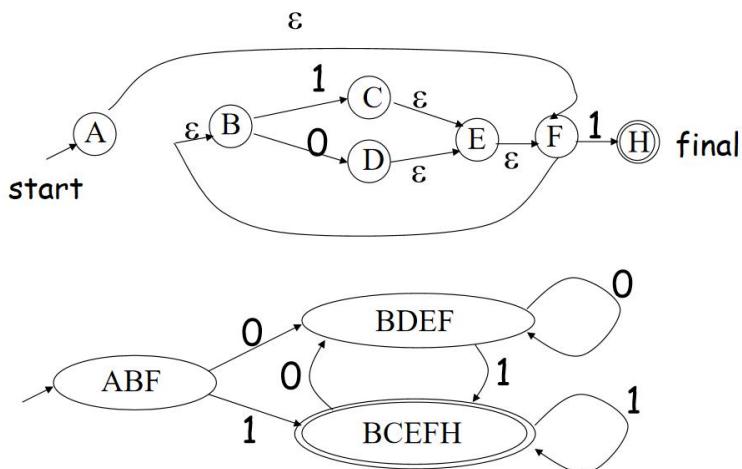
## NFA 转化为 DFA

接下来我们要把 NFA 变成 DFA，NFA 是不确定的不可能是查表的过程，所以我们要转化为 DFA 再进行查表。在 NFA 中一个 Input 会同时到好几个状态中去，我们用 DFA 中模拟 NFA 的这个过程，模拟过程中 DFA 的状态是 NFA 的状态的子集（若干个状态，non-empty subset of states of NFA）。NFA 同时到几个状态的时候，DFA 中就是一个状态。

构造出来的 DFA 的初始状态是什么？NFA 看似初始状态只有一个，其实开始状态能够  $\epsilon$  到达的状态都记为 DFA 的初始状态。

从一个状态  $S$  接收一个字符  $a$  到另一个状态  $S'$ ，我们要看看碰到每个字母表中的字符会变到哪里去，从而增加状态转移。

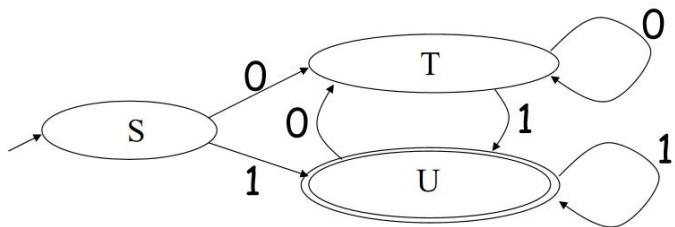
## NFA -> DFA Example



按照这个步骤执行肯定是可以把一个 NFA 转换成 DFA 的，只是可能指数爆炸。

## 用查表来描述一个 DFA

用表来描述一个 DFA



|   | 0 | 1 |
|---|---|---|
| S | T | U |
| T | T | U |
| U | T | U |

上表中，横轴是字符集合，纵轴是状态集合。

有了这个表以后，我们就从 `start_state` 开始一个个字符走，比如我们在状态  $i$  接收到了字符  $a$ ，直接查  $T[i, a] = k$  跳转到状态  $k$  即可。如果走到一半的时候走不下去了，也是 `reject`。

我们有一些自动化的工具，一般叫做 `lex`，比如 `flex` 和 `jlex`。也就是我们写 `regular expression`，它会生成 NFA, DFA, DFA 的表，然后我们就可以进行查表判定。实际上有时候 DFA 会比较大，也不一定要完全变成一个 DFA，可以中间自己多记录几个状态。

## 压缩 DFA 的状态数量

正规式->最小化 DFA 说明 <https://zhuanlan.zhihu.com/p/37900383>

## Minimize DFA state number

$s_1$  accepts  $\sigma$  iff  $s_2$  accepts  $\sigma$   
 $s_1$  and  $s_2$  are equivalent

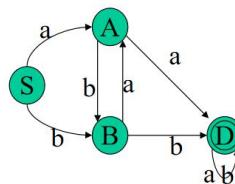
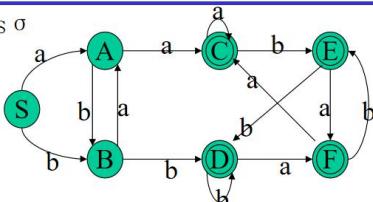
$\Pi 0: \{S, A, B\}$  (nonfinal)  
 $\{C, D, E, F\}$  (final)

a  $\{S, A, B\} \rightarrow \{S, B\}$  {A} (inequivalent destinations)

$\Pi 1: \{S, B\}$  {A} {C, D, E, F}

b  $\{S, B\} \rightarrow \{S\}$  {B}

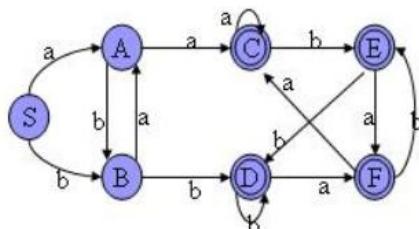
$\Pi 2: \{S\}$  {A} {B} {C, D, E, F}



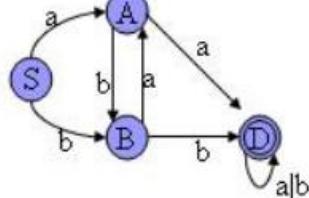
NFA 变 DFA 的状态会有很多，我们希望对 DFA 的状态进行化简一下，压缩一下状态。比如说我们有上面的这个 DFA，有 7 个状态 ABCDEF。要做化简实际上是做一个等价类，也就是说当我们状态  $S_1$  能够接收字符串  $s$  的时候， $S_2$  也能接收  $s$ 。

一开始我们可以把自动机的状态分成两类：

1. 非终结状态(non-final)集合
2. 终结状态集合



==MINIMIZE==>



步骤如下：

1. 将 M 的状态分为两个子集一个由终态  $k_1 = \{C, D, E, F\}$  组成，一个由非终态  $k_2 = \{S, A, B\}$ 。
2. 考察  $k_2 = \{S, A, B\}$  是否可分，因为状态 A 读到 a 之后到达了状态  $C \in k_1$ ，而状态 S 和状态 B 读到 a 之后到达状态  $A \in k_2$ ，所以可以分成  $k_3 = \{S, B\}$  和  $k_4 = \{A\}$
3. 考察  $k_3 = \{S, B\}$  是否继续可分，因为 S 读到 b 之后到达状态  $B \in k_3$  而 B 读到了 b 之后到达状态  $D \in k_1$ ，所以可以继续拆分成  $k_5 = \{S\}$  和  $k_6 = \{B\}$
4. 我们继续考察  $k_1 = \{C, D, E, F\}$  是否可分，因为 C,D,E,F 读到 a 和 b 都是转移到  $k_1$ ，所以  $k_1$  不再可分。我们可以选用其中的一个状态来代替所有状态，比如我们选择 D。
5. 重新连接对应的边即可。

Regular expression->NFA->DFA->Table 这就是一个 table-driven 的 algorithm。

因为我们的词法分析不是回答 yes/no 的事情，它的 specification 是这样写的  $R = R1 | R2 | \dots | Rn | error$ ，而我们输入的是  $x_1x_2\dots x_n$ ，我们是要切割出  $x_1x_2\dots x_k$ ，判断它是不是属于  $L(R_i)$ ，还要满足 maximal munch 和 priority rule。已经落到某一类的时候，我们需要继续往前走来找到更长的符合条件的情况。

所以刚才那个 table 我们只是有了一个基础的这个东西，我们要把这个东西扩展，扩展成我们的词法可以用的这个东西。

## 让 DFA 支持 Maximal Munch 和 Priority Rule

### Max length match (example)

|                                              |                     |
|----------------------------------------------|---------------------|
| if                                           | {return IF;}        |
| [a-z][a-zA-Z]*                               | {return ID;}        |
| [0-9]+                                       | {return NUM;}       |
| ([0-9]+ “.” [0-9]* ) ([0-9]* “.” [0-9]+)     | {return REAL;}      |
| (“--” [a-zA-Z]* “\n”)   (“ ”   “\n”   “\t”)+ | { /*do nothing */ } |
|                                              | {error();}          |

#### if --not-a-com

– (Fig.2.4, Fig 2.5)

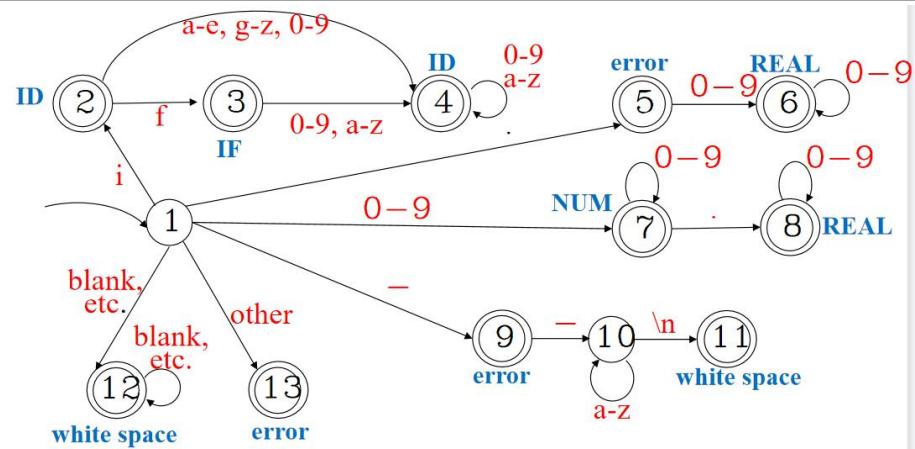
这就是一个词法

分别 keyword(if), identifier(返回 ID), number, 实数, 注解(两个"--"+一串字母+\n, 或者 space, \n, tab)，错误(如果前面的都不能匹配, 就报错了)

现在我们 Input: if --not-a-com

到 if --not 都能识别, 但是-a 我们不能识别出来, 就报错了。

这么 5~6 个 regular expression, 画出来的图是这样的。

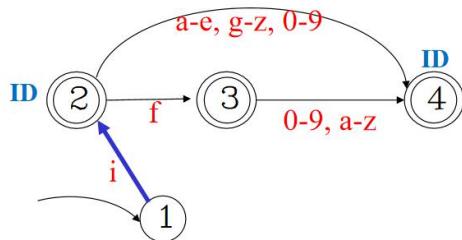


14 期末考试，给 3 个正则表达式画一下这张图。期中考试就是画这种图，两个小时三道题。这个过程需要练习一下。

我们现在有了这张图以后，我们需要识别输入的字符串。我们需要增加一些状态。

| Last Final | Current State | Current Input  | Accept Action |
|------------|---------------|----------------|---------------|
| 0          | 1             | If --not-a-com |               |

- The **T** is the input position at the last final
- The **L** is the current input position
- The **I** is the beginning of the current lexeme

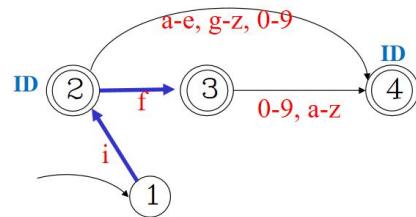


因为我们碰到 maximal munch，比如 i 进入到了 id 的 accept state，我们不能直接作为结果，读到 f 以后就变成了 keyword，如果读到了 if0，那么它又是一个 identifier。所以我们要继续读到后面。我们遇到 if 后的空格时，自动机报错我们就要回退。上一次进入 final 的时候，我们要记录一下。一开始是三者合一的工字型。一旦到了 2 以后，就有：

| Last Final | Current State | Current Input  | Accept Action |
|------------|---------------|----------------|---------------|
| 2          | 2             | if --not-a-com |               |

因为 maximal munch，我们需要继续往前走。

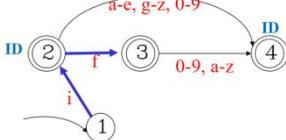
| Last Final | Current State | Current Input           | Accept Action |
|------------|---------------|-------------------------|---------------|
| 3          | 3             | if <u>T</u> --not-a-com |               |



| Last Final | Current State | Current Input           | Accept Action |
|------------|---------------|-------------------------|---------------|
| 3          | 0             | if <u>T</u> --not-a-com | return IF     |

• When a dead state is reached, the variables tells what token was matched and where it ended

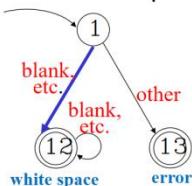
- Token: IF
- Lexeme: if



这时候空格转移不出去，我们就报错了。所以我们只要知道 I 和 T 之间的时候，我们就可以说这之间的是我们的 lexeme。所以我们只要知道 3 这里的 token 是 if 即可。然后我们继续回退空格，从空格开始继续走状态机。

| Last Final | Current State | Current Input           | Accept Action |
|------------|---------------|-------------------------|---------------|
| 0          | 1             | if <u>T</u> --not-a-com |               |

- The current input position is backtracked

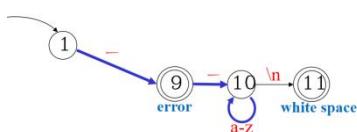


| Last Final | Current State | Current Input           | Accept Action |
|------------|---------------|-------------------------|---------------|
| 12         | 12            | if <u>T</u> --not-a-com |               |

| Last Final | Current State | Current Input           | Accept Action             |
|------------|---------------|-------------------------|---------------------------|
| 12         | 0             | if <u>T</u> --not-a-com | found white space; resume |

我们回退到-即可，输出 space 为 token。

| Last Final | Current State | Current Input           | Accept Action                     |
|------------|---------------|-------------------------|-----------------------------------|
| 9          | 0             | if <u>T</u> --not-a-com | error, illegal token '-' ; resume |



我们退回到第一个横杠，就报错。退回到 9 的情况：

| Last Final | Current State | Current Input           | Accept Action                     |
|------------|---------------|-------------------------|-----------------------------------|
| 9          | 0             | if <u>T</u> --not-a-com | error, illegal token '-' ; resume |

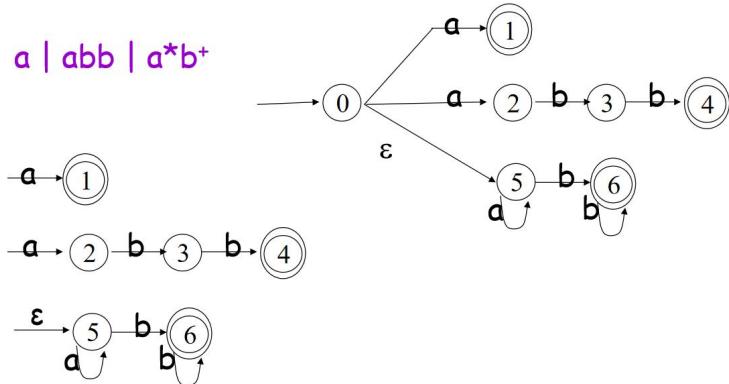
报错两次以后，我们就会认为 not 是一个 identifier，- 是一个错，a 是一个 identifier，- 继续报错，com 是一个 identifier。

所以在实际执行的时候，我们需要有三个 position 才能实现往前走+回退的机制。这是

maximal munch 需要的事情。

第二个事情是我们的 priority-rule。

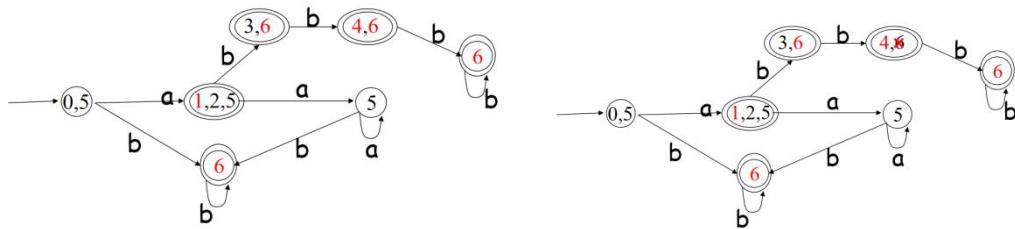
### Priority Rule



显然  $abb$  是  $a^*b^+$  的子集，但是  $abb$  优先，同时进入 4 和 6 作为 accept states 的时候，应该是 4 优先，因为 4 的优先级高。我们把 NFA 画成 DFA。

Two NFA final states are in one DFA state:  
- 4 and 6

the DFA state as the higher priority one in final  
Input:  $abb$ : accepted as  $abb(4)$  not  $a^*b^+(6)$



有问题的就是  $\{4,6\}$  这个状态，到了这个状态的时候，同时是 4 和 6 的时候我们只能是 4，需要在最后这个 DFA 中去掉优先级较低的 6。注意前面压缩 DFA 的时候不能随便压缩了。

### Lex 工具

到这里，我们工具的原理都讲完了，我们也处理了两条规则。

接下来我们来介绍 lex 工具。

## Lex specification of tokens (Cont'd)

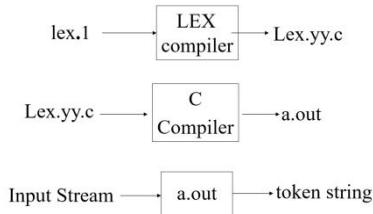
```

/*Regular Expressions and Actions: */
if {ADJ; return IF;}
[a-z] [a-z0-9]* {ADJ; yylval.sval=String(yytext); return ID;}
{digits} {ADJ; yylval.ival=atoi(yytext); return NUM;}
({digits}.””[0-9]*)([0-9]*.””{digits})
 {ADJ; yylval.fval=atof(yytext); return REAL;}
(“--”[a-z]*”\n”)|(“ ”|”\n”|”\t”)+
 {ADJ;}
.
 {ADJ; EM_error("illegal character");}

```

**The string matched by  
the regular expression**

我们写代码的时候还是写一串 **declaration**, 我们还需要写一串花括号。报错的时候我们要往前走。我们不但要有识别，我们还要做一些动作。花括号里的东西都是 C 程序。



我们写一些 **declaration** (regular expression) 和 **action** (C), 有一个简单的编译器生成 C 代码。生成了 C 以后，就可以放到编译器里头，编译出可执行的文件。**a.out** 就是输入一个 tiger program, 通过 **a.out** 运行以后能就变成了一个 token 串。

### Lex specification of tokens

```

%{
/*C Declarations: */
#include "tokens.h" /*definition of IF, ID, NUM, ...*/
#include "errmsg.h"
union {int ival; string sval; double fval;} yylval;
int charPos=1;
#define ADJ {EM_tokPos=charPos, charPos+=yylen}
%}
/*Lex Definitions: */
digits [0-9]+
%%

The length of the matched string
The semantic value of the matched string

```

**The string matched by  
the regular expression**

```

/*Regular Expressions and Actions: */
if {ADJ; return IF;}
[a-z] [a-z0-9]* {ADJ; yylval.sval=String(yytext); return ID;}
{digits} {ADJ; yylval.ival=atoi(yytext); return NUM;}
({digits}.””[0-9]*)([0-9]*.””{digits})
 {ADJ; yylval.fval=atof(yytext); return REAL;}
(“--”[a-z]*”\n”)|(“ ”|”\n”|”\t”)+
 {ADJ;}
.
 {ADJ; EM_error("illegal character");}

```

Lex 分成三个部分：

- 第一个部分是`%{ ... %}` 这是我们的一串 C 代码。有 `include`, 也会定义一些 `union`。`Tokens.h` 就会定义一些 `token` 的类型。因为我们的每个 `token` 有一个 `lexeme` (真正的字符串是什么)，对于 `token number` 的话，一个 `lexeme` 是真正要变成一个整数值的。`Identifier`、`if` 的话就是变成字符串，实数的话要变成 `double`。不同 `token` 的 `lexeme` 是不一样的，我们就用到了一个 `yylval` 这个 `union`, 里面可能是整数、字符串或 `double`。还有我们当前 `character` 的这个位置，一开始都是 1，每一次可以调整。可以算当前吃到的 `lexeme` 有多长。调整的话就是 `error` 的地方。
- 第二部分就是 `lex definition`, 可以定义一些宏，比如用 `digits` 替换掉`[0-9]+`
- 第三部分就是两个`%%`开始，把我们的 `regular expression` 一行行写出来，每个后面

都有一个 action。if 的话直接往前走；identifier，除了继续往前走以外，还要把 lexeme 算出来，其中 yytext 就是|到 T 之间的字符串，有了它以后我们就可以转化为 string；整数、浮点数同理。红色的都是 lex 提供的。

## 处理嵌套注释

最后我们来讲一下怎么处理嵌套的 comments。它在 lex 里头是可以支持的，这个支持就是说我们会有两个 start\_state。一个叫 initial，一个 comment，在 lex definition 声明了两个初始状态。正常情况下都是 initial 的 if，字符串。如果我们碰到了(\*，就说明我们进入了一个可能 comments 的情况，也就是我们进入了一个新的 start\_state (COMMENT)，在这个状态下碰到任意输入都无所谓，直接过滤掉。过滤掉以后，一直到\*)，它要回到 INITIAL 的情况。一开始我们想让它进入 INITIAL 状态，我们就用最后一行。也就是 BEGIN INITIAL; 吃掉的一个字符要还回去，也就是 yyless(1)。

## Use start state

```
%Start INITIAL COMMENT
%%
<INITIAL>if {ADJ; return IF;}
<INITIAL>[a-z]+ {ADJ; yyval.sval=String(yytext); return ID;}
<INITIAL>"(*" {ADJ; BEGIN COMMENT;}
<INITIAL>. {ADJ; EM_error("illegal character");}
<COMMENT>"*)" {ADJ; BEGIN INITIAL;}
<COMMENT>. {ADJ;}
.
.
{BEGIN INITIAL; yyless(1);}


```

4

这个东西还不能直接处理嵌套。嵌套是要回去自己想的。

2021/9/26

## 使用 lex 的好处

- **Hand-coded scanner (like any ordinary program):**
  - Programmer creates types, defines data & procedures, designs flow of control, implements in source language.
- **Lex-generated scanner:**
  - Programmer writes patterns
  - (Declarative, not procedural)
  - Lex implements flow of control
  - Much less hand-coding, but
  - code looks pretty alien, tricky to debugs

只需要写之前所提到的 pattern，只需要写少量代码。这种方式就叫做声明式的过程。真正的代码和控制流是由 lex 帮我们实现。好处就是代码量比较少，但是 pattern 写错的时候，debug 的技巧就和平时不一样了。

接下来我们介绍 tiger 语言，第二个 lab 交的时候需要加上 tiger.lax，说一些怎么处理的 comment,error handler, etc.

这本书编译器的前端相对比较简单。

Tiger 语言在 lex 中有

1. 保留字 (reserved word) : while, for , to, break, let, etc...
2. 标点符号: -, :, ; 等
3. Identifier, 开始必须是字母，是字母数字和下划线的组合
4. Comments: /\* \*/，允许嵌套
5. 整数常量，只支持非负整数，如果是负的就相当于一个表达式。
6. String 常量，引号之间的 printable character, space, escape sequences (转义串)。

#### escape sequences

- \n, \t: linefeed, tab
- \ddd: ddd is a 3-octal number
- \"", \\: ", \
- \f\_\_\_\_f: this sequence is ignored, where f\_\_\_\_f stands for sequence of one or more formatted characters including at least space, tab, or newline, form feed

#### • \^c:

- \^c:
  - ^@ (0, null, NUL, \0), mark the end of a string
  - ^G (7, bell, BEL, \a), emit a warning of some kind
  - ^H (8, backspace, BS, \b), may overprint the previous character
  - ^I (9, horizontal tab, HT, \t), moves the printing position right to the next tab stop
  - ^J (10, line feed, LF, \n), moves the print head down one line
  - ^K (11, vertical tab, VT, \v), vertical tabulation
  - ^L (12, form feed, FF, \f), to cause a printer to eject paper to the top of the next page, or a video terminal to clear the screen.
  - ^M (13, carriage return, CR, \r), moves the printing position to the start of the line, allowing overprinting.
  - ^Z (26, Control-Z, SUB, EOF). Acts as an end-of-file for the Windows text-mode file i/o.
  - ^[ (27, escape, ESC, \e (GCC only)). Introduces an escape sequence

最典型的是换行符和 tab 用\n 和\t 表示。\\ddd 是可以用 ASC2 的八进制表示来表示一个 ASC2 字符。在字符串里头出现引号的时候，我们就要使用\"的情况。在两个斜杠中呢出现的格式化的字符就是要在字符串中踢掉的。最后它还支持\\c 的情况，它实际上就代表了特殊的字符。

词法分析我们讲了 regular expression，我们写好 regular expression 的 declaration 给到 lex 就可以产生出相应的程序。

## 语法分析

下面我们来讲语法分析，语法分析显然比词法分析更重要。

首先，我们来简单看一下。做语法分析的时候，它们都有一些理论来支持（形式化语言）。有限状态的自动机，输入比较长的时候一定会重复的进入某些状态。像计数这种事情在 FA 中不太可能去做这件事情，因为 FA 里面是一张表是有限内存，只能数一个事先给定的长度。像要容易匹配的括号对是不能实现的。

- E.g., language of balanced parentheses is not regular:  $\{( )^i \mid i \geq 0\}$

像 comments 嵌套其实就是再做上面的这样的一个匹配。我们知道 lex 可以解决这个问题。因为 lex 本身它不是一些 regular expression。

所以我们要用 CFG 来表示我们程序的语法。

词法分析器：把程序原先的字符串变成了 token 串。

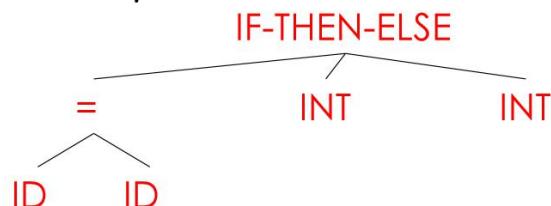
语法分析器：拿到 token 串以后生成 parse tree（语法树）。

if x = y then 1 else 2 fi

- Parser input

IF ID = ID THEN INT ELSE INT FI

- Parser output



Parser 的作用就是生成一个 parse tree，但是 input 是 token 串，有可能还是错的 token 串。所以 parser 要先把非法的 token 串排除掉，对于合法的就构造出一个 parse tree。我们构造语法就是通过 CFG。它来自于这个启发：我们的程序的构造都是递归构造出来的。递归的定义就是有一些开始的部分，比如一个 identifier 可以作为最基础的部分。两个表达式通过运算符连起来的还是表达式，(statement, expression)也是一个表达式。这种就是我们语言 construct，都是 recursive defined。

# 用上下文无关文法(CFG)来描述语法

Programming language constructs have recursive structure

An EXP is

$$\begin{aligned} \text{Exp} &\rightarrow \text{Exp Binop Exp} & (\text{OpExp}) \text{ or } \\ \text{Exp} &\rightarrow (\text{Stm}, \text{Exp}) & (\text{EseqExp}) \end{aligned}$$

...

Context-free grammars are a natural notation for this recursive structure

A CFG consists of

- A set of terminals  $T$
- A set of non-terminals  $N$
- A start symbol  $S$  (a non-terminal)
- A set of productions

Assuming  $X \in N$ , a production has the following forms

$$\begin{array}{ll} X \rightarrow \varepsilon & , \text{ or} \\ X \rightarrow Y_1 Y_2 \dots Y_n & \text{where } Y_i \subseteq N \cup T \end{array}$$

Terminal 是预先定义的, non-terminal 是需要定义的, non-terminal 中有一个 start symbol。接下来我们需要用 production 来定义 non-terminal。

在我们的讲义中 non-terminal 都是大写的, terminal 都是小写的。Start-symbol 就是第一个 production 最左边的。以下是一些简写的规范。

Write

$$X_1 \dots X_n \rightarrow^* Y_1 \dots Y_m$$

**Productions**

if

$$X_1 \dots X_n \rightarrow \dots \rightarrow \dots \rightarrow Y_1 \dots Y_m$$

-  $X \rightarrow \alpha$ , A left side,  $\alpha$  right side

- A set of production  $X \rightarrow \alpha_1, X \rightarrow \alpha_2, \dots, X \rightarrow \alpha_k$ ,

- with the same left hand side can be written as

-  $X \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_k$ ,

in 0 or more steps

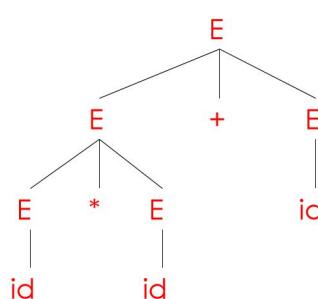
以下定义了一个简单的加和乘的表达式:

$$\begin{array}{l} E \rightarrow E * E \\ | \quad E + E \\ | \quad (E) \\ | \quad id \end{array}$$

在做语法分析的时候, 是给定一个 token 串构造出一个 parse tree。这个 production 在构造 parse tree 的时候起到的是 replacement rules, 也就是  $X$  出现的地方用  $Y_1 \dots Y_n$  替换掉,

( $X \rightarrow Y_1 \dots Y_n$ )。从 start symbol 开始, 我们反复使用替换规则。我们不断地把 non-terminal 换成另外的 non-terminal 和 terminal。最后我们希望产生出一串 terminal, 是正好和我们输入一致的, 这就是一个 parser 的过程。

$$\begin{array}{l} E \\ \rightarrow E + E \\ \rightarrow E * E + E \\ \rightarrow id * E + E \\ \rightarrow id * id + E \\ \rightarrow id * id + id \end{array}$$



在每一步怎么选择合适的 production 做替换，这就是语法分析中很重要的事情。我们从 start symbol 开始用不同的方式去做替换，最后我们替换出 sequence of terminal。

Let  $G$  be a context-free grammar with start symbol  $S$ . Then the language of  $G$  is:

$$\{ a_1 \dots a_n \mid S \rightarrow^* a_1 \dots a_n \text{ and every } a_i \text{ is a terminal} \}$$

CFG 是个语法，它可以推导出来的 terminal 串，就是这个语法构造出的语言。对于一个程序设计语言，如果我们用 cfg 构造出来以后，所有我们合法的程序就构成了我们的语言。

$L(G)$  is the language of CFG  $G$

Strings of balanced parentheses  $\{(^i)^i \mid i \geq 0\}$

Two grammars:

$$\begin{array}{lll} S \rightarrow (S) & \text{OR} & S \rightarrow (S) \\ S \rightarrow \epsilon & & | \quad \epsilon \end{array}$$

用这个 CFG 可以描述出这个括号匹配的集合，而 regular expression 不行。

The idea of a CFG is a big step. But:

- Membership in a language is "yes" or "no"
  - we also need parse tree of the input
- Must handle errors gracefully
- Need an implementation of CFG's (e.g., bison)

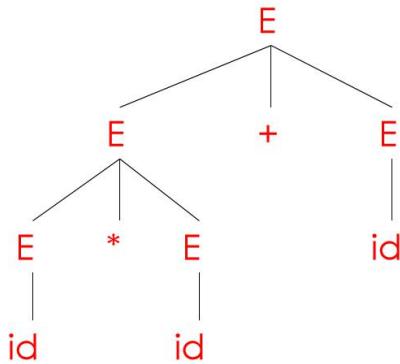
我们要做的事情

1. 判断这个语言是否属于某个集合
2. 构造 parse tree
3. 出错处理

实际上对于一个 language，我们可以用多个 grammar 或者多个 regular expression 来描述。在语法分析的过程中，因为我们最后是使用工具来构造它。工具对语法是比较敏感的，比如说我们有两种写法，可能一种工具能处理而另一种写法工具不能处理（工具会报错处理不了）。Lex 做词法分析的时候也敏感但是碰到的少一些。

从 Start symbol 开始通过一系列的替换，出现了一串 terminal，这个过程叫做 derivation（推导），它也对应了一个 parse tree。在推导的过程当中，一个 non-terminal 被右边替换掉了的话。这个 non-terminal 就成为父节点，而推导出的就变成了子节点。Parse tree 的叶子结点一定是 terminal。我们对于叶子结点按顺序来看，一定是我们最初的 input。Parse tree 还能反映出是先乘后加还是先加后乘。

实际上  $id * id + id$  这个表达式我们在不知道先验优先级的时候，我们是不知道乘和加的顺序。



但是这个 **parse tree** 我们就很明显的知道是先乘后加的。

在替换过程中选择先替换谁都不是什么问题，但是有可能的两种我们需要额外关注一下。中间的叶子结点在某一步上有很多的 **non-terminal**，一个选择是替换最左边的那个，另一个选择是替换最右边的哪一个。上例中就是最左推导。下图就是最右推导的例子，不断替换最右边的 **non-terminal**。两种情况只是 **parse tree** 出现的时间不一样，但最后的结果是一样的。

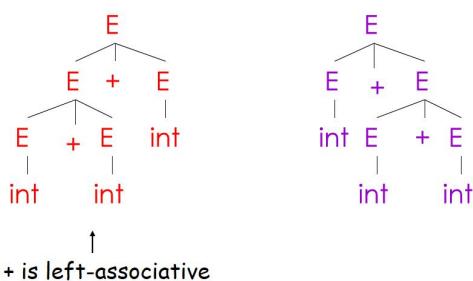
$$\begin{aligned}
 & E \\
 \rightarrow & E+E \\
 \rightarrow & E+id \\
 \rightarrow & E * E + id \\
 \rightarrow & E * id + id \\
 \rightarrow & id * id + id
 \end{aligned}$$

下面我们来讲一下歧义，一个语法构造完以后，可能会有不同的 **parse tree**。可能最后这些不同的 **production** 最后得到的序列是一样的。

## CFG 的歧义

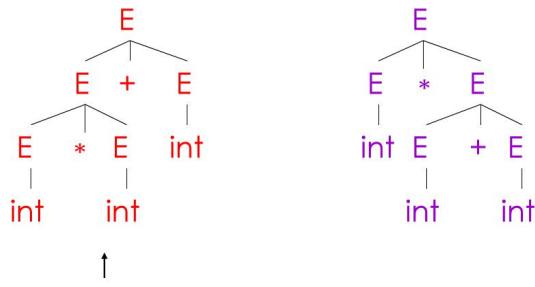
### Ambiguity. Example

This string has two parse trees



左图是最左推导出来的 **parse tree**，左边的加法是先执行的（左结合）。我们也可以像右图一样最右推导，加法就变成右结合了。实际上我们希望的肯定是加法符合左结合。

This string has two parse trees



\* has higher precedence than +

另外一个就是  $\text{int} * \text{int} + \text{int}$

左图就是先乘后加，而右图就是先加后乘。显然我们希望的是先乘后加，但是在原始输入的时候是没有优先级这个信息的。这就导致了我们用 CFG 去描述的时候出现了这两种不同的情况。

这种我们就称之为 **ambiguous**, 也就是有歧义的语法。

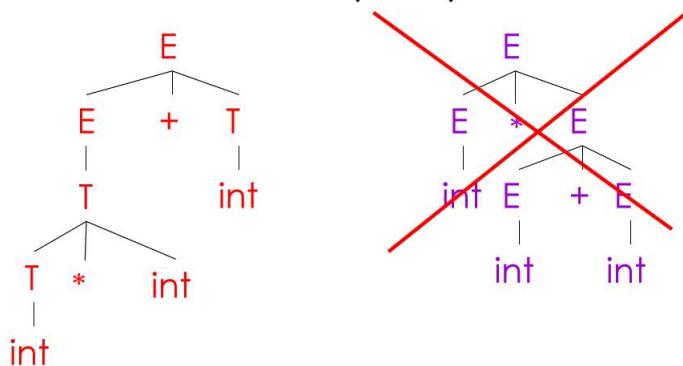
对于某些串来说，出现多个最左推导和最右推导。这个时候就是有歧义的，也就是 **ill-defined**。歧义性的消除需要我们事先的约定, eg: 先乘除后加减、加法都是左结合。

### Dealing with Ambiguity

- There are several ways to handle ambiguity
- Most direct method is to rewrite the grammar unambiguously
 
$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T^* \text{int} \mid \text{int} \mid (E) \end{aligned}$$
- Enforces precedence of \* over +
- Enforces left-associativity of + and \*

我们可以重写一个没有歧义的语法。我们修改原先的文法，引入一个 term 的东西。 $T^*$   $\text{int}$  而不是  $T*T$  就表现了乘法是左结合的。 $E+T$  而不是  $E+E$ ，也能保障加法是左结合的。 $E$  和  $T$  分开是现有  $T$  再有  $E$ ，这就保证乘法是有限的。有了这样的情况以后，我们就只能推导出来合法的情况了。

The  $\text{int} * \text{int} + \text{int}$  has only one parse tree now



## Ambiguity: The Dangling Else

- Consider the grammar

$$\begin{aligned} E \rightarrow & \text{if } E \text{ then } E \\ | & \text{if } E \text{ then } E \text{ else } E \\ | & \text{OTHER} \end{aligned}$$

- This grammar is also ambiguous

If-then-else 的情况，它可以是只有 if-then，也可以都有。

The expression

if  $E_1$  then if  $E_2$  then  $E_3$  else  $E_4$   
has two parse trees



Typically we want the second form

此时的 else 和哪组 if-then 就有两种情况了。

### The Dangling Else: A Fix

- else matches the closest unmatched then
- We can describe this in the grammar  
(distinguish between matched and unmatched "then")

$$\begin{aligned} E \rightarrow & \text{MIF} \quad /* \text{all then are matched */} \\ | & \text{UIF} \quad /* \text{some then are unmatched */} \\ \text{MIF} \rightarrow & \text{if } E \text{ then MIF else MIF} \\ | & \text{OTHER} \\ \text{UIF} \rightarrow & \text{if } E \text{ then E} \\ | & \text{if } E \text{ then MIF else UIF} \end{aligned}$$

- Describes the same set of strings

58

我们可以 rewrite 这个 CFG，要求 else 要 match 最近的 then。我们把 IF 分成两类（一类是 if-then-else，另一类是 if-then 的 if）。MIF 中要求里面都是 Match 的，UIF 只有可能在 else 中有 unmatch if。

如果碰到歧义了，我们就要去修改这个文法，但是修改这个文法没有什么通用的技巧，也不可能用一个自动的东西把一个有歧义文法翻译成一个没有歧义的文法。

在文法中有歧义事实上也是有好处的，我们发现 fix 后的无歧义的文法都比较复杂，而原始的有歧义的文法是比较简洁的。我们可以想想怎么就用原先的有歧义的文法去消除歧义。

我们可以给一些规则来消除歧义。

- Consider the grammar  $E \rightarrow E + E \mid int$
  - Ambiguous: two parse trees of  $int + int + int$
- 
- Left-associativity declaration: `%left +`
  - Consider the grammar  $E \rightarrow E + E \mid E * E \mid int$   
And the string  $int + int * int$
  - Precedence declarations: `%left +`  
`%left *`

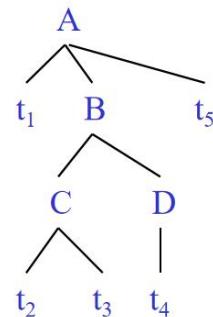
我们只要声明一下，加法是符合左结合律的，以及乘法的优先级比加法高。那在这个过程中自然消除掉了。`%`是一种声明，说明乘法和加法都是左结合的，在后面的优先级比前面高。我们就可以在工具中增加一些规则去消除这些歧义。

给定一个语法，我们要用 **CFG** 来描述，然后那个工具就会告诉我们是否合法并且工具会构造一个 **parse tree**。剩下的就是写程序比较容易解决的事情。

## 递归下降法构造 parse tree

Top-down 的方法就是从根节点开始，从左向右逐个消除 non-terminal。最后试图得到一个 parse tree，从叶子节点来看和我们的输入是一样的。

- Terminals are seen in order of appearance in the token stream:  
 $t_1 \ t_2 \ t_3 \ t_4 \ t_5$
- The parse tree is constructed
  - From the top
  - From left to right



我们写了一个加分和乘法右结合、乘法优先级比加法高的情况。

Consider the grammar

$$\begin{aligned} E &\rightarrow T + E \mid T \\ T &\rightarrow int \mid int * T \mid (E) \end{aligned}$$

Token stream is:  $int_5 * int_2$

Start with top-level non-terminal **E**

Try the rules for **E** in order

因为我们有两个 production，一开始我们就随便选一个，第二件事情是要把  $T_1$  给消除掉， $T_1$  又有 3 个 production。我们发现  $(E)$  是和 Int 不匹配的；我们发现 int 虽然和  $int_5$  匹配了，但是  $T+E$  需要有个加号，是不匹配的；我们选择  $int * T$  的话，我们发现 int

和 $*$ 对了，剩一个  $T$  要消除，然后我们再去看，发现  $\text{int}$  都对了。但是我们发现最后的  $T+E$  会多一个 $+E$  出来。我们只能在第一个推导  $E$  的时候选择推导出  $T$ 。也就是退回到第一次推导的时候，再继续尝试  $E \rightarrow T$ 。我们一个个去试， $T$  只能再选择  $\text{int} 5^* T$ 。然后  $T$  再推导出  $\text{int}2$ 。

这种我们叫做递归下降法（recursive descent parsing），也就是从根节点开始一个个去试，一旦发现 unmatched，就回退（有可能回退多级）。这就是带回溯的递归。要把中间状态记住。这个东西手工实现是比较容易的，但是不是总能很好的工作。

## 2021/9/28

上节课介绍了 CFG，以及怎么用 CFG 描述我们的文法。有了 CFG 可以定义一个 CFL，可以定义出一系列的 terminal 串来属于这个语言，下面给定一个 terminal，我们怎么判定在不在这个语言中，方法分两种：

1. top-down 方法，其中又分为递归下降法（实际上就是，从一个 start symbol 开始，每一个 production 都去尝试一下，见上面  $\text{int}5 * \text{int}2$  的例子，如果不符就有个回退的过程。）

从 start symbol 开始对所有可能性做尝试，从某个时间点来看，前面  $t_1 \dots t_k$  都是 terminal 了，而  $A$  是第一个 non-terminal 节点，把  $A$  的所有 production 都拿过来试一下。

## 左递归和消除左递归

Recursive-decent parser 是比较容易实现的，也开始流行使用 ANTLR 方法。Top-down 方法比较害怕的是一个 production 情况： $S \rightarrow S\alpha$ ，这样它就会变成死循环，一直在尝试消除  $S$ 。这种情况我们称之为左递归文法。不仅仅是  $S \rightarrow S\alpha$ ，还包括通过一系列的推导后，间接地变成这样  $S \rightarrow^+ S\alpha$ ，也属于左递归文法（left-recursive grammar）。

### Elimination of Left Recursion

- Consider the left-recursive grammar

$$S \rightarrow S\alpha \mid \beta$$

- $S$  generates all strings starting with a  $\beta$  and followed by a number of  $\alpha$

- Can rewrite using right-recursion

$$S \rightarrow \beta S'$$

$$S' \rightarrow \alpha S' \mid \epsilon$$

### More Elimination of Left-Recursion

- In general

$$S \rightarrow S\alpha_1 \mid \dots \mid S\alpha_n \mid \beta_1 \mid \dots \mid \beta_m$$

- All strings derived from  $S$  start with one of  $\beta_1, \dots, \beta_m$  and continue with several instances of  $\alpha_1, \dots, \alpha_n$

- Rewrite as

$$S \rightarrow \beta_1 S' \mid \dots \mid \beta_m S'$$

$$S' \rightarrow \alpha_1 S' \mid \dots \mid \alpha_n S' \mid \epsilon$$

我们可以消除左递归，也就是要重新写一下这个文法。通常情况下，我们把不属于左递归的拿出来和  $S'$  连接，然后  $S'$  中放入左递归即可。可以自动地消除左递归。在实际的情况下，有很多的文法是不需要回退的。

比如  $E$  和  $T$  在  $T+E$  中二选一的情况下，因为我们有一串 INPUT，我们把 input 和  $E$  的推导放在一起，实际上是可以预测选择的。所以，我们现在用的比较多的是 predictive parser。我们在做 basic 解释器的时候，会碰到 IF,GOTO。这是整句 statement 的第一个 token，拿到以后整句话的语义就清楚了。

## Predictive Parsers

---

- Like recursive-descent but parser can "predict" which production to use
  - By looking at the next few tokens
  - No backtracking
- Predictive parsers accept LL( $k$ ) grammars
  - L means "left-to-right" scan of input
  - L means "leftmost derivation"
  - $k$  means "predict based on  $k$  tokens of lookahead"
- In practice, LL(1) is used

## LL(1)

一边 topdown 地消除 non-terminal，一边扫描我们的 input。每次我们都是在消除最左边的 non-terminal (left-most derivation)，以及从左向右扫描 Input terminals (left-to-right)。像一般的语言，看一个 token 就可以了，如看到了 IF,GOTO,LET 等。根据 Input token 来决定我们选择哪个 production。往前看几个决定了  $k$  到底是几。

### LL(1) language. Example

---

```
S → if E then S else S L → end
S → begin S L L → ; S L
S → print E E → num = num
enum token {IF,THEN,ELSE,BEGIN,END,PRINT,SEMI,NUM,EQ}
extern enum token getToken(void);
enum token tok;
void advance() {tok=getToken();}
void eat(enum token t) { if (tok==t) advance(); else error();}
```

定义了 begin s1; s2; sn end 和 begin s end。这样我们可以先把 token 定义出来。词法分析就是 getToken，拿出下一个扫描到的 token。

```

S → if E then S else S | begin S L | print E
void S(void) {
 switch(tok) {
 case IF:
 eat(IF); E(); eat(THEN); S(); eat(ELSE); S(); break;
 case BEGIN:
 eat(BEGIN); S(); L(); break;
 case PRINT:
 eat(PRINT); E(); break;
 default:error();
 }
}

```

我们定义一个 `S` 函数，也就是我们想看到 `if / begin/ print` 作为合法的开头。看到了以后，我们就 `eat` 掉，让 `token` 前进一格。比如看到 `if` 的时候要求中间一定要看到 `then` 和 `else`。

```

L → end | ; S L
void L(void) {
 switch(tok) {
 case END:
 eat(END); break;
 case SEMI:
 eat(SEMI); S(); L(); break;
 default:error();
 }
}
E → num = num
void E(void) {
 eat(NUM); eat(EQ); eat(NUM);
}

```

我们可以进一步构造 `L` 和 `E`，这两个都比较好写。

`LL(1)` 语言处理 `statement` 比较方便，也就是 `token` 开头看到了它就可以知道了整个 `statement` 长什么样子。在 `basic` 中，`expression` 我们用的是中缀变后缀的方法，栈上的东西根据操作符的优先级进栈/出栈。

### Left-Factoring Example

---

- Recall the grammar
 
$$\begin{aligned} E &\rightarrow T + E \mid T \\ T &\rightarrow \text{int} \mid \text{int}^* T \mid ( E ) \end{aligned}$$
- Factor out common prefixes of productions
 
$$\begin{aligned} E &\rightarrow TX \\ X &\rightarrow + E \mid \varepsilon \\ T &\rightarrow ( E ) \mid \text{int} Y \\ Y &\rightarrow ^* T \mid \varepsilon \end{aligned}$$

我们定义了右结合的乘法和加法，还有括号表达式。在这个里头我们要消除 `E` 的时

候，我们没有办法确定是 int 还是 int \*，这时候我们就来一个左分解，相当于是提取公因式，我们把 E 改成 TX，公共部分 T 被提取出来了。这样原本 LL(1) 只往前看一步解决不了的文法就可以处理了。

## LL(1) 的 Parsing Table

### LL(1) Parsing Table Example

---

- Left-factored grammar

$$\begin{array}{l} E \rightarrow TX \\ T \rightarrow (E) \mid \text{int } Y \end{array} \quad \begin{array}{l} X \rightarrow +E \mid \varepsilon \\ Y \rightarrow *T \mid \varepsilon \end{array}$$

- The LL(1) parsing table:

|   | int   | *  | +             | (   | )             | \$            |
|---|-------|----|---------------|-----|---------------|---------------|
| T | int Y |    |               | (E) |               |               |
| E | TX    |    |               | TX  |               |               |
| X |       |    | +E            |     | $\varepsilon$ | $\varepsilon$ |
| Y |       | *T | $\varepsilon$ |     | $\varepsilon$ | $\varepsilon$ |

- \$ means end-of-input

- Sometime a new production is introduced  $S' \rightarrow S \$$

左边都是 non-terminal，上面是当前的 input 是什么。我们可以根据文法构造出这张表，来替换。空格就是出错的情况。上面的 \$ 符号就是我们在 token 串的最后加一个符号代表了 token 结束。换句话说，当我们消除 X 且发现 token 串已经到最后的时候，我们直接消除为  $\varepsilon$  即可。

### Using Parsing Tables

---

- Method similar to recursive descent, except
  - For each non-terminal S
  - We look at the next token a
  - And choose the production shown at [S,a]
- We use a stack to keep track of pending non-terminals
- We reject when we encounter an error state
- We accept when we encounter end-of-input

我们给定一个文法，有可能构造出这个表，有了这张表以后就可以查表做 parser 了，也是可以用一个自动工具来实现这件事情。我们去查询 S,a 对应的表的产生式，可以把 S 替换掉。

|   | int   | *   | +          | (     | )          | \$         |
|---|-------|-----|------------|-------|------------|------------|
| T | int Y |     |            | ( E ) |            |            |
| E | TX    |     |            | TX    |            |            |
| X |       |     | + E        |       | $\epsilon$ | $\epsilon$ |
| Y |       | * T | $\epsilon$ |       | $\epsilon$ | $\epsilon$ |

| Stack      | Input        | Action     |
|------------|--------------|------------|
| E \$       | int * int \$ | TX         |
| TX \$      | int * int \$ | int Y      |
| int Y X \$ | int * int \$ | terminal   |
| Y X \$     | * int \$     | * T        |
| * TX \$    | * int \$     | terminal   |
| TX \$      | int \$       | int Y      |
| int Y X \$ | int \$       | terminal   |
| Y X \$     | \$           | $\epsilon$ |
| X \$       | \$           | $\epsilon$ |
| \$         | \$           | ACCEPT     |

查表消除的例子如上图所示，栈顶和 Input match 后，就可以直接消掉了。

## LL(1) Parsing 算法

### LL(1) Parsing Algorithm

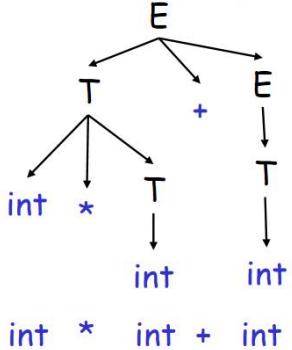
```

initialize stack = <S $> and next (pointer to tokens)
repeat
 case stack of
 <X, rest> : if T[X,*next] = Y1...Yn
 then stack ← <Y1... Yn rest>;
 else error ();
 <t, rest> : if t == *next ++
 then stack ← <rest>;
 else error ();
 until stack == <>

```

算法的形式化定义如上图所示：如果栈顶是一个 nonterminal，我们拿这个来查表，如果表里有这个东西，我们就用表里的元素来替换这个东西。如果栈顶是一个 terminal，如果匹配了，那么 next 指针就往后移动一个。一直到栈为空或者中间报错。这里要求表上的替换是唯一确定的，如果表里有多个 production，那么我们就不知道要选谁了。

- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal



- The leaves at any point form a string  $\beta A \gamma$ 
  - $\beta$  contains only terminals
  - The input string is  $\beta b \delta$
  - The prefix  $\beta$  matches
  - The next token is  $b$

21

## 构造 LL(1) Parsing Table

下面我们来说怎么来构造这张表, LL(1)替换到某个时刻, parsing tree 的叶子结点会呈现出  $\beta A \gamma$ , 最左边是一串终结符  $\beta$ , 然后出现一个非终结符  $A$ , 然后跟一串 symbol (可能是终结符也有可能是非终结符), 我们用  $\gamma$  表示。这样子如果这个推导最后是 match 的话, 我们假设接下来输入的第一个 token 为  $b$ , 那么整个需要 parsing 的 token 串就长成  $\beta b \delta$ 。

即:

树上的叶子节点:  $\beta A \gamma$

正在处理的字符串:  $\beta b \delta$

在这个情况下, 我们拿到了一个  $A$  和一个  $b$ , 我们希望找到一个 production  $A \rightarrow \alpha$  来把  $A$  替换掉。替换出来的结果一定是希望最后换到的东西的开头是  $b$ 。

在能够正确识别的情况下, 就等于我们有这么一个推导  $S \rightarrow^* \beta A \gamma$ , 且我们看到了  $b$ , 我们最后一定要推导出  $\beta b \delta$ 。存在两种可能:

1.  $b$  确实是通过  $A$  推导出来的, 那么我们可以使用任何一个能够推导出第一个字符是  $b$  的推导  $A \rightarrow \alpha$ , 这种我们就称之为  $b \in First(A)$ 。
2.  $b$  不是通过  $A$  推导出来的, 换句话说,  $A \rightarrow \epsilon$ , 而  $b$  是通过  $A$  后的  $\gamma$  推导出来的。

这意味着在最终的推导中,  $b$  会出现在  $A$  之后, 也就是  $S \rightarrow^* \beta A b \omega$ , 在这种情况下, 我们就说  $b \in Follow(A)$ 。并且在这种情况下, 我们会选择任意一个最终能使得  $A$  扩展成  $\epsilon$  的推

导  $A \rightarrow \alpha$ 。那么我们就有  $\varepsilon \in First(A)$ 。

## First 集合和 Follow 集合

所以我们定义了 first 和 follow:

如此，我们就定义了这两个集合：

First 集合

$$First(X) = \{b \mid X \rightarrow^* b\alpha\} \cup \{\varepsilon \mid X \rightarrow^* \varepsilon\}$$

并且终结符的 First 就是它本身：  $First(b) = \{b\}$

### Computing First Sets

Definition  $First(X) = \{b \mid X \rightarrow^* b\alpha\} \cup \{\varepsilon \mid X \rightarrow^* \varepsilon\}$

1.  $First(b) = \{b\}$

2. For all productions  $X \rightarrow A_1 \dots A_n$

- Add  $First(A_1) - \{\varepsilon\}$  to  $First(X)$ . Stop if  $\varepsilon \notin First(A_1)$
- Add  $First(A_2) - \{\varepsilon\}$  to  $First(X)$ . Stop if  $\varepsilon \notin First(A_2)$
- ...
- Add  $First(A_n) - \{\varepsilon\}$  to  $First(X)$ . Stop if  $\varepsilon \notin First(A_n)$
- Add  $\varepsilon$  to  $First(X)$

First 一个 symbol，也就是从这个 symbol 出发可以推出一系列东西，第一个元素就叫做  $First(X)$ 。Terminal 的 first 就是它自己。

左边的 X 就是我们要计算的 Symbol，我们就来看第一个  $A_1$  的 first 都是 X 的 first，除了  $\varepsilon$  的情况。如果  $\varepsilon$  不属于  $A_1$  的话，此时这个事情就结束了  $First(A_1) = First(X)$ ，但是如果  $A_1$  被  $\varepsilon$  消除掉的情况， $A_2$  就会起作用了。如果中间都停不了， $A_1$  到  $A_n$  都是  $\varepsilon$ ，那么我们认为  $\varepsilon$  是  $First(X)$ .

举个例子来说：

• Recall the grammar

$$\begin{array}{ll} E \rightarrow TX & X \rightarrow +E \mid \varepsilon \\ T \rightarrow (E) \mid \text{int } Y & Y \rightarrow *T \mid \varepsilon \end{array}$$

• First sets

$$\begin{array}{ll} First(()) = \{()\} & First(()) = \{()\} \\ First(\text{int}) = \{\text{int}\} & First(+) = \{+\} \\ First(*) = \{* \} & \\ First(T) = \{\text{int}, ()\} & \\ First(X) = \{+, \varepsilon\} & First(Y) = \{*, \varepsilon\} \\ First(E) = \{\text{int}, ()\} & \end{array}$$

第二个我们要算 follow，每一个 Symbol 的 first 是从这个 symbol 开始推导出一串东西，这个东西是以这个东西开始的。

而 follow 是从 start symbol 开始， $b$  跟在  $X$  后面，计算 follow 的时候，需要计算出所有的 first，然后再加一个\$到其之后。

**Definition**  $\text{Follow}(X) = \{ b \mid S \xrightarrow{*} \beta X b \delta \}$

1. Compute the First sets for all non-terminals first
2. Add \$ to Follow(S) (if S is the start non-terminal)
3. For all productions  $Y \rightarrow \dots X A_1 \dots A_n$ 
  - Add  $\text{First}(A_1) - \{\epsilon\}$  to Follow(X). Stop if  $\epsilon \notin \text{First}(A_1)$
  - Add  $\text{First}(A_2) - \{\epsilon\}$  to Follow(X). Stop if  $\epsilon \notin \text{First}(A_2)$
  - ...
  - Add  $\text{First}(A_n) - \{\epsilon\}$  to Follow(X). Stop if  $\epsilon \notin \text{First}(A_n)$
  - Add Follow(Y) to Follow(X)

我们找到一个 productionX， $X$  后面的这一串东西和之前的 first 计算是一样的，如果  $A_1$  能被消掉，那么  $A_2$  的 first 就 followX，如果  $X$  是最后一个（ $A_1$  到  $A_n$  都被消掉的情况），那么  $Y$  的 follow 就是  $X$  的 follow。

**Recall the grammar**

$$\begin{array}{ll} E \rightarrow TX & X \rightarrow +E \mid \epsilon \\ T \rightarrow (E) \mid \text{int } Y & Y \rightarrow *T \mid \epsilon \end{array}$$

**Follow sets**

$$\begin{array}{ll} \text{Follow}(+) = \{ \text{int}, () \} & \text{Follow}(*) = \{ \text{int}, () \} \\ \text{Follow}(()) = \{ \text{int}, () \} & \text{Follow}(E) = \{ \}, \$ \} \\ \text{Follow}(X) = \{ \$, () \} & \text{Follow}(T) = \{ +, (), \$ \} \\ \text{Follow}(()) = \{ +, (), \$ \} & \text{Follow}(Y) = \{ +, (), \$ \} \\ \text{Follow}(\text{int}) = \{ *, +, (), \$ \} & \end{array}$$

+和\*，因为我们  $E$  和  $T$  的 first 我们都有，所以就加入(, int。而左括号后面跟了  $E$ ，所以就加入  $E$  的 first。而  $E$  的 follow，包括有括号， $E$  还出现在  $X \rightarrow +E$ ，我们不知道  $\text{follow}_X$  是谁，但是  $E$  又是 start symbol，我们会把 dollar 加进去，但是我们不知道  $X$  的 follow 是谁，暂时我们只能算成这样，等一下  $X$  算完还要迭代回来。 $X$  在这， $X$  的 follow 就是  $E$  的 follow，而  $E$  的 follow 包含了  $X$  的 follow，这样  $X$  和  $E$  的 follow 就可以唯一确定了。 $T$  出现在  $E \rightarrow TX$  和  $Y \rightarrow *T$ ， $Y$  的 follow 我们还不知道， $\text{First}(X)$  是 + 和  $\epsilon$ ，因为  $\epsilon$  代表了  $X$  被消掉了，那么我们还要加入  $\text{Follow}(E)$ ，还要加入 \$ 进来。接下来我们看有括号，因为其在最右边，所以就是  $\text{Follow}(T)$  和 \$。Int 的话首先要加入  $\text{First}(Y)$ ，因为  $\text{First}(Y)$  中包含  $\epsilon$ ，所以我们还要加入  $\text{follow}(T)$

## Constructing LL(1) Parsing Tables

- Construct a parsing table  $T$  for CFG  $G$
- For each production  $A \rightarrow \alpha$  in  $G$  do:
  - For each terminal  $b \in \text{First}(\alpha)$  do
    - $T[A, b] = \alpha$
  - If  $\alpha \rightarrow^* \epsilon$ , for each  $b \in \text{Follow}(A)$  do
    - $T[A, b] = \alpha$
  - If  $\alpha \rightarrow^* \epsilon$  and  $\$ \in \text{Follow}(A)$  do
    - $T[A, \$] = \alpha$

根据 CFG，我们设立这些规则来构建这张表。如果  $A \rightarrow^* \epsilon$ ，那么我们可以加入  $T[A, \text{Follow}(A)] = \alpha$

- Recall the grammar

$$\begin{array}{ll} E \rightarrow T X & X \rightarrow + E \mid \epsilon \\ T \rightarrow ( E ) \mid \text{int } Y & Y \rightarrow * T \mid \epsilon \end{array}$$

- Where in the line of  $Y$  we put  $Y \rightarrow * T$ ?

- In the columns of  $\text{First}(^* T) = \{ *\}$

|   | int | *     | +          | ( | )          | \$         |
|---|-----|-------|------------|---|------------|------------|
| Y |     | $* T$ | $\epsilon$ |   | $\epsilon$ | $\epsilon$ |

- Where in the line of  $Y$  we put  $Y \rightarrow \epsilon$ ?

- In the columns of  $\text{Follow}(Y) = \{ \$, +, ) \}$

我们就一个个 production 看，然后填到表里去。有了这个表以后，刚才的推导我们就有了一张表。

## 出错处理

我们讲了一种通用的方法来构造表，根据查表用产生式来替换。还有一种就是我们最早讲的手写代码。

接下来以手写代码为例来讲出错处理，我们希望尽量恢复报错继续做下去。这时候就有几种办法：

1. 假装插入一个 token

## Error Recovery (Insert a Token)

---

```
void T(void)
{
 switch(tok) {
 case INT:
 eat(INT); Y(); break;
 case LPAREN:
 eat(LPAREN); E(); eat(RPAREN); break;
 default:
 /* pretend that an INT token was here */
 printf("expected an integer or left-paren") ; Y();
 }
}
```

- **Dangerous to introduce infinite loop**

下面这个东西就用  $Y$  去分析，如果我们碰到错误就往里插入的时候，可能它就停不下来导致死循环。

2. 删一些 token

## Error Recovery (delete tokens)

---

```
int T_follow[] = {+,), $,-1(相当于数组的结束符)};
void T(void)
{
 switch(tok) {
 case INT:
 eat(INT); Y(); break;
 case LPAREN:
 eat(LPAREN); E(); eat(RPAREN); break;
 default:
 printf("expected +, right-paren, or $");
 skipTo(T_follow);
 }
}
```

因为我们是想分析出一个  $T$ ，我们分析出这里肯定不是一个  $Y$ ，但是我们希望一个  $Y$ ，我们找到一个  $T$  的 follow 停下来，之前的都认为是  $T$ 。把前面的这一段都当成是  $Y$  处理掉。

## Notes on LL(1) Parsing Tables

---

- If any entry is multiply defined then  $G$  is not LL(1)
  - If  $G$  is ambiguous
  - If  $G$  is left recursive
  - If  $G$  is not left-factored
  - And in other cases as well
- Most programming language grammars are not LL(1)
- There are tools that build LL(1) tables

这个就是我们的 LL(1)parsing table 的情况。并不是所有文法都可以构造出这张表来，比如有歧义，比如不能左递归。对于大部分语法来说，就不是 LL(1)，比如表达式就是一个左递归的。所以 LL（自顶向下方法）是一个比较简单的语法分析方法，具有比较多的限制。

## LR Parser

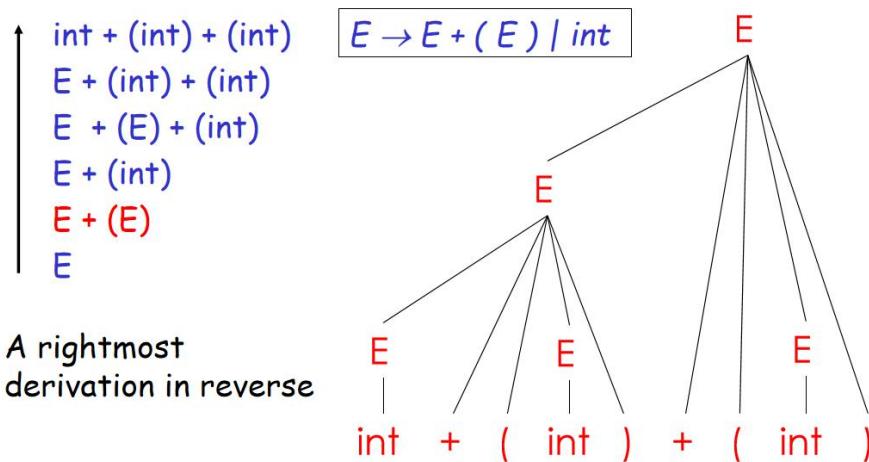
我们接下来介绍 LR parser，刚才的方法是自顶向下的方法从 start symbol 开始逐步 expand 我们的叶子结点。而我们的 LR parse 是从叶子节点开始逐步地向上伸展来构造出一个根节点。L (token 串从左到右扫描) R (每次消除右边的)，好处就是可以处理左递归也不需要左分解。我们考虑  $E \rightarrow E+E \mid \text{int}$ , 怎么构造出  $\text{int}+(\text{int})+(\text{int})$  呢？

- LR parsing reduces a string to the start symbol by inverting productions:

```
str ← input string of terminals
repeat
 - Identify β in str such that $A \rightarrow \beta$ is a production
 (i.e., str = $\alpha\beta\gamma$)
 - Replace β by A in str (i.e., str becomes $\alpha A \gamma$)
until str = S
```

我们不断地从 input string 中，不断地找出  $\beta$  使得  $A \rightarrow \beta$ ，然后我们把  $\beta$  替换成  $A$ 。一直换掉最后变成了一个 start symbol 就结束了。

## A Bottom-up Parse in Detail (6)



我们把这个过程反过来就是最右推导的过程，整个过程就是最右推导的逆过程。

**2021/10/8**

上节课讲了 LL parser，如果说用递归下降的方法，它的适应性强一些，后来用的 predictive LL parser，根据当前状态和当前 token，马上做出一个决定用哪个 production 来替换 non-terminal，我们看到的东西比较少，所以 LL(1) 的能力比较弱。

上节课还提到了 LR parser（L 指的是从左到右扫描 token 串，R 指的是 right-most derivation）。为什么是 right-most 呢，上节课我们提出了如下的这个 grammar。

$$E \rightarrow E + (E) \mid \text{int}$$

- Why is this not LL(1)?

- Consider the string:  $\text{int} + (\text{int}) + (\text{int})$

这是一个左递归的文法，LL 是不能用的。对于 LL，是自顶向下看到一个 non-terminal，把 production 的左边替换成右边。而 LR 是 bottom-up，把看到的一串 terminal 串，规约成 production 的左边。

```

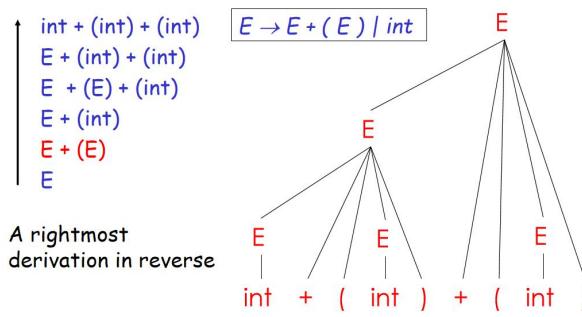
str ← input string of terminals
repeat
 - Identify β in str such that A → β is a production
 (i.e., str = αβγ)
 - Replace β by A in str (i.e., str becomes αAγ)
until str = S

```

我们把 string 中的 β 换成这个 A，那么我们的 string 就从  $\alpha \beta \gamma$  变成了  $\alpha A \gamma$ ，直到这个 string 变成了我们的 start symbol。

我们就要按照刚才的说法，把输入串  $\text{int} + (\text{int}) + (\text{int})$  规约。

### A Bottom-up Parse in Detail (6)



- LR parser 其实就是记录了最右推导的逆过程，所以在一个 string 中要 identify 一个  $\beta$  然后替换成  $\alpha$ ，在这个过程中有一个性质，就是  $\gamma$  本身都是 terminal，而  $\alpha$  就不一定是 non-terminal 和 terminal 都有，因为我们把  $\beta$  替换成  $\alpha$  的过程是最右推导的逆过程，所以  $\alpha$  是最右推导的第一个 non-terminal，所以  $\gamma$  肯定都是 terminal。

因此我们在刚才算法中，有一个 string 这个东西，一开始都是 terminal，过程中逐步地变成了 non-terminal。因此，我们把这个 string 分成两部分，一部分是  $\gamma$  这部分全部是 terminal，剩下的左边的部分就可能都存在。

- The dividing point is marked by a **I**
  - The **I** is not part of the string
- Initially, all input is unexamined: **I** $x_1x_2\dots x_n$

就引入了这个 split 符号。

## Shift 和 Reduce

Shift 就是把右边 string 的 symbol 放到左边来，而 reduce 就是把  $\beta$  变成  $\alpha$  的过程。

### Shift

*Shift: Move I one place to the right*  
- Shifts a terminal to the left string

$$E + (\text{I } \text{int}) \Rightarrow E + (\text{int I } )$$

### Reduce

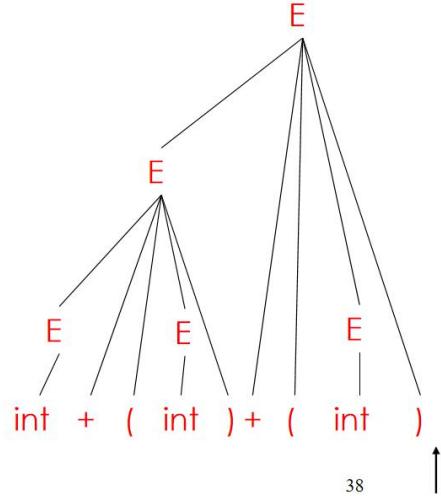
*Reduce: Apply an inverse production at the right end of the left string*  
- If  $E \rightarrow E + (E)$  is a production, then

$$E + (E + (E) \text{I}) \Rightarrow E + (E \text{I})$$

一开始是一个 input string，不断去做 shift 和 reduce，逐渐替换为一个 start symbol，parser 任务就结束了。

## Shift-Reduce Example

|                                                 |                                 |
|-------------------------------------------------|---------------------------------|
| $I \text{ int} + (\text{int}) + (\text{int})\$$ | shift                           |
| $\text{int } I + (\text{int}) + (\text{int})\$$ | red. $E \rightarrow \text{int}$ |
| $E \text{ I} + (\text{int}) + (\text{int})\$$   | shift 3 times                   |
| $E + (\text{int } I) + (\text{int})\$$          | red. $E \rightarrow \text{int}$ |
| $E + (E \text{ I}) + (\text{int})\$$            | shift                           |
| $E + (E) \text{ I} + (\text{int})\$$            | red. $E \rightarrow E + (E)$    |
| $E \text{ I} + (\text{int})\$$                  | shift 3 times                   |
| $E + (\text{int } I)\$$                         | red. $E \rightarrow \text{int}$ |
| $E + (E \text{ I})\$$                           | shift                           |
| $E + (E) \text{ I}\$$                           | red. $E \rightarrow E + (E)$    |
| $E \text{ I}\$$                                 | accept                          |

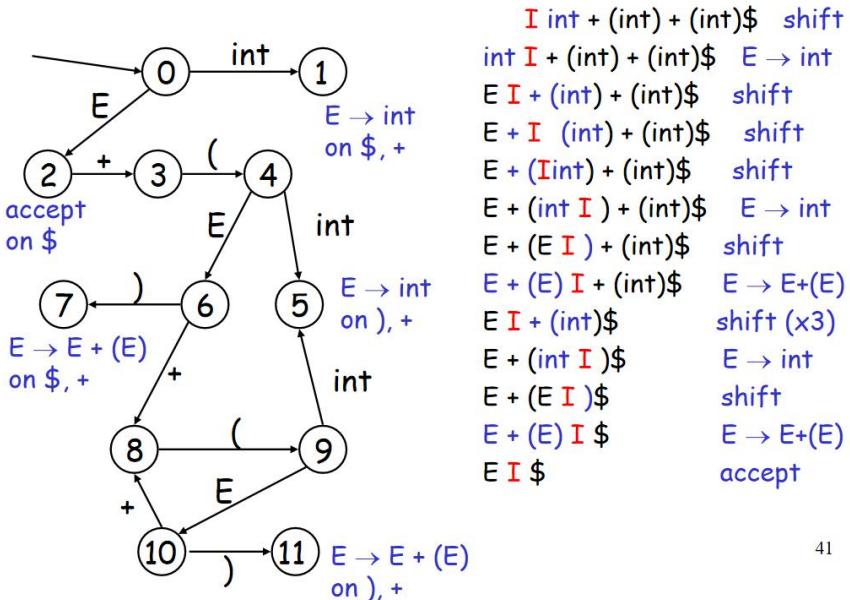


Shift 的过程就是不断向右移动竖线，然后左边 string 就会发生变化，我们左边的实现就可以使用栈来实现，我们总是把栈顶的东西换掉。这也是符合之前的  $\alpha\beta\gamma \rightarrow \alpha A\gamma$ ，其中的  $\beta$  肯定是我们栈顶上的东西，因为要符合最右推导。

## LR(1)

一个东西进栈，要么让更多的东西进栈（shift），要么对栈顶做 reduce。到底怎么决定呢？我们可以根据栈上的内容来决定。左边的这些 non-terminal 和 terminal 符号串，怎么来决定呢，在我们的例子中，我们可以构造出一个 DFA。这个 DFA，有状态的 transition，9 这个状态看到了 E 以后，input 就是 string 中左边的东西。

### LR(1) Parsing. An Example



不一样的就是在 1,2,5,7,11，这几个状态后头跟了一个东西，这个东西代表到了这个状

态之后要做一些 reduce 的 action。比如 9 读到了一个 E 到了 10, 10 可能是一个终结状态, 当前为  $E+(E|)$ , 我们从 0 出发走到了 6, 然后我们 shift 一个看到了右括号, 我们再重新走一遍, 走到了 7, 到了 7 这个状态, 栈上一定有  $E+(E)$  这个东西, 有可能要做 reduce 操作。使用  $E \rightarrow E+(E)$  是有条件的, 要么后面 ( $\gamma$  的第一个) 必须跟了 \$ 或者 +, 才能使用这个 production 来做 reduce。

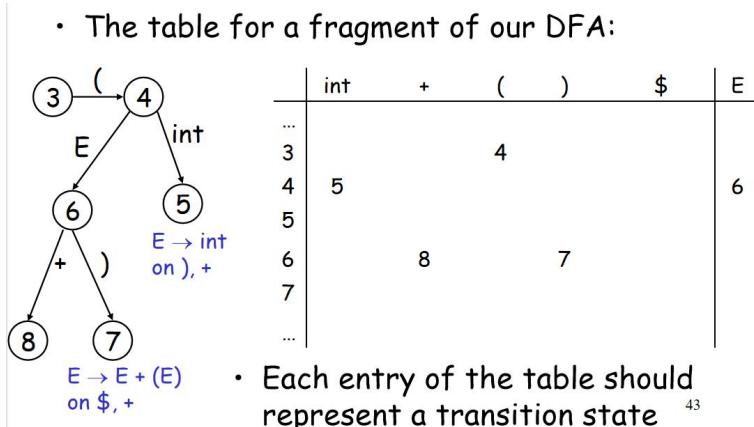
这个就叫做 LR(1) parser。

一开始栈上为空, 就从 start state 从 0 开始, 停止在 0 这个状态。然后我们看到了  $\gamma$  的第一个是 int, 那么 0 有转移到 int 跑到 1 状态的 transition, 所以可以做 shift。把 int 吃掉以后就听到了 1 这里, int 可以变成 E 的条件是后面有 \$ 或者 +, 确实是这样, 我们就可以做 reduce 了。所以根据栈上的状态和  $\gamma$  的第一个字符是什么, 就可以判断是做 shift 还是 reduce 还是 error。在 2 这个地方既可以做 reduce 也可以做 shift, 如果后面跟的是 +, 就可以做 shift, 如果是 \$ 就可以做 reduce。走到 3 以后, 有一个左括号 transition, 确实 match 的情况下, 我们就可以 shift 一个, 走到了 4.

## Representing the DFA

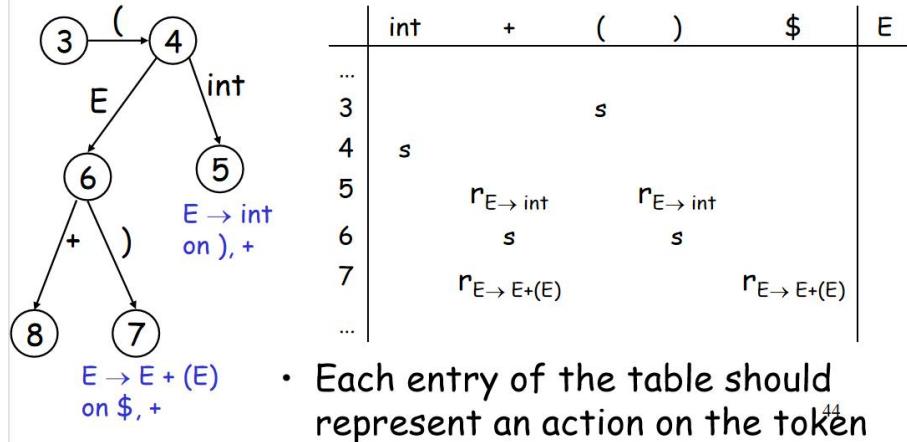
- Parsers represent the DFA as a 2D table
  - Recall table-driven lexical analysis
- Lines correspond to DFA states
- Columns correspond to terminals and non-terminals
- What should an entry of a table represent for ?

DFA 每次都要重走, 很麻烦。DFA 要实现的时候就是一个 2 维的表, 行代表状态, 列代表 symbol。



状态机对应了这么一张表, 代表了在 3 这个状态看到左括号进入 4 这个状态。只有这张表就只有状态转移, 而我们还需要把 reduce 的动作告诉我们, 所以还需要再来一张 action 表。

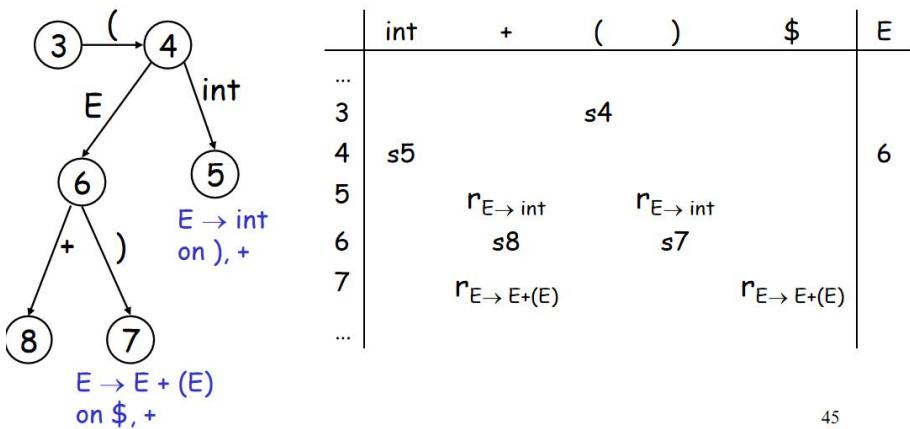
- The table for a fragment of our DFA:



也就是 5 看到+和)要做 reduce，所以上面这张 action 表是 DFA 表的扩展。

我们可以把这两张表合并一下，成为一张 combine 表。

- The table for a fragment of our DFA:



- Typically columns are split into:

- Those for terminals: action table
- Those for non-terminals: goto table

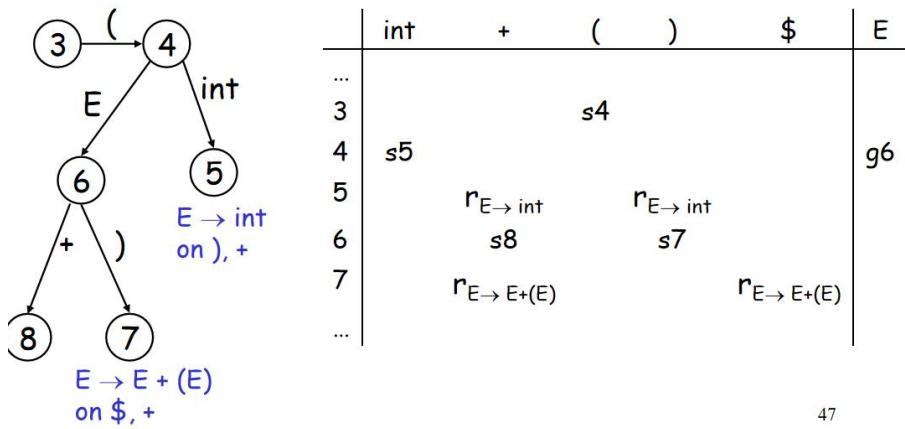
Goto  
table

| Action table | int                  | +                  | ( )                | \$                   | E |
|--------------|----------------------|--------------------|--------------------|----------------------|---|
| ...          |                      |                    |                    |                      |   |
| 3            |                      |                    |                    | s4                   |   |
| 4            | s5                   |                    |                    |                      | 6 |
| 5            |                      | r <sub>E→int</sub> | r <sub>E→int</sub> |                      |   |
| 6            |                      | s8                 |                    | s7                   |   |
| 7            | r <sub>E→E+(E)</sub> |                    |                    | r <sub>E→E+(E)</sub> |   |
| ...          |                      |                    |                    |                      |   |

46

然后我们把最右边一些分成 Goto Table。因为从 4 看到 E，一定这个 E 是在栈上的，一定是要继续走到下一个状态的，所以这里不能做 shift 或者 reduce。表的最终状态如下：

- The table for a fragment of our DFA:



47

如果我们把状态机扩展一下，栈上除了有 symbol 以外，还有 symbol 停在的状态上，其中  $state_k$  就是走到 symbol k 的时候停在的状态。

在每次 shift 操作或者 reduce 操作后，我们需要在空栈上重新走一遍这个 DFA，比较耗时。我们就可以在每个栈中的元素维护一下走到这个元素时，到了 DFA 中的那个状态。

#### The LR Parsing Algorithm

```

Let I = w$ be initial input
Let j = 1
Let DFA state 0 be the start state
Let stack = < dummy, 0 >
repeat
 case action[top_state(stack), I[j]] of
 shift k: push < I[j++], k >
 reduce X → α:
 pop |α| pairs,
 push < X, Goto[top_state(stack), X] >
 accept: halt normally
 error: halt and report error
 endcase
endrepeat

```

最初始的时候，没有东西在栈上，我们用 dummy 来代替。然后我们根据栈顶的状态和 next token 去查 action table，然后就会告诉我们是 shift 还是 reduce，

Shift: 比如 s8 就是 shift 后状态变成 8 了，然后就把当前 symbol 和对应的转移到的状态压栈。然后就不需要从头走了。

Reduce: reduce 的 production 是在 action 中有的，栈顶的 symbol 一定是  $\alpha$ ，pop 掉对应元素后，再填入  $< X, Goto[\text{top\_state(stack)}, X] >$

$\langle sym_1, state_1 \rangle, \dots, \langle sym_k, state_k \rangle, \underbrace{\langle sym_{k+1}, state_{k+1} \rangle, \dots, \langle sym_n, state_n \rangle}_{|\alpha|}$

$state_0 \xrightarrow{X} *state_k \xrightarrow{} state_t$ ，根据 goto table 就可以知道， $state_t = Goto(state_k, X)$ ，然后我们把 state 填回去就行了：

$$\langle sym_1, state_1 \rangle, \dots, \langle sym_k, state_k \rangle, \langle X, state_t \rangle$$

LR 的特性：

- LR 一定能够处理更多的文法，因为在 LL 的时候，根据当前的状态和下一个 symbol，一定要找到自己的产生式。而 LR，确定不了的时候可以进栈，比如可以 shift 三次，再来做决定。所以 LL 是看到了马上要做决定，而 LR 做不了决定的时候可以进栈，之后再做决定，所以 LR 比 LL 要 powerful，大部分程序设计语言都是可以变成 LR 的。
- LR 的核心就是一个 table，有了这个 table 就是上面很简单的 algorithm，输入、查表、做 action。

这张表也可以通过构造出来，根据 DFA 构造表是讲词法的时候讲的，是很容易的事情，所以核心就是构建我们的 DFA。

DFA 做的事情：栈上有一堆 symbol，下面就是我们想通过 DFA 知道当前我们正在找哪一个 non-terminal，这实际上就是找 non-terminal 对应 production 的右边，比如我们拿到了一个  $X$ ，我们一定是看栈顶上是不是有  $\alpha$ ，可以把  $\alpha$  规约成  $X$ 。可能只走了一半，栈中只有一半的  $\alpha$ ，此时可能就选择 shift 操作。

比如我们想把目前的栈上符号规约成非终结符  $X$ ，对应的规则是  $X \rightarrow \alpha \bullet \beta, a$ ，也就是已经看到了  $\alpha$  这部分，在这种情况下因为我们还没看完，是要做 shift，此时要看下一个 token。

## LR(1) item

一个 LR(1) item 是一个 pair， $X \rightarrow \alpha \bullet \beta, a$

其中第一个元素是一个推导规则，而第二个元素是一个终结符（也叫作前看终结符）这个 pair 描述了 parser 的一个语义，我们最终希望规约出  $X$ ，栈顶已经有了  $\alpha$ ，希望结束的时候停在了  $Xa$  这个状态。所以下边要做的事情就是看看剩下的  $\beta a$  和当前的 input:

$$\alpha | \underbrace{t_{k+1}, \dots, t_n}_{\gamma} .$$

DFA 的目的就是描述出上面这些东西。

我们规定 start symbol 只有一个 production:  $S \rightarrow E$ 。最初的状态是把 input string 换成  $S$ ，我们希望后头看到的是个  $E$  并且跟了一个 \$，这样我们的目的就达成了。我们的期望是这样子。

最初的 parse 上下文就是  $S \rightarrow \bullet E, \$$ ，且栈为空，我们尝试从  $E\$$  规约出  $S$  来。

那么，我们继续有如下的 LR(1) item。

我们有了这个期望以后，就会有  $\bullet E$  在这个地方。因为是  $E$  是一个 non-terminal，比较难处理。

- $E \rightarrow E + \bullet(E), +$ ，直接做 shift，也就是如果我们看到左括号之后，就可以进入

$E \rightarrow E + (\bullet E), +$ 。如果后头没有跟左括号就报错。

- $E \rightarrow E + (E)\bullet, +$  此时一点跟在最后，那也是简单的，直接做 reduce。

但是  $E \rightarrow E + (\bullet E), +$  这种情况比较复杂，因为点在 non-terminal 前，稍微复杂一点，我们希望看到  $E$ ，但是  $E$  是 non-terminal 的，所以我们需要根据  $E$  的两个 production 来把后面的 terminal 串规约出  $E$  来，这两个 production 为：

$$\begin{aligned} E &\rightarrow \bullet \text{int}, ) \\ E &\rightarrow \bullet E + (E), ) \end{aligned}$$

所以希望看  $E$  就转换成了希望看到 int) 或者  $E+(E))$ ，而到了这里  $E \rightarrow \bullet \text{int}, )$  就是比较好的处理的情况了，第二种还是 non-terminal 的情况。但是这是我们希望看到  $E$  后头跟加号的情况。

这个红点如果跟在 non-terminal 前头，我们就把这个 non-terminal 用 production 来扩展，就有更多的东西出现，这些东西都叫做一个 LR(1) 的 item（含红点的 production 和逗号后的 symbol）。

对于一个 LR(1) item:  $X \rightarrow \alpha \bullet y\beta, a$ ，因为点跟在一个非终结符  $y$  前面，这说明了我们的希望，但是没有办法来做动作。此时我们就把  $y$  的所有 production 都拿过来，有了这个东西以后，然后我们取出  $First(\beta a)$ ，加到 item 里头，这时候就是算一个 closure（闭包），这样做了以后有可能就出现红点+terminal 的情况，但是还可能继续出现红点+non-terminal 的情况，一定要继续展开，直到集合中的东西不断变大，token、production、红点的位置都是有限的，只要的这个过程不断去重复，最后这个东西一定就稳定了，不会再变化了。

使用 item 扩展 context 的操作叫做 closure operation。注意  $First(\beta a)$  就是它能够推导出的所有字符串的第一个字符的集合。

```
Closure(Items) =
repeat
 for each [X → α•yβ, a] in Items
 for each production Y → γ
 for each b ∈ First(βa)
 add [Y → •γ, b] to Items
until Items is unchanged
```

这样就把我们的希望换成了一些具体的行动，一开始我们看到这些东西：

原先我们只有  $E \rightarrow E + (E)$  和  $E \rightarrow \text{int}$  这两个 production。我们从  $S \rightarrow \bullet E, \$$  开始 expand，最后就变成了三个 item。这就是开始状态的 closure。

## Constructing the Parsing DFA (1)

---

- Construct the start context: Closure({ $S \rightarrow \bullet E, \$$ })

$$\begin{aligned} S &\rightarrow \bullet E, \$ \\ E &\rightarrow \bullet E + (E), \$ \\ E &\rightarrow \bullet \text{int}, \$ \\ E &\rightarrow \bullet E + (E), + \\ E &\rightarrow \bullet \text{int}, + \end{aligned}$$

- We abbreviate as:

$$\begin{aligned} S &\rightarrow \bullet E, \$ \\ E &\rightarrow \bullet E + (E), \$/+ \\ E &\rightarrow \bullet \text{int}, \$/+ \end{aligned}$$

- A DFA state is a **closed** set of LR(1) items
- The start state contains [ $S \rightarrow \bullet E, \$$ ]

这个 closure 算出来以后，就是我们 DFA 的一个状态。如果某个状态包含了 item [ $X \rightarrow \alpha \bullet, b$ ]，此时就可以做 reduce。

从状态 0，我们要构造出状态 1 和状态 2.

- A state "State" that contains [ $X \rightarrow \alpha \bullet y \beta, b$ ] has a transition labeled  $y$  to a state that contains the items "Transition(State,  $y$ )"
  - $y$  can be a terminal or a non-terminal

$\text{Transition}(\text{State}, y)$

```

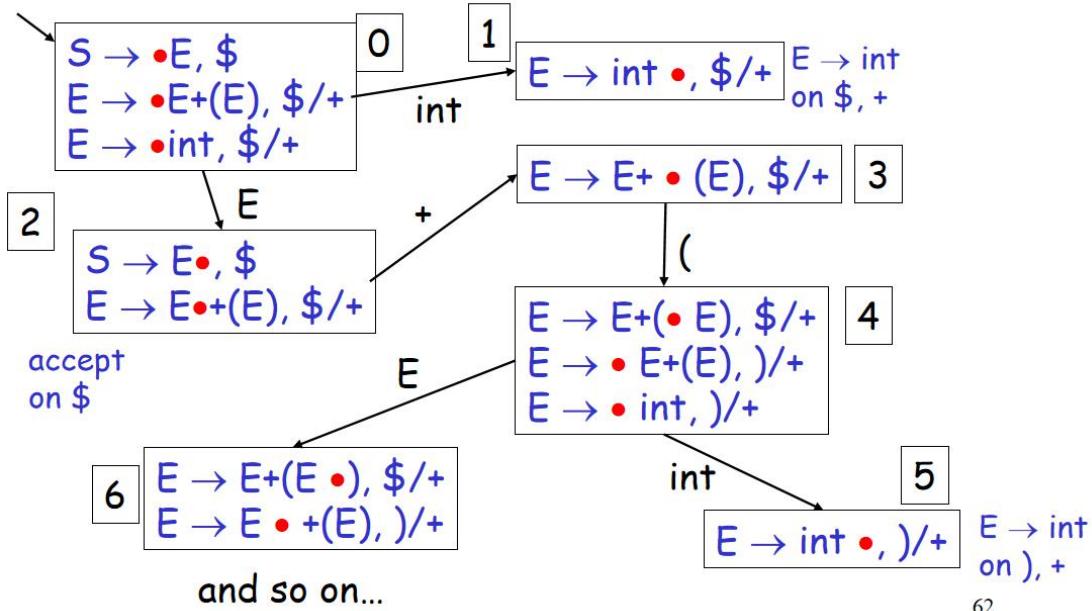
Items = ∅
for each [X → α • yβ, b] ∈ State
 add [X → αy • β, b] to Items
return Closure(Items)

```

如果  $state_k$  中包含了 [ $X \rightarrow \alpha \bullet y \beta, b$ ] 这个 item， $y$  是一个 symbol，它可以是 terminal

也可以是 non-terminal，下面我们想构造出一个从  $state_k$  到  $state_{k+1}$  的 transition。 $state_{k+1}$  一定包含了  $[X \rightarrow \alpha y \bullet \beta, b]$  这个 item。

## Constructing the Parsing DFA. Example.



62

从开始状态 0 开始，小红点后面跟了 int 或者 E，所以出现了状态 1 和状态 2，对于状态 1，小红点在最后了，此时做 reduce。对于状态 2，读到了一个 E 以后，小红点移到后面一个，也就是 shift 一次，对于状态 2 中的一个小红点在最后的情况，做一次 reduce，也就是 accept 的 action。状态 3 就是状态 2 的小红点+terminal 的情况，直接 shift 即可。状态 3 到状态 4 同理。状态 4 依旧需要算 closure。然后根据小红点后面跟了什么，继续 expand。走到最后，状态不再增加的时候，这个 DFA 就构造出来了，我们通过这个方法，从一个最初的状态通过计算 closure 算出了状态 0 的所有的 item，然后通过 transition 派生出其他的状态，逐步把 DFA 的状态一个个派生出来，DFA 的每个状态就告诉我们希望干什么和已经干了什么，有了这个状态以后就可以自动做 LR(1) parser 了。

**2021/10/12**

上节课讲到了这么根据 CFG 来自动地构造 DFA，这个事情可以自动地来做，但是问题是说，我们构造出 DFA 以后，我们的 parser generator 就产生了呢？我们需要的不仅仅是构造出自动机来，在自动机上还有 action (shift 或 reduce)，当我们在 DFA 上设置 action 的时候会不会出什么问题。很有可能在构造 DFA 的时候，一个个 item 是一个集合，每个集合是 item 的集合。

- If a DFA state contains both  
 $[X \rightarrow \alpha \bullet a\beta, b]$  and  $[Y \rightarrow \gamma \bullet, a]$
- Then on input "a" we could either
  - Shift into state  $[X \rightarrow \alpha a \bullet \beta, b]$ , or
  - Reduce with  $Y \rightarrow \gamma$
- This is called a shift-reduce conflict

这是 shift-reduce conflict，因为是我们的语法有二义性

- Typically due to ambiguities in the grammar
- Classic example: the dangling else  
 $S \rightarrow \text{if } E \text{ then } S \mid \text{if } E \text{ then } S \text{ else } S \mid \text{OTHER}$
- Will have DFA state containing
  - $[S \rightarrow \text{if } E \text{ then } S \bullet, \text{ else}]$
  - $[S \rightarrow \text{if } E \text{ then } S \bullet \text{ else } S, \text{ x}]$
- If **else** follows then we can shift or reduce
- Default (bison) is to shift
  - Default behavior is as needed in this case

当我们看到 else 应该是做 reduce 还是 shift，我们在文法中是没有界定清楚的

1. 修改文法，使得文法没有二义性，但是这没有自动化方法，需要人手动去写
2. 让 parser generator 在发现 shift-reduce conflict 的时候，让它自己选一种（现在的工具默认都会选择 shift）。

像这样有二义性的文法还是挺多的，比如加法和乘法在一起的优先级顺序、是否左结合、右结合。所以在构造 DFA 状态的时候不可避免会出现这个情况：

- Consider the ambiguous grammar  
 $E \rightarrow E + E \mid E * E \mid \text{int}$
- We will have the states containing
  - $[E \rightarrow E * E \bullet, +]$
  - $[E \rightarrow E \bullet + E, +]$
- Again we have a shift/reduce on input +
  - We need to reduce ( $*$  binds more tightly than  $+$ )
  - Recall solution: declare the precedence of  $*$  and  $+$

显然乘法的优先级比加法高，所以我们希望做的是 reduce。

- Same grammar as before  
 $E \rightarrow E + E \mid E^* E \mid \text{int}$
- We will also have the states  
 $[E \rightarrow E + E \bullet, +]$   
 $[E \rightarrow E \bullet + E, +]$

- Now we also have a shift/reduce on input +  
  - We choose reduce because  $E \rightarrow E + E$  and + have the same precedence and + is left-associative

同样的，此处我们如果选择 reduce，就代表加法是左结合的；如果选择 shift，那么就代表加法是右结合的。所以我们就定义了在自动生成之外的规则来解决这个问题。

- In bison declare precedence and associativity:  

```
%left +
%left *
```
- The precedence of the rule is determined by the last token in its rhs
- Resolve shift/reduce conflict with a shift if:
  - no precedence declared for either rule or terminal
  - input terminal has higher precedence than the rule
  - the precedences are the same and right associative

所以在 bison 中，可以定义结合律和优先级的顺序（写在后面的优先级高），当我们定义完了以后，我们就根据三条准则来选取 shift，只有在这三个条件都不满足的情况下，就选择 reduce。我们这些定义的优先级都是对于 token 和 terminal 定义的，production 的优先级是由这个 production 的最后一个 token 来确定的。

- 如果说 production 和 terminal 都没有优先级的时候，就使用 shift，也就是如果 if-then-else 没有定义优先级的情况下，默认选择 shift。
- 输入的 terminal 大于 production 的优先级的情况，选择 shift，比如  $E \rightarrow E + E, ^*$  的时候，因为乘法的优先级大于加号（production）的优先级，我们就做 shift。
- 优先级一样的时候，就要看结合律了，右结合做 shift 而左结合做 reduce，对于幂次运算，它是右结合的，eg:  $4^{3^2} \rightarrow 4^9$ .

我们为了避免这个情况，最好是让我们的 parse tree 没有那么多的二义性。

## Reduce/Reduce Conflicts

- If a DFA state contains both  
 $[X \rightarrow \alpha \bullet, a]$  and  $[Y \rightarrow \beta \bullet, a]$ 
  - Then on input "a" we don't know which production to reduce
- This is called a reduce/reduce conflict

下面的叫做 reduce-reduce conflict，也就是栈上是要把不同的东西规约为不同的 non-terminal，这时候可能做两种不同的动作，这也是因为文法中的二义性导致的。

- Usually due to gross ambiguity in the grammar
- Example: a sequence of identifiers
  - $S \rightarrow \epsilon \mid id \mid id S$
- There are two parse trees for the string id
  - $S \rightarrow id$
  - $S \rightarrow id S \rightarrow id$
- How does this confuse the parser?

这个文法定义了任意长度的 id 串，如果不写中间这个，就没问题了，但是为了让文法有二义性，就多写了一个 production，但是写不写定义出来的文法是一样的。对于 id 这个叶子结点，我们可以使用两种不一样的 tree 的形式来导出它，这就会产生一个 conflict。

- Consider the states
  - $[S' \rightarrow \bullet S, \$]$
  - $[S \rightarrow \bullet, \$]$
  - $[S \rightarrow \bullet id, \$]$
  - $[S \rightarrow \bullet id S, \$]$
  - $[S \rightarrow id \bullet, \$]$
  - $[S \rightarrow id \bullet S, \$]$
  - $[S \rightarrow \bullet id, \$]$
  - $[S \rightarrow \bullet id S, \$]$
- Reduce/reduce conflict on input \$
  - $S' \rightarrow S \rightarrow id$
  - $S' \rightarrow S \rightarrow id S \rightarrow id$
- Better rewrite the grammar:  $S \rightarrow \epsilon \mid id S$

在转移后的新的 state 中，我们看到了有两个 reduce，这就导致了 reduce-reduce conflict。把这个二义性消掉的话就重写一下这个文法。

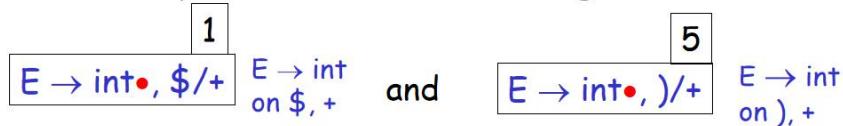
这样的话，parser generator 是根据一个给定的 CFG，可以构造出一个 DFA，然后我们用规则来消除一些 conflict，这样我们消除 conflict 规则是自动的，构造 DFA 方法也是自动的，所以最后根据一个 input of tokens 来走这个 DFA，走的时候会有一个栈来放 symbol 和状态，走一遍全部 reduce 成一个 start symbol 和 \$ 的时候，就成功了；如果中间停下来不能走了，那么就是 error。

## LALR(1):合并 LR(1) 的 DFA 状态

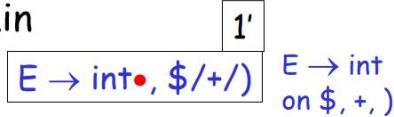
到这里这件事情似乎结束了,但是真的 parser generator 没有到此为止,理由是因为 LR(1) 的构造出来的 DFA 的状态太多了,一个很小的语言就会有几千个状态,效率比较差。进一步研究,我们就发现很多 state 状态长得差不多。

### LR(1) Parsing Tables are Big

- But many states are similar, e.g.



- Idea: merge the DFA states whose items differ only in the lookahead tokens
  - We say that such states have the same core
- We obtain



我们发现上例中, rule (production) 是一样的,只是 followed symbol 是不一样的,我们就想把这样的状态做一个合并,我们把前头一样的部分叫做 core,也就是我们希望把相同 core 的 state 合并。

### The Core of a Set of LR Items

- Definition: The core of a set of LR items is the set of first components
  - Without the lookahead terminals
- Example: the core of
$$\{ [X \rightarrow \alpha \bullet \beta, b], [Y \rightarrow \gamma \bullet \delta, d] \}$$
is
$$\{ X \rightarrow \alpha \bullet \beta, Y \rightarrow \gamma \bullet \delta \}$$

LR items 的 core 就是集合里每个 item 的第一部分,构成的这个集合就是 core。

- Consider for example the LR(1) states
  $\{[X \rightarrow \alpha \bullet, a], [Y \rightarrow \beta \bullet, c]\}$ 
 $\{[X \rightarrow \alpha \bullet, b], [Y \rightarrow \beta \bullet, d]\}$
- They have the same core and can be merged
- And the merged state contains:
  $\{[X \rightarrow \alpha \bullet, a/b], [Y \rightarrow \beta \bullet, c/d]\}$
- These are called LALR(1) states
  - Stands for LookAhead LR
  - Typically 10 times fewer LALR(1) states than LR(1)

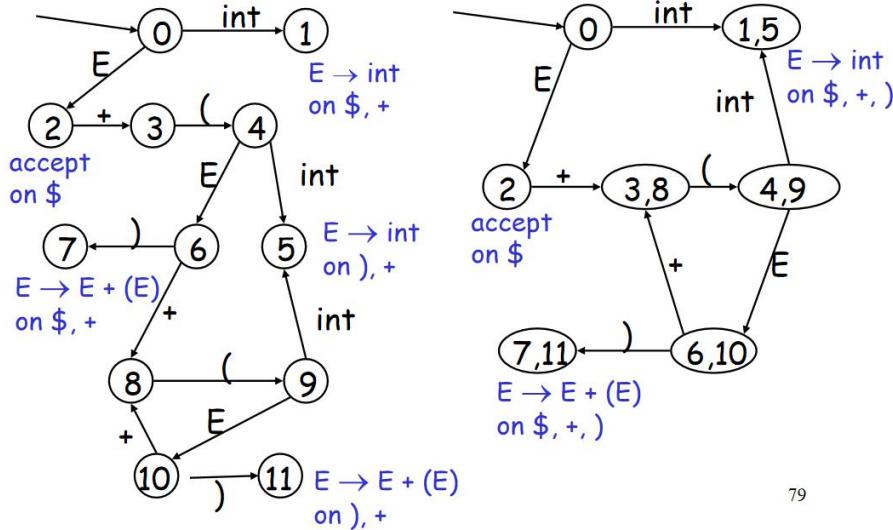
原来我们是 LR(1)，我们就把 core 一样的做一个合并，合并完以后的状态叫做 LALR(1) 状态。LA 叫做 look ahead。这样的合并会把状态减掉十倍。

- Repeat until all states have distinct core
  - Choose two distinct states with same core
  - Merge the states by creating a new one with the union of all the items
  - Point edges from predecessors to new state
  - New state points to all the previous successors



在前面做 DFA minimize 的时候，我们做过类似的事情。

## Conversion LR(1) to LALR(1). Example.



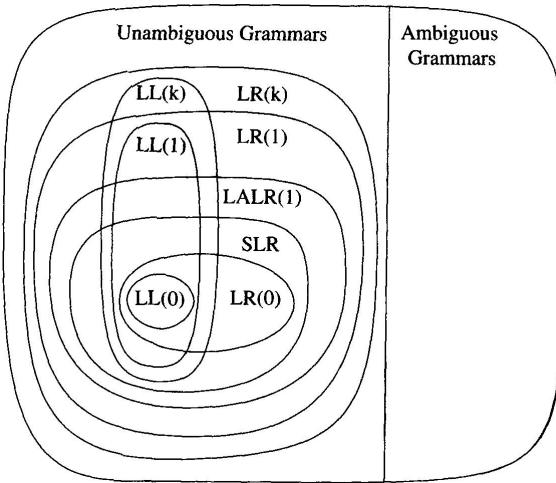
79

我们把两个 LALR(1)的状态合并之后，会引入一些 conflict。

- Consider for example the LR(1) states
  $\{[X \rightarrow \alpha \bullet, a], [Y \rightarrow \beta \bullet, b]\}$ 
 $\{[X \rightarrow \alpha \bullet, b], [Y \rightarrow \beta \bullet, a]\}$
- And the merged LALR(1) state
  $\{[X \rightarrow \alpha \bullet, a/b], [Y \rightarrow \beta \bullet, a/b]\}$
- Has a new reduce-reduce conflict
- In practice such cases are rare

这就会导致新的 reduce-reduce conflict，幸运的是这种情况非常少见。

- LALR languages are not natural
  - They are an efficiency hack on LR languages
- Any reasonable programming language has a LALR(1) grammar
- LALR(1) has become a standard for programming languages and for parser generators



我们的工具是对文法非常敏感的，那么文法首先分成 LL 和 LR，我们只讲了 LL(1) 和 LR(1)，我们也可以往前看  $k$  个。我们这里也没讲什么都不看的情况。

一般做 parser，是基于 CFG 的；手写的 parser，就基于 LL(1) 即可，要处理比较 general 的 language，就可以使用 LR(1)，为了提高效率，我们就可以使用 LALR(1)。

我们再讲一下 error recovery，遇到 error 可以分成以下几类：

1. 立即停止，解决一个问题要重新编译一次。
2. 记录错误，并且尝试继续，一个问题可能造成一大堆的错误。

如果不让它停下，我们就要做 error recovery，让程序尽可能地 parser 下去。Recovery 的方法是删除、加入和替换。Recovery 还分成 local recovery（一点报错就在这一点插入删除替换）、global recovery（一点的错误，在这点之前的地方也去做插入删除替换）。

$$\text{exp} \rightarrow \text{NUM}$$

$$| \text{exp PLUS exp}$$

$$| (\text{exp})$$

$$\text{exp} \rightarrow \text{exp}$$

$$| \text{exp ; exp}$$

- *General strategy for both bottom-up and top-down:*
  - look for a **synchronizing token**
- *Accomplished in bottom-up parsers by adding error rules to grammar:*

$$\text{exp} \rightarrow (\text{error})$$

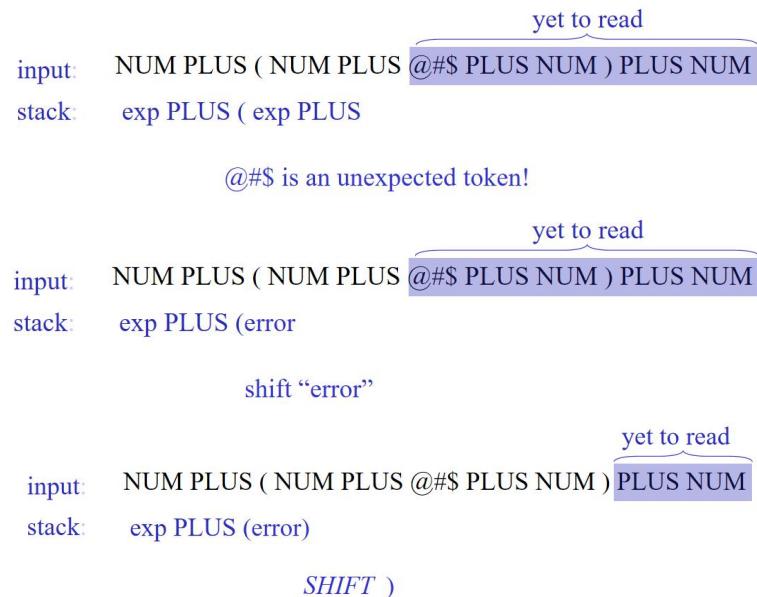
$$\text{exp} \rightarrow \text{exp}$$

$$| \text{error ; exp}$$

比如我们在处理 expression 的时候碰到问题的时候，我们怎么来 recover。其实我们都是要增加一个出错处理的 production，加的时候就有技巧了，其中的 error 是人为构造出的新的 token，当我们发现 error 的时候，我们是想找 error 后头的一个 token，其实也就是 synchronizing token，比如说上例中的右括号和分号，一旦我们出错的情况下，我们就会：

- In general, follow error with a synchronizing token. Recovery steps:
  - Pop stack (if necessary) until a state is reached in which the action for the error token is shift
  - Shift the error token
  - Discard input symbols (if necessary) until a state is reached that has a non-error action
  - Resume normal parsing

我们需要把栈上的东西 pop 掉，然后 shift 一个 error token 进来。



第二种方法是 global 的，这个时候我们就是说我们要做插入删除和替换，当我们发现出错的时候，我们在出错之前的地方去做插入删除和替换。具体的有个方法是 Burke-Fisher error repair 方法，当前我们发现了一个 error，我们往前退 k 个 token，在这 k 个 token 中，我们来做 single token 的操作，要么插入一个、要么删除一个、要么替换一个。比如说，我们的 parser 在第 500 个的时候出错了，我们就在 480~500 的范围内来做动作。我们做了一个动作以后，我们就去做 parser，我们就去看最长的 parse 是我们要的 repair。

- Burke-Fisher error repair
  - tries every possible single-token insertion, deletion or replacement at every point in the input up to K tokens before the error is detected
    - eg: K = 20; parser gets stuck at token 500; all possible repairs between token 480-500 tried
    - best repair = longest successful parse

- Consider Burke-Fisher with
  - K-token window
  - N different token types
- Total number of repairs =  $N + 2K^N$ 
  - deletions (K) +
  - insertions  $(K + 1)^N$  +
  - replacements  $K(N - 1)$
- Affordable in the uncommon case when there is an error

对于 k 个 token 来说，delete 可以 k 个；插入有  $k+1$  个位置，选择有 N 种；替换有 k 个位置，选择有  $N-1$  种，所以一共有  $N + 2KN$  种操作，每次做完操作我们还要停下来 parse 一次。也就是在错误不常见的时候，勉强能来做这个事。因为出错的时候需要往前退 K 个，所以我们需要两个栈。始终保持 old stack 和 new stack 之间保持 K 个的差距。

## 语法分析工具 yacc

最后我们来讲一下工具，

### **YACC Source Format**

---

%{Supplementary Code%}

%definition

.....

%definition

%%

### Rules

一上来是 %{ C 代码 %}，然后定义哪些是 token、terminal、start symbol，再往下还可以定义一些结合律和优先级之类的； %%后面就是 production。

|                                                        |               |
|--------------------------------------------------------|---------------|
|                                                        | %%            |
|                                                        | prog: stmlist |
| <b>Yacc (example 1)</b>                                |               |
| %{                                                     |               |
| int yylex(void);                                       |               |
| void yyerror(char *s) { EM_error(EM_tokPos, "%s", s);} |               |
| %}                                                     |               |
| %token ID WHILE BEGIN END DO IF THEN ELSE ESMI ASSIGN  |               |
| %start prog                                            |               |
|                                                        | Terminal      |
|                                                        | Nonterminal   |
| stm : ID ASSIGN ID                                     |               |
| stm : WHILE ID DO stm                                  |               |
| stm : BEGIN stmlist END                                |               |
| stm : IF ID THEN stm                                   |               |
| stm : IF ID THEN stm ELSE stm                          |               |
| stmlis : stm                                           |               |
| stmlist SEMI stm                                       |               |

前面 lex 帮我们做好了，拿来给我们的 parser 用。在下面我们可以定义 token 和 non-terminal。然后就是一些我们定义的 production。

### Yacc (example 2)

```
%{ declarations for yylex and yyerror %}
%token INT PLUS MINUS TIMES UMINUS
%start exp

%left PLUS MINUS
%left TIMES
%left UMINUS

%%
exp : INT
exp : exp PLUS exp
exp : exp MINUS exp
exp : exp TIMES exp
exp : MINUS exp %prec UMINUS
```

注意其中左结合和优先级（负号的优先级最高）的定义方法。注意最后一条，在 lex 的时候，我们认为-1 和 2-1 中的 minus 是同一个 token，但是我们又想保证负号的优先级比减号高，那么我们可以附加定义最后一条规则的优先级为 UMINUS，这条规则的优先级变高了。

```
%%
stm: ID ASSIGN ae
 | ID ASSIGN be
be: be OR be
 | be AND be
 | ae EQUAL ae
 | ID
ae: ae PLUS ae
 | ID
e : ID
 | e OR e
 | e AND e
 | e EQUAL e
 | e PLUS e
reduce/reduce conflict for ID to ae or ID to be Defer the error report for expression "a + b&c" to semantic phase
```

注意，在上面这个语法中， $a+b\&c$  是符合语法的，但是不符合语义，到时候我们还会做一次 type checking 来做语义分析。

- ML-Yacc provides additional support for Burke-Fisher
  - to continue parsing, we need semantics values for inserted tokens
- some multiple-token insertions & deletions are common, but it is too expensive for ML-Yacc to try every 2,3,4- token insertion, deletion

```
%change EQ -> ASSIGN
| SEMI ELSE -> ELSE
| -> IN INT END
```

*ML-Yacc would do this  
anyway but by specifying,  
it tries it first*

在 ML-Yacc 中，是支持 Burke-Fisher error repair 方法的。可以用%value 去指定一些值。有了这些东西，recovery 就可以做了。还有一个就是这个方法一次只能换一个，当我们想换多个的时候，不能枚举去换，但是我们可以指定某些替换。使用%change 来把 equal 替换成 assign。

第四章：分析程序对错的时候，我们要生成内部表示。Source program 是 asc2 字符构成的，在编译的时候要引入中间结构。是有很多层级的 IR 的，最高层级的 IR 是抽象语法树，它像我们的高级语言。一个 syntax tree 是要记录程序的层次结构，而符号表是在程序中有很多的变量、函数、用户定义的类型，这些都是符号，所以我们需要记录下来。我们还需要知道变量的类型、如果变量是一个数组，那么我们还需要知道变量元素的类型是什么；函数的参数和返回类型是什么；user-defined type 到底是什么 type。

## AST

AST (抽象语法树) 是一个树，我们现在已经教过些什么呢？程序的组成是递归定义的。描述这样的构造就是 CFG，它对应的是一个 parse tree，树可以表示为一个层次结构。

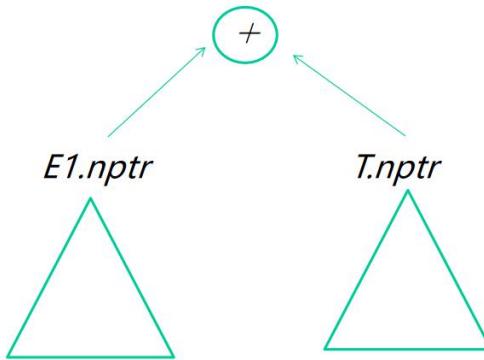
Syntax tree 和 parse tree 有一点点区别，syntax tree 中没有必要把 parse tree 中的一些中间结果保留，比如 parse 中有 E->T->1，我们在 syntax tree 中只需要一个 1 就行了。

我们要引入 attribute (属性) 的概念，我们在描述文法的时候是用 CFG，其中的 terminal 和 non-terminal 都是 grammar symbol，接下来我们要给每个 symbol 对应一个 value。在我们构造 syntax tree 的时候，对应的 value 就是树的根。我们一边在 parser 的时候，一边要计算它的 value。

## Attributes (2)

- For a production:  $E \rightarrow E_1 + T$
- The pointer to the root of the AST is denoted as  $E.nptr$
- The pointer to the root of left subtree is  $E_1.nptr$
- The pointer to the root of right subtree is  $T.nptr$
- The  $E.nptr$ ,  $E_1.nptr$  and  $T.nptr$  are attributes for  $E$ ,  $E_1$  and  $T$  respectively

也就是每一个 symbol 现在有了一个 value，也就称为 attribute。Parser 的过程也就是把  $E_1 + T$  规约成  $E$ ，显然前面两个 pointer 已经有了。



An AST node is associated with the production

- This node is constructed according to the production

Now, each production is also associated with a semantic rule

- Semantic rules define how the value of an attribute at a parse-tree node is computed
- A semantic rule may have side effects
  - Printing a value or updating a global variable

Define operations on the data structures

- `mknode(op, left, right)`

- `mkleaf(id, entry)`

- `mkleaf(num, val)`

我们有一些函数 mknnode, mkleaf。

| Production                 | Semantic Rules                                                         |
|----------------------------|------------------------------------------------------------------------|
| $E \rightarrow E_1 + T$    | $E.\text{nptr} := \text{mknnode}('+', E_1.\text{nptr}, T.\text{nptr})$ |
| $E \rightarrow E_1 - T$    | $E.\text{nptr} := \text{mknnode}('-', E_1.\text{nptr}, T.\text{nptr})$ |
| $E \rightarrow T$          | $E.\text{nptr} := T.\text{nptr}$                                       |
| $T \rightarrow (E)$        | $T.\text{nptr} := E.\text{nptr}$                                       |
| $T \rightarrow \text{id}$  | $T.\text{nptr} := \text{mkleaf}(\text{id}, \text{id}.entry)$           |
| $T \rightarrow \text{num}$ | $T.\text{nptr} := \text{mkleaf}(\text{num}, \text{num}.val)$           |

那么我们就可以对应地去写 semantic rule 了，如果  $T \rightarrow (E)$ ，那么  $T$  的节点就直接赋值给  $E$ ，是一个压缩的过程，没有冗余了。这里我们都是通过右边的 attribute 来计算左边的。

### Synthesized attributes

- The value of an attribute at a node is computed from
  - values of the attributes at the children of that node in the parse tree

刚才是构造树的过程，我们也可以构造一个计算器，一边 parser，一边把结果算出来。

Eg: 构造计算器的过程

| Production                   | Semantic Rules                                  | 3*5+4             |
|------------------------------|-------------------------------------------------|-------------------|
| $L \rightarrow E \ n$        | $\text{Print}(E.\text{val})$                    | $L$               |
| $E \rightarrow E_1 + T$      | $E.\text{val} := E_1.\text{val} + T.\text{val}$ | $E.\text{val}=19$ |
| $E \rightarrow T$            | $E.\text{val} := T.\text{val}$                  | $n$               |
| $T \rightarrow T_1 * F$      | $T.\text{val} := T_1.\text{val} * F.\text{val}$ | $+$               |
| $T \rightarrow F$            | $T.\text{val} := F.\text{val}$                  | $T.\text{val}=4$  |
| $F \rightarrow (E)$          | $F.\text{val} := E.\text{val}$                  | $T.\text{val}=15$ |
| $F \rightarrow \text{digit}$ | $F.\text{val} := \text{digit}.lexval$           | $*$               |
|                              |                                                 | $F.\text{val}=5$  |
|                              |                                                 | $digit.lexval=5$  |
|                              |                                                 | $digit.lexval=3$  |

- Modify the data structure for parser stack

| state    | val        |
|----------|------------|
| ...      | ...        |
| <b>X</b> | <b>X.x</b> |
| <b>Y</b> | <b>Y.y</b> |
| <b>Z</b> | <b>Z.z</b> |
| ...      | ...        |

2021-10-12                          45

这样在 parser generator 的时候，栈上多了一列以后，就可以把 attribute 记录进去了。

Eg: 栈上的例子

| Production              | Semantic Rules                      |
|-------------------------|-------------------------------------|
| $L \rightarrow E n$     | <code>print(val[top])</code>        |
| $E \rightarrow E_1 + T$ | $val[top] := val[top-2] + val[top]$ |
| $E \rightarrow T$       |                                     |
| $T \rightarrow T_1 * F$ | $val[top] := val[top-2] * val[top]$ |
| $T \rightarrow F$       |                                     |
| $F \rightarrow (E)$     | $val[top] := val[top-1]$            |
| $F \rightarrow digit$   |                                     |

| Input         | State        | Val   | Production used       |
|---------------|--------------|-------|-----------------------|
| $3 * 5 + 4 n$ | -            | -     |                       |
| $* 5 + 4 n$   | <i>digit</i> | 3     |                       |
| $* 5 + 4 n$   | <i>F</i>     | 3     | $F \rightarrow digit$ |
| $* 5 + 4 n$   | <i>T</i>     | 3     | $T \rightarrow F$     |
| $5 + 4 n$     | $T *$        | 3 -   |                       |
| $+ 4 n$       | $T * digit$  | 3 - 5 |                       |
| $+ 4 n$       | $T * F$      | 3 - 5 | $F \rightarrow digit$ |
| $+ 4 n$       | <i>T</i>     | 15    | $T \rightarrow T * F$ |

| Input    | State       | Val    | Production used       |
|----------|-------------|--------|-----------------------|
| $+ 4 n$  | <i>E</i>    | 15     | $E \rightarrow T$     |
| $4 n$    | $E +$       | 15 -   |                       |
| $n$      | $E + digit$ | 15 - 4 |                       |
| $n$      | $E + F$     | 15 - 4 | $F \rightarrow digit$ |
| $n$      | $E + T$     | 15 - 4 | $T \rightarrow F$     |
| $n$      | <i>E</i>    | 19     | $E \rightarrow E + T$ |
| $E n$    |             | 19 -   |                       |
| <i>L</i> |             | 19     | $L \rightarrow E n$   |

## 2021/10/15

上节课我们讲了一边 parser 一边做动作。早期的编译器没有那么多动作，一边做 parser 一边生成汇编，这件事比较复杂。现在我们在做语法分析的时候就生成 AST (抽象语法树)，

上节课还举了一个例子，一边 parse 一边生成计算结果，所以也不是要一定生成 AST。

## 语法分析工具 Bisonc++

最早用的是 yacc，现在我们比较用的多的是 Bison C++，这次我们的 lab 和书上是不一样的，一开始也是：

### Bisonc++ Example

```
%scanner scanner.h /* include declarations of lex and error */
%union {int num; string id; }

%token <num> INT
%token <id> ID
%type <num> exp
%start exp

%left PLUS MINUS
%left TIMES
%left UMINUS
```

这个 type 是生成 terminal 和 non-terminal 的。它会跟一个 symbol list。然后我们有优先级，left right 还有 non-associative。

### Useful Bison Directives

- **%token [ <type> ] terminal token(s):**  
The %token directive is used to define one or more symbolic terminal tokens.  
The <type> specification is optional, and specifies the type of the semantic value when receiving one of the subsequently named tokens is received.
- **%type <type> symbol-list**  
- To associate (non-)terminals with specific semantic value types the %type directive is used.
- **%start nonterminal symbol**  
- defining the start rule
- **%nonassoc %right %left**  
- These directives are called precedence directives  
- used to declare tokens and to specify their precedence and associativity, all at once

## Bisonc++ Example

```
%%
exp : INT Semantic action
exp : exp PLUS exp {$$ = $1 + $3; }
exp : exp MINUS exp {$$ = $1 - $3; }
exp : exp TIMES exp {$$ = $1 * $3; }
exp : UMINUS exp %prec UMINUS {$$ = -$2; }
$2
```

%prec directive gives the rule as the same priority as the UMINUS

上节课我们讲 yacc 的时候是没有后头这个东西。今天我们实际上就是说要把 semantic action (一边做 parser 的时候一边做什么动作) 加进去。上例是计算器的例子，每一个表达式中间都有 semantic value，这就是我们的值。当我们把 int 规约为 expression 的时候，int 的 semantic value 就被赋值给了 expression。左边的 semantic value 就用 \$\$ 来表示，右边的符号比较多，就从左到右按照 \$1, \$2, \$3 来表示。

给定一个存在+-\*的表达式，它就可以把对应的值算出来。这就像在 basic 解释器中，表达式我们中缀变后缀的时候，一边变、一边把值算出来，这就和我们这里一边分析一边求值一样。

下面我们再举一些例子来看看 bison c++。

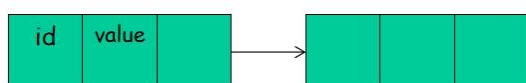
### CFG for straight-line programs

|                                 |                      |
|---------------------------------|----------------------|
| $S \rightarrow S ; S$           | $L \rightarrow$      |
| $S \rightarrow id := E$         | $L \rightarrow L, E$ |
| $S \rightarrow \text{print}(L)$ |                      |
| $E \rightarrow id$              | $B \rightarrow +$    |
| $E \rightarrow \text{num}$      | $B \rightarrow -$    |
| $E \rightarrow E B E$           | $B \rightarrow *$    |
| $E \rightarrow S, E$            | $B \rightarrow /$    |

勘误：上文的  $L \rightarrow$  应该为  $L \rightarrow E$

这是我们在 lab1 中使用的语句。Expression sequence: S, E 也就是 statement 先算，然后再来计算这个表达式。

在这个里头，因为我们碰到了 id，id 可能会被赋值，执行到某个地方 id 的值可能会变掉，所以处理 id 变量的时候是用一个 table。Table 是不覆盖的，有一个新的值就把值插入在最前面，虽然在表中可能出现了好几次，但是我们只返回第一个找到的值。



```

class Table {
public:
 Table(std::string id, int value, const Table *tail)
 : id(std::move(id)), value(value), tail(tail) {}
 int Lookup(const std::string &key) const;
 Table *Update(const std::string &key, int val) const;

private:
 std::string id;
 int value;
 const Table *tail;
};

```

这个 `Table` 数据结构提供了查询、`update` 操作。

```

%scanner scanner.h /* include declarations of lex and error */
%filenames parser

%union { int num; string id; }

%token <num> INT
%token <id> ID
%token ASSIGN PRINT LPAREN RPAREN SEMICOLON

%type <num> exp
%left COMMA
%left PLUS MINUS
%left TIMES DIV

%start prog

```

```

class Parser {
...
private:
 Table _table;
...
}

```

然后我们来看我们的 `bison`, 第一行讲的是和 `scanner` 相关的东西。

第二个部分的东西是和我们 `parser` 相关的一些内嵌的 C++ 的代码。其他东西都一致, 和我们的文法相关。比如 `int` 和 `id` 都是一个 `terminal`, 还有一些 `terminal` 都是 `assgin`、分号、括号等。然后下面这个地方是 `exp`, 它是一个 `non-terminal`, `value` 是 `number`。然后我们定义了左结合的 `COMMA`, `+*/`, 并且乘除在最后, 说明其优先级最高。

```

%%%
prog : stm

stm : stm SEMICOLON stm
stm : ID ASSIN exp { _table.update($1, $3); }
stm : PRINT LPAREN exps RLPAREN { cout << "\n"; }

exp : exp { cout << $1; }
exp : exps COMMA exp { cout << $3; }

```

把 statement 规约为 program, 是没有 action 的。Action 都在每个单独的 statement 里头, 在我们赋值的时候, id 有一个新的 value, 要去修改这个表, \$1 的 value 就是 string, \$3 的 value 就是 number。碰到 print 的时候, 里面已经打印过了, 外面只需要再打印一个回车即可。注意我们这里的 action 是有副作用的。某些 semantic action 不该发生的发生了, 或者应该发生了但是没发送。在 parser 到 exps COMMA exp, 我们只需要打印新的 expression 就可以了, 因为前面的 exps 的东西已经都在之前被打印过了。

```

exp : INT { $$ = $1; }
exp : ID { $$ = _table.lookup($1); }
exp : exp PLUS exp { $$ = $1 + $3; }
exp : exp MINUS exp { $$ = $1 - $3; }
exp : exp TIMES exp { $$ = $1 * $3; }
exp : exp DIV exp { $$ = $1 / $3; }
exp : stm COMA exp { $$ = $3; }
exp : LPAREN exp RPAREN { $$ = $2; }

```

COMA 的最终结果是最后的 expression 的表达值。当我们在表达式中碰到 ID 的时候, 就要把对应 id 的值拿出来, 然后赋值给 expression 的 semantic value。这样我们一边做语法分析, 一边打印出中间的结果。

## Error Recovery with Side Effects Semantic Actions

---

- Error recovery may pop states from the stack
- It may result “impossible” semantic actions

```
statements : statements exp SEMICOLON
 | statements error SEMICOLON
 | /* empty */
exp : increment exp decrement
 | ID
increment : LPAREN { nest += 1 ; }
decrement : RPAREN { nest -= 1 ; }
```

Side-effect 对 side recovery 是有影响的，比如我们出错的时候，需要把栈上的一系列东西 pop 掉，找一个 synchronizing token，比如找到左括号，其后前的全部 pop 掉。我们在 semantic action 中，遇到左括号就 nest+1，遇到右括号就 nest-1，在出现 error 的时候，很多东西 pop 掉了可能导致到最后左右括号就不一致了。所以我们最好的方法是在 semantic action 的时候不要做有副作用的动作。

## Error Recovery with Side Effects Semantic Actions

---

- Whenever a semicolon is reached, the value of **nest** should be zero
  - Because increment and decrement are balanced
- If a syntax error is found after some left parentheses have been parsed
  - The states will be popped from the stack without “completing” them
- Best solution
  - Side-effect-free semantic actions

所以我们要选用 side-effect-free 的 semantic action。

### Burke-Fisher in ML-Yacc

---

ML-Yacc provides additional support for Burke-Fisher

- to continue parsing, we need semantics values for inserted tokens

```
%value ID {make_id "bogus"}
%value INT {0}
%value STRING {"“”"}
```

这个插入的时候是一个 token, token 有时候需要 semantic value, 我们就可以使用%value 的方式去说明这些在 parser 中插入的 token 的 value 是什么。

## Syntax Tree Node for Tiger

---

```
class Var {
public:
 int pos_;
 * some interface for subclass to implement *
}
class Exp {
public:
 int pos_;
 * some interface for subclass to implement *
}
```

下面我们就来说 tiger 中的 Syntax Tree, 在 lab3 中, 我们需要一边做 parser, 一边生成 syntax tree。比如我们这里有一个 Var, 这个变量是 tiger 中的变量, 它是一个类, 并且有几个子类, 这些子类公共的部分只有一个 pos\_, 其他东西都在我们的子类中。变量的子类有三个, 而表达式的子类有一大堆。

先要表明我们的 expression 在源文件的哪一个位置上, 这个东西主要是在我们需要报错的时候, 在第五章我们做类型检查的时候, 如果发现要把字符串赋值给一个整数, 这就会报错, 我们需要报出它在哪个位置上出了错, 大家在做 lex 的时候这件事很简单, 因为 lex 是一边扫描一边定位。而当我们 AST 都产生的时候, parser 已经扫描过一边文件了, 这时候如果我们使用扫描位置来报错, 那么都在文件结束的地方, 所以我们报错的时候需要这个来报出对应的位置。

我们要去做一个可变参的插入的事情。任何一个变量都在 symbol table 上做。所以简单变量最终要做的事情就是找到这个 table。

## Syntax Tree Node for Tiger

---

```
absyn::SimpleVar(int, sym::Symbol);
absyn::FieldVar(int, absyn::Var, sym::Symbol);
absyn::SubscriptVar(int, absyn::Var, absyn::Exp);

absyn::VarExp(int, absyn::Var);
absyn::NilExp(int);
absyn::IntExp(int, int);
absyn::StringExp(int, string);
absyn::CallExp(int, sym::Symbol, absyn::ExpList);
absyn::OpExp(int, absyn::Oper, absyn::Exp, absyn::Exp);
absyn::RecordExp(int, sym::Symbol, absyn::EfieldList);
absyn::SeqExp(int, absyn::ExpList);

absyn::AssignExp(int, absyn::Var, absyn::Exp);
absyn::IfExp(int, absyn::Exp, absyn::Exp, absyn::Exp);
absyn::WhileExp(int, absyn::Exp, absyn::Exp);
absyn::BreakExp(int);
absyn::ForExp(int, sym::Symbol, absyn::Exp, absyn::Exp,
 absyn::Exp);
absyn::LetExp(int, absyn::DecList, absyn::Exp);
absyn::ArrayExp(int, sym::Symbol, absyn::Exp, absyn::Exp);
```

Tiger 中只有变量和表达式，没有 statement。Symbol variable 和 field variable。第一个就是 pos，简单变量的话就是跟了 symbol。而 field variable，比如 a.f，其中是 Symbol 指的就是 f。SubscriptVar 的节点中，左儿子就是 a，右儿子就是一个整数表达式。

字符串、整数、function call、双目运算、record 都是表达式。

```
a := 5; a+1
absyn::SeqExp (int pos, absyn::ExpList exp);

absyn::SeqExp (2,
absyn::ExpList (
 absyn::AssignExp(4, absyn::SimpleVar(2, sym::Symbol("a")),
 absyn::IntExp(7,5)),
 absyn::OpExp(11, absyn::PLUS_OP,
 absyn::VarExp(new absyn::SimpleVar(10, sym::Symbol("a"))),
 absyn::IntExp(12,1))))
```

像这样的一个东西，这是我们第一节课中举的例子。

a=5; a+1 是我们 straight-line 语言中的 sequence expression，  
a 的位置是 2，冒号的位置是 4。

## Positions

---

- Keep track of the position in AST nodes
- How to get position in parser generator?
  - Add a position stack along with semantic value stack
  - Beginning and end positions of each token and phrase are available for the semantic actions
  - Bison parser generator can do this
  - Yacc does not

正如之前所说，在 AST 中，我们需要维护 pos 来便于报错。在 parser generator 的时候，有一个 DFA，它在做的时候就是在进栈，一开始我们进栈的时候，只进了一些 Symbol，因为 DFA 每次大家都要从头走，比较 naive，先进 terminal 和 non-terminal，然后再把 state 放在里面，然后因为我们要做 semantic action，所以我们又多了一些放入 semantic value，下面我们要做 position 的话，那么就再加一列。

| Symbol | State | Semantic Value | Position |
|--------|-------|----------------|----------|
|        |       |                |          |

所以 Bison 就帮我们做了这件事情，为了支持 pos，我们需要修改我们原先的数据结构。

内嵌文件中改为

```
extern absyn::OpExp(absyn::Exp, absyn::BinOp, absyn::Exp, position);
%union {int num; string id; position pos; ...};
%type <pos> pos
```

```
pos: { $S = errmsg_->GetTokPos(); }
exp: exp PLUS pos exp { $S = new absyn::OpExp($1, absyn::PLUS_OP,
$4,$3); }

exp: pos exp PLUS exp { $S = new absyn::OpExp($2, absyn::PLUS_OP,
$4,$1); }
```

我们还需要增加 position 的 semantic value。所以我们增加一个类型叫 pos，也就是一个 non-terminal 叫 position，它的 semantic value 就是 position。这样我们就可以在栈上拿到这个 position 的值。我们在表达式的时候，我们想把 position 放到什么地方，让 pos 代表什么位置，那么我们就在表达式中放到哪里。

## Mutually Recursive Functions

- Must be adjacent function declarations

```
let var a :=5
 function f() : int = g(a)
 function g(i: int) = f()
 in f()
end
```

介绍完了前面的变量和表达式之后，我们来看一些声明：变量声明、类型声明、函数声明，我们现在来看函数声明，我们需要让它支持递归调用。那么这个 `f` 和 `g` 在 `tiger` 中必须是挨着的（连续声明的），那么 `f` 和 `g` 才可以递归调用。这样定义我们处理起来稍微简单一点。

### Syntax Tree Node for Tiger

```
absyn::FunctionDec(int, absyn::FunDecList);
absyn::VarDec(int, sym::Symbol, sym::Symbol, absyn::Exp);
absyn::TypeDec(int, absyn::NameAndTyList);

absyn::NameTy(int, sym::Symbol);
absyn::RecordTy(int, absyn::FieldList);
absyn::ArrayTy(int, sym::Symbol);

absyn::Field(int, sym::Symbol, sym::Symbol);
absyn::FieldList(absyn::field, absyn::fieldList);
absyn::ExpList(absyn::Exp);

absyn::Fundec(int, sym::Symbol, absyn::FieldList, sym::Symbol,
 absyn::Exp);
absyn::FundecList(absyn::FunDec);
absyn::DecList(absyn::Dec);
absyn::NameAndTy(sym::Symbol, absyn::Ty);
absyn::NameAndTyList(absyn::NameAndTy);
absyn::EField(sym::Symbol, absyn::Exp);
absyn::EFieldList(absyn::EField);

enum Oper {PLUS_OP, MINUS_OP, TIMES_OP, DIVIDE_OP,
 EQ_OP, NEQ_OP, LT_OP, LE_OP, GT_OP, GE_OP}
```

一开始我们要定义一个 `function declaration list`, `FuncDec` 是我们真正的函数声明，函数有函数名、参数、返回类型、函数体。`Record` 中一般都有 `a:int, b:int, c:string`。返回类型也是一个 `symbol`，因为在 `tiger` 中是支持变量声明的，是可以通过 `type` 来声明一个类型名。第四个参数就是函数体就是一个表达式。

## AST Builder for straight-line programs

---

```
%{ #include "absyn.h" %}

%union {int num; string id; absyn::stm stm; absyn::Exp exp;
 absyn::ExpList expList}
%token <num> INT
%token <id> ID
%token ASSIGN PRINT LPAREN RPAREN SEMICOLON
%type <stm> stm prog
%type <exp> exp
%type <expList> exps
%right COMMA
%left PLUS MINUS
%left TIMES DIV
%start prog
2021/10/15
```

我们在这里声明了一些 semantic value，哪些是 token，哪些是 non-terminal，和之前说过的都是一样的，只是最后的 semantic value 会发生一些变化。

```
%%

prog : stm {$$=$1; }

stm : stm SEMICOLON stm {$$ = new absyn::CompoundStm($1, $3);}
stm : ID ASSIN exp {$$ = new absyn::AssignStm($1, $3); }
stm : PRINT LPAREN exps RLPAREN {$$ = new absyn::PrintStm($3); }

exps : exp {$$ = new absyn::ExpList($1); }
exps : exp COMMA exps {$3->Prepend($1); }

exp : INT {$$ = new absyn::NumExp($1); }
exp : ID {$$ = new absyn::IdExp($1); }
exp : exp PLUS exp {$$ = new absyn::OpExp($1, absyn::PLUS_OP, $3); }
exp : exp MINUS exp {$$ = new absyn::OpExp($1, absyn::MINUS_OP, $3); }
exp : exp TIMES exp {$$ = new absyn::OpExp($1, absyn::TIMES_OP, $3); }
exp : exp DIV exp {$$ = new absyn::OpExp($1, absyn::DIV_OP, $3); }

exp : stm COMMA exp {$$ = new absyn::EseqExp($1, $3); }
exp : LPAREN exp RPAREN {$$ = $2; }
```

剩下的事情就差不多，现在我们构造了一棵树。

```

let var a :=5
 function f() : int = g(a)
 function g(i: int) = f()
in f()
end

```

我们再回到这个函数递归调用的情况，我们来看一下这个 tiger 程序。

```

absyn::LetExp(
 absyn::DecList(
 absyn::VarDec(sym::Symbol("a"), NULL, absyn::IntExp(5)),
 absyn::FunctionDec(absyn::FuncdeclList(
 absyn::Fundec(sym::Symbol("f"), NULL, sym::Symbol("int"),
 absyn::CallExp(sym::Symbol("g", ...)),
 absyn::Fundec(sym::Symbol("g"),
 absyn::FieldList(sym::Symbol("i"),sym::Symbol("int")),
 NULL,
 absyn::CallExp(sym::Symbol("f"),NULL)))),
 absyn::CallExp(sym::Symbol("f"), NULL)))
)
)

```

所以一开始是 LetExp 然后是 DecList, Call 的时候因为没有参数，所以第二个参数为 NULL。 Dec 中包括了两部分，一个是 a 的变量声明以及指明其类型，因为一开始没有指明 a 的类型，所以 a 的第二个参数类型为 NULL，到第五章中，需要根据赋的初值的类型补回 a 的类型。 回过来看我们的 dec 有函数声明、变量声明和类型声明。

我们接下来来看 FunctionDec。里面是一个 FuncDeclList，里面有 f 和 g 的函数声明。f 的参数为空、返回值为 int。而 g 是有参数。

### Must be adjacent type declarations

```

type tree = { key: int, children: treelist }
type treelist = { head: tree, tail: treelist}

```

```

absyn::TypeDec(
 absyn::NameAndTyList(
 absyn::NameAndTy(sym::Symbol("tree"),
 absyn::RecordTy(
 absyn::FieldList(
 absyn::Field(sym::Symbol("key"),
 sym::Symbol("int")),
 absyn::Field(sym::Symbol("children"),
 sym::Symbol("treelist"))))),
 absyn::NameAndTy(sym::Symbol("treelist"),
 absyn::RecordTy(
 absyn::FieldList(
 absyn::Field(sym::Symbol("head"),
 sym::Symbol("tree")),
 absyn::Field(sym::Symbol("tail"),
 sym::Symbol("treelist"))))))))

```

## Other Syntax Tree Design Issues

---

- No AST for "&" , "|" and unary "-" expressions
  - $e1 \& e2$  is translated into  $\text{if } e1 \text{ then } e2 \text{ else } 0$
  - $e1 | e1$  is translated into  $\text{if } e1 \text{ then } 1 \text{ else } e2$
  - $-i$  is translated into  $0-i$
  - Keep the AST data structure smaller
  - Make fewer cases for the semantic analysis phase to process
  - Harder for the type-checker to give meaningful error messages that relate to the source code

有些 ast 中有的，我们 ast 中不表示，直接处理掉了。

## 2021/10/19

上节课我们提到了 AST，比如变量和 function 怎么表示。在生成 AST 的时候生成了很多接口。但是 tiger 中还有很多语法运算没有用 AST 表达出来，比如逻辑运算或者单目运算符负号。我们在做 parser 的时候要做一个变换， $e1 \& e2$  可以被翻译成  $\text{if } e1, \text{ then } e2, \text{ else } 0$ ， $e1 | e2$  可以被翻译成  $\text{if } e1 \text{ then } 1 \text{ else } e2$ 。 $-i$  可以被翻译成  $0-i$ ，但是这样我们就不能表示 2's complement 的最小数了。这里强调的是并不是每个语法结构都需要一个 AST，而是可以转化成别的表达来减少我们 AST 的类型，AST 的类型少了，我们在马上要做的语义分析里头，也会减少一些工作。我们变化以后，如果真的检查出错误来，可能就和原来不太相关了。

我们再提一下，在 AST 中有很多 identifier，表现形式就会有很多符号。

```

let type a = int
 var a : a := 5
 var b : a := a
 in b + a
end

```

- **Symbol "a" denotes**

- The type "a" in syntactic contexts where type identifiers are expected
- The variable "a" in syntactic contexts where variables are expected

有很多符号 a 和 b，出现的时候都是字符串，在我们的 tiger 中，要把字符串处理掉，我们并不是直接来存字符串，而是用指针指向字符串的位置。我们通常用 Symbol 来表示字符串，这样它的大小就统一了，是类似指针一样的东西。可以通过这种指针拿回字符串。

- **Semantic analysis needs to keep track of**
  - which local variables are used from within nested functions
- **Field type is used for both formal parameters and record fields**
  - **absyn::FunDec**(int pos, sym::Symbol name,  
absyn::FieldList params, sym::Symbol result, absyn::Exp body);
  - **absyn::FieldList**(absyn::Field head);
  - **absyn::Field**(int pos, sym::Symbol name, sym::Symbol type);

在第五章很快就会介绍 symbol table，是记录符号的数据结构。它要把嵌套的、类型和变量分开。FuncDec 怎么用 AST 来表示，它的 node 有很多

1. 参数，在其中我们需要考虑参数名和参数类型
2. 返回值的类型
3. 函数名
4. 函数体是什么

所以 FunDec 中就有 name, FieldList(形参), result(返回类型), exp (函数体，所有计算的代码都是表达式)。因为我们在 semantic analysis 的时候需要方便我们报错，所以还有一个 pos 记录当前的位置。

每一个 Field 其中就是一个参数名、一个参数类型。

在我们 parser 的时候，把所有东西都当字符串处理存进去。但是我们的类型，比如 int 是有意义的，在生成 AST 的时候，都当字符串处理。在我们语义分析的时候，我们再把这些有意义的字符串提取出来处理。

我们还要提一下，在 AST 里头，当我们到 register allocation 的时候，有些变量是要进寄存器的，还有一些变量是不能进寄存器的，比如 C 语言里，我们希望取变量的地址，那么这个变量不能放到寄存器了。又比如我们是 struct 变量、数组变量，都是不能放到寄存器里的。在 tiger 里头，虽然没有指针，但是有 record 和 array，这种变量都需要放到内存里，而不能

放到寄存器里。

```
f(x){
 int a;
 g(int y){
 = a + x;
 }
}
```

Tiger 还有函数嵌套，也就是在 f 的声明里头可以声明另一个 g，在 f 里面 g 前声明了 a，那么在内函数 g 中可以使用到 a。这种内函数调用的外函数的变量 a、x，也不能放到寄存器里，因为进到里面的函数的时候，会更新寄存器，我们需要把 a、x 放到内存中。

### The **escape component** of a **varDec** or **field** is used to keep track of this

- absyn::VarDec(int pos, sym::Symbol var, sym::Symbol type, absyn::Exp init);
- Not mentioned in the constructors
- All initialized to TRUE
- escape has meaning only for formal parameters

在 tiger 的 AST 里头，一个变量声明的时候有一个 escape，说明是不是要放到寄存器里去。Exp init 是初始化的时候的表达式，目前这个 VarDec 中是没有 escape 变量的。

AST 我们讲完了，前面构造 AST 都是使用 Bison 来构造的，在每个 production 的最后，有一个 semantic action，在规约的时候我们生成一个 AST。我们在 top-down parsing 的时候，这件事情是怎么做的。

- CFG and its semantic actions
- $E \rightarrow E_1 + T \quad \{ E.val = E_1.val + T.val \}$
- $E \rightarrow E_1 - T \quad \{ E.val = E_1.val - T.val \}$
- $E \rightarrow T \quad \{ E.val = T.val \}$
- $T \rightarrow (E) \quad \{ T.val = E.val \}$
- $T \rightarrow \text{num} \quad \{ T.val = \text{num}.val \}$

我们使用 CFG 来定义这个语法，定义完我们就知道了每个 production 代表什么含义，这个例子是支持加减、括号的计算器。而我们现在 topdown 是 LL 文法，不支持左递归，所以我们先要消除左递归。

| Production                | Semantic action                   |
|---------------------------|-----------------------------------|
| $E \rightarrow E_1 \pm T$ | $\{ E.val = E_1.val \pm T.val \}$ |
| $E \rightarrow T$         | $\{ E.val = T.val \}$             |

- $E \rightarrow T \{ R.i = T.val \} R \{ E.val = R.s \}$
- $R \rightarrow + T \{ R_1.i = R.i + T.val \} R_1 \{ R.s = R_1.s \}$
- $R \rightarrow - T \{ R_1.i = R.i - T.val \} R_1 \{ R.s = R_1.s \}$
- $R \rightarrow \epsilon \quad \{ R.s = R.i \}$

消除左递归后，我们会得到  $E \rightarrow TR$ ，那么消除左递归后，semantic 该怎么办呢？它就会变得复杂，我们先碰到  $T$ ，算完  $T.value$  以后，我们要去处理这个  $R$ ，但是第一个  $T$  的这个值对  $R$  是有用的，我们需要根据  $T$  和  $R$  的 value 来算出  $E$  的 value。

所以在这边，我们给  $R$  两个 semantic value， $R.i$  和  $R.s$ 。如果  $R$  为空，那么把  $R.i$  赋给  $R.s$ ， $R.s$  是  $R$  的最终的值，然后我们通过第一行的“ $E.val = R.s$ ”就可以把右边最终的值赋值给  $E$ 。

## Inherited & Synthesized Attributes

---

- Inherited Attribute
  - The value of an attribute at a node is computed from values of the attributes at the siblings and parent of that node in the parse tree
  - e.g.  $R.i$
- Synthesized Attributes
  - The value of an attribute at a node is computed from values of the attributes at the children of that node in the parse tree
  - e.g.  $T.val, E.val, R.s$

Attribute 分为两类， $R.i$  的特征是由它的 parent 和 sibling 来计算的，而  $R.s$  的特点是它的值都是由 children 来计算出来的。

我们 Parser 生成 AST 都需要把 semantic action 在最后，是 bison 可以处理的东西，而 action 写在中间的情况下，这叫做 Translation Scheme（可以出现在 production 的任意地方），bison 可能不能处理这种。

## Top-Down Translation

$$E \rightarrow TR \mid T \quad R \rightarrow + TR \mid - TR \mid \epsilon \quad T \rightarrow (E) \mid \text{num}$$

```
void R()
{
 if (lookahead == addop) {
 eat(addop);
 T();
 R();
 }
}
```

$$R \rightarrow + T \{ R_1.i = R.i + T.val \} R_1 \{ R.s = R_1.s \}$$

对于每个 non-terminal 都有一个函数，函数体其实就是把 production 写进去。对于每个 production 有 terminal 和 non-terminal，遇到 terminal，我们就使用 eat 把对应的 terminal 吃掉，继续往后走，对于每个 non-terminal，就调用对于的名字。这个只能判定是否正确，不能做动作、不能生成 AST，我们要做动作的话，我们就要把 translation scheme 放进去。

我们先来看红色的，分成 inherited attribute  $R.i$  是  $R$  的初值，而  $R.s$  是  $R$  的终值，所以我们做如下处理：

- 每个 non-terminal 还是对应一个函数，但是这个函数是有参数的，我们把 inherited attribute 作为函数传入的，把 synthesized attribute 就作为函数的返回值。

其中用到的 attribute，都作为这个函数的局部变量，下面我们来改写一下上面的图，改写成我们想生成一个 AST。

$$\begin{array}{ll} R \rightarrow & \text{addop} \\ & T \quad \{R_1.i = \text{makenode}(addop.lexeme, R.i, T.nptr)\} \\ & R_1 \quad \{R.s = R_1.s\} \\ R \rightarrow & \epsilon \quad \{R.s = R.i\} \end{array}$$

```
syntax_tree_node* E(void);
syntax_tree_node* R(syntax_tree_node*);
syntax_tree_node* T(void);
```

第一步规约完了以后，我们把  $R_1$  的 tree node 给成  $R$  的 tree node。

```

SyntaxTreeNode* R(syntax_tree_node* i)
{
 syntax_tree_node* t_nptr, ri, rs, s;
 char addoplexeme;
 if (lookahead == addop) {
 addoplexeme = lexval ;
 eat(addop);
 t_nptr = T();
 ri = mknnode(addoplexeme, i, t_nptr);
 rs = R(ri) ;
 s = rs;
 } else
 s = i ;
 return s ;
}

```

刚才我们提到 `R` 有两个 attribute，一个作为参数、一个作为返回值。当我们看到加号的时候，把加号吃掉，然后我们调用 `T`，它会生成一个 `T` 的语法树。我们把这个加一下，构成一棵新的语法树。调用 `R` 以后，返回值是 `R.s` (synthesized attribute)，这里也就是，如果在产生式中有 non-terminal 的情况下，会有参数传进去。

### How to write the parsing code by hands? (3)

---

- generating an assignment  $c = B(b_1, b_2, \dots, b_k)$  for each nonterminal symbol  $B$ , where  $b_1, b_2, \dots, b_k$  are the variables for the inherited attributes of  $B$  and  $c$  is the variable for the synthesized attribute of  $B$
- copying the action code into the parsing code and replacing the reference to an attribute by the variable for that attribute

我们把 `T` 和 `R` 之间的 action 直接 copy 过来。这就是 Top-down parsing，一般现在我们都用 `bison` 来做，不会牵涉到这个 top-down parsing 生成 AST。

## Symbol Table

我们现在开始讲第五章的 Symbol Table，它里面还有一个 Symbolic Table。基于 Symbolic Table 我们构造了 Symbol Table。为什么要有 Symbol Table 呢？在第四章中，我们把所有东西都放进到了 AST 里去了，我们在做语法分析的时候生成了 AST，在生成 AST 的时候能不能搞清楚  $x$  是一个变量、类型是 int 呢？也不是不行。但是我们现在语法分析的时候，尽量用 `bison` 处理 semantic action。如果 semantic action 太复杂跑到中间去了，`bison` 可能就不能处理这件事情了。

现在第五章还是前端的部分，前端的目的是把源文件变成一个中间表示，AST 是中间表示的一部分。我们 lab1 里，除了有 AST 以外还有 Symbol Table，变量是要写到 Symbol Table 里头的，所以我们在表示 front end 的时候，需要有 tree 和 symbol table。其包含定义了哪些

变量、类型是什么，我们还需要表示出变量的有效范围(scope)这件事情。比如 let D in E end，D 中声明的变量的有效范围只有在 E 中。一个 Symbol Table 里面要存所有变量及其类型：

- Binding gives a symbol a meaning.
- Denotes by  $\mapsto$  arrow
- An environment is a set of bindings
- An environment  $\sigma_0$  contains the bindings  $\{g \mapsto \text{string}, a \mapsto \text{int}\}$ , meaning that
  - the identifier a is an integer variable
  - the identifier g is a string variable

我们把 D 声明的变量叫做一个 environment，它包含了很多变量的声明。在程序中 environment 就有可能嵌套：

- ```

1. function f(a:int, b:int, c:int) =
2.   (print_int(a+c));
3.   let var j := a+b
4.     var a := "hello"
5.     in print(a); print_int(j)
6.   end;
7.   print_int(b)
8. )
  
```
- σ_1
 - On line 1, the formal parameters give us a new table σ_1 equals to
 - $\sigma_1 + \{ a \mapsto \text{int}, b \mapsto \text{int}, c \mapsto \text{int} \}$
 - The identifiers in line 2 can be looked up in σ_1
 - σ_2
 - At line 3, $\sigma_1 + \{ j \mapsto \text{int} \}$
 - Suppose at the very beginning the environment is σ_0
 - σ_3
 - At line 4, $\sigma_2 + \{ a \mapsto \text{string} \}$

最早的 environment 是参数 env。如果进入函数体之前，外面的 env 叫做 σ_0 。函数的参数定义是在 σ_1 。后面一行 `print_int(a+c)`，a 和 c 会先去找 σ_1 ，找不到的情况下才会找 σ_0 ，再找不到就报错。在第三行开始，又开始一个新的 environment σ_2 ，并且声明了 a 把之前的 a 覆盖掉了。注意第三行的 a 还是 σ_1 中的 a。所以，我们就会发现在扫描我们语法树的时候，environment 会变化。

Environment 会起什么作用？

- In semantic analysis
 - The symbol table is searched every time a name is encountered
 - Changes to the table occur if a new name or new information about an existing name is discovered
- In data layout
 - The symbol table serves for providing the input information and keeps the results which can be used by code generation
- In code generation
 - Symbol table can be used to handle names and labels

Scope information, 在后序扫描（先扫描儿子再扫描父亲）语法树的时候，我们扫描到父节点的时候，environment 就变掉了，我们在语义分析的时候有两种处理方式，

- 1.functional style, 每次发生变化的时候老的都保留着。
- 2.imperative style, 有了新的就看不到老的，但是我们退出 scope 的时候还能回得去。还有些语言支持多个 symbol table。

Multiple Symbol Tables

```

package M;           σ1 = {a ← int}
class E {           σ2 = {E ← σ1}
    static int a = 5;
}
class N {           σ3 = {b ← int, a ← int}
    static int b = 10;
    static int a = E.a + D.d;
}
class D {           σ5 = {d ← int}
    static int d = E.a + N.a;
}
σ7 = σ2 + σ4 + σ6

```

第一遍扫描只是构造出这个 symbol table，第二遍扫描扫描引用，看到哪个引用就去查对应的 symbol table。

接下来，我们来看一下怎么实现 symbol table。Tiger 中用的是一个 open-hash。

```

typedef struct bucket {
    std::string key_;
    void *binding_;
    struct bucket *next_;
    struct bucket(std::string key, void *binding, struct bucket
        *next) : key_(key), binding_(binding), next_(next) {}
} bucket_t;

constexpr unsigned int SIZE = 109;
struct bucket *table[SIZE];

void Insert(std::string key, void *binding)
{
    int index = hash(key) % SIZE ;
    table[index] = new struct bucket(key, binding, table[index]);
}

void Pop(std::string key)
{
    int index = Hash(key) % SIZE ;
    table[index] = table[index]->next_ ;
}

```

```

void *Lookup(std::string key)
{
    int index = hash(key) % SIZE ;
    struct bucket *b ;
    for (b=table[index]; b; b=b->next_)
        if (0==b->key_.compare(key)) return b->binding_ ;
    return NULL ;
}

```

它就声明了一个开链的 hash，其中有一个 key，有一个 binding。Pop 操作，给定一个 key，把链里头的第一项给它拿掉，相当于一个 delete。

2021/10/22

Symbol Table 类似是一个 hashmap，把变量和函数定义为原先的类型。

Imperative style 就是直接插入。

Functional Symbol Tables

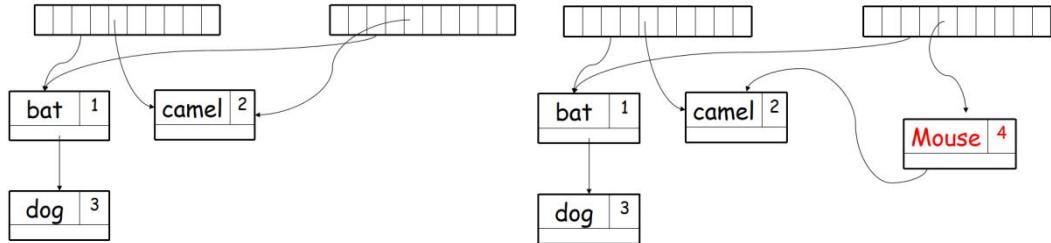
- In imperative style
 - Alter a table by adding a binding to it
- In the functional style, we wish
 - To compute $X + Y$
 - We still have X available to look up identifiers
 - So, we create a new table by computing the "sum" of an existing table and a new binding

我们在 imperative 中写的是 $X = X + Y$ ，算完就把原先的 X 覆盖了，而在 function 中就不存在覆盖的情况，如果要赋值就创建一个新的值，也就是 function-style 是有“不可更改性”。这个方法的好处就是可以控制对一个变量的修改，那么我们很容易控制一个变量有多个读者而只有一个写者。

对应到 Symbol Table 中， X 和 Y 对应是两个 Symbol Table，编译器的分析中有不同的 Scope，从父函数进入到子函数的 Scope 中，会声明一些新的 declaration、定义一些新的（局部）变量，这就是一个不断叠加的过程。在 imperative 中，全程只有一个 Symbol Table，进入的时候更新 Symbol Table，退出的时候做减法。而在 Function 中， X 和 Y 和 $X+Y$ 都存在。

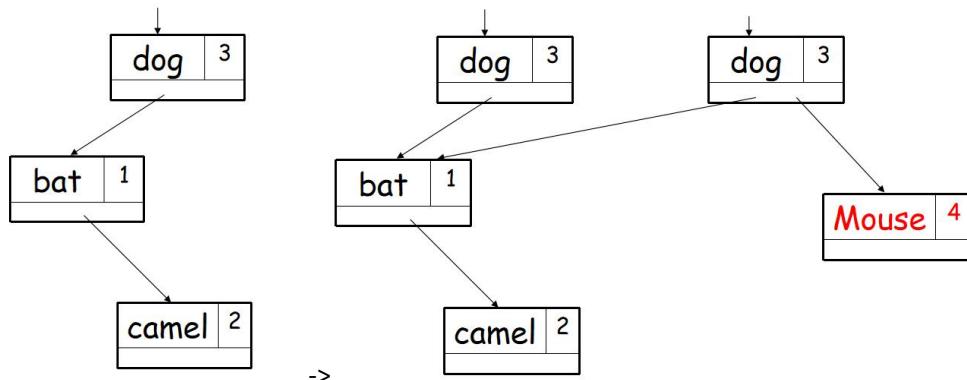
在 functional 里面，hash 的效率可能比较差。

非覆盖的更新对于哈希实现是比较低效率的。当我们想插入 binding 的时候，我们可以复制 array，不过 bucket 可以共享，然后再加入一个新的 binding。



左图是插入前复制完的情况，而右图就是插入完的情况，插入了 Mouse 的情况。如果我们把新创建的 Array 删除了，那么原先的 Array 还可以使用。

这个教材提出不要使用哈希表来避免 copy，它提出的二分搜索树。左子树小于当前节点、右子树大于当前节点。



使用二叉搜索树的好处是我们不需要 copy 整个 hash array，我们只需要复制个别节点就可以了，当我们需要插入一个新的节点在第 n 层的时候，只需要 copy 上面的 n-1 层节点即可。

为什么我们要复制一遍 dog 节点呢？对于 dog 来说原先左儿子是 bat，右儿子是 null；而更新完 dog 左儿子是 bat，右儿子是 mouse。dog 节点左右节点不一样了，我们可以理解为 copy on write 的操作，所以我们需要复制一份 dog。

二叉搜索树的深度不会很高，所以每次更新的效率也在 $\log N$ ，可以接受。

Symbols in the Tiger Compiler

- Using pointer instead of string
- Using destructive-update hash table to map strings to symbols
 - All the occurrences of a single string will be mapped to a single symbol

Symbol Table 是装 table 的。我们可以把 string 换成引用。所有的 string，如果内容一样应该 map 到同一个 symbol。这里给了一个静态方法 unique_symbol

```

struct Binder {
    KeyType *key;      ValueType *value;
    Binder *next;      KeyType *prevtop;
    Binder(KeyType *key, ValueType *value, Binder *next, KeyType *prevtop)
        : key(key), value(value), next(next), prevtop(prevtop) {}
};


```

Tables in the Tiger Compiler

- Imperative style method is implemented in **tab::Table** using hash tables

```

static constexpr unsigned long TABSIZE = 127;
namespace tab {
template <typename KeyType, typename ValueType> class Table {
public:
    /* Some public methods definition */
protected:
    /* Definition of Binder */
    Binder *table_[TABSIZ];
    KeyType *top_;
};

}


```

Tables in the Tiger Compiler

- tab::Table implements following functions**
- Constructor, Enter, Set, Look, and Pop

```

Table() : top_(nullptr), table_() {}
void Enter(KeyType *key, ValueType *value);
ValueType *Look(KeyType *key);
void Set(KeyType *key, ValueType *value);
KeyType *Pop();


```

这个 **table** 里存的一个是 **binder** (**bucket** 的数组)。这个 **table** 就是我们比较熟悉的哈希表，增删改查都齐了。

- The stack can be integrated into the Binder
 - Having a protected member "top_"
 - When pushing (entering), top is copying into prevtop field in the Binder
- Symbol table
 - the TAB is wrapped

```
namespace sym {
    class Table : public tab::Table<Symbol, ValueType>
}
```
- A mark symbol is introduced


```
Symbol marksym_ = {"<mark>", nullptr};
```

为什么要引入 top 呢？因为在 Symbol Table 有嵌套的 Scope, eg:

```
{
    int a = 10;          [a->int]
    {
        double a = 20;      [a->double]
        //这里要用到 a 的话，肯定得到的是 20，此时用到的 a，其 double 要覆盖 int
    }
}
```

我们分析到红色}的时候，我们需要把 `double` 重新改为 `int`，方法一是我们往上回溯，然后一个个去掉，教材中的方法是我们记录到上一个{是在哪开始的，记录其开始的位置，我们不断 `pop`，这样就可以把它清空了。

所以实际上[a->int]和[a->double]是相连的，在 `pop` 的时候，我们会先 `pop` [a->double]，然后再去 `pop` [a->int]。如果我们只是这样做，我们其实不知道 `pop` 到哪结束。这里就加了新的 symbol 也就是<mark> symbol 来表示我们开始了。所以在具体实现的时候，会是这样的：[a->int]->[<mark>->null]->[a->double]，我们在 `pop` 的时候，`pop` 直到 mark。

```
namespace sym {
    Table() : tab::Table<Symbol, ValueType>() {}
    /* Because of template, no need to implement methods
     * Enter, Pop, Look, Set and Constructor again.
     */
    template <typename ValueType> void Table<ValueType>::BeginScope() {
        this->Enter(&marksym_, nullptr);
    }
    template <typename ValueType> void Table<ValueType>::EndScope() {
        Symbol *s;
        do {
            s = this->Pop();
        }
        while (s != &marksym_);
    }
}
```

BeginScope 的实现，就是把 Mark symbol 插入到表中，而 EndScope 就是如果不是 mark symbol 的话就不断 Pop.

- The undo stack must be provided
- A beginscope operation pushes a special marker onto the stack
- endScope operation pops off the stack down to and including the topmost marker
- As each symbol is popped, the head binding in its bucket is removed

然后我们来看一下 binding，每一个 symbol table 中都有一个 binding，也就是映射中的 T 要对应一个值，在 tiger 中把类型的命名空间和函数、变量的命名空间是分开的。

```
let type a = int
    var a : a := 5
    var b : a := a
    in b + a
end
```

- Symbol "a" denotes
 - The type "a" in syntactic contexts where type identifiers are expected
 - The variable "a" in syntactic contexts where variables are expected

比如上图中的定义在 tiger 语言中是允许的，即类型系统的名字和变量系统的名字是分开的。那么我们怎么判断呢？如果一个指令是合法的语法指令，我们是可以判断出来的。

比如 VAR ID : ID := exp，那么我们是可以知道哪个是哪个的。

所以我们需要搞两个 Symbol Table，一个做变量检查，一个做类型检查。在后面做语义分析的时候，就会越来越多地用到这样的类型。

- For a type identifier, we only remember the type it stands for
 - Type environment maps a symbol to type::Ty
 - That is, when looking up the sym::Table, it always returns a type::Ty pointer

```

namespace type {
    class TyList;
    class Field;
    class FieldList;

    class Ty {
        public:
            virtual Ty *ActualTy();
            virtual bool IsSameType(Ty *);
        protected:
            Ty() = default;
    };
}

class TyList {
    private:
        std::list<Ty *> ty_list_;
};

class Field {
    public:
        sym::Symbol *name_;
        Ty *ty_;
};

class FieldList {
    private:
        std::list<Field *> field_list_;
};

```

这里大概展示了一些实现，这里是一些抽象的定义。这里要再讲第一下，`expList` 左节点是一个 `exp`，右节点是一个 `expList`，这在树中是很自然的递归类型，但是在 C++ 中不太好做，所以我们上右图中给出的是 `std::list`。所以我们整体的 `list` 结构都统一成右图这样。

Type Equivalent

```

let type a = { x: int, y:int }
type b = { x: int, y:int }
var i : a := ...
var j : b := ...
in i := j
end /*illegal */
let type a = { x: int, y:int }
type c = a
var i : a := ...
var j : c := ...
in i := j
end /* illegal */

```

在 tiger 中，哪怕名字不同、也是不相同的；但是类型别名的情况是可以这样做的。我们刚才说了变量和类型的 namespace 是分开的。

Environments

- Type of the `tEnv_` (type environment) is `TEnv`

```

namespace env {
...
using TEnv = sym::Table<type::Ty>;
...
}

```

- `tEnv_` initially maps

- the symbol "int" to `type::IntTy`
- the "string" to `type::StringTy`
- use `FillBaseTEnv` to initialize

`TEnv` 中，其实 key 就是 `Symbol`，而 value 是 `type::Ty` 结构。我们有预定的类型去做初始

化。

上节课讲到了 Type，这节课我们讲 value binding，其分为变量和函数。

Value Bindings in the Tiger Compiler

- For a value identifier, it may be
 - A variable or a function
- If a variable, we need to remember
 - What is its type
- If a function, we need to remember
 - What are its parameter types
 - What is the result type

变量记录的是类型，而 function 要记录的是参数类型和结果的返回值类型。我们引入 environment 的这个概念，里面也是哈希表的结构。

```
namespace env {
    class EnvEntry {
        public:
            bool readonly_;
            explicit EnvEntry(bool readonly = true) :
                readonly_(readonly) {}
            virtual ~EnvEntry() = default;
    };
}
```

为什么需要 readonly_ 呢？在 for 循环中我们可以定义 i=1，后面我们就不能赋值 i=3 了，tiger 中不支持在循环体中对循环变量赋值，所以这里我们为这种情况添加了 readonly_ 变量，如果我们尝试对 readonly_ 变量赋值，那么会报出一个 readonly_ 的语义错误。

Tiger 中是不允许隐式的类型转换的，但是在别的语言中很常见，我们可以定义一个 EnvEntry e = true，因为 EnvEntry 说白了就是一个 readonly 和一个变量。我们这样写就会隐式地转化为了 EnvEntry e = EnvEntry(true)；为了避免这样的隐式转换，我们这里加了一个 explicit，要求代码必须显式的调用这个构造函数。

```
class VarEntry : public EnvEntry {
    public:
        tr::Access *access_;
        type::Ty *ty_;
        /* Constructor */
    };
    class FunEntry : public EnvEntry {
        public:
            tr::Level *level_;
            temp::Label *label_;
            type::TyList *formals_;
            type::Ty *result_;
            /* Constructor */
    };
}

• Type of the venv_ (value environment) is VEnv
namespace env {
    ...
    using TEnv = sym::Table<type::Ty>;
    using Venv = sym::Table<env::EnvEntry>;
    ...
}

• venv_ contains the bindings for predefined
functions
- flush, ord, chr, size, ....
- use FillBaseVEnv to initialize.
```

Type Checking

```
class Ty {
public:
    virtual Ty *ActualTy();
    virtual bool IsSameType(Ty *);
protected:
    Ty() = default;
};

SubClass:
NilTy, IntTy, StringTy, VoidTy, RecordTy, ArrayTy, NameTy
```

函数为空的时候是 VoidTy, NilTy 是 record 类型, 后面几种就是真正的类型消息。NameTy 是类似一个占位符。

Type Checking Expressions

```
class Exp/Dec/Var {
...
virtual type::Ty *SemAnalyze(
    env::VEnvPtr venv,
    env::TEnvPtr tenv,
    int labelcount,
    err::ErrorMsg *errormsg) const = 0;
...
}
```

不同的类型, 我们都定义了这样一个 SemAnalyze 的方法, 任务就是对不同的类型写一个 semantic analysis。

Type Checking Expressions

```
type::Ty *OpExp::SemAnalyze(...) const {
    type::Ty *left_ty = left_->SemAnalyze(...)->ActualTy();
    type::Ty *right_ty = right_->SemAnalyze(...)->ActualTy();
    if (oper_ == absyn::PLUS_OP || oper_ == absyn::MINUS_OP
    || oper_ == absyn::TIMES_OP || oper_ == absyn::DIVIDE_OP) {
        if (typeid(*left_ty) != typeid(type::IntTy)) {
            errormsg->Error(left_->pos_, "integer required");
        if (typeid(*right_ty) != typeid(type::IntTy)) {
            errormsg->Error(right_->pos_, "integer required");
        return type::IntTy::Instance();
    } else {
        if (!left_ty->IsSameType(right_ty)) {
            errormsg->Error(pos_, "same type required");
            return type::IntTy::Instance();}}
        return type::IntTy::Instance();
    }
}
```

这里展示了一个 Opexp, 左边要通过 semantic analysis 拿到真实的类型。然后我们判

`operation` 是不是这四种，`+*/`，别的不支持，如果是这四种，那么我们进入，并且我们只支持 `int` 的加法，所以要求两边的类型都是 `int`。`c++` 中也可以在运行时得到变量的 `typeinfo`，如果 `left_ty` 和 `right_ty` 的 `type` 不相等，那么就报错左右两边不相等。

Syntax Tree Node for Tiger

```
class SimpleVar : public Var {
public:
    sym::Symbol *sym_;
    SimpleVar(int pos, sym::Symbol *sym) : Var(pos), sym_(sym)
    {}
    ~SimpleVar() override;
    type::Ty *SemAnalyze(env::VEnvPtr venv, env::TEnvPtr tenv,
    int labelcount, err::ErrorMsg *errormsg) const override;
};
```

现在我们去判断 `variable`（简单变量、`record`、`array`），我们在 `symbolVar` 中引入 `SemAnalyze`。

```
class EnvEntry {
public:
    bool readonly_;
    explicit EnvEntry(bool readonly = true) : readonly_(readonly) {}
    virtual ~EnvEntry() = default;
};

class VarEntry : public EnvEntry {
public:
    tr::Access *access_;
    type::Ty *ty_;
    explicit VarEntry(type::Ty *ty, bool readonly = false)
        : EnvEntry(readonly), ty_(ty), access_(nullptr){};
};
```

我们再复习一下两个 `Entry` 的定义。在 `c++` 中 `nullptr` 表示为指针为 0

```
type::Ty *SimpleVar::SemAnalyze(...) const {
    env::EnvEntry *entry = venv->Look(sym_);
    if (entry && typeid(*entry) == typeid(env::VarEntry)) {
        return (static_cast<env::VarEntry *>(entry))->ty_
            ->ActualTy();
    } else {
        errormsg->Error(pos_, "undefined variable %s",
                           sym_->Name().data());
    }
    return type::IntTy::Instance();
}
```

也就是我们拿到 `Symbol Variable` 在 `Venv` 中找一下，看看 `entry` 是哪个，如果 `entry` 不为空，先判断一下它所表示一个 `VarEntry`，如果是就直接返回了。

最后是 `declaration`:

```

type::Ty *LetExp::SemAnalyze(...) const {
    venv->BeginScope();
    tenv->BeginScope();
    for (Dec *dec : decs_->GetList())
        dec->SemAnalyze(venv, tenv, labelcount, errmsg);
    type::Ty *result;

    if (!body_) result = type::VoidTy::Instance();
    else result = body_->SemAnalyze(venv, tenv, labelcount,
                                     errmsg);

    tenv->EndScope();
    venv->EndScope();
    return result;
}

```

简单来说，`stdlist` 我们拿出来以后可以简单的用 `for` 循环去访问它。接下来我们拿到 `body`，如果 `body` 是空，那么返回一个 `void`。否则我们就对 `body` 做 semantic analysis。
这是 `let` 的。

后面我们先复习一下语法树的 node:

Syntax Tree Node for Tiger

```

absyn::FunctionDec(int, absyn::FundecList);
absyn::VarDec(int, sym::Symbol, sym::Symbol, absyn::Exp);
absyn::TypeDec(int, absyn::NameAndTyList);

absyn::NameTy(int, sym::Symbol);
absyn::RecordTy(int, absyn::FieldList);
absyn::ArrayTy(int, sym::Symbol);

abysn::Field(int, sym::Symbol, sym::Symbol);
abysn::FieldList(absyn::Field);
absyn::ExpList(absyn::Exp);

```

在语法解析的时候，也有一些 type 的内容，在语法解析的时候，我们大概知道有些内容是什么。比如 `a : int = 3`, 我们可能定义为的不是 `int` 而是 abstraction type，在做类型检查的时候我们需要进一步翻译一下，还有一些 field，定义一个 record 的时候需要定义一些 fieldlist。`ExpList` 的话就是需要重新分析一遍。

Type Checking Variable Declarations

```

void VarDec::SemAnalyze(...) const{

    type::Ty *init_ty = init_->SemAnalyze(venv, tenv,
                                             labelcount, errmsg);
    venv->Enter(var_, new env::VarEntry(init_ty));

}

```

- The above code is ok for the declaration `var x := exp`
- for `var x : type_id := exp`,
 - it must check that `type_id` and `type of exp` are compatible
- if the `type of exp` is `NilTy`, `type_id` must be a `RecordTy` type
 - `var a : my_record := nil`

前面我们一直在讲这个是怎么用的，在 `expression` 中我们拿到的是直接从 `semantic analysis` 中拿出来的。换句话说，我们之前的都是从 `venv` 中 `lookup` 出来对应的 `symbol`，而上图是指我们在定义变量的时候，我们找到 `Symbol` 对应的 `type` 是什么，然后加入到 `venv` 中。

比如 `var x: type_id :=exp`，这个情况我们需要检查 `type_id` 和 `exp` 的类型是否是一致的。还有一个是初始化 `nil` 的时候，前面一定是 `record` 类型。

```
absyn::Fundec(int, sym::Symbol, absyn::FieldList, sym::Symbol,
               absyn::Exp);
absyn::FunDecList(absyn::FunDec);
absyn::DccList(absyn::Dcc);
absyn::NameAndTy(sym::Symbol, absyn::Ty);
absyn::NameAndTyList(absyn::NameAndTy);
absyn::EField(sym::Symbol, absyn::Exp);
absyn::EFieldList(absyn::EField);

enum Oper {
    PLUS_OP, MINUS_OP, TIMES_OP, DIVIDE_OP, EQ_OP,
    NEQ_OP, LT_OP, LE_OP, GT_OP, GE_OP}
```

上图复习了一下 AST Node

```
void TypeDec::SemAnalyze(..) const {
    absyn::NameAndTy *type = types_->GetList()->front();
    tenv->Enter(type->name_, type->ty_->SemAnalyze(tenv,
                                                       errormsg));
}

• Not very general, only handles type-dec list of length 1

• absyn::Ty::SemAnalyze(env::TEnvPtr, err::ErrorMsg *)
  - translates type expressions absyn::Ty to the digested type
    descriptions type::Ty that will be put into tenv_
  - recurs over the absyn::Ty, turning absyn::RecordTy into
    type::RecordTy
  - just looks up any symbols it finds in the tenv_
```

我们拿到 `name_ty` 调用 `SemAnalyze`，它会分析出 `RecordTy`。第一步语法解析的时候，我们是粗略地分析成了 `absyn::RecordTy`，然后这里我们分析成了 `type::Record` 放进 `tenv_`。

至少大家要写一个 `for` 循环，都扫描一遍然后 `enter` 进去。

```
void FunctionDec::SemAnalyze(..) const {
    absyn::FunDec *function = functions_->GetList()->front();
    type::Ty* result_ty = tenv->Look(function->result_);
    type::TyList formals = params->MakeFormalTyList(tenv, errormsg);
    venv->Enter(function->name_, new env::FunEntry(formals,
                                                       result_ty));
    venv->BeginScope();
    auto formal_it = formals->GetList().begin();
    auto param_it = params->GetList().begin();
    for (; param_it != params->GetList().end(); formal_it++, param_it++)
        venv->Enter((*param_it)->name_, new env::VarEntry(*formal_it));
    type::Ty *ty =
        function->body_->SemAnalyze(venv, tenv, labelcount, errormsg);
    venv->EndScope();
}
```

函数的定义，就是找一下第一个，然后去判。参数的类型还是 `abstract::Ty`，这里面就会做一个转换，变成 `formals`。这里我就会使用迭代器去写 `for` 循环。

Cycles in Representations of Types

```
typedef cell *link ;  
typedef struct {  
    int info ;  
    link next ;  
} cell ;
```

自递归问题，这是 C 里面的代码，其实 `cell *` 在 `cell` 定义前出现了，并且 `cell` 里面用了 `cell*`

Recursive Declarations

- Undefined type or function identifiers will be encountered
 - absyn::RecordTy::SemAnalyze for recursive record types
 - absyn::Exp::SemAnalyze for recursive functions
- Put all “headers” in the environment first
 - Let the body to be NULL at first

还有一种可能是 function a 里面会调用 b, function b 里面又会调用 a。思路就是我们放一个占位符进去。第一遍我们做 a 的时候看到 b, 知道肯定 b 是个函数, 同理 b 调用 a 的时候, 我们也知道 a 是个函数。第二遍的时候, 我们再去检查类型是不是符合。

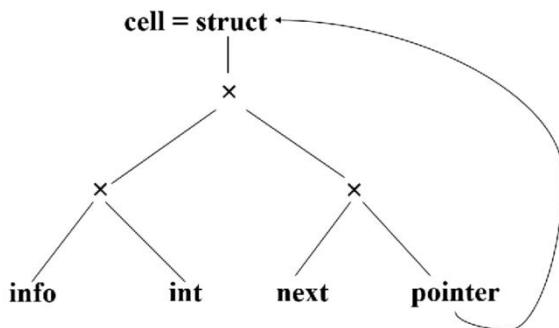
Tiger 中的递归声明

Recursive Declarations

- type `list = { first: int, rest: list }`
- type `list = is header`
 - `tenv->enter(name, new type::NameTy(name, NULL))`
- `Body is { first: int, rest: list }`
 - When `absyn::Ty::SemAnalyze` on the body of the type declaration
 - it stops as soon as it gets to any `type::NameTy` type
 - Cannot behave like `ActualTy`
 - When `absyn::Ty::SemAnalyze` returns a `type::Ty`,
 - it can be used to modify the `ty_` field of the `type::NameTy` class in the `tenv` for which is `NULL` now

在 tiger 中，所有的 array 和 record 都是一个指针。第一部分把 TypeName 拿出来，也就是先定义好 list（把类型 List 登记到 type environment 里去），然后一遍扫描写进去即可。

Cycles in Representations of Types



通过 structure 引入的环是允许的。

Type Checking Recusive Declarations

- Every cycle in a set of mutually recursive type declarations must pass through a record or array declaration
- The following illegal type declaration must be detect by the type-checker


```

type a = b
type b = c
type c = d
type d = a
      
```

而这种环就不允许了。

Tiger 中的递归函数调用

Recursive Function Calls

- absyn::FunctionDec(int, absyn::FundecList);
 - For FunctionDec Node there are two passes for its children nodes
- First pass
 - Only recursive functions themselves are entered into the venv with their prototypes
 - function name
 - types of formal parameters
 - type of return value
- Second pass
 - Trans the body using the new environment
 - The formal parameters are processed again
 - This time entering params as env::VarEntries

2021/10/26

20

Node 里是一个 list, 下面是一个个的 func。也是分两步, 第一步处理 function name、formal parameters、return value, 把这些登记到变量表里去。第二遍我们扫描的时候, 除了函数名以外, 还有个函数体。我们从变量环境中可以找到这些参数。对于递归的函数, f 调用了 g, g 里又调用了 f。总会有一个函数先引用再被定义。在 C++ 中, 就是先定义其 prototype: f(), g()。而 C 里比较特殊, C 引用函数的时候如果还没有定义, 会自动把缺省值补上。而 tiger 中选了一个折中, 只要我们放在一起定义, 就可以通过两边 parse 处理掉。

Names for Type Expressions

```
typedef cell *link;
link next, last;
struct A *p, *q, *r;
```

- next, last link
- p, q, r pointer(cell)
- Name Equivalence
 - Views each type name as a distinct type
- Structural Equivalence
 - Names are replaced by the type expressions they defined

刚才我们提到了 name-equivalent 和 structure-equivalent。Cell 就是一个 list, 第一部分是 information, 第二部分是个 list。实际上 next 和 last、p、q、r 都是指向 cell 的指针, 但是如果在 tiger 中, 它就会认为 next、last 和 p、q、r 不是同一个东西。这就是 name-equivalent, 这就是把每个名字当做不同的类型, 这在处理递归的时候是比较简单的, 我们可以停在 name 这里, 不需要再去展开。而另外一种就是 structure-equivalence, 替代掉以后就是同一个东西。我们要判断两个类型是不是结构等价的话, 我们可以通过递归来做: 基础类型 (primitive type) 是可以比较的, 然后根据 array 和 record 构造出的更加复杂的类型。我们不关心 field 的 name, 我们只关心 field 的类型是什么。

Algorithm

Non Primitive Types

array [] of T₁ (array)
array(T₁.type)
T₁×T₂ (record)
T.type = T₁.type × T₂.type
***T₁ (pointer)**
T.type = pointer(T₁.type)
T₁ → T₂ (function)
T.type = T₁.type → T₂.type

```
int sequiv(s, t) {  
    if ( s and t are the same basic type ) return 1 ;  
    else if ( s == array(s1) && t == array(t1) )  
        return sequiv(s1, t1) ;  
    else if ( s == s1×s2 && t == t1 × t2 )  
        return sequiv(s1, t1) && sequiv(s2, t2)  
    else if ( s == pointer(s1) && t == pointer(t1) )  
        return sequiv(s1, t1)  
    else if ( s == s1→s2 && t == t1 → t2 )  
        return sequiv(s1, t1) && sequiv(s2, t2)  
    return 0;  
}
```

如果我们只有四种构造类型，给定两个类型，我们就可以根据上右图的递归方法来判定两个类型是否等价。

我们讲 tiger 选用的是 name-equivalent，否则我们还要展开到下面。我们后面讲的这部分都是 tiger 中没有的东西，但是一般的编译语言要处理的。比如我们有 int 和 float，又有一个加法，那么 int + float 怎么办？这件事情就是编译器帮我们做的事情。

Type Conversion

- X + I
 - X real, I int
 - Insert inttoreal before I
 - X I inttoreal real+
- Implicit type conversion
 - Coercions
 - No information lost
 - int to double, not vice-versa
- Explicit type casting

比如上面实数和整数相加，我们就要把 I 转化为实数，然后加法一定是浮点数加法。转的方法第一种就是编译器帮我们去做，第二种就是在代码中 I 前添加(float)做强制类型转换。

Type Conversion

Production	Semantic rule
$E \rightarrow \text{num}$	$E.\text{type} = \text{integer}$
$E \rightarrow \text{num.num}$	$E.\text{type} = \text{real}$
$E \rightarrow \text{id}$	$E.\text{type} = \text{lookup}(\text{id}.entry)$
$E \rightarrow E_1 \text{ op } E_2$	$E.\text{type} = \begin{cases} \text{integer} & \text{if } (E_1.\text{type} == \text{integer} \& \& E_2.\text{type} == \text{integer}) \\ \text{real} & \text{else if } (E_1.\text{type} == \text{integer} \& \& E_2.\text{type} == \text{real}) \\ \text{real} & \text{else if } (E_1.\text{type} == \text{real} \& \& E_2.\text{type} == \text{integer}) \\ \text{real} & \text{else if } (E_1.\text{type} == \text{real} \& \& E_2.\text{type} == \text{real}) \\ \text{error, integer} & \text{else} \end{cases}$

在 tiger 中，只有 int 和 string 两种，所以 tiger 中有 type checking 的动作，还是没有 conversion 的动作。

Simple Overloading

- Overloaded symbol has different meanings depending on its arguments
 - $1.0 + 2.0$
 - $1 + 2$
- Resolution
 - Determine a unique meaning for an occurrence of an overloading symbol
 - Well known as operator identification

这种根据参数决定 operator 的具体含义的，这就是 simple overloading。

General Overloading

- The meaning of an overloading symbol is depending on its context
- An example from Ada, operator “*”
 - A pair of integers to an integer
 - A pair of integers to a complex
 - A pair of complexes to a complex
 - $2*(3*5)$
 - $(3*5)$ must be integer
 - $(3*5)*z$
 - $(3*5)$ must be complex

有人提出过 general overloading。在 Ada 中，它会根据 environment 来判断而不是只看局部的，把第一步计算 $3*5$ 也直接变成复数乘法。

General Overloading

$E \rightarrow E_1(E_2)$

```
E.type =  
(E2.types == s && E1.types == s → t ) ?  
t : type_error }
```

```
E.types = { t | there exists an s in E2 types  
such that (s → t) is in E1 types }
```

做法无非就是从表达式最上面看下来。

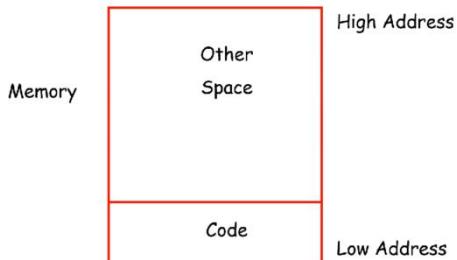
第一部分就是彻底结束了。高层次的语言->AST + Symbol Table，前端就结束了。后端就是 AST + Symbol Table -> IR -> 汇编。

从 AST 可以看出源程序长什么样子，基本上可以退回去。再变一次以后，它就不像高级语言了。后端和前端最大的区别就是，前端可以大量地使用工具解决，而后端需要手写代码。我们要把 AST 变成一个汇编，我们要先讲 activation record。

Activation Record (Frame)

Activation record (AR) 其实就是 frame。Frame 是 ICS 中比较喜欢用的词。

Memory Layout



最早的计算机是没有程序的概念的，只有执行的概念。后来我们可以把程序作为数据存到内存中，那么我们找一个地方就把代码放进去了。后来，我们慢慢发现有 procedure，函数之间可以相互调用。Activation 就是函数的一次调用。从函数被调用到返回，就是这个 activation 的 lifetime。其中变量的 lifetime 也是类似的定义。

Activations

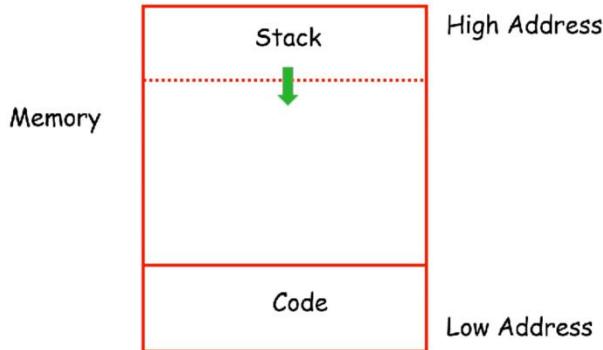
- An invocation of procedure P is an activation of P
- The lifetime of an activation of P is
 - All the steps to execute P
 - Including all the steps in procedures that P calls

Lifetimes of Variables

- The lifetime of a variable x is the portion of execution in which x is defined
- Note that
 - Lifetime is a dynamic (run-time) concept
 - Scope is a static concept

有了这件事情以后，变量会变的，一会儿有一会儿没有，Fortran 会直接分配变量，但是这样不合理，我们发现可以使用一个栈来存放这些变量，这样符合函数 P 调用函数 Q 后，生成 P 的变量->生成 Q 的变量->收回 Q 的变量->收回 P 的变量的特点。

Memory Layout



在很多机器上，栈从高地址开始向着低地址增长。在一次函数调用中需要维护的数据叫做 activation record 或者 frame。

如果函数 F 调用了 G，会有一些参数的传递，F 要向 G 传递一些实参，而 G 在执行的过程中是 formal parameters (形参)，早期在执行的过程中是一个东西。本来我们的参数都是从右向左依次压栈的，这就是同一个东西。F 向其中压栈，而 G 从中读出来，那么它们的 frame 就会有 overlap。

Higher-Order Functions

- Some languages support both
 - Nested functions
 - Function-valued variables
- Higher-order function
 - Functions are nested
 - Functions are returnable values
 - ML, Scheme etc
- Pascal and Tiger
 - Functions are nested but not returnable
- C
 - Functions are not nested but returnable

C 这样的程序有一个特点：它是有函数指针的，并且可以作为返回类型，但是没有函数嵌套。而在 tiger 中函数允许嵌套，但是函数指针不能作为返回值。

Higher-Order Functions

```
fun f(x) =          int (*)() f(int x) {  
    let fun g(y) = x+y      int g(int y) {  
        in g                  return x + y;  
    end                      }  
    val h = f(3)            return g;  
    val j = f(4)            }  
    val z = h(5)            int (*h)() = f(3);  
    val w = j(7)            int (*j)() = f(4);  
                           int z = h(5);  
                           int w = j(7);
```

两次调用 `f` 返回了两个函数，我们再去执行就可以拿到两个 value。这是既嵌套又返回的情况，编译中比较难处理这种情况，所以我们不处理。

The Contents of a Typical ARs

Incoming arguments	Argument n ... Argument 1 Static link	↑ higher address Previous frame
	Return address	
Frame pointer→	Saved registers Local variables Temporaries	Current frame
Outgoing arguments	Argument m ... Argument 1 Static link	
Stack pointer→		Next frame ↓ Lower addresses 9

在过程中，我们也会生成一些 local 的 temporary variable。传参的时候就是自右向左压栈，有寄存器了以后那就是前 4~6 个参数放到寄存器里。在 tiger 中要支持函数嵌套，并且内函数可以引用外函数的变量。`g` 需要访问 `f` 的 activation record。但是 `f` 和 `g` 的 activation record 在栈上并不一定是相邻的。我们要让 `g` 能够找到 `f` 的 frame，所以我们需要加一个 static link，让 `g` 能够直接跑到 `f` 的 frame 中。

The Main Point

The compiler must determine, at compile-time,
the layout of activation records and generate
code that correctly accesses locations in the
activation record

*Thus, the AR layout and the code generator
must be designed together!*

Discussion

- There is nothing magic about this organization
 - Can rearrange order of frame elements
 - Can divide caller/callee responsibilities differently
- An organization is better if it improves execution speed or simplifies code generation

Registers

- Real compilers hold as much of the frame as possible in registers
 - Especially the method result and arguments
 - Return address is stored in a register in RISCs
 - Locals and temporaries are also saved in registers

Lab5 就是 code gene, 我们尽量希望把变量能放寄存器就放寄存器(假设是无限个),
但是最后一个 lab 的寄存器是有限个的。

Call-by-Value

- A formal parameter is treated just like a local name
- The storage for the formals is in the activation record of the call
- The caller evaluates the actual parameters and places their r-values in the storage for the formals

Call-by-Reference

- Call-by-address
- If an actual parameter is a name or an expression having an l-value, then that l-value itself is passed
- If the actual parameter is an expression that has no l-value, then the expression is evaluated in a new location and the address of that location is passed

call-by-ref (call-by-addr) 实际上就是传入 actual parameter 的地址传进去, 也就是传入的是左值。

Call-by-Restore

Call-by-Restore

- A hybrid between call-by-value and call-by-reference, also known as
 - Copy-restore linkage
 - copy-in copy-out
 - value-result

- Before control flows to the callee
 - The actual parameters are evaluated
 - r-values of the actuals are passed to the callee
 - l-values of the actuals if having l-values are determined before the call
- When control returns
 - the current r-values of the formals are copied back into the l-values of the actuals
 - using the l-values computed before the call

call-by-restore 是一个混合的方法，它有很多别名。实际上就是在 calling 之前的数据参的时候，我们把右值传给 callee，同时如果有左值的话，我们把左值的值（右值）传递过去。Callee 的时候都是拿 r-value 计算，而在 return 的时候，会把计算完的右值 copy 到左值里去。

Call-by-Name

- The procedure is treated as if it were a macro
 - Macro-expansion
 - In-line expansion
- The local names of the callee are kept distinct from the names of the calling procedure
- The actual parameters are surrounded by parentheses if necessary to preserve their integrity

我们用的宏就是 call-by-name。

Parameter Passing

- Parameters are passed on the stack
 - For most machines designed in 1970s
 - Caused the memory traffic
- For RISC calling conventions
 - The first k parameters are passed in registers
 - The next parameters are passed on the stack
 - Few functions have parameters more than 4
 - Almost none have parameters more than 6

Parameter Passing

- Function $f(a_1, \dots, a_n)$ has the value a_1 in register r_1
- What happens if it calls $h(z)$?
 - z is passed into r_1
 - Previous value in r_1 (for a_1 this time) must be saved on the stack
 - Still memory traffic!

Parameter Passing

- How has the use of registers saved any time?
 - Leave procedures
 - Inter-procedure register allocation
 - $f(x)$ might receive x in r_1
 - but call $h(z)$ with z in r_7
 - a_1 is dead before f calls $h()$
 - Register windows

Parameter Passing

- How has the use of registers saved any time?
 - Leave procedures
 - Inter-procedure register allocation
 - $f(x)$ might receive x in r_1
 - but call $h(z)$ with z in r_7
 - a_1 is dead before f calls $h()$
 - Register windows

使用寄存器存在调用别人的时候，还是需要把参数存到栈上，但是对于叶子函数（不

调用别人的函数），就没有这种问题。如果寄存器传参的话，最终编译器要处理这些事情，在 ICS 里头我们举过一个例子。

```
P(x,y):
{
    return Q(y) + Q(x);
}

这时候我们传参的时候
%rdi<-x
%rsi<-y
我们在调用第一个 Q 的时候
t1<-%rdi
%rdi<-y
call Q
t2<-%rax
%rdi<-t1
call Q
add %t2 %rax
Ret
这时候我们发现 t1 和 t2，必须能跨过 Q。所以我们需要把 t1 和 t2 作为 callee-save register
```

Variable-length Argument Lists

```
#include<stdarg.h>
void minprintf(char *fmt,...)
{
    va_list ap; /* points to each unnamed arg in turn */
    char *p,*sval;
    int ival;
    va_start(ap,fmt); /* make ap point to 1st unnamed arg */

    for(p=fmt; *p; p++) {
        if(*p != '%') {
            putchar(*p);
            continue;
        }
        switch(*++p) {
            case 'd':
                ival = va_arg(ap,int); printf("%d",ival);
                break;
            case 's':
                for(sval = va_arg(ap,char *);*sval;sval++) putchar(*sval);
                break;
            default:
                putchar(*p); break;
        } // end switch
    } // end for
    va_end(ap); /* clean up when done */
} // end function
```

可变长参数至少有一个是固定的，也就是参数数量必须是 1~任意多。这就是一本书上的例子，我们要访问这个变参，就通过 `va_list` 的一个宏。我们传入第一个参数到 `va_start` 中，就可以定位到第一个参数。注意 `va_arg(ap,)` 这个函数每调用一次就会往后走。

大家可以看到 `va_start`, `va_arg` 有一个潜在的要求是 `formal parameter` 必须是连续放的，否则我们的宏不能处理。所以，我们希望 `formal parameter` 是从 `memory` 中连续拿的。

Parameter Passing

- varargs feature such as `printf()` in C
 - the address of the formal parameters can be accessed
 - all the formal parameters must be at consecutive address
- How to handle the in-register parameters
 - Allocate space on the stack but not written by caller
 - Callee writes register values onto the stack when necessary

但是前 6 个又是要放寄存器里的。一个比较好的参数是把栈上的前 6 个位置空出来，等到 callee 需要的时候，把相应寄存器的值一个个写到内存中去。

Frame-resident variables

- Values are allocated to memory (in the frame) only when necessary for one of these reasons:
 - Is passed by reference
 - Its address is taken
 - Is accessed by a procedure nested inside the current one
 - The value is too big to fit
 - The register holding the variable is needed for a specific purpose
 - Too many locals and temporaries - "spill"

2021/10/29

15 号期中考试

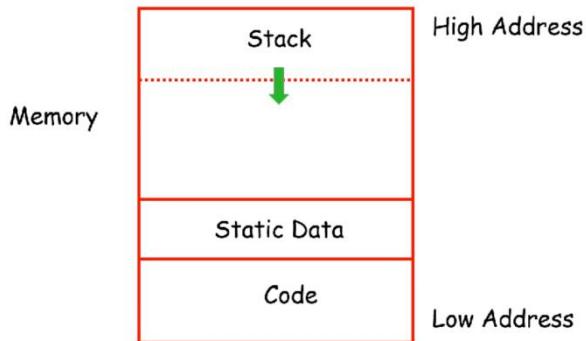
考到 type checking，我们说过的，我们是两次考试，每次考试 20 分，我们的 Project 是 60 分

上节课讲到参数通过压栈/寄存器来传参，并且传参的时候会有什么问题。今天我们来讲的主要是 local variables。变量的话一种就是说放到寄存器里，因为寄存器比较快。但是有些情况下我们也不得不把变量放到内存中，大部分时间是放到栈上。Eg: call by ref，引用变量地址，nested function（内函数需要用到的外函数的变量），结构、数组等寄存器中放不下的，又比如 callee saved register，又比如 spill（局部变量和临时变量太多了，寄存器放不下，多出来的都放到内存里去）。

这里有很多种变量进内存的情况，其中前三种称为 variable escape（前两种需要地址，第三种是嵌套函数引用，在 tiger 中主要是第三种情况）

这些讲的都是局部变量，也就是函数体中定义的变量需要放到内存里。还有一种是全局变量，在 C 和 C++ 中比较明显。一旦生成了以后，程序的任何地方都可以访问这个数据，所

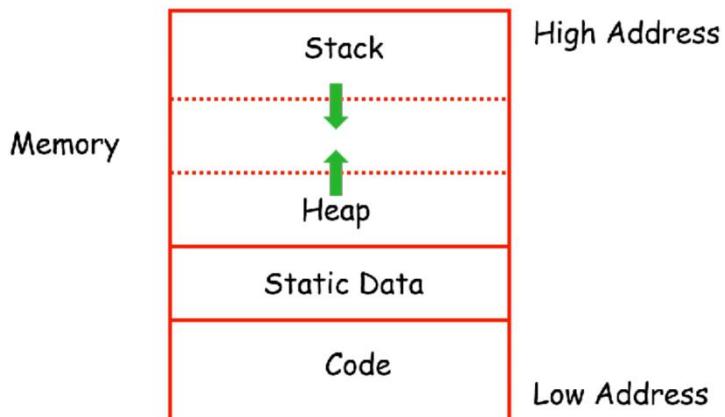
以不能放到栈上，因为栈会随着函数调用消失。



但是在 tiger 中，把引号中的字符串常量，最后都是放到 Static Data 里头。在 C 里面也是这样处理的。我们知道在 ICS 里，Read-Only Data 是和 Code 放在一起的。

第三种情况是堆，在 C++ 里就是用 new 申请的空间。在 Tiger 中，是把 record 和 array 在内存中分配固定大小的空间。在 Tiger 处理中，是把 record 和 array 放到堆上去的。因为使用了堆，所以在函数中声明数组和 record 以后，会在堆上分配一块空间。当我们函数结束以后，这两个变量就消失了。但是在堆上的两个空间还在对应的地方，tiger 没有显式地回收掉这两个东西。在第二部分里头，就有 garbage collection，在哪个地方我们需要用 compiler 把 record 和 array 在堆上的空间回收掉。在某些点会触发这件事情，让 runtime 来接手回收掉垃圾。

Memory Layout with Heap



Code 是可读的，static data 是可能是可读的，也有可能读写的。栈和 heap 是随着程序运行的时候会变的。对于每个栈本身的一个 frame，通常 frame 的大小是固定的。我们现在知道在这两个增长的中间，我们又插入了 shared library。在考虑我们程序编译的时候，就要考虑到内存的布局，这都是在 user 态的东西，往上的 kernel 态我们也没去管。

下面我们重点提到在 tiger 中的 variable escape

Lexical Scope with Nested Procedures

```
1. type tree = {key:string, left: tree, right: tree}
2.
3. function prettyprint(tree: tree): string =
4.     let
5.         var output:= ""
6.
7.         function write(s:string) =
8.             output := concat(output, s)
9.
10.        function show(n:int, t:tree) =
11.            let function indent(s:string) =
12.                (for(i:=1 to n)
13.                    do write(" "));
14.                output := concat(output,s);write("\n"))
15.            in if t=nil then indent(".")
16.            else (indent(t.key);
17.                  show(n+1, t.left);
18.                  show(n+1, t.right))
19.        end
20.
21.    in show(0, tree); output
22. end
```

Lexical Scope with Nested Procedures	
main	1
prettyprint	2
write	3
show	3
indent	4

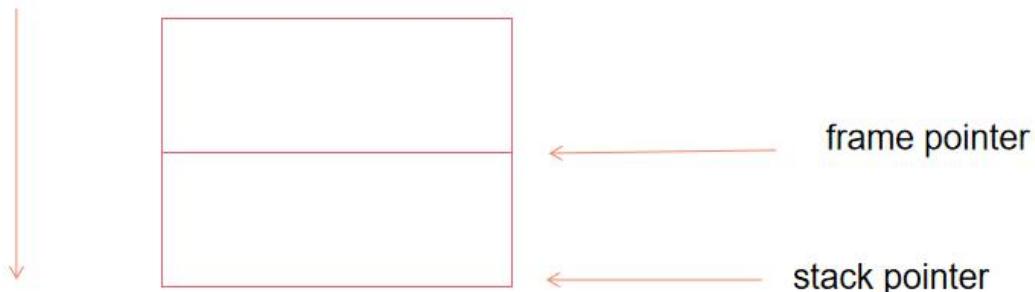
这个部分 main 函数在最外面定义了 tree 这个类型。

我们关心的是，main 里面定义了 prettyprint，prettyprint 里又定义了 write 和 show，show 里面又定义了 indent 这个函数。我们把 main 称为第 1 层函数，以此类推，prettyprint 第 2 层，write 和 show 第 3 层，indent 第 4 层。

在每一层的函数中都可以声明一些变量。比如在 prettyprint 中声明了 tree 和 output 变量。第三层的函数 write 可以用到外层的 output 变量。C 因为没有函数嵌套，所以只有 global 和 local 变量，而在 tiger 中是可以的。因为只有在 prettyprint 调用的时候才有可能调用 write 和 show，所以 show 和 write 一定是可以用到外层的 output 变量的。

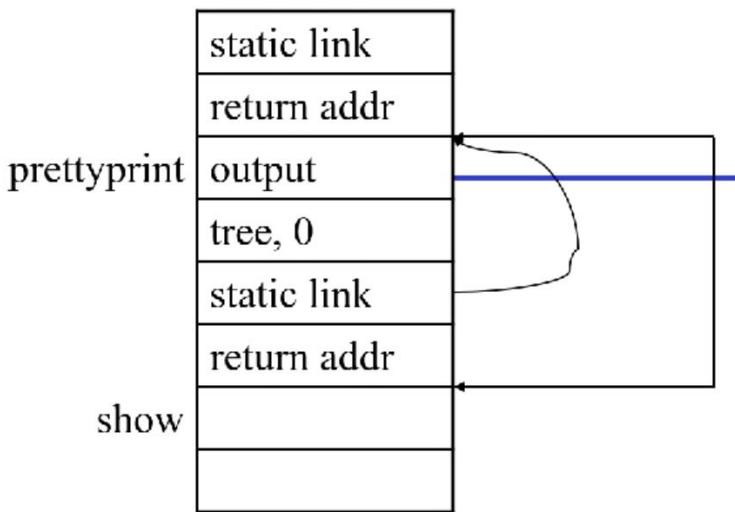
同样的，第 4 层函数 indent 用到了第 3 层的变量 n、第 2 层的变量 output。

那么这个 indent 怎么去访问到 n 和 output 呢？我们需要传递一个 frame pointer 进去。

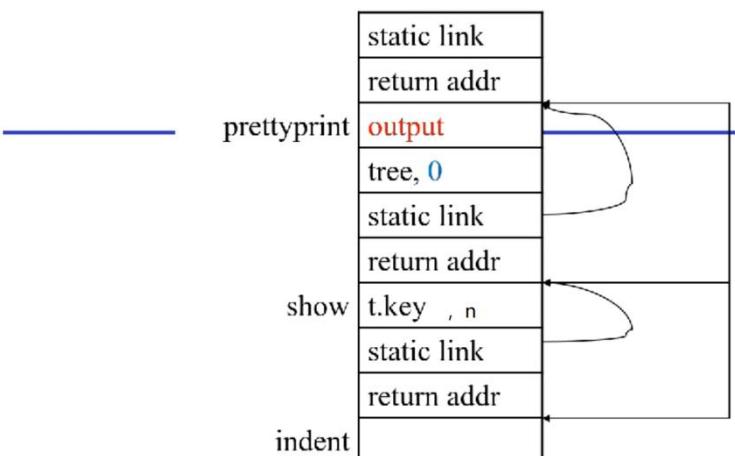


要让内函数访问到外函数的变量，我们需要把 frame pointer 当做参数传递进去。Indent 如果能拿到 prettyprint 的 frame pointer，那么我们就可以访问到 prettyprint 中定义的 output。但是传递进去没那么简单，因为在 indent 中用到了两个变量 n 和 output。我们在这个时候，是传递 show 的 frame pointer 还是 prettyprint 的，还是两个都传？

我们先看一个简单的情况，`show` 会用到 `prettyprint` 中的 `output`。我们令 `0` 和 `tree` 压栈，演示调用 `show` 的时候会传参。在 tiger 中，是把 `0` 当做第二个参数，`tree` 是第 3 个参数，而第一个参数是 `static link`，它是一定要在栈上的。



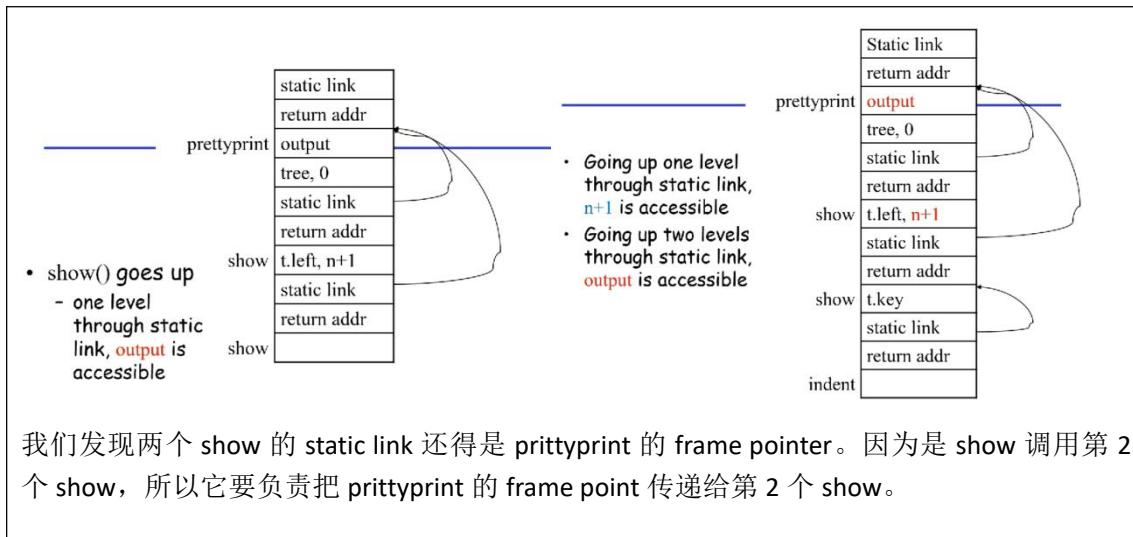
这样 `show+8` 就可以拿到 `static link`。Tiger 假设生成的代码都是 `frame pointer` 的，后面我们可以用一个宏把 `frame pointer` 替换成 `stack pointer`。这就是 `static link` 作为参数传递进去的情况。



- `indent()` goes up
 - one level through static link, `0` is accessible
 - two levels through static link, `output` is accessible

在调用 `indent` 的时候，我们可以传递 `show` 的 `static link` 进去。要访问 `n` 的时候，可以直接通过 `show` 的 `static link` 进去直接访问 `n`，要访问 `output` 的情况下，就需要走两次 `static link` 到 `prettyprint` 里去找到 `output`。

我们的直觉似乎是相邻的上层传下层。那么 `show` 递归调用了怎么办呢？



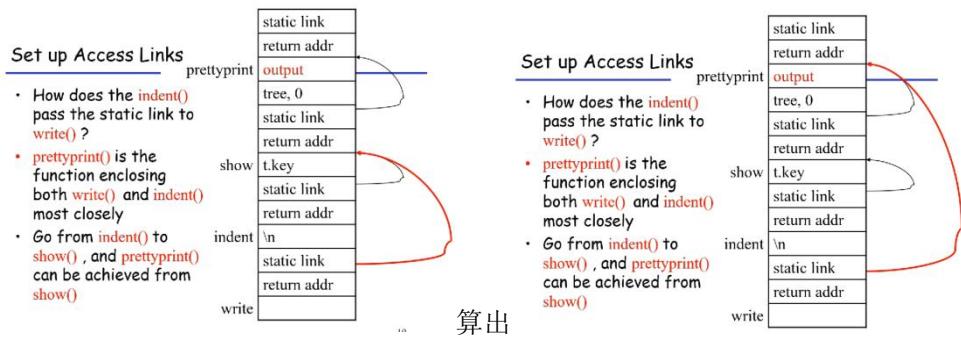
我们发现两个 `show` 的 static link 还得是 `prettyprint` 的 frame pointer。因为是 `show` 调用第 2 个 `show`，所以它要负责把 `prettyprint` 的 frame point 传递给第 2 个 `show`。

下面我们来讲算法，这本书里这部分写的太烂了，龙书上写的是比较好的。

Access Nonlocal Variables

- Procedure **p** at nesting depth n_p
- Procedure **p** refers to a nonlocal **a** with nesting depth $n_a < n_p$
 - When control is in **p**,
 - The frame for **p** is at the top of the stack
 - Follow $n_p - n_a$ static links from the frame at the top of the stack
 - After following $n_p - n_a$ links, we reach a frame for the procedure that **a** is local to.
 - $n_p - n_a$ can be computed at compile time

有一个 `procedure p`，每一个函数/过程都是有深度的 n_p 。`p` 想要访问一个非自己局部的变量，也就是外函数的变量 `a`，也就是 `a` 的层次一定是比 `p` 小的。那么我们在 `p` 里要访问 `a`，当前我们已经在 `p` 里了。`p` 在栈的底部是有一个 `static link` 的。那么我们在 `p` 里要访问 `a` 要走 $n_p - n_a$ 次 `static link`，这都是我们在编译时可以确定的，生成出来的代码是可以确定的。在第 7 章的时候，我们翻译到 `non-localvariable` 的时候会产生出一个非常深的数据结构。当然，`a` 在对应深度的 `frame` 的 `offset` 的多少，编译器也是知道的。



第二个问题，是我们的 static link 怎么建立起来。这个问题更加复杂。这里的 indent 最后怎么访问到 output，我们这里需要建立两次。

Set up Access Links

- Procedure **p** at nesting depth n_p calls procedure **x** at nesting depth n_x .
- $n_p < n_x$
 - x must be declared within p ($n_x = n_p + 1$), static link in x must point to the frame pointer of the activation record of the caller p just below in the stack
 - Eg. `show()` calls `indent()`
 - Static link of `indent()` points to the frame pointer of `show()`

建立的过程如上图所示，在编译的时候，我们发现函数 p 调用了函数 x。

第一种情况，浅的调用深的。也就是把自己的 frame pointer 传进去，在我们这个函数里，`prettyprint` 也不能直接调用 `indent`，所以要保证深度层次相差 1。

第二种情况，它可以调用外函数，比如第 4 层的 `indent` 可以直接调用第 3 层 `write`。我们怎么样把 `prettyprint` 的 static link 传递给 `write`。也就是 `indent` 和 `write` 需要找到最近共同祖先，传入最近共同祖先即可。找到了以后，我们看 `indent` 到 `prettyprint` 要走几下，算一下，我们就知道 `indent` 在执行的时候沿着 static link 需要走几下。然后把它作为参数传进去。

Displays

Displays

- An array d of pointers to frames
- Storage for a nonlocal a at nesting depth i is in the frame pointed by display element $d[i]$
- Suppose control is in an activation of a procedure p at nesting depth j
 - The first $j-1$ elements of the display point to the most recent activations of the procedures that lexically enclose procedure p
 - $d[j]$ points to the activation of p
- Using a display is generally faster than following access links

在中文字的 95 页，6.1.6 静态链

有几种方法可实现这一目的：

- 每当调用函数 f 时，便传递给 f 一个指针，该指针指向静态包含 f 的那个函数，称这个指针为静态链（static link）。
- 建立一个全局数组，该数组的位置 i 处包含一个指针，它指向最近一次进入的，其静态嵌套深度是 i 的过程的栈帧，这个数组叫做嵌套层次显示表（display）。
- 当 g 调用 f 时， g 中每一个实际被 f （或被嵌套在 f 内的任意函数）访问了的变量，都将作为额外的参数而传递给 f 。这称之为 λ -提升（lambda lifting）。

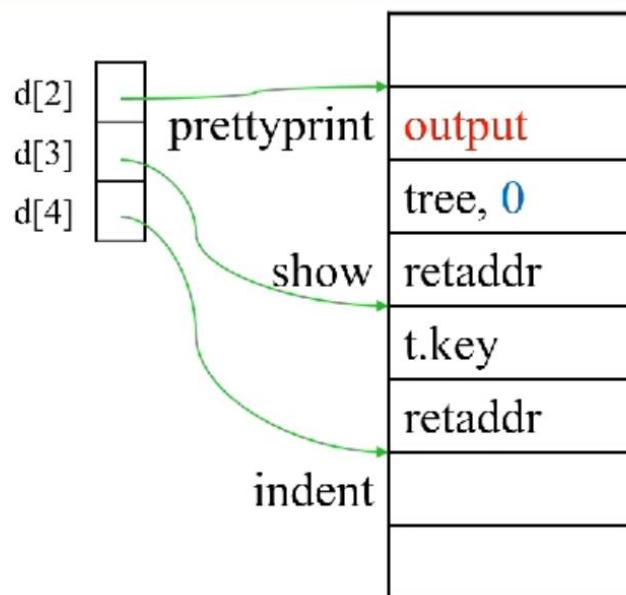
我这里只详细描述静态链的方法。至于实际中应当使用哪种方法，见习题 6.7。

每当函数 f 被调用时，都将传递给它一个指针，该指针指向在程序正文中直接包含 f 的函数 g 的“当前”（最近一次进入的）活动记录。

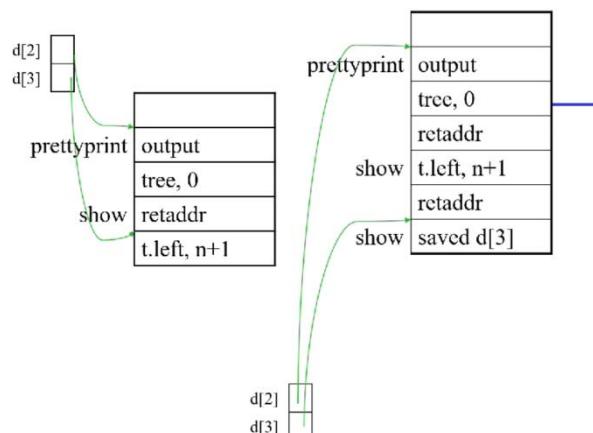
例如，在程序 6-2 中：

下面我们来看看 display 是什么意思，实际上就是在运行的时候每个函数都有一个深度，排序完就是 1234。

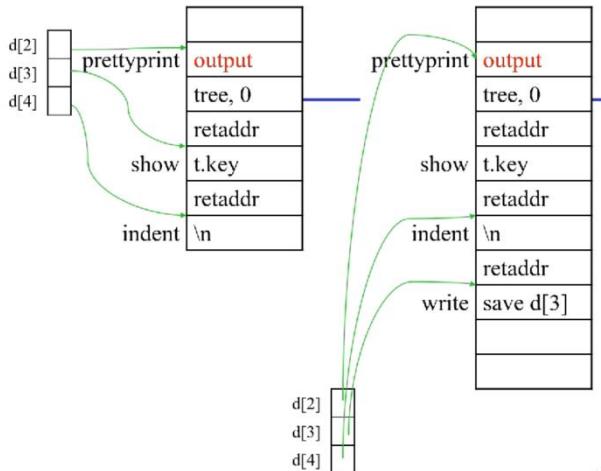
Displays



有一个指针，指过去到栈帧开始的地方。



show 自己调用自己的情况，那么第三层有两个 show，display 中我们保留最近的一个 show，因为第一个 show 除了退出，不会再调用了。当第 2 个 show 结束的时候，上图右边的 display 需要恢复成上图左边的情况，所以在第二个 show 中要保存前一个 show 的地址，这样第二个 show 退出的时候就可以把第 1 个 show 的位置填进去。



29

最后一个 lambda lifting

Lambda Lifting

- When g calls f, each variable of g that is actually accessed by f (or by any function nested inside f) is passed to f as an extra argument.

当我们 g 调用 f 的时候，escape analysis 需要的变量全部传递进去。

我们开始讲 tiger 中的数据结构。

Frame

```
/* frame.h */
namespace frame {
    class Frame {
        frame::Frame NewFrame(temp::Label *name, std::list<bool> formals);
    };
}
```

- Frames holds information about
 - ① formal parameters**
 - ② local variables allocated in the this frame**
- To make a new frame for a function f with k formal parameters
 - call NewFrame(f, l)
 - l is a std::list<bool> of k Booleans
 - True for escape

在我们的书上就是大写的 F_作为前缀。在这个 frame 里，比较重要的就是遇到一个函数之后，我们需要 new 一个 frame 出来，也就是给一个新的函数分配一块 frame。我们讲到 new frame 的第一步没什么太多的事情，

- 把函数名字传递进去。但是这个时候的函数名就不是 symbol 了。把一个函数名放在那里，其实就是一个 label（在汇编中的函数入口），第二个参数实际上就是一个 bool 的链表，其实就是{true, false, false}，代表着哪些参数应该放到内存里去，第一个参数是 static link，是一定要放到内存中去。

Example

```
g(arg1, arg2, arg3);
• The first one is escaped while the others are
not
NewFrame(g, {true, false, false});
```

第 7 章，我们要做 translation，AST 在变过来的时候是没有 static link 的，但是在变完之后，就有 static link 了，然后我们就需要调用 new frame。我们在设计的时候，frame 对于高层来说是一个不可见的 namespace。因为我们要生成代码，一定是要和某个体系结构相关的。Frame 是每个体系结构单独写的，但是其他模块都是体系结构无关的。

我们回到这里，传入函数的 label 和一个 bool 链表，对应每个 formal parameters 要不要进栈。

Access

Frame

- The implementation of frame.h is in a module specific to the target machine
- The interface is abstract
 - /* mipsframe.h */
#include "frame.h"
namespace frame {
class Access {
public:
 virtual tree::Exp *ToExp(tree::Exp *framePtr) const = 0;
};
};
- tree is the middle IR data structure and will be introduced in the next chapter

有了 frame 以后，变量无非就是分配在栈上或者寄存器里。所以我们要设计两个数据结构，一个是放在寄存器中的变量，一个是放在内存中的变量。我们用一个统一的封装接口 Access。Access 类就是用来描述函数的局部变量和形参（formals）是存放在寄存器上还是存放在栈上。

```
/* mipsframe.h */
#include "frame.h"
class Access {
    public: virtual tree::Exp *ToExp(tree::Exp
*framePtr) const = 0;
}
//其中 tree 是中间表达 tree language 的数据
结构
```

```
/* mipsframe.cc */
#include "mipsframe.h"
class InRegAccess : public Access {
public: temp::Temp *reg;
    explicit InRegAccess : (temp::Temp
*reg):reg(reg) {}
    tree::Exp *ToExp(tree::Exp *framePtr)
const override
    { return new tree::TempExp(reg);}
};
```

	<pre>class InFrameAccess : public Access { public: int offset; explicit InFrameAccess : (int offset): offset(offset) {} tree::Exp *ToExp(tree::Exp *framePtr) const override; };</pre>
<pre>/* frame.h */ class Frame { public: ... std::list<frame::Access *> *formals_;</pre>	

变量要么是在寄存器中（InRegAccess），里面分配了一个 virtual 的 register。

要么在栈上（InFrameAccess），核心问题就是 offset，也就是在 frame 中，从 frame pointer 开始是负几的位置上存了这个变量。

所以在 Frame 类中，我们定义 formals（函数的形参）是一个 Access 的链表，也就是形参要么在寄存器上要么在栈上，

<pre>/* frame.h */ class Frame { public: /* other members and methods */ std::list<frame::Access *> *formals_;</pre>	<ul style="list-style-type: none"> • formals_ extracts a list of k "accesses" <ul style="list-style-type: none"> - denoting the locations where the formal parameters will be kept at run time as seen from inside the callee
--	---

这个是说 formal 是一个 access 的链表，也就是把 formal 的这些形参分配了寄存器或者是栈上的东西。讲到这个形参，上节课讲到参数用寄存器来传的时候，虽然速度很快，但是会引入比较大的麻烦。

比如在 ICS 中

<pre>P(x,y): { return Q(y) + Q(x); }</pre>	<p>这时候我们传参的时候 %rdi<-x</p>
--	----------------------------------

```
%rsi<-y
```

我们在调用第一个 Q 的时候

```
t1<-%rdi
```

```
%rdi<-y
```

```
call Q
```

```
t2<-%rax
```

```
%rdi<-t1
```

```
call Q
```

```
add %t2 %rax
```

```
Ret
```

这时候我们发现 t1 和 t2，必须能跨过 Q。所以我们需要把 t1 和 t2 作为 callee-save register

Shift of View

- “shift of view”
 - Parameters may be seen differently by the caller and the callee
 - Handled by NewFrame()
- For each formal parameter, NewFrame() must calculate:
 - How the parameter will be seen from inside the function
 - In a register or in a frame location
 - What instructions must be produced to implement the “view shift”
 - If a parameter will be seen as “memory at offset X from the frame pointer
 - “view shift” must copy the stack pointer to the frame pointer on entry to the procedure

Callee-saved 参数进寄存器会很讨厌。

MIPS

- Formals
 - InFrameAccess(0)
 - InRegAccess(t_{157})
 - InRegAccess(t_{158})
- Why should not g just access r4 and r5 directly?
 - fuction m(x:int, y:int) = (h(y,y), h(x,x))
- Register allocator will choose machine register
 - r4 may be chosen to hold t_{157} and mov may be eliminated finally
- View Shift
 - $sp \leftarrow sp - K$
 - $M[sp+K+0] \leftarrow r2$
 - $t_{157} \leftarrow r4$
 - $t_{158} \leftarrow r5$

像 MIPS 中，传入 r4 和 r5，进来以后需要把它们移走，移到两个临时寄存器里去。参数超过 6 个也要进栈，在这种情况下，我们都要把这些东西放到栈上。这个栈我们可以算出大小，那么上来第一条指令就可以调整栈指针分配 k 个空间。第二条指令是 r2 是第一个参数，r4、r5 是第三第四个参数。R2 是 static link，不能分到寄存器里去，必须要进栈，所以它必须去栈底。后面的参数我们根据 bool list 中是否为 true 来判断是要放到栈上还是要放到寄存器里去。

我们在后端生成，在 tree 上翻译完了以后，头尾还要添加很多东西。

Frame

- frame::Frame type is hidden
- It is a data structure holding
 - The locations of all the formals
 - Instructions required to implement the "view shift"
 - The number of locals allocated so far
 - And the label at which the function's machine code is to begin

这个 frame 是一个隐藏的类型，这个 frame 要包含 locations of all the formals，要么在寄存器里，要么在栈上；还需要包含 view shift 相关的指令，不同的函数的指令会不一样，但是要生成在 frame 里，这是一开始。

可能过段时间，我们还会发现代码有很多局部变量。以及函数是一个 label，在汇编的时候我们是需要继续操作的。

Local Variables

- Frame::AllocLocal(TRUE)
 - Returns an InFrameAccess with an offset from the frame pointer
- Frame::AllocLocal(FALSE)
 - Returns a register InRegAccess(t₄₈₁)
- Register allocator will use as few registers as possible to represent the temporary
- Frame size may also be optimized by noticing when two frame-resident variables could be allocated to the same slot

有了这些东西以后，我们刚才讲到了中间要处理局部变量。来一个局部变量我们是要分配空间的，在 frame 上我们需要 allocate local，在栈上我们就传递 True，在寄存器里就传递 False，等一下我们要讲 escape analysis。并且可能有第二个变量用到的时候第一个变量的生命周期已经结束了，那么这种时候 x 和 y 就可以共享一个空间。

第三个事情，我们的局部变量是可以嵌套的。在 C 里的话，

```
P(x, y)
{
    int v;
    {
        int v;
    }
}
```

当退出 block 的时候，v 还在栈上，我们认为其中变量的 lifetime 是整个函数的生命周期，等到函数结束的时候才会被回收。

Escape Analysis

如果一个函数体中定义的变量没有被嵌套的内函数访问过，那么这个变量就是没有逃逸的（do not escape），在 runtime 的时候，可以分配在临时寄存器中；如果它在内函数中被访问过，那么在 runtime 的时候，这个变量一定要分配在栈上（frame 中），这样调用内函数的时候，内函数就可以通过 static link 进行一次或多次跳转访问到这个变量，此时这个变量就叫做逃逸变量（escape variable）。确定一棵 AST 中所有变量是否是逃逸的行为，称为逃逸分析（escape analysis）。

具体操作方法其实就是在遍历整棵 AST，查看每个变量是否在内函数中被调用过。每当我们看到一个变量或形参的声明时，我们就在 Symbol Table 的一张专门用来做逃逸分析的表中，添加一项<key = 变量 id, value = {depth (静态嵌套深度) = d, escape_ref = 变量的 escape 域的地址}>；当我们看到一个变量在深度 d_1 被访问时，我们就去这张表中查看是否 $d_1 > d$ ，如果确实是这样，说明是在某个内函数中访问到的这个变量，那么我们就可以直接根据表中维护的 escape 的地址，设置对应的 escape 为 True，也就是指明这个变量确实逃逸了，需要在运行时分配在栈上。

C++ 的 escape analysis：（来自 lab5 源码）

```
escape.h
class EscapeEntry {
public:
    int depth_;          // 变量的静态嵌套深度
    bool *escape_;       // 变量的 escape 在 AST 中的位置
    EscapeEntry(int depth, bool *escape) : depth_(depth), escape_(escape) {}
};
```

```

using EscEnv = sym::Table<esc::EscapeEntry>; //逃逸分析表
using EscEnvPtr = sym::Table<esc::EscapeEntry> *;

escape.cc
void AbsynTree::Traverse(esc::EscEnvPtr env) {
    /* TODO: Put your lab5 code here */
}

void SimpleVar::Traverse(esc::EscEnvPtr env, int depth) {
    /* TODO: Put your lab5 code here */
}

```

我们可以看到其实就是一个递归向下遍历整棵 AST 的形式。

2021/11/02

我们把上次的 frame 的一点点剩下的讲完，然后开始讲中间表示，其实这已经是中间表示了，Frame 的结构应该是什么样的。它应该表示一些函数的数据信息，包括函数转化为汇编以后有一个 label 来指定跳转的位置，也就是 call by name。我们上节课讲了 shift of view，比如我们 callee 传参的时候，如果是 x86-64，参数在前面 6 个寄存器，在 callee 看的时候，它认为这些形参都不是在特殊寄存器，它认为都是在 temporary 里。在做 escape analysis 的时候，如果内函数并没有访问到这个形参的话，它认为这些参数都会在临时寄存器里；如果通过 escape analysis 发现某个变量是被内函数访问过的，那么这个变量就应该放在栈上，这些都是 callee 做的事情。而 caller 总是认为前 6 个参数在固定的寄存器中，后面的参数在栈上。Callee 认为参数都不在指定的寄存器中，因为如果在指定的寄存器中会有很多麻烦。上来我们就会写很多代码，比如：

```

pushq %rdi #把参数放到栈上去
movq %rsi, t1

```

这些都是我们需要补的代码，我们用另外的方法生成出来以后贴到函数的头上去。这是我们要在栈上做的事情。

临时的寄存器叫做 temporary，我们就在这里定义了这样的一个类：

<u>temp.h</u>	<u>temp.cc</u>
<pre> class Temp { friend class TempFactory; public: [[nodiscard]] int Int() const; private: int num_; explicit Temp(int num) : num_(num) {} }; class TempFactory { public: static Temp *NewTemp(); private: int temp_id_ = 100; static TempFactory temp_factory; </pre>	<pre> Temp *TempFactory::NewTemp() { Temp *p = new Temp(temp_factory.temp_id_++); std::stringstream stream; stream << 't' << p->num_; Map::Name()->Enter(p,new std::string(stream.str())); return p; } </pre>

```
};
```

我们认为此时有无限的临时寄存器，有限寄存器的情况我们要等寄存器分配的时候再说。

还有一个就是在 `function call` 的时候，比如我们有一个函数 `f`，汇编的时候一定会有一个对应的 `label`，比如说 `f_`。包括说 `jump`、`conditional move` 等都会有 `label`，其实就是一个字符串。因此在代码实现中，我们也有 `label` 这样一个类：

temp.h

```
using Label = sym::Symbol;
class LabelFactory {
public: static Label *NewLabel();
    static Label *NamedLabel(std::string_view name);
    static std::string LabelString(Label *s);
private: int label_id_ = 0;
    static LabelFactory label_factory;
};
```

temp.cc

```
Label *LabelFactory::NewLabel() {
    char buf[100];
    sprintf(buf, "L%d", label_factory.label_id_++);
    return NamedLabel(std::string(buf));
}

Label *LabelFactory::NamedLabel(std::string_view s) {
    return sym::Symbol::UniqueSymbol(s);
}

std::string LabelFactory::LabelString(Label *s) { return
s->Name(); }
```

在这里我们就可以 `new` 这个 `label`，其实就是把一个 `symbol` 和一个指针关联起来，这样在之后我们就可以用这个 `label` 了。我们现在已经有了 `Frame` 类来描述我们这个栈、`Access` 类来描述我们的变量，这个变量可能在临时寄存器上也有可能在 `Frame` 上。临时寄存器我们就用 `Temp` 类来描述，最后还有一个 `label` 类就是来描述汇编中的 `Label` 的。

下边我们来看一下 `Translation` 的过程。

`Type-checking` 是自底向上扫描 `AST`，那么扫描树的时候我们也可以处理 `declaration`，一边做一边 `environment` 就有了。对于 `declaration` 的处理是自左向右的，比如：

```
Let dec1; dec2; dec3; in expseq
```

它是自左向右处理 `declarations` 的，并且处理完右边的 `expseq` 就可以直接开始用这些 `declaration`。

我们马上要讲到的事情是要把 `high-level` 的 `AST` 变成一个中间表示 `IR` (`intermediate representation`)，也就是第 7 章中的 `tree`。它是机器语言的一种抽象，因为我们要支持不同的高级语言和不同的机器语言，支持不同的高级语言我们是用 `AST` 来做到的，也就是用 `AST` 来表示不同的高级语言；支持不同的机器语言就是通过我们这个中间表示 `tree language` 来做到的，所以接下来我们要做的事情就是把我们的 `AST` 变成我们的 `tree language`。

书上给的方法就是我们扫描整个 `AST` 一次，同时做 `type-checking` 和 `translation`。`type-checking` 还是 `semantic analysis` 的过程，里面有一部分函数就是在做 `translation`，一边扫描一边我们生成 `tree language`。注意，我们生成的 `tree language` 是体系结构无关的，但是我们最终生成的汇编语言一定是和 `ISA` (指令集架构) 有关的，所以我们如下图所示做了两层抽象的封装：

```

semant.c
translate.h
translation.c
frame.h      temp.h
μframe.c     temp.c

```

这样.c 在 implement 不同函数的时候，可以使用不同的体系结构的指令集，但是向上提供接口的时候，frame.h 和 temp.h 就屏蔽掉了底层实现的细节。包括 frame 我们是不希望被 semant 看到，但是一定要给 translation 看到的，所以我们沿用这样的思路把底层实现以接口的形式暴露给上层。

在 translation.h 中有一个 tr 命名空间，其中我们定义了一个 Level 类，其实就是 translation 里的 frame，一个 Level 类包含了一个 Frame，但是它还有别的东西，比如一个 Parent，就是哪个函数定义我的。当然我们 tiger 语言最外面有一个 outermost 的 frame，里面定义了 main。所以除了 outermost，每个函数都是可以找到自己的 parent 的。

第二件事情就是我们把 frame.h 里的 Access 类，重新定义成了在 translation.h 里的 Access 类。这两个的区别就是 Frame::Access 是针对变量的，表明一个变量要在寄存器中要么在栈上，而在 translation 中，增加了这个 Access 是在哪个 frame（函数里）。我们多了一层封装，我们同时扫描的时候能看到的数据结构都是 translation 里的，frame 里的结构我们是看不到的。

```

translate.h
class Access {
public:
    Level *level_;
    frame::Access *access_;
    Access(Level *level, frame::Access *access) : level_(level), access_(access) {}
    static Access *AllocLocal(Level *level, bool escape);
};

class Level {
public:
    frame::Frame *frame_;
    Level *parent_;
    /* TODO: Put your lab5 code here */
};

```

在 type-checking 的时候，我们会构造 symbol table，我们只关心变量的类型是什么。在 translation 的时候，我们遇到一个变量的时候，还要关系变量应该是放在寄存器中还是放在栈上，我们在做过 escape analysis 以后，要把 access 的信息也要填到 function 里去。这里填的就是 function 的 local 变量。

```

env.h
// 蓝色表示是 type-checking 中需要关注的变量，而 橙色 表示是 translation 中需要关注的变量
class VarEntry : public EnvEntry {
public:
    tr::Access *access_;
    type::Ty *ty_;
};

```

```

// For lab4(semantic analysis) only
explicit VarEntry(type::Ty *ty, bool readonly = false)
    : EnvEntry(readonly), ty_(ty), access_(nullptr){};

// For lab5(translate IR tree)
VarEntry(tr::Access *access, type::Ty *ty, bool readonly = false)
    : EnvEntry(readonly), ty_(ty), access_(access){};
};

class FunEntry : public EnvEntry {
public:
    tr::Level *level_;
    temp::Label *label_;
    type::TyList *formals_;
    type::Ty *result_;

    // For lab4(semantic analysis) only
    FunEntry(type::TyList *formals, type::Ty *result)
        : formals_(formals), result_(result), level_(nullptr), label_(nullptr) {}

    // For lab5(translate IR tree)
    FunEntry(tr::Level *level, temp::Label *label, type::TyList *formals,
             type::Ty *result)
        : formals_(formals), result_(result), level_(level), label_(label) {}
};

```

我们在遇到一个 `function` 的时候，我们要填入 `function` 的 `frame`（也就是 `level`），另外还要填入 `label`。这个就是对我们原先的 `EnvEntry` 做了一个修改，增加了一些 `translation` 关注的变量。

那么我们新增了这两个东西以后，一遍扫描（包含 `type-checking` 和 `translation`）合起来就叫 `semant`。

我们在做 `translation` 的时候：

- 如果我们在 `translate` 的一个 `function`，它会调用 `tr::Level::NewLevel` 这个函数，`new` 出来以后，就会把这个 `level` 记录到 `FunEntry` 里去。
- 如果我们 `translate` 的是一个 `variable declaration`，那么它就会去调用 `tr::Access::AllocLocal(level, escape_)`。`escape_` 是在 `semant` 阶段的最开始，遍历整棵 AST 进行 `escape analysis` 得到的。`Semant` 会在生成机器码的时候使用到这个 `Access`。

第三个事情就是我们有 `static link`，它不是一个程序中本身有的东西，而是我们为了让内函数访问外函数的变量增加出来的结构，当然如果不用 `static link` 的话，我们也可以使用 `display` 或者 λ `lifting`，不过在我们的 `compiler` 中用的还是 `static link`。在 `translate` 的时候，我们看到的如下：

当我们在函数 `g` 中尝试调用 `f(x,y)` 的时候，我们其实就是在调用：

```
tr::Level::NewLevel(level(g), {false, false});
```

在这个函数内部，因为 Level 只是对 frame::Frame 的一层封装，它事实上会在形参的最前面添加上一个 true，再调用 NewFrame(label, formals)；所以 NewLevel 传入的是两个参数而 NewFrame 传入的是三个参数。

刚才我们讲的有个 outermost，我们现在最外面的就当做 main_level_，在 ProgTr 里。这就是最外的一层特殊函数。当然我们还有一个 NewLevel 函数，里面主要是 parent, label name 以及一个 List 定义了 escape analysis 的信息。

整个 Tiger 最外面的一层叫做 main Tiger program，它里面包含一个 main 函数，并且和 main 函数并列的还有一些 library function (print、scan、stringequal 等)。

这就是我们的第六章。下面我们开始讲第七章。

我们要有一个更像机器的中间表示 IR。实际上 AST 也算一种中间表示，属于 High IR。现在我们的 tree language 叫做 Middle IR，下面我们还有一个 Assem 叫做 Lower IR。编译的过程就是不断的中间表示变换的过程。我们通过 bison 把 high level 的过程变成 AST，把 HIR 变成 IR 的过程会复杂一点。

IR 要求非常简单，AST 的一部分翻译成 IR，IR 的 group 就可以真正变成机器指令。现在我们就来介绍这个 tree language

Tree Language

Expression

Tree 分成两部分，第一部分叫做 Expression，第二部分是 Statement。我们先来看 expression。Expression 分成如下：

1. Constant 常数
2. Name (symbolic constant) : 比如 goto .L 这个.L 就是 name
3. Temporary (virtual register)
4. Binary operation
5. Memory access (访存)：比如说(%rax)就是一个 expression，要访问某个内存地址上的一个 value。
6. Expression sequence：前面的都比较像汇编，而这里的不像，就是(seq, exp)。比如说 exp 是 a+5，而 seq 就是 a=5，这就是一个 side-effect，最终的结果就是 10.
7. Function call

这样我们就可以把 expression 定义成一个类，在 tree 的命名空间中。我们接下来就根据这 7 种情况定义 7 个子类。

```

class Exp {
public:
/* interface for subclass to implement */
};

class BinopExp : public Exp {
public:
    BinOp op_;
    Exp *left_, *right_;
    BinopExp(BinOp op, Exp *left, Exp *right)
        : op_(op), left_(left), right_(right) {}
    /* implement Exp interface */
};

enum BinOp { PLUS_OP, MINUS_OP, MUL_OP, DIV_OP, AND_OP,
    OR_OP, LSHIFT_OP, RSHIFT_OP, ARSHIFT_OP, XOR_OP};

```

注意到这里没有关系表达式。因为是二元运算，所以就有左右子树。

```

tree::BinopExp(tree::BinOp, tree::Exp*, tree::Exp*) ;
tree::MemExp(tree::Exp*) ;
tree::TempExp(temp::Temp*) ;
tree::EseqExp(tree::Stm*, tree::Exp*) ;
tree::NameExp(temp::Label*) ;
tree::ConstExp(int) ;
tree::CallExp(tree::Exp*, tree::ExpList*) ;

tree::ExpList(std::list<tree::Exp *>) ;

```

我们就这样一个个去写。

为了在后面讲解起来方便，我们定义一些缩写：

CONST(i): 整数常量 i

NAME(n): 符号常量 (Label) n

TEMP(t): 临时寄存器 t

Binary(o, e₁, e₂): 对表达式 e₁ 和 e₂ 进行对应 o 的二元运算

Mem(e): 从内存地址 e 开始，访问 wordSize 个 Byte (int 为 4)

ESEQ(s, e): 也就是 (seq, exp)

Call(f, l): 也就是调用 f，传入的参数是 l

Statement

在我们定义 Tree 中的 expression 的时候，已经和 tiger 没有关系了，我们更希望抽象出一种机器语言。

第二类的 Tree 就是 statement，分成如下：

1. MOVE
2. Expression: 把刚才的 expression class 拿过来执行计算，但是我们不要结果。比如说 a++, 我们并没有把 a 的 old value 拿出来。

3. **Jump:** 无条件跳转
4. **Conditional Jump:** 有条件跳转
5. **Statement Sequence**
6. **Label:** 注意 Label 和 Exprssion 中的 Name 的区别。Name 是使用这个 symbol, 比如 goto .L, 而 Label 是定义这个 symbol, 即.L:

```

class Stm {
public:
/* interface for subclass to implement */
};

class SeqStm : public Stm {
public:
    Stm *left_, *right_;

    SeqStm(Stm *left, Stm *right) : left_(left), right_(right) { assert(left); }
/* implement interface */
};

```

同样地，我们会定义 6 个子类。

```

tree::SeqStm(tree::Stm, tree::Stm) ;
tree::LabelStm(temp::Label) ;
tree::JumpStm(tree::Exp*, std::vector<temp::Label *> *) ;
tree::CjumpStm(tree::RelOp, tree::Exp*, tree::Exp*, temp::Label*,
temp::Label*) ;
tree::MoveStm(tree::Exp*, tree::Exp*) ;
tree::ExpStm(tree::Exp*) ;

tree::StmList(std::list<Stm *>) ;

```

在 Conditional Jump 的时候，会有 Relational Operator，也就是：

```

enum RelOp {
EQ_OP, NE_OP, LT_OP, GT_OP, LE_OP, GE_OP,ULT_OP, ULE_OP, UGT_OP, UGE_OP
};

```

类似地，我们定义如下的一些缩写：

MOVE: 分为两类，一类是把表达式移动到寄存器中，一类是把表达式移动到内存中，即：

MOVE(MEM(e_1), e)

MOVE(TEMP t , e)

Exp: 只求值不要结果

Jump(e , $labs$): 跳转到 e ，这个 e 可能是一个 label 或者是一个地址的表达式（其值为 $labs$ ）

CJump(o , e_1 , e_2 , t , f): conditional jump 写的比较复杂一点，必须是两个表达式做关系运算，这和汇编里的条件跳转比较一致，比如 **JL, JNE** 等。和汇编中不一样的是，一般汇编中只有一个 target，条件成立跳转，条件不成立就继续执行。但是在我们这里，如果条件成立就跳转到 t ，条件不成立就跳转到 f 。

SEQ(s_1, s_2): 就是两个 statement 合在一起

LABEL(n): 就是定义了 n 作为当前机器地址的 label, 要使用的时候就可以使用 Name(n)

这个是我们的 tree, 注意一遍扫描的时候, tree 也是不让 type-checking 看到的。type-checking 能看到的是 translation 出来的东西。

现在我们有了一个 Exp, 这就是我们翻译出来的 Tree。这个 class 有三个子类, 本来我们讲 AST 表示了一个 expression, 那么我们翻译出来得到 expression 就可以了。翻译出来的只有三样东西, Ex、Nx 和 Cx。

翻译出来的 Ex 就是 expression; 第二个 Nx 就是一个 statement; 第三个 Cx 比较复杂, 是一个 statement 加两个 patchlist, 定义如下:

```
/* translate.cc */
class Cx {
public:
    PatchList trues;
    PatchList falses;
    tree::Stm *stm;
    Cx(PatchList *trues, PatchList *falses, tree::Stm *stm)
        : trues(trues), falses(falses), stm(stm) {}
};
```

Cx 我们还看不太清楚是什么东西, 前两者是比较清楚的, 要计算表达式的值我们就选用 Ex, 如果不需要表达式的值, 就选 Nx。

下面我们来看一下, 原来在做 type-checking 的时候, 同时做了 translation, 也就是一遍扫描的时候, 返回的就是如下的一个 ExpAndTy:

```
class ExpAndTy {
public: tr::Exp *exp;
    Type::Ty *ty;
    ExpAndTy(tr::Exp *exp, type::Ty *ty) : exp(exp), ty(ty) {}
};

ExpAndTy *absyn::Var::Translate (env::VEnvPtr venv, env::TEnvPtr tenv);
ExpAndTy *absyn::Exp::Translate (env::VEnvPtr venv, env::TEnvPtr tenv);
void absyn::Dec::Translate (env::VEnvPtr venv, env::TEnvPtr tenv);
type::Ty *absyn::Ty::Translate (env::TEnvPtr tenv);
```

下面我们来看 Cx 到底是什么东西, 为什么会变的这么复杂。简单来看, $x < 5$ 就会被翻译成一个 Cx:

stm = CJUMP(LT, x, CONST(5), NULL, NULL)

Cjump 后面应该要跟两个 label, 在 Cx 里, 我们没有把 true 和 false 对应的 label 没有填进去, 把这两个变成了两个 PatchList trues 和 falses。也就是把这两个 NULL 的位置, 指针指到 True 的 label 的位置和 False 的 label 的位置。

为什么要这样处理呢? 在 Tiger 的 AST 中, 是没有逻辑运算的, Tiger 语言是有逻辑运算的, 因为要支持 short-cut。支持 short-cut, 也就是对于表达式 $a > b \mid c < d$, 如果 $a > b$ 成立的时候, 就不要计算 $c < d$ 了。我们在生成 AST 的时候, 我们把&和|变成了 if expression。

也就是对于 tiger 语句:

$\text{if}(a > b \mid c < d) \text{then } e_1 \text{ else } e_2$

要处理嵌套 if 的 short-cut，本书作者就认为他的这种方式比较好。

$a > b \mid c < d$ 的翻译语句如下：

```
temp::Label::*z = temp::LabelFactory::NewLabel();
tree::CJumpStm *s1 = new tree::CjumpStm(tree::GT_OP, a, b, NULL, z);
tree::CJumpStm *s2 = new tree::CjumpStm(tree::LT_OP, c, d, NULL, NULL);
tree::Stm *s1 = tree::SeqStm(s1, new tree::SeqStm(new tree::LabelStm(z), s2));

tr::PatchList trues = PatchList({&s1->true_label_, &s2->true_label_});
tr::PatchList falses = PatchList({&s2->false_label_});

tr::Exp e1 = new Tr::CxExp(trues, falses, s1);
```

在上例中，对于 $s1$ 来说，如果 $a > b$ 的值为 True，它暂时还不知道要跳转到哪里，但是值如果值为 False，它知道要跳转到判定第二个条件，所以我们可以 $s1$ 和 $s2$ 之间插入一个 Label z ，使其 False 的时候跳转到 z 。而对于 $s2$ 来说，它暂时都不知道要跳转到哪里去。但是我们知道对于 $s1$ 和 $s2$ 的 True 的情况，它们应该是跳转到一个相同的地方的。注意上例中，最后我们还是创建了一个 $CxExp$ ，没有转换为 0 和 1。这是因为我们的关系表达式可能非常长。

但是如果我真的需要关系表达式来求值怎么办呢？比如 $flag := (a > b) \mid (c < d)$ ；这时候就需要把 Cx 转化为 Ex 。

```
class Exp {
public:
    [[nodiscard]] virtual tree::Exp *UnEx() const = 0;
    [[nodiscard]] virtual tree::Stm *UnNx() const = 0;
    [[nodiscard]] virtual Cx UnCx (err::ErrorMsg *errormsg) const = 0;
};
```

所以在 Exp 中定义了三种强制类型转换的函数。分别把三样东西转换成 Exp 、 Stm 和 Cx 。我们先来看 Nx 和 Ex 怎么转化为 Ex ：

```
tree::Exp *NxExp::UnEx() const override {
    return new tree::EseqExp(stm_, new tree::ConstExp(0));
}

tree::Exp *ExExp::UnEx() const override {
    return exp_;
}
```

要么就是在后面加一个 0，要么就是把自己返回。

那么我们怎么把 Cx 转化为 Ex 呢？

上例的 Cx 中，有三个位置和两个 label，现在我们要做的事情就是把 Cx 的值求出来。接下来就是我们要把 true 跳转的地方设置一个寄存器值为 1，false 要跳转到地方设置这个值为 0，这样 Cx 就变成了 Ex 。

```
tree::Exp *CxExp::UnEx() const override {
    temp::Temp *r = temp::TempFactory::NewTemp();
    temp::Label *t = temp::TempFactory::NewLabel();
    temp::Label *f = temp::TempFactory::NewLabel();
    tr::DoPatch(trues_, t); tr::DoPatch(falses_, f); //回填操作
}
```

```

return new tree::EseqExp(new tree::MoveStm(new tree::TempExp(r), new tree::ConstExp(1)),
    new tree::EseqExp(cx.stm,
        new tree::EseqExp(new tree::LabelStm(f),
            new tree::EseqExp(new tree::MoveStm(new tree::TempExp(r), new tree::ConstExp(0)),
                new tree::EseqExp(new tree::LabelStm(t), new tree::TempExp(r))))));
}

using PatchList = std::list<temp::Label**>;
void DoPatch(PatchList t_list, temp::Label *label) {
    for (t : t_list)
        *t = label ;
}

PatchList JoinPatch(PatchList first, PatchList second) {
    /* use std::list function can simply implement this */
}

```

其中 `DoPatch` 就是把对应的 `PatchList` 都用第二个参数填入。其实很容易理解，我们初始化临时寄存器为 1，考察 `conditional jump` 这个 `statement`，如果为 `TRUE`，直接跳转到 `t` 这个 `label`，返回 `TempExp`，也就是 1。如果为 `FALSE`，那么就跳转到 `f` 这个 `label`，重新设置寄存器的值为 0，并且返回。

第八章我们要把这个代码正规化，这样才能生成代码。

注意 `Nx` 是不能转化为 `Cx` 的，因为它没有值，所以需要报错。

我们还是一遍扫描做两件事情，要把两件事情分隔好，不要让一边可以看到对面的乱七八糟的太多。

比如说我们可以在 `translate` 中添加 `SimpleVar` 接口，可以转化成一个 `tr::Exp`，其实就是一个 `Ex`。它要去翻译的时候，`Access` 已经可以从表中有了，也知道在哪个 `level`，我们传进去以后就可以处理 `SimpleVar` 了。在 `translation` 中我们是看不到 `tree` 中的东西，看到的都是 `tr`。

另外，我们在 `frame.h` 中，还有一个 `RegManager`，因为我们有一些特殊的寄存器，比如 `frame pointer`，比如 `return value` 的 `register` 称为 `rv`，有些机器它的 `return address` 也是一个特殊的寄存器。

```

/* frame.h */
Class RegManager {
...
virtual temp::Temp *FP(void);
virtual int WordSize() = 0;
}
tree::Exp *Access::ToExp(tree::Exp *framePtr);
If acc is inReg, the result is simply the register
If acc is inFrame(k), the result is
    MEM(BINOP(PLUS, TEMP(fp), CONST(k)))

```

下边我们来看，这边要定义一个 `ToExp` 函数。我们要把 `tree node` 产生出来。`Access` 如果是在寄存器中的变量，那么它返回出来的就是寄存器，如果是在内存中的，那么就会使用这个 `framepoint` 来计算 `memory` 的地址，来拿到这个变量。`AST` 中是一个 `variable node`，在 `Tree` 中就变成了一个 `Expression Node`。

2021/11/5

上节课主要讲了 `Cx`、`Nx`、`Ex` 三个部分。`Lab5` 大部分都是关于翻译，算法上没有什么难度，主要是工程上要考虑到各种各样的结构。

Translate simple variables

- Add an interface to Semant
 - `tr::Exp tr::SimpleVar(tr::Access, tr::Level);`
- Semant
 - never gets its hands dirty with a `tree::Exp`
 - should not contain any direct reference to the Tree or Frame module.
- `tr::Access`
 - can be obtained by querying `venv`
 - class `Accessss` { public: `Level level_;` `frame::Access access_;` };

语义分析需要关系程序本身，而不需要关心运行时，栈帧都是通过 `Access` 结构之间传来传去的，但是对于 `Semant` 模块是不可见的。这里的问题是能不能拆成两个 `parse` 来做呢？`Semantic` 结束的时候，`imperative symbol table` 就被完全销毁掉了。

`Tr::SimpleVar()` 也就是在 `translate` 的时候顺带做了语义分析。

刚才说过了 `Semant` 不会碰 `tree::Exp` 这种事情，而关注的是把 `symbol variable` 放进 `symbol table`。`Tr::Access` 是通过 `venv` 获取的，是变量的环境中获取的（变量 `symbol->变量 type`）。通过这个 `Access`，里面有这个 `Scope` 的层次，`frame` 里面的 `access`（在栈上和在寄存器中）。寄存器我们要分配编号，而栈上的我们需要分配偏移量。

Translate simple variables

```
/* frame.h */  
Class RegManager {  
...  
virtual temp::Temp *FP(void) ;  
virtual int WordSize() = 0;  
}  
tree::Exp *Access::ToExp(tree::Exp *framePtr);  
If acc is inReg, the result is simply the register  
If acc is inFrame(k), the result is  
    MEM(BINOP(PLUS, TEMP(fp), CONST(k)))
```

这里我们加了一个 register manager，抽象一下硬件上的寄存器，在不同的硬件上有不同的寄存器，所以我们专门定义了 RegManager，我们可以继承这个 Manager 在不同的硬件上做实现。*FP 不一定要返回一个固定的寄存器，wordsize 是根据硬件的情况来决定的。下面还定义了一些和 Access 相关的东西。Frame = k 对应的是从栈帧到实际变量的偏移量，也就是位置(fp + k)上的值。

Translate simple variables

- When a variable x is used, semantic calls
 - tr::Exp tr::SimpleVar(tr::Access access, tr::Level level);
 - Eventually calls
tree::Exp *frame::Access::ToExp(tree::Exp *framePtr);
- If access is in the level
 - MEM(BINOP(PLUS, TEMP(FP), CONST(k)))
- Otherwise, static link must be used
 - MEM(+(CONST k, MEM(+(CONST 2*wordsize, (...MEM(+(CONST 2*wordsize, TEMP FP) ...))))))
 - Follows n-1 static links in nested functions
 - k is the offset of x in the defined function

Semant 和 translate 在同一个 parse 做完，在我们要 translate 的时候，我们调用 tr::SimpleVar，最终它会调用到 ToExp，翻译成一个 memory 的 exp 或者是一个寄存器的 exp。

因为我们的语言包括嵌套函数，这是 tiger 不同于 C 和 C++ 的一点。它就需要用我们之前提到的 static link 来取 memory 的位置。Static link 是通过一个固定的位置把栈帧连在一起的。因为它在栈上的位置是固定的，只要我们拿到 FP，就可以通过 FP+4 来拿到 static link 往上走，并且编译器会计算出来需要走几层，所以我们只需要往上走到对应的位置即可。

数组和 Record 的翻译

Translate subscripting and field selection

- $A[i]$
 - Calculate the address $(i-l)*s + a$
 - l is the lower bound of the index range
 - s is the size of each array element
 - a is the base address of the array elements
 - If a is global, $a-l*s$ can be done at compile time
- $a.f$
 - Add the constant field offset of f to the address a

因为它是定长比较简单，对于 i 的访问就是 $(i-l) * s + a$ ， a 是 array 的起始地址。在 C 语言里的情况，lower bound = 0，那么要访问 $A[i]$ ，那就是 $\text{addr}(A) + i * 4$ (4 是 int 数组的情况)。

field 就是地址加上整个在里头的 offset，比如 ICS 里的 alignment 和 padding。下面主要是要讲一下 offset 和 element 还是比较容易算的。我们主要的还是计算变量的地址。

数组和 record 在不同语言中代表什么含义呢？

Translate array variables

- In Pascal

- Array-valued variable stands for contents of the array

```
var a, b : array [1..12] of integer
```

```
begin
```

```
    a := b
```

```
end ;
```

Copies the contents of array a into array b

这就是 a 和 b 代表整个数组的内容，所以 a:=b 就是把 b 中的内容整个 copy 到 a 里。a 出现在了赋值语句的左边，所以取的是这个表达式的 lvalue (基本上指的是地址)。在 C 里的左值都称为一个 scalar。

- In C

- It is a pointer constant

```
{
```

```
    int a[12], b[12];
```

```
    a = b; /* illegal statement because a is const */
```

```
}
```

```
{
```

```
    int a[12], *b;
```

```
    b = a; /* legal statement */
```

```
    /* B points to the beginning of the array a */
```

```
}
```

而在 C 里，数组的 a 代表是一个地址常量。因为 a 不能当做左值，所以前半部分 a=b 是不合法的。而后半部分是把常量赋值给一个变量，这是合法的。

- In Tiger, Java and ML

- It behaves like a pointer

- In Tiger

- Array name is not a pointer constant

- Created and initialize by $t_a[n]$ of i

- t_a is array type

- n is the number of elements

- i is the initial value of each element

在 tiger 和 java 中，array 像一个指针一样，但它不是一个常量。在 tiger 中，12 个整数的一个数组，就是在堆 (heap) 上找 48 个 Byte，然后在栈上再分配一个 8 个 Byte 的地址空间，这里填的是数组的首地址，在 tiger 中数组名代表了一个指针变量，所以在 tiger 中虽然没有指针，但是 array 和 record 的行为和 pointer 是一样的。它们所指向的空间都是放在堆上的，所以不同的语言对于不同的东西是用不同的方式来处理的。

```

let
    type intArray = array of int
    var a := intArray[12] of 0
        var b := intArray[12] of 7
    in  a := b
    end
    - the array variable a ends up points to the same 12
      sevens as the variable b
    - the original 12 zeros allocated for a are discarded
• In Tiger
    - Record values are also pointers

```

在 tiger 中，这样处理就是允许的。a 赋值为 b 之后，a 所指向的原先的 48 个 Byte 就变成 garbage 了，等我们讲完 garbage collection 就会回收掉这块堆上的空间。

Structured L-values

- l-value is the result of an expression
 - that can occur on the left of an assignment
- r-value is one
 - that can only appear on the right of an assignment
- l-value denotes a location
 - Assigning to an l-value means storing to that address
- l-value can occur on the right of an assignment
 - It means the contents of the location
- An integer or pointer value is a "scalar"
 - It has only one component
- Structured l-values
 - Array variables in Pascal, struct variables in C

Translate subscripting and field selection

- For structured l-value
 - tree::Exp tree::MemExp(tree::Exp*, int size) should be introduced
 - an address calculation should be MEM(+TEMP fp, CONST K), S)
 - S indicates size of object to be stored

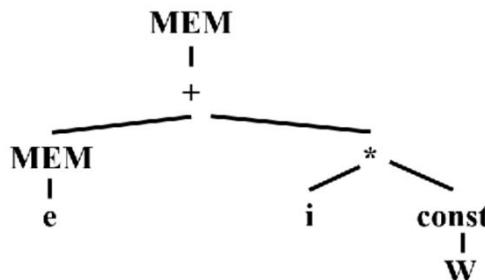
现在我们的 mem expression 只是给了一个地址，如果我们要支持 structured l-value，那么我们还需要加一个 size。从 frame pointer 加一个常数，然后再说要多少个。

- In Tiger there is no structured l-valued
 - The "base address" e of the array is really the contents of a pointer variable
 - So MEM is required to fetch this base address MEM(c) which is one-word pointer value p
 - The contents of address p, p+W, p+2W, ... will be the elements the array

我们在 tiger 中的 tree 就没有 size 了，只有首地址。在 tiger 中，数组也好，结构也好，是一个 aggregated data type。

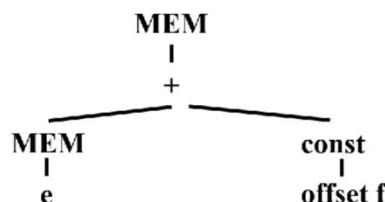
Translate subscripting

- $a[i]$ is just
- $\text{MEM}(\text{+(MEM}(e), *(i, \text{CONST } w))$



这就是 $a[i]$ 用树结构怎么表示，我们怎么计算出 memory 的地址。**a** 其实是栈上的一个地址或者是一个寄存器，我们其实是要把这里面的值拿出来作为地址，在堆上找这个地址 $\text{Mem}(e) + i * w$ 。然后继续对这个地址访存找到 $a[i]$ 的值。到底是放在寄存器上还是放在栈上，这是做 escape analysis 的结果。

- $A.f$
- $\text{MEM}(\text{+(MEM}(e), \text{CONST offset } f))$



Filed 是类似的，编译的时候我们可以根据 field 直接算出 offset。

A Sermon of Safety

- Memory bugs are very common
- Array bound check
 - Time consuming
 - Optimization
- Null check

我们讲了 array 和 record，record 比较简单，引用 field 的时候一定 declaration 存在的，一定有个 field 是 **f**。但是数组的话，**i** 可能越界（负的、太大），如果 **i** 不在合法的范围内，访问数组就会导致程序出错。这是 memory 中的 bug。在 C 中产生 garbage 的话，其实就是 memory leak，如果在循环/递归中一直做这件事情，就很快没有空间了。所以我们在编译的时候可以做一些检查，比如 bound check。碰到 **i** 的话，我们要看一下这个 **i** 是不是在范围内。Java 的 interpreter 就是这么做的，每次访问的时候都做一个 bound checking，但是现在我们要把下标和两个 bound 做 checking，比较慢。但是 Java 确定 **i** 不会越界的时候，可以把 bound

checking 去掉。

到目前为止，我们讲了简单变量、array、record 怎么从 AST 生成一个 IR tree。下面我们将来讲 arithmetic（算术运算）怎么变成 tree。

- In Tiger it is easy
- Binaries are straight forward
- No unary
 - Unary negation can be implemented as subtraction from zero
 - Unary complement can be implemented as XOR with all ones
- No floating point

在 tiger 中比较简单，算数也不多。逻辑运算、关系运算都变成 Cx 了。等会儿我们看 if 的时候，一个要支持 short-cut，一个是不想支持太多临时变量，当我们需要的时候才回填回去。还没翻译完的时候，它就留着。

因为 AST 中也没有 Unary、求补我们可以使用异或来代替。翻译就没有太多的事情。

Translate conditionals

- If e_1 then e_2 else e_3
 - Treat e_1 as a Cx (apply unCx to e_1)
 - Treat e_2 and e_3 as Ex (apply unEx to e_2 and e_3)
 - Make two labels t and f to which the conditional will branch
 - Allocate a temporary r,
 - after label t, move e_2 to r
 - after label f, move e_3 to r
 - Both branches should finish by jumping newly created "joint" label

这个翻译起来就有点事情，我们可以再次看到为什么在 tiger 中要选出特别不直观的 Cx 来。Ex 就是一个表达式，Nx 就是一个 tree，而 Cx 就是非常奇怪。

其实 Cx 就是说 relational 和 logical 的 expression 最常见就出现在 e_1 这个位置上。而且出现的时候，我们还要做 short-cut。我们在翻译 e_1 的时候，它的两个 target，true 和 false 都是要跳转的。这样的表示出来的 Cx 就会和 assembly 不一样，因为 assembly 讲的是 true 要跳，而 false 就继续执行后面的。

在现在来讲，他就要两跳。在翻译 e_1 的时候，两跳的 target 就不给他确定下来。因为它希望在这个地方代码可以间接一点，否则我们还需要再来一遍。而到了后面就是把 true 的地方填入 e_2 ，false 的地方填入 e_3 。

因此，这里我们引入了寄存器 r，它在 true 的时候赋值为 e_2 表达式的值，在 false 的时候赋值为 e_3 表达式的值，最终两个分支会有一个 joint label 就是 r。

事实上， e_1 不一定是 Cx，所以我们引入了 unCx，强制把 e_1 变成一个 C jump。

- Treat all of these cases specially
 - If e2 and e3 are both "statements" (N_x)
 - Translate the result as statement
 - If e2 or e3 is a Cx

if $x < 5$ then $a > b$ else 0
 $x < 5$ is translates as a CX

true = { t }

false = { f }

stmt = CJUMP(LT, x, CONST(5), □_t, □_f)

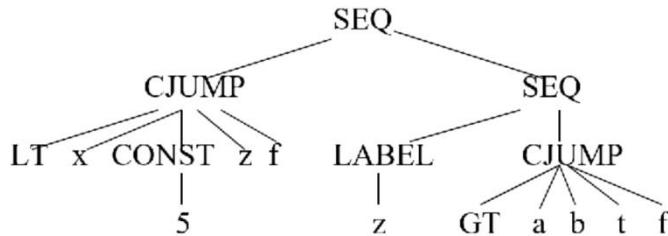
e1 和 e2 都是 relational 的东西。

Translate conditionals

if $x < 5$ then $a > b$ else 0

translate $x < 5$ as $Cx(s_1)$, translate $a > b$ as $Cx(s_2)$

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



- String comparison
 - For string equal just calls runtime -system function `stringEqual`
 - For string unequal just calls runtime -system function `stringEqual` then complements the result

If 拿出来主要是展现一下，为什么要有 Cx 这个东西。在讲到 relational operator 的时候，有一个特殊的是比较两个 string 是否相等。它实际上是调用了一个库，run time system，写出来的 behavior 像 relational 的 operator。

Strings in Assembly

```

.section .data
string:
    .ascii "hello, world\n"
string_end:
    .equ len, string_end - string
.section .text
.globl main
main:
    Printf("hello world");
  
```

`hello world` 是一个字符串的常量，放在汇编中的 `datasection` 中。加点的东西是伪汇编，这是一些宏，告诉汇编器最终生成的 `.o`(relocatable object file)这些东西该放到哪里去。`len` 这里也可以当做参数直接用。

在 `tiger` 中是函数嵌套，而没有全局变量，它依然会产生出 `datasection` (`C` 里是全局变量和字符串常量产生的)，因为我们有 `string constant`。

- How to represent a string literal
 - Fixed length such as in Pascal
 - Varied length with `\0` terminated such as in C
 - One word followed by characters such as in Tiger
- How to generate string literal(string function)
 - Label definition
 - Pseudo code to generating word constant
 - Pseudo code to emit character data
 - It is implemented in Frame module

在 `C` 里，加一个 `\0` 作为字符串的结尾；在 `Pascal` 中是固定长度的字符串处理。而 `tiger` 支持 128 个 `ascii` 字符全部出现在字符串里，分成两个部分，一个是 `word` (4Byte，最长表示 2^{32} 的一个字符串的长度)，后面放的是字符串。这样我们就不需要找 `\0` 来计算字符串的长度了。

我们要生成字符串代码的时候，`label` 是肯定要的，第二个后头会跟两个 `pseudo code`。然后往里放常数和字符串。这是 `tiger` 中的方式。

- For string literal `lit`
 - Make a new label `lab`
 - Returns the tree `tree::NameExp(lab)`
- Puts the assembly language fragment
`frame::StringFrag(lab, lit)` onto a global list
- All string operations are performed in functions provided by the runtime system
- These functions heap-allocate space for their results, and return pointers

生成好了以后，`label` 就能用了。要用字符串的时候，其实就是一个 `Name expression`。它可能是一个字符串的首地址，拿了这个就可以访问我们的字符串。

`Fragment` 其实就是 `datasection` 中的东西。

还有一个操作是字符串的拼接操作，拼出来的字符串是在运行过程中生成的，我们就要放到堆上，返回出来的首地址，我们就当 `label` 来用。

Record and Array Creation

- Record creation and initialization
 - { f1 = e1 ; f2 = e2; ... ; fn = en }
- A record may outlive the procedure activation that creates it
- It cannot be allocated on the stack
- It must be allocated on the heap

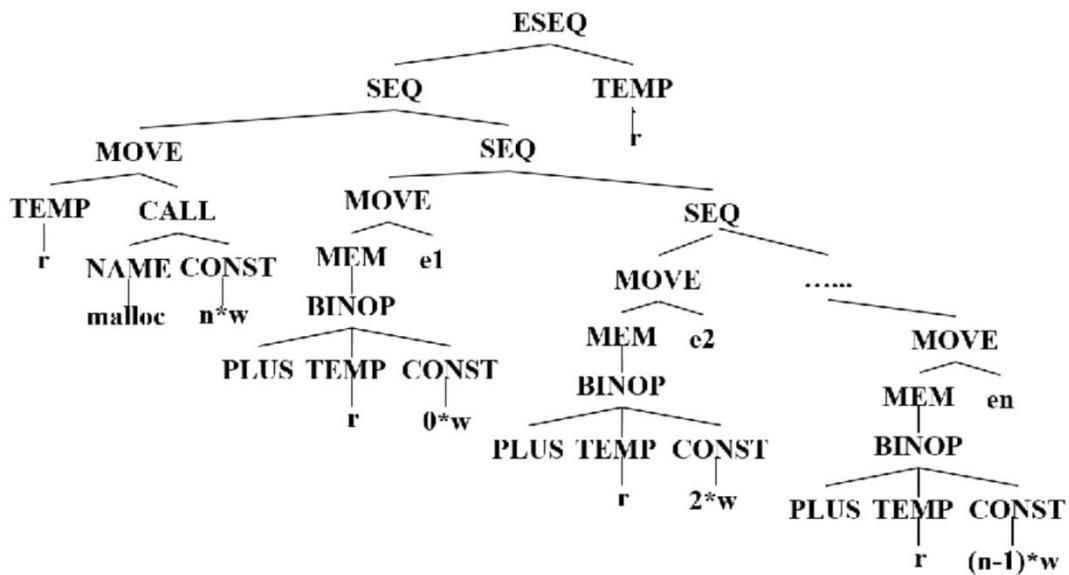
在声明 array 和 record 的时候，我们会对其赋初值。它们是都在堆上的。

Record Creation

- Call an external memory-allocation function
 - creates an n-word area
 - returns the pointer into a new temporary r
- A series of MOVE trees can initialize offsets 0, 1w, 2w, ..., (n-1)w from r with the translations of expression ei
- The result of the whole expression is r

声明了一个 n 个 field 的 record，我们就去堆上要一个 4n Byte 的空间，把空间地址放给 temp 寄存器 r。我们有了首地址之后，可以对空间里的每一个 field 去赋值。

Record Creation



最左边有一个 function call, malloc n 个整数，我们把这个地址赋值到虚拟寄存器里去。然后往 memory 里放一个 e1，第二个位置我们就去放 e2，……以此类推。malloc 是我们编译里提供的外部函数。

Array Creation

- Call an external initArray function
 - creates an n-word area
 - initialize all the element with the same value
 - returns the pointer into a new temporary r
 - The result of the whole expression is r
- init_array(n, b)
 - n: array length
 - b: initializing value
- CallExp(NameExp(label of “init_array”)),
 ExpList({n,b}))

Array 初始化每个是一样的，直接调用一个外部函数 init_array 直接初始化即可。Tiger 会用到一些外部函数，外部函数只有在编译器生成代码的时候才能用这些东西。要支持 external call。

Call runtime-system functions

- External functions are implemented in C or assembly language
- ```
/* frame.h */
tree::Exp *frame::ExternalCall(std::string s, tree::ExpList *args);
```
- Simple implementation should be
- ```
tree::Exp *frame::ExternalCall(std::string s, tree::ExpList *args) {
    return tree::CallExp(tree::NameExp(temp::LabelFactory::NamedLabel(s)),
        args);
}
```
- Must handle
 - Underscore function
 - different calling conventions(such as no static link)

第一个传入函数名，第二个要传入函数的参数。在汇编中生成出来的代码名字和源程序的名字可能不一样，我们要用真正汇编里生成出来的函数的名字。

Translate while loops

```
test:
    if not(condition) goto done
    body
    goto test
done:
    • A break statement simply jump to done
        - Exp::Translate has a new formal parameter break
        - done label must be passed as the break parameter
            when translate body
```

在 while 里要支持 break，到了 translation 里也要支持 break，break 其实就是跳出第一

层循环。这边就是要把 `done` 作为一个参数传给 `while`, `break` 的时候就直接 `jump done` 即可。

Translate for loops

- Similar to while loop
- Rewrite the AST into the following code

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

- Be careful to the overflow

- Put the test at the end of loop

```
if i > limit goto done
body
if i == limit goto done
Loop: i := i + 1
    body
    if i <= limit goto Loop
done:
```

Declarations

- Exp::Translate and Dec::Translate
 - Take tr::Level as an extra argument
- tr::Exp *Dec::Translate(env::VEnvPtr venv, env::TEnvPtr tenv, tr::Level *level, temp::Label *label) const = 0;
- Dec::translate returns an extra result tr::Exp as initialization
 - Variable declaration
 - Tr::Exp to reflect the initialization
 - An assignment must be put just before the body of Let
 - Function and type declaration
 - Return "no-op" expression such as Ex(CONST(0))

Dec 的返回值返回了一个 Tree 的表达式。我们拿到了这些 tree 的返回值以后，最后我们要加一些 move 语句，在 let 结束的时候，马上要进 in 了，所以我们要把 MOVE 都做好，放到最前，这样执行 in 的时候，就有这些值作为初值了。

- Prologue
- Body
- Epilogue

Body 都翻译完了以后，还需要一些指令。

Example

```
.text
.type      incr, @function
incr:
    pushl    %ebp
    movl    %esp, %ebp
    movl    count.1231, %eax
    addl    $1, %eax
    movl    %eax, count.1231
    popl    %ebp
    ret
.size      incr, .-incr
.local     count.1231
.comm     count.1231,4,4
```

我们 tiger 的翻译，`ret`、一开始的两行都不是我们 `trans tree` 产生出来的东西。从头上到 `incr` 的第二行，从 `ret` 往下也不是 `transverse tree` 产生的东西，这就是 `prologue` 和 `epilogue`。`Body` 我们已经产生出来了。

2021/11/9

上节课我们讲到把 AST 翻译成我们的 tree (middle IR)，但是我们翻译出来的 tree 基本上是整个汇编语言的核心一块，但是每一段汇编上来会有一块声明函数的入口等基本操作，要放在 `.text` 这个 section 中，进入到函数之前从函数出去的时候可能都要做一些准备工作、收尾工作，包括还有一些没有初始化的静态变量的声明。

因此，我们在一个 function 除了中间的 `body`，其他还有两部分是 `prologue` 和 `epilogue`。这也是我们在生成代码的时候需要生成出来的东西。

Function Definition

- Prologue
- 1. Pseudo-instructions to announce the beginning of a function;
- 2. A label definition of the function name
- 3. An instruction to adjust the stack pointer
- 4. Instructions to save “escaping” arguments - including the static link - into the frame, and to move nonescaping arguments into fresh temporary registers
- 5. Store instructions to save any callee-saved registers- including the return address register - used within the function.

Prologue:

1. 伪指令
2. 函数名 label 的定义
3. 调整 stack pointer
4. 因为 tiger 有函数嵌套，内函数可以访问外函数所定义的变量，所以参数也可能被内函数访问，所以这些访问都需要放在栈上，但是传参的时候参数都是在寄存器中的，所以进入到了函数之后，我们要把这些参数放到栈上去，对于不需要放到栈上去的参数，我们也需要把这些参数从固定的寄存器移动到 temp 虚寄存器中。比如

```
pushq %rdi  
movq %rsi, t1
```

以上其实是处理 ICS 中比较经典的例子，也就是

```
P(X,Y){  
  
    return Q(Y)+Q(X);  
}
```

这个并不是我们做到这里的时候才做，是一开始，我们就需要把 X 和 Y 移到其他的寄存器中，把用于传参的寄存器空出来。这些工作也是在 body 之前需要做的。

5. 还有一些 callee-saved register

也就是 prologue 是在之前 translate 没有的语义，需要我们额外的生成出来。同理，函数结束的时候，也有一些 epilogue 代码所需要做的工作。

Function Definition

- epilogue
- 7. An instruction to move the return value (result of the function) to the register reserved for that purpose
- 8. Load instructions to restore the callee-save registers
- 9. An instruction to reset the stack pointer (to deallocate the frame)
- 10. A return instruction (Jump to the return address)
- 11. Psuedo-instructions, as needed, to announce the end of a function

1. 恢复栈指针
2. 把 callee-saved register 从栈上 pop 出来
3. Return 的 value 要放到固定的寄存器中（比如 x86 中的%rax）。

这些工作也是要专门做的。

仔细分析一下，操作 3 和 9 需要用到栈指针。所以我们就把 1~3、9~11，我们用一个单独的函数去处理它。

- 1,3,9,11 depend on exact knowledge of the frame size
 - Should be generated later F_procEntryExit3
- 2, 10 are also handled at that time
- To implement 7
 - MOVE(RV, body)
 - /* frame.h */
 - temp::Temp frame::RegManager::ReturnValue(void)

为什么要用一个单独的函数？Compiler 希望做到的事情是希望支持多种 target，支持不同的体系结构。每一个不同的体系结构，prologue 和 epilogue 是不同的。中间的 4 这个操作是 *shift of view*，传的参数两边看过来是不一样的，不同的机器这部分也是不一样的，所以我们尽量让有些代码可以重用，有些代码可以修改。只有当所有寄存器都分配完了，我们才能执行这个函数。

我们翻译完 body 以后，实际上是一个 Expression，我们把 expression 的值 move 到 ReturnValue register，每个体系结构，这个 register 都是固定的。剩下的都是和 view shift 有关的。

Function Definition

- 4, 5, 8 are part of the view shift
- It should be done by
 - frame::ProcEntryExit1(frame::Frame*, tree::Stm*)
 - in /* frame.h */
- Translate should apply this function to each procedure as its body is translated

这就是我们头尾的一些处理函数。

到目前为止，我们的一个函数就翻译完了，产生出了两个 fragment (procedure fragment,

产生出来的汇编代码，最终要放到.text 中) 在翻译的过程中还会产生一些字符串常量，生成 StringFrag，最终是要放到.data 里去的。

```
/* frame.h */
class Frag {
/* some definitions and interface here */
};

class StringFrag : public Frag {
public:
    temp::Label *label_;
    std::string str_;
    ...
};

class ProcFrag : public Frag {
public:
    tree::Stm *body_;
    Frame *frame_;
    ...
};

}; 2021/11/9
```

ProcFrag 是包含指令的 tree，还包括了 frame 的信息，主要是变量等。最后我们一边翻译一边把 fragment 放到一个 list 里面去，有一个就插入一个。

Fragment

```
/* frame.h */
tr::ProcEntryExit(tr::Level *level, tr::Exp *body, std::list<tr::Access*> formals);

class Frags {
public:
    Frags() = default;
    void PushBack(Frag *frag);
    const std::list<Frag*> &GetList() { return frags_; }
private:
    std::list<Frag*> frags_;
};
```

在 translation 这一层我们做了封装，屏蔽掉了底下不同体系结构的细节。这里就只需要翻译好的 body 和函数的 level，在 translation 的时候，我们主要就是要生成 view shift。所以我们还需要传入 formals 需要进栈的信息。

我们拿到的是一个 AST，我们在 traverse 这个 AST 的时候，我们一边做 typechecking 一边做 translation。Typechecking 某种程度还是前端的一部分，而 translation 是前端到后端的过渡，它碰到一个函数的时候，会先 new 一个 level，然后再根据 AST 调用一系列的函数。它一边翻译，一边把 string，也就是字符串常量 (data fragment) 加到 fragment list 中去。

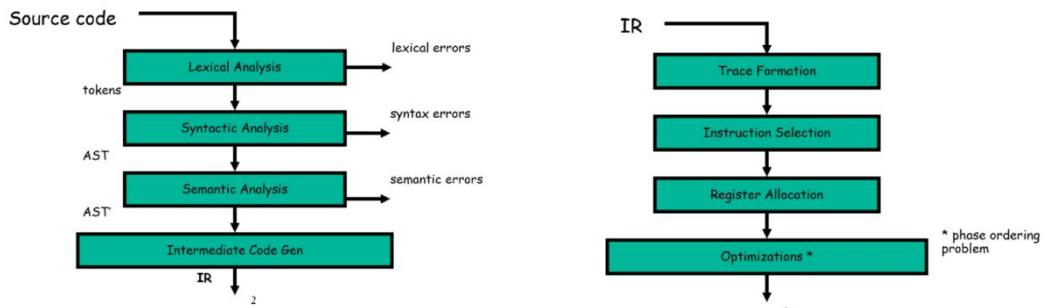
整个翻译完了以后，我们再把头尾加上去，就会得到一个 procedured fragment。

Fragment

- Semant
 - Calls upon `tr::Level::NewLevel` in processing a function header
 - Translates the body
 - By calling other interface functions of `Translate`
 - Remember `DataFrag` for any string literals
 - Calls `procEntryExit1` to remember a `ProcFrag`
 - All the remembered fragments go into a private fragment list within `Translate`

所有 fragment 都会被记录到 fragment list 里去，我们翻译就到此为止了。

第八章：Basic Blocks and Traces



第八章是这门课比较难的部分。我们先来看一下，我们现在做完了一系列步骤，得到了一个 IR。但是 IR 还是和我们的 AST 比较接近，即使它已经有点像汇编了。它还是一个树状的结构，而我们的汇编是链状的结构，所以第八章我们主要处理这个问题，处理完了还是一个和具体机器无关的汇编表示。

到第九章，我们就要把体系结构无关的汇编表示变成一个 ISA（指令集），当然寄存器还是虚的，假设无限个寄存器，最终到了寄存器分配讲完，才会变成真的 memory。

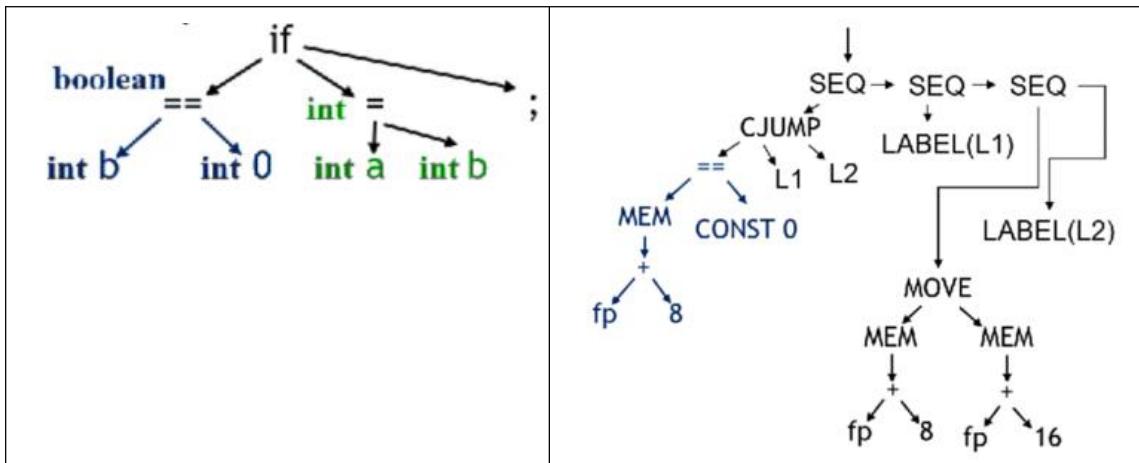
我们先来看一个例子：

对于 tiger 语句

`if (b==0) a = b;`

我们假设变量 a 和变量 b 都放在栈上，那么我们可以通过 MEM 来对 a 和 b 取值。

其 AST 和 tree language structure 如下图所示：

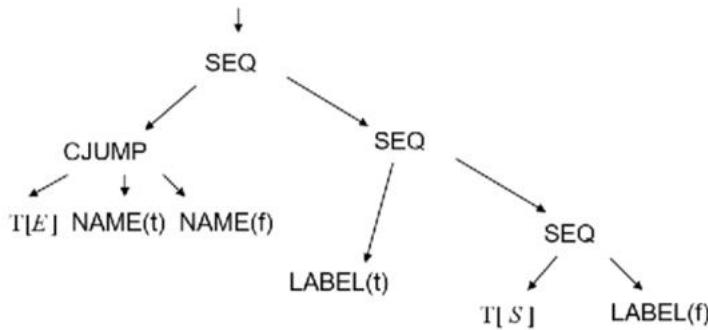


接下来我们再来考虑翻译一个比较通用的情况。

我们有语句 $T[\text{if}(E) S] = \dots$

tree 语句为：

$\text{SEQ}(\text{CJUMP}(T[E], \text{NAME}(t), \text{NAME}(f)), \text{SEQ}(\text{LABEL}(t), \text{SEQ}(T[S], \text{LABEL}(f))));$

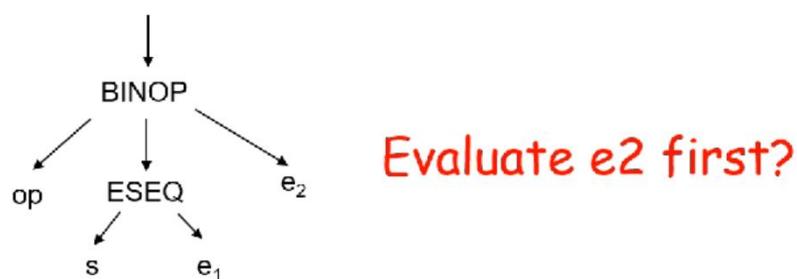


我们注意 E 的部分就翻译成 CJUMP。从这个地方来看，CJUMP 是一个树的表示，和代码中的表示有一点点不一样。真正的汇编指令里，我们有 `conditional jump`，比如 `jl`，它实际上是条件成立会跳转而条件不成立的情况下就执行自己的后一条指令。而在 tree 里无论成立还是不成立，都需要跳转。所以这个差异是在我们把 tree language 转变为 assem 的时候需要处理的一个问题。

第 2 个 tree 里的问题是，

Mismatches: Trees vs. Code

- CJUMP can jump to two labels (real code "falls through")
- ESEQ nodes within expressions are inconvenient (order of evaluation matters)



如果计算 s 的时候有 side-effect，那就会导致谁先做谁后做，生成出来的指令不一样，所以我们也要处理这个问题。

第三个问题也是比较类似的，把 s 换成一个 call，也会改变。

Mismatches: Trees vs. Code (Cont.)

- CALL nodes within expressions also depend on order (have side effects)
- CALL nodes within call nodes cause problems
- Transform IR to eliminate above cases

我们需要把这些不一致的地方都消除掉。

我们先考虑后几种情况：表达式中嵌入了其他表达式，并且嵌入的表达式有 side-effect。

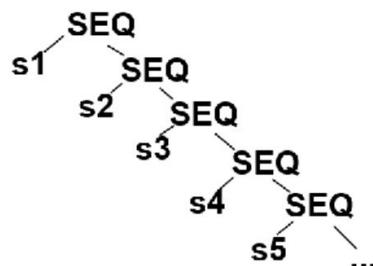
Why canonical form?

- Intermediate code has general tree form
 - easy to generate from AST, but...
- Hard to translate directly to assembly
 - assembly code is a sequence of statements
 - Intermediate code has nodes corresponding to assembly statements deep in expression trees
- Canonical form: all statements brought up to top level of tree -- can generate assembly directly

Canonical Form

Canonical form

- In canonical form, all SEQ nodes go down right chain:



- Function is just one big SEQ containing all statements:
 $\text{SEQ}(s_1, s_2, s_3, s_4, s_5, \dots)$
- Can translate to assembly more directly

我们要把 statement 全部移上去，每个 statement 最后就会和我们的汇编比较接近了。到此我们的执行的顺序就定下来了。

第一步：线性化
主要把 expression sequence 和 call 都处理掉。

Canonical Tree

- $ESEQ(s_1, ESEQ(s_2, e)) \Rightarrow ESEQ(SEQ(s_1, s_2), e))$
- $BINOP(op, ESEQ(s, e1), e2) \Rightarrow ESEQ(s, BINOP(op, e1, e2))$
- $MEM(ESEQ(s, e1)) \Rightarrow ESEQ(s, MEM(e1))$
- $JUMP(ESEQ(s, e1)) \Rightarrow SEQ(s, JUMP(e1))$
- $CJUMP(op, ESEQ(s, e1), e2, l1, l2) \Rightarrow SEQ(s, CJUMP(op, e1, e2, l1, l2))$

它基本规则就是一些 rewriting rule。比如在做 e 之前，要先做 s_1 再做 s_2 ，我们把 statement 合在一起，把表达式放到最后去。

两个表达式做双目运算的情况下，也是我们先让 s 先做，然后再做 e_1 和 e_2 的运算。
Memory 也是，计算地址的时候也会不同，所以我们就把语句提出来，再去计算地址。

CJUMP 中也是，把 s 提出来再计算。但是如果 s 出现在两个地方了怎么办？Expression sequence 出现在第二个怎么办？

- $BINOP(op, e1, ESEQ(s, e2)) \Rightarrow ESEQ(MOVE(TEMP t, e1), ESEQ(s, BINOP(op, TEMP t, e2)))$
- $CJUMP(op, e1, ESEQ(s, e2), l1, l2) \Rightarrow SEQ(MOVE(TEMP t, e1), SEQ(s, CJUMP(op, TEMP t, e2, l1, l2)))$

我们先算 e_1 ，再执行 s ，它会影响 e_2 的计算。所以我们不能把 s 提到最前面去，否则 e_1 就会受到影响。所以我们要先把原先的 e_1 用一个额外的寄存器 t 存起来，然后我们再把 s 提出来即可。

Canonical Tree

- **If s, e_1 commute**

$$\begin{array}{ll} \text{BINOP(op, } e_1, \text{ESEQ}(s, e_2)) & \text{CJUMP(op, } e_1, \text{ESEQ}(s, e_2), l_1, l_2) \\ \Rightarrow \text{ESEQ}(s, \text{BINOP(op, } e_1, e_2)) & \Rightarrow \text{SEQ}(s, \text{CJUMP(op, } e_1, e_2, l_1, l_2) \end{array}$$

- **Commute function**

- Constant commutes with any statement
- Empty statement commutes with any expression

如果 e_1 本身是 constant expression 或者 s 是 nop, 这种情况下是显而易见没有相互影响的。

```
bool Stm::IsNop() {
    if (typeid(*this) == typeid(tree::ExpStm)) {
        auto exp = static_cast<tree::ExpStm *>(this)->exp_;
        return typeid(*exp) == typeid(tree::ConstExp);
    } else return false;
}
bool Stm::Commute(tree::Stm *x, tree::Exp *y) {
    if (x->IsNop())
        return true;
    if (typeid(*y) == typeid(tree::NameExp) ||
        typeid(*y) == typeid(tree::ConstExp))
        return true;
    return false;
}
```

我们先判这个 expression 是不是 const, statement 是不是 nop。我们根据下式可以来判定一个 s 和一个 e 是否是可交换的。

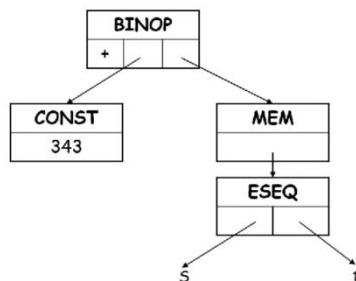
Generally, how to eliminate all ESEQ?

- Idea: extract the statement out of ESEQ, move it higher, with the help of temporaries
- Example:

```
[e1, e2, ESEQ(s, e3)]  
=>  
(SEQ(MOVE(t1, e1), SEQ(MOVE(t2, e2), s));  
[TEMP(t1), TEMP(t2), e3])
```

刚才我们提到了有一些规则，最终我们希望尽量把 s 往外提出去。我们在 s 提出去之前，我们需要把 e1 和 e2 的值算好放到寄存器中。然后再算 s 和 e3.

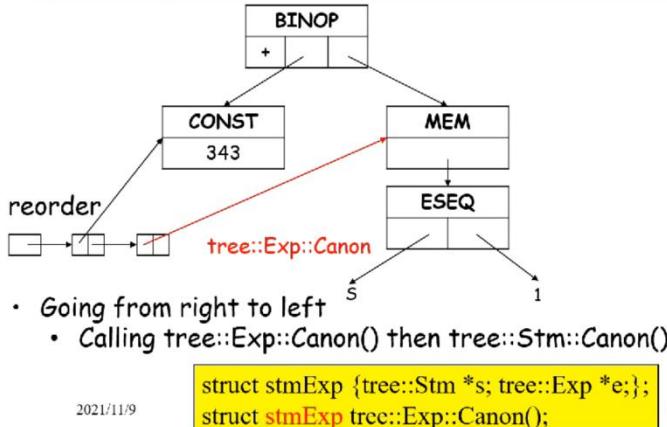
General Rewriting Rules



下边我们先来看这个例子，这是一个表达式的 tree，我们的目标是把 s 提到最前面去。我们先来了一个 expression reference list，分别指向当前尝试正规化的两个儿子中的 Expression，在这个例子中就是指向 tree 的两个子节点。然后有了这个以后，我们从右向左调用 canon 函数，把它正规化，从右向左的过程就是 reorder。

Example

```
struct ExpRefList {  
    std::list<std::reference_wrapper<tree::Exp *>> refs;  
    tree::Stm *Reorder();};
```

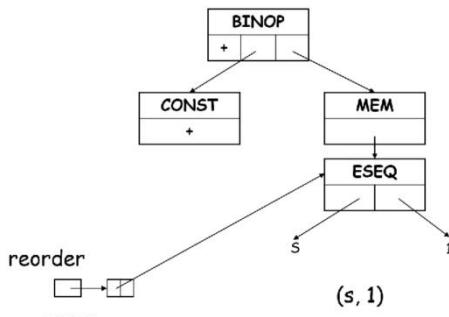


2021/11/9

17

最终的结果就是分成了 statement 和 expression。我们实际上是对 tree node 做标准化，建立一个 list。我们在处理 MEM 的时候，我们又要建立一个 expression reference list，不过此时只有一个节点指向 ESEQ 节点。这样我们就可以递归地往下做。

General Rewriting Rules

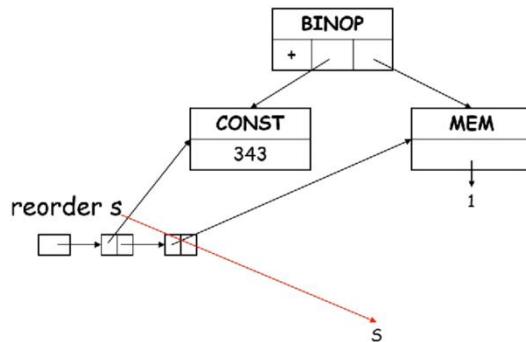


2021/11/9

18

我们再递归地做，就可以得到了 $(s, 1)$ 。然后我们就把这里面的 s 拿出来， 1 就被我们处理完成了。然后我们回到上一层递归的 expression reference list。如果 **CONST** 和 s 是可以交换的，那么 s 就直接提出去了，如果是不可交换的，那么 s 就要生成一个 move 语句，把现在 s 的地方 Move($e1, t1$)。

General Rewriting Rules



因为这个例子中是可以交换的，所以我们直接把 s 往前移动。到最后 expression reference

list 中就没有待处理的表达式了，最终只剩下了一个 seq 和一个 exp 的形式，这就是 rewriting rule。

General Rewriting Rules

```
struct StmAndExp { tree::Stm *s_; tree::Exp *e_;};  
struct ExpRefList {  
    std::list<std::reference_wrapper<tree::Exp *>> refs;  
    tree::Stm *Reorder();  
}  
canon::StmAndExp tree::Exp::Canon() ;  
tree::Stm *Canon();  
• Above functions are mutually called each other  
static Stm *Stm::Seq(tree::Stm *x, tree::Stm *y) {  
    if (x->IsNop())  
        return y;  
    if (y->IsNop())  
        return x;  
    return new tree::SeqStm(x, y);  
}  
2021/11/9
```

我们现在有 StmAndExp 这个结构，statement 的 Canon 返回的就是 statement，expression 的 canon 返回的就是一个 StmAndExp 结构。

我们会产生多个 statement，合成一个 statement 就成为了 statement sequence。这个时候我们可以判定一下，如果存在 nop 的情况，我们扔掉即可。

General Rewriting Rules

```
#define NOP (new tree::ExpStm(new tree::ConstExp(0)))  
  
canon::StmAndExp BinopExp::Canon() {  
    return {ExpRefList(left_, right_).Reorder(), this};}  
canon::StmAndExp MemExp::Canon() {  
    return {ExpRefList(exp_).Reorder(), this};}  
canon::StmAndExp TempExp::Canon() { return {NOP, this};}  
canon::StmAndExp EseqExp::Canon() {  
    canon::StmAndExp x = exp_->Canon();  
    return {tree::Stm::Seq(stm_->Canon(), x.s_), x.e_};}  
canon::StmAndExp NameExp::Canon() { return {NOP, this};}  
canon::StmAndExp ConstExp::Canon() { return {NOP, this};}  
canon::StmAndExp CallExp::Canon() {  
    return {GetCallRlist(this)->Reorder(), this};}
```

我们可以看一下，BinopExp 的 Canon 函数其实就是对其左 exp 和右 exp 构造一个 expression reference list，有了这个 list 以后，我们调用 reorder。刚才的例子中就是一个 binary operation。

如果是 memory 的话，它就调用 Mem 自己的那个 expression，如果是 Temp 寄存器，那就不需要，直接 return 即可。

```

Stm *SeqStm::Canon() {
    return tree::Stm::Seq(left_->Canon(), right_->Canon());
}

Stm *LabelStm::Canon() { return this; }

Stm *JumpStm::Canon() {
    return tree::Stm::Seq(ExpRefList((tree::Exp *&)exp_).Reorder(), this);
}

Stm *CjumpStm::Canon() {
    return tree::Stm::Seq(ExpRefList(left_, right_).Reorder(), this);
}

```

如果是 seq，就是左右各自 canon 一下，然后再合并。Jump 主要是把 jump 后面跟的 expression，reorder 一下。Jump 本身也是一个 statement，所以这就是一个 sequence。

2021/11/12

上节课我们做了线性化之后，把一些有 expression sequence 有 side-effect 的事情，我们就调整了。包括 function call 出现在表达式当中，我们都把这些提出来了，结果就是变成了一串 statement。

第二个 mismatch 就是在 assembly 中，conditional jump 只有一个 target，它的 false 是继续执行后面的指令，但是在我们的 IR 中，conditional jump 有 true target 和 false target。

Taming Conditional Branches

- NO counterpart for two-way branch
 - On most machines
- Rearrange the trees so that
 - CJUMP(cond, l_t, l_f) is immediately followed by LABEL(l_f)
- Two stages
 - Takes a list of canonical trees from them into basic blocks
 - Order the basic blocks into a trace

要做这件事情就要提到 basic block。它是一个 statement sequence，具有唯一的入口和出口，所以 basic block 它们之间的顺序是可以随便放的，因为都是通过 jump 指令跳入跳出的。但是因为我们的汇编都是串行的，我们要把图状的 basic block 变成一个线性的结构。这样就是一个 trace(把 basic block 一个个排下来)。我们尽量地把 CJUMP(cond, l_t, l_f) 和 LABEL(l_f) 放在一起。有时候可能不一定完全一样，因为 basic block 可能从多个地方跳进来，那么我们只能让一个跟到后面来。

上节课我们已经提到了，linearize 之后，我们有一串 statement，s1~sn。我们从头到尾扫描，看到一个 Label，就是一个 basic block 的开始，前面就是上一个 basic block 的结束。如果我们扫描到一个 Jump 或者 Conditional Jump，那么这个 basic block 就结束了，下一个 basic block 就开始了。最后我们就把线性表划分成了一个个 basic block。因为看到 Label 的时候是强行划分的，所以我们要把上一个的 Basic block 末尾加一行 Jump L；靠 Jump 划分开的，下面的 basic block 就加一个 label。

并且，如果我们扫描到最后并且不是 jump，我们需要加一个 done：的 label，因为后面还有 epilogue 需要添加。

线性表的划分其实就是一个 StmListList (每个 basic block 是 StmList，组成的就是

```

StmListList)

/* canon.h */

namespace canon {
class Block {
public:
    temp::Label *label_;      /* special label done */
    StmListList *stm_lists_;
Block(temp::Label *label, StmListList *stm_lists)
    : label_(label), stm_lists_(stm_lists) {}
};

class StmListList {
    friend class Canon;
public: /* some methods definition */
private: std::list<tree::StmList*> stmlist_list_;}
} 2021/11/12

```

因为 Basic Block 都是跳进跳出的，所以可以任意排序，我们先想做的就是把 CJump 和 Label false 排序在一起。

Trace

Put all the blocks of the program into a list **Q**

While **Q** is not empty

start a new (empty) trace, call it **T**

remove the head element **b** from **Q**

while **b** is not marked

 mark **b**; append **b** to the end of the current trace **T**;

 examine the successors of **b**

 if there is any unmarked successor **c**

$b \leftarrow c$

end the current trace **T**

我们遍历 **Q**，拿出一个 **Q** 的第一个 basic block **b**

我们要 check 一下是不是已经放进去 trace 过了，如果没有就 mark 一下加入到 trace 中。

我们扫描 **b** 的后继，如果存在没有标记的后继 **c**，那么用 **c** 代替 **b**，继续走循环。一直到找不到的情况，这个 trace 结束，内层循环结束。

最终我们就变成了一个个的 trace，它是线性的东西。Trace 谁先谁后也是无关的，我们再做一遍排序。Trace 的好处就是 jump 的 target 永远是挨在一起的。

有了 trace 之后，我们再来查一遍：

Finishing Up

- For any CJUMP followed by its false label
 - we let it alone
- For any CJUMP followed by its true label
 - Switch the true and false labels and
 - Negate the condition
- For any CJUMP(cond, a, b, L_t, L_f) followed by neither label

```
CJUMP(cond, a, b, Lt, L't)
LABEL L'f:
JUMP(NAME Lf)
tree::StmList *TraceSchedule();
```

如果 CJump 和 Lf 正好挨在一起，那么任务就达到了；如果不是，那么可能是 CJump 和 Lt 挨在一起了，我们需要把 Conditional Jump 条件反一反，那么也完成了我们的任务。

最糟糕的情况是一个孤零零的 Cjump，两个 target 都被别人拿走了。这样是不能生成汇编的，此时我们就强行增加两行，把零跳改成一跳。

- For any CJUMP followed by its false label
 - we let it alone
- For any CJUMP followed by its true label
 - Switch the true and false labels and
 - Negate the condition
- For any CJUMP(cond, a, b, L_t, L_f) followed by neither label

```
CJUMP(cond, a, b, Lt, L't)
LABEL L'f:
JUMP(NAME Lf)
tree::StmList *TraceSchedule();
```

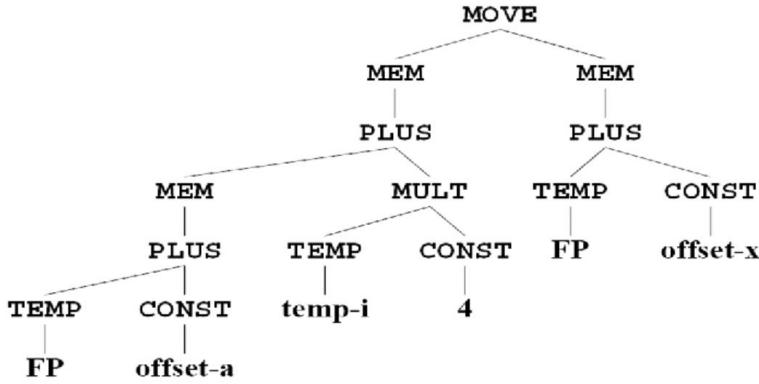
然后，我们还可以再做一点优化，有可能 Jump 和 Target 挨在一起了，那么我们可以直接消掉相邻的情况。

Basic block 在上半本书里，主要的任务是调整 CJump。它是串行执行的，分析代码就很好分析。Basic block 之间会构成一个图，我们称之为控制流图（cfg, control flow graph）。优化没有在这门课中作为重点。如果要做编译相关的工作，那么就要以 cfg 作为基础。

我们已经把 AST (HIR) 变成了 IR (MIR)。接下来我们要把 MIR 变为 LIR。大家看到 tree language 有点接近汇编，但是还不是汇编。我们接下来是要把 tree 变成伪汇编，因为伪汇编中的指令是真的，但是寄存器是假的，我们现在还是假设有无限多的寄存器。我们最后一步就是分配寄存器，变成 LIR 的时候基本上汇编的样子都出来了。

我们先来举个例子，要把 Tree 变成汇编：

比如我们现在有一个赋值语句 $a[i] := x$ ，它本身是一个 MOVE statement，左边是 x ，右边是 $a[i]$ 。在这里我们要做一些假设，我们做完 escape analysis 后就知道放到哪里了。我们现在假设 i 在寄存器中， a 和 x 在 memory 中。这样我们的 tree 就出来了。



我们现在有两个指针 FP 和 SP。x 和 FP 有一个 offset，也就是 offset-x。这个是在第 7 章 translation 的时候，看到一个 variable declaration，我们会从 env 找 escape analysis 的结果是应该在 memory 中与否。一旦到了 memory 里，我们从栈顶 FP 开始分配一个空间给这个变量，所以每个变量和 FP 的 offset 我们是清楚的。最终 x 的地址就是 FP+offset-x。然后到内存当中去取 x 的值：MEM(+, FP, offset-x)。

左边呢，数组在 tiger 中是一个 pointer。我们先取 MEM 中 a 的值作为基址，然后再根据取数组中的第几个值来计算偏移量。

第九章：Instruction Selection

下面我们就生成指令了，一定要指定 ISA 是什么。

Jouette Instructions

Name	Effect
—	r_i
ADD	$r_i = r_j + r_k$
MUL	$r_i = r_j \times r_k$
SUB	$r_i = r_j - r_k$
DIV	$r_i = r_j / r_k$
ADDI	$r_i = r_j + c$
SUBI	$r_i = r_j - c$
LOAD	$r_i = M[r_j + c]$

Name	Effect
—	$M[0]$
STORE	$M[r_j + c] = r_i$
MOVEM	$M[r_j] = M[r_i]$

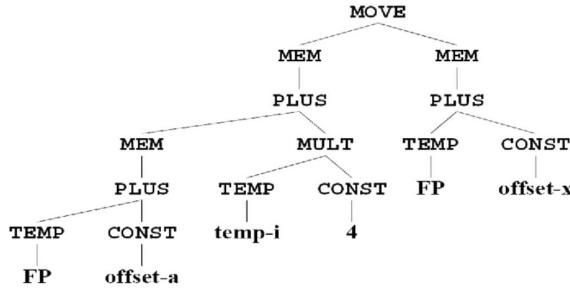
书上用的是教学用的 Jouette 指令集。这边我们并不限制有几个寄存器。ADDI、SUBI 也就是可以加一个 immediate。注意这里没有寄存器之间的 MOVE，在 Jouette 有一个特殊的寄存器 r0，放了一个只读的 0，比如我们要 r_j 移动到 r_i 中，那么就是 ADD $r_i r_j + r0$ 。同样的一个常数如果要放到寄存器 r_i 中，那么就是 ADDI $r_i r0 + c$ 。

然后就是一个 LOAD 指令，它支持的 base register 加一个常量来寻址。STORE 就是把一个寄存器的值放到 Memory 中。

到目前为止，Jouette 的指令集都是很 RISC 的，为了有点变化，它加了最后一条指令 MOVEM，允许两个 Memory 互相传递值。

我们认为除了 MOVEM，其他指令都是一个 cycle 能完成的。

我们接下来的目标就是把 tree 变成一串指令。



- LOAD $r_1 = M[fp+a]$
- ADDI $r_2 = r_0 + 4$
- MUL $r_2 = r_1 * r_2$
- ADD $r_1 = r_1 + r_2$
- LOAD $r_2 = M[fp+x]$
- STORE $M[r_1] = r_2$

实现一：

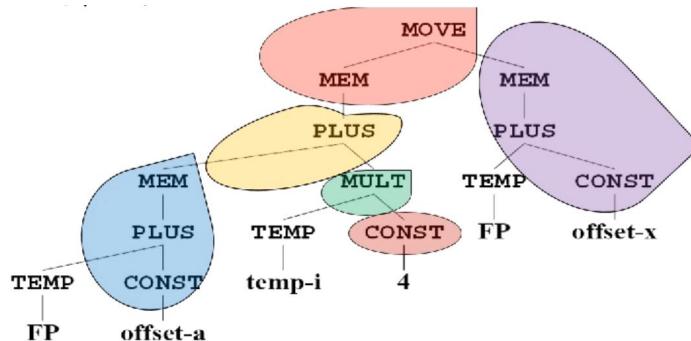
因为数组 a 是放到 memory 里的，我们先把数组 a 的首地址读出来放在 r1 中。因为数组的每个元素的偏移量为 4，我们先把常数 4 放在 r2 中。然后我们计算 r_1+r_2 就是我们要取的 $a[i]$ 的内存地址。

实现二：

- LOAD $r_1 = M[fp + a]$
- ADDI $r_2 = r_0 + 4$
- MUL $r_2 = r_1 * r_2$
- ADD $r_1 = r_1 + r_2$
- ADDI $r_2 = fp + x$
- MOVEM $M[r_1] = M[r_2]$

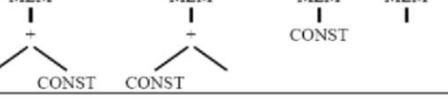
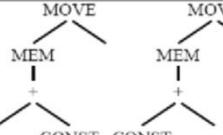
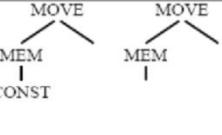
我们现在只算了 x 的地址，但没有把值取出来，我们直接把 x 的值放到 $a[i]$ 的内存地址中去。

这一章的题目叫做 **instruction selection**。也就是一个 tree 可能对应多个不同的指令序列完成相应功能。最后我们就要选一个指令序列。



一条指令可以看成一个 tree 上的 pattern。

Instruction Tree Patterns

Name	Effect	Trees
—	r_i	TEMP
ADD	$r_i = r_j + r_k$	
MUL	$r_i = r_j \times r_k$	
SUB	$r_i = r_j - r_k$	
DIV	$r_i = r_j / r_k$	
ADDI	$r_i = r_j + c$	
SUBI	$r_i = r_j - c$	
LOAD	$r_i = M[r_j + c]$	
STORE	$M[r_j + c] = r_i$	
MOVEM	$M[r_j] = M[r_i]$	

PLUS 下面不是两个寄存器怎么办呢？因为除了 MOVEM 之外，其他指令的执行结果都是放到寄存器中的。所以一条指令，在 tree 中是一个 tile，或者叫做 tree 的一个 pattern。

ADDI 因为加法有交换律，所以有两种情况，再加上一个 $r0+c$ 的情况。

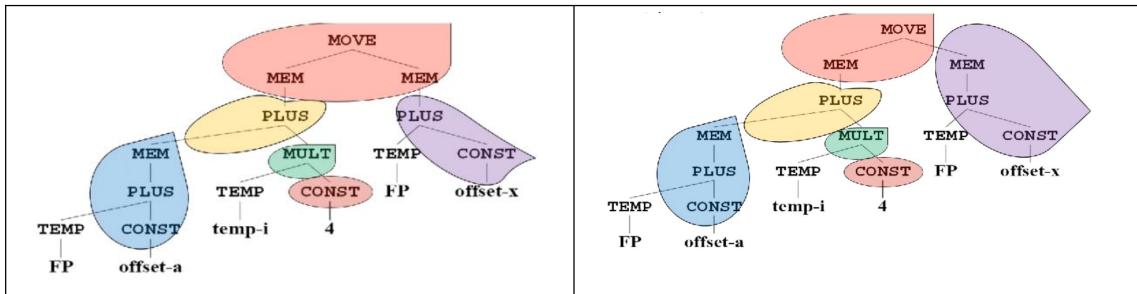
LOAD 的话，可以是一个单独的寄存器访存，也可以是常量访存，也有可能是常量加寄存器的访存。

我们在计算中间结果的时候，会引入很多中间寄存器。

所以指令选择的过程，实际上是把 tree 做 tiling 的过程，也就是拿一个个 tile 来覆盖我们的 tree。注意覆盖的过程中，两个 tile 不能存在 overlap。

所有地方都覆盖好了以后，我们自底向上把指令写出来，那么我们的 tree 就转变为了 一串的指令。有不同的 tiling 的时候，生成出来的指令是不一样的。我们的目标是明确的，我们希望生成的指令执行时间快一点。我们在做选择的时候，如果对于 RISC 这种指令，如果一条指令执行时间 (time cost) 是一样的，我们选择指令数最少的。但是在 Jouette 中有一条指令 cost 为 2，所以不能使用这个方法。

方法 1：全局最优，因为我们要做 tiling，每个 tile 都是有权重的。



上图中的 cost = $1 * 5 + 1 * 2 = 7$	上图中的 cost = $1 * 6 = 6$
---------------------------------	-------------------------

方法 2：局部最优，我们看相邻的两个能不能合成，如果合成出来代价更小，那么就选择。其实就是一个局部的贪心。对于 RISC 指令集，局部最优基本上是全局最优。大家要做 lab 就是一个局部最优的 maximal munch。

Maximal Munch (局部最优)

Optimal Instruction Selection - Maximal Munch

- Cover root node of IR tree with largest tile t that fits (most nodes)
 - Tiles of equivalent size => arbitrarily choose one
- Repeat for each subtree at leaves of t .
- Generate assembly instructions in reverse order
 - instruction for tile at root emitted last

我们从根开始做，根可能有好几种 tile 可以去 cover 的方法，我们要选择最大的那个 tile 去 cover，cover 完以后，tree 就变成两棵子树了，我们接下来可以继续递归地去做这件事情。

- Maximal Munch is quite easy to write in Cpp
- Two recursive functions:
 - tree::Stm::Munch for statements
 - tree::Exp::Munch for expressions
- Each clause in these functions matches pattern/tile
- Clauses ordered in order of tile preference

```
namespace tree{
void MoveStm::Munch(assem::InstrList &il) {
  if (typeid(*dst_) == typeid(MemExp)) {
    MemExp* dst_mem = static_cast<MemExp*>(dst_);
    if (typeid(*dst_mem->exp_) == BinopExp) {
      BinopExp* dst_binop = static_cast<BinopExp*>(dst_mem->exp_);
      if (dst_binop == PLUS_OP &&
          typeid(*dst_binop->right_) == typeid(ConstExp) {
        Exp *c1 = dst_binop->left_; Exp *c2 = src_;
        /*MOVE(MEM(e1+i), e2) */
        e1->Munch(il);
        c2->Munch(il);
        il.emit("STORE");
      }
    }
  }
}
```

右边是 const 的情况：

我们来以 MoveStm 看一下。接下来我只需要把根是 MOVE 的之前表格中的 pattern 一个个情况去写即可。

生成出来的指令就是：STORE M[r1 + c], r2。在此之前已经计算出了 $r1 <- \dots, r2 <- \dots$

```
.....
if (typeid(*dst_) == typeid(MemExp)) {
  MemExp* dst_mem = static_cast<MemExp*>(dst_);
  if (typeid(*dst_mem->exp_) == BinopExp) {
    BinopExp* dst_binop = static_cast<BinopExp*>(dst_mem->exp_);
    .....
    else if (dst_binop->op_ == PLUS_OP &&
             typeid(*dst_binop->left_) == typeid(ConstExp) {
      Exp *c1 = dst_binop->left_; Exp *c2 = src_;
      /*MOVE(MEM(i+e1), e2) */
      e1->Munch(il);
      c2->Munch(il);
      il.emit("STORE");
    }
  }
}
```

左边是 const 的情况：

```

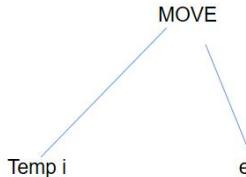
.....
if (typeid(*dst_) == typeid(MemExp)) {
    MemExp* dst_mem = static_cast<MemExp*>(dst_);
.....
else if(typeid(*src_) == typeid(MemExp)) {
    MemExp *src_mem = static_cast<MemExp*>(src_);
    Exp *e1=dst_mem->exp_, e2=src_mem->exp_;
    /*MOVE(MEM(c1), MEM(c2)) */
    e1->Munch(il);
    e2->Munch(il);
    il.emit("MOVEM");
}

```

两个子树都是 **MEM** 的情况:

2021/11/16

我们上节课在讲 **instruction selection**, 讲了一个 **maximal munch**, 每一个指令就变成了 **tree** 上的一个个 **tile**, 然后拿 **tile** 去覆盖我们的 **tree**。覆盖好以后, 剩下的就是有一些子树, 我们递归地去覆盖, 结束之后, 我们自底向上地来生成我们的指令。后来我们看了代码, 当我们看的 **MOVE** 的时候, 实际上会根据 **MOVE** 的左子树和右子树的地方来做 **tiling**。实际上, 在我们 **tree** 的表达里, 因为 **tree** 是一个 **statement**, 还会有如下图所示:



那么我们要生成指令是:

```

r<-e1
add r1<-r0+r

```

```

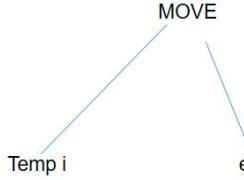
.....
if (typeid(*dst_) == typeid(tree::MemExp)) {
.....
}
else if (typeid(*dst_) == typeid(TempExp)) {
    Exp *e2=src_;
    /*MOVE(TEMP~i, e2) */
    e2->Munch(il);
    il.emit("ADD");
}

```

针对 **statement** 的 **munch**, 我们就来判各种 **substatement**, 我们分门别类地一个个写出来可能的 **statement**。但是 lab5 的第二部分, 因为我们对 x86-64 做 **instruction selection**, 所以 **tiling** 的方式完全不一样, 并且改成了 **c++** 了以后, 这段代码也不是非常好的风格。

这边我们有一个参数 **instruction list**, 实际上 **munch** 和 **emit** 这样的结果我们都会生成一条指令, 最后我们就变成了一个指令的序列, 每生成一个都 **append** 到 **list** 最后。

接下来我们来看 **expression** 的 **maximal munch**, 当然我们也会产生几条指令放到最后, 区别在于 **expression** 要返回一个寄存器, 而 **statement** 的 **maximal munch** 返回的是 **void**。



比如上例中， e 的结果会放在返回的寄存器中。

expression 的 maximal munch 的例子：

```

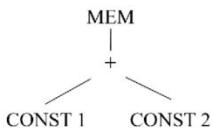
temp::Temp *Exp::Munch(assem::InstrList &il)
BINOP(+,e1,CONST(i))=>
    e1->Munch(il); il.emit(r ← “ADDI”); return r;
BINOP(+,CONST(i),c1)=>
    e1->Munch(il); il.emit(r ← “ADDI”); return r;
CONST(i)=>
    il.emit(r ← “ADDI”); return r;
BINOP(+, c1, c2)=>
    c1->Munch(il); c2->Munch(il); il.emit(r ← “ADD”); return r;
TEMP(i)=> return i ;

```

注意我们先 tile CONST + e ，如果不匹配再 tile i ，这才满足 maximal munch。

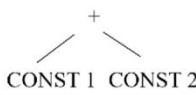
这就是局部最优的 maximal munch，如果我们要做全局最优。在判断一个 root 节点，我们对任何子树都要得到最优情况，这样我们就可以继续判断当前的情况。在这边，我们用到的是动态规划的想法。因为我们讲的是先对 subtree 做 tiling，做好以后，把 subtree 的 cost（生成需要几条指令，加权以后得到 cost），我们把 cost 记录在每个 subtree 的根节点上。我们来看一个例子：

- MEM(BINOP(PLUS, CONST(1), CONST(2)))
- MEM(PLUS(CONST(1), CONST(2)))



我们现在对每个 subtree 都来计算 cost

- The only tile that matches CONST 1 is an ADDI instruction with cost 1
- Similarly, CONST 2
- Several tiles match the + node



Tile	instruction	Tile Cost	Leaves Cost	Total cost
	ADD	1	1+1	3
	ADDI	1	1	2
	ADDI	1	1	2

要 tile 单纯的一个 const，需要一条 addi 指令。如上表所示，后两个 tiling 方式的 cost 更少。最后还有一个 memory 的情况。

Tile	instruction	Tile Cost	Leaves Cost	Total cost
MEM I	LOAD	1	2	3
MEM CONST	LOAD	1	1	2
CONST	LOAD	1	1	2

这种方式实现起来比较复杂，所以我们 lab 使用的是 maximal munch，再下来，如果到了 cisc 这种 style。它有一些特征，当然有些特征和 instruction selection 没什么关系。

For CISC Machine, some new problems are

issued:

- Few registers
- Classes of registers
 - Some operations available only on certain registers
 - division instruction in Pentium needs both %rax and %rdx
- Two-address instructions
- Arithmetic operations can address memory
- Several addressing modes
- Variable-length instructions
- Instruction with side effects
 - Auto-increment addressing modes

关系比较大的是：每次寻完址就自动地加 4 加 8。有影响是前面几个特点：寄存器少也不影响 instruction selection，到时候分配寄存器的时候可能一点麻烦。

2,3,4,5 可能对我们有点影响。比如我们要支持整数除法，它用到了 IDIV

- **Classes of registers** $t1 \leftarrow t2/t3$

```
movq t2, %rax
cqto
idivq t3           //%rdx used
movq %rax, t1
```
- **Two-address instructions** $t1 \leftarrow t2+t3$

```
movq t2, t1
addq t3, t1
```

这条指令它用到了 %rax 和 %rdx，虽然只有一条指令，但是用到了两个额外的寄存器。注意 cqto 是把 64 位变成 128 位（%rdx 是高位，%rax 是低位，根据 %rax 的最高位填入全 0 或全 1）。除来的结果商是放到 %rax 里，余数是放到 %rdx 里。这样一个 tree 就变成了 4 条指令，在寄存器分配的时候，我们就必须知道这 4 条指令都用到了 %rax 和 %rdx。

第二点是，在 x86-64 里算术逻辑运算只有两个操作数，它的 destination 和 source 是合而为一的。所以 $t1 \leftarrow t2+t3$ 需要用两条指令去实现它。

我们还有更多的 addressing mode，在访问数组的时候，我们就可以用更复杂的 addressing mode 去支持数组的访问。这就是针对一些特殊的情况我们来做优化，如果我们只是类似于 jouette 的方法来做的话，我们只有简单的 addressing mode。

在我们的 x86-64 中，还支持 (r_a, r_b, s) 。所以我们 tiger 的数组访问可以通过这样的二重寻址方式来访问。我们可以去做优化。

如下的算术运算，在 x86-64 中只需要生成右边的一条指令即可。

Arithmetic operations can address memory

```
movq -8(%rbp), %rax           addq %rcx, -8(%rbp)
addq %rcx, %rax
movq %rax, -8(%rbp)
```

最后我们来看代码。我们的目的是要把 tree language (MIR) 变成一个 assem (LIR)。
这个已经基本上是汇编指令了，除了寄存器还是虚的（无限多的）。

```
/* assem.h */
namespace assem {
    class Targets {
        public:
            std::vector<temp::Label *> *labels_;
            explicit Targets(std::vector<temp::Label *> *labels) :
                labels_(labels) {}
        };
        class Instr {
            public:
                /* Destructor and some interface for subclass to implement */
            };
        class OperInstr : public Instr {
            public:
                std::string assem_;
                temp::TempList *dst_, *src_;
                Targets *jumps_;
                OperInstr(std::string assem, temp::TempList *dst, temp::TempList *src,
                          Targets *jumps)
                    : assem_(std::move(assem)), dst_(dst), src_(src), jumps_(jumps) {}
                /* some interface to implement */
            };
}
```

大部分的指令都是 operation instruction，还有一个是 label instruction

```

class LabelInstr : public Instr {
public:
    std::string assem_;
    temp::Label *label_;
    LabelInstr(std::string assem, temp::Label *label)
        : assem_(std::move(assem)), label_(label) {}
    /* some interface to implement */
};

class MoveInstr : public Instr {
public:
    std::string assem_;
    temp::TempList *dst_, *src_;
    MoveInstr(std::string assem, temp::TempList *dst, temp::TempList *src)
        : assem_(std::move(assem)), dst_(dst), src_(src) {}
    /* some interface to implement */
};

```

因为它只有一个标号。为什么要把 move instruction 单独从 operation instruction 拿出来呢？因为它在寄存器分配中比较特殊，比如我们要做 shift-of-view，我们可能上来就要做在 callee 中把真实 register 中的参数放到虚的 register 中去（callee-saved register）：

```
t1<%rdi
t2<%rsi
t3<%rdx
```

只是这种情况，我们需要把%rdi 移走，但是大部分情况是不需要移走的。所以在 register allocation 的时候，是要做 move 的 coerce 的。我们在生成汇编的时候单独作为一个类拿出来比较好。

所有的 instruction 第一部分都是一个字符串。下面是 jouette 的 instruction。

```

MEM(+(Temp fp, CONST 8))
assem::OperInstr("LOAD `d0 <- M[`$0+8]", 
                  temp::TempList(temp::TempFactory::NewTemp()),
                  temp::TempList(reg_manager->FramePointer()),
                  nullptr)


- The actual assembly language of Jouette after register allocation
  - LOAD r1 <- M[r27+8]
  - r27 is the frame pointer fp
  - register allocation assign the new temp to register r1
- The assem instruction does not know about register assignment
  - Talks about the source and destination
  - One source '$0, one destination 'd0

```

'这个符号不会在汇编中出现，但是标识着在指令中对应寄存器的位置。

在我们生成 lower IR 的时候，第一部分就是把这条指令写出来。但是它和真的 assembly 还有区别，因为两个寄存器还没定下来。虚的寄存器就是一个 TempList，分为 destination (结果寄存器) 和 source (操作数寄存器)。比如 add 的话就会有两个 source register。

第三个参数就是 Target* jumps，如果我们的 jump 指令，那么 jump 的 label 就会出现在 target 这里。

现在上例中，是对 fp 寄存器加 8，也就是取 static link。因为取出来结果要放到寄存器

里，所以我们 new 一个新的寄存器。

Abstract Assembly-Language Instructions

`*(+(TEMP t87, CONST3),MEM(TEMP t92))`

- Could be translated as follows

assem	dst	src
ADDI `d0 <- `s0+3	t908	t87
LOAD `d0<-M[`s0+0]	t909	t92
MUL `d0<- `s0 * `s1	t910	t908, t909

- After register allocation, it might look like

ADDI r1 <- r12+3
LOAD r2<-M[r13+0]
MUL r1<- r1 * r2

- Two address instructions

$$t_1 \leftarrow t_1 + t_2$$

- Can be described as

assem	dst	src
ADD `d0, `s1	t1	t1,t2

- Where `s0 is implicitly mentioned in the assem string

在寄存器分配的时候，我们必须把 t1 放进去

我们现在要做的是 control flow

- An LabelInstr is a point in a program to which jumps may go
 - An assem_component showing how the label will look in the assembly-language program
 - A label_component identifying which label-symbol was used
- An MoveInstr is like OperInstr, but must perform only data transfer
 - if dst_and src_temporaries are assigned to the same register, the MOVE can later be deleted

Move 是我们在做寄存器分配的时候要对寄存器做特殊处理，做 collace。

这样我们就把 lowerIR 变成了 instruction 的形式。在我们 instruction 这个 class 里，要支持我们最后的输出，也要支持我们的调试。我们希望能输出这样的格式。这样我们还要提供给一个 Print 函数，给了一个 instruction 后，我们能够打印出来具体的寄存器是通过 map 来定的。

```

/* temp.h under namespace temp: */
class Map {
public:
    ...
    void Enter(Temp *t, std::string *s);
    std::string *Look(Temp *t);
    static Map *Empty();
    static Map *LayerMap(Map *over, Map *under);
private:
    tab::Table<Temp, std::string> *tab_;
    Map *under_;
    Map() : tab_(new tab::Table<Temp, std::string>()), under_(nullptr) {}
    Map(tab::Table<Temp, std::string> *tab, Map *under)
        : tab_(tab), under_(under) {}
};



- A temp::Map is just a table
  - whose keys are temp::Temp* and
  - whose bindings are strings
- One mapping can be layered over another
- When to use the temp::Map operations
  - The register allocation
  - Frame module makes it to describe the names of all the preallocated registers
    - Such as %rsp for SP, %rax for RV
  - For debugging purpose
    - Just maps each temporary to its name

```

这个 table 和 environment 一样是会一级级叠起来的。这个 table 实际上就是把 register 和 string 做一个 mapping。对于一些特殊的预先有的寄存器，我们也要添加到 map 中。另外我们的 map 还要负责 register allocation，把虚的寄存器变成我们实的寄存器。有了这些东西之后，我们来看前面的 maximal munch 函数怎么来写：

```

#define L (new temp::TempList /* just for slide neat */)
temp::Temp *MemExp::Munch(assem::InstrList &il) {
    temp::Temp *r = temp::TcmpFactory::Newtemp();
    (switch by exp_)
    case MEM(BINOP(PLUS, e1, CONST(i))): {
        il.Append(new assem::OperInstr("LOAD `d0<- M[`s0+" + i +
        "]\n", L(r), L(e1->Munch(il)), nullptr));
        return r;
    }
    case MEM(BINOP(PLUS, CONST(i), c1)): {
        il.Append(new assem::OperInstr("LOAD `d0 <- M[`s0+" + i +
        "]\n", L(r), L(e1->Munch(il)), nullptr));
        return r;
    }
}

```

我们就是对一个个 subclass 去写。在做 MEM(e1+i) 的时候，会先做 e1 的 maximal munch，会 return 一个 register，作为 MEM 指令的 source。MEM 对应的是一个 LOAD 指令，结果放到 d0，其中的 d0 和 d1 会从 map 中拿了以后填入。

```

void MoveStm::Munch(assem::InstrList &il){
    (switch by src_ and dst_)
    case MOVE(MEM(BINOP(PLUS, c1, CONST(i)))):
        il.Append(new assem::OperInstr("STORE M[`s0+" + i + "]<-`s1\n",
        L(), L({e1->Munch(il), e2->Munch(il)}), nullptr));
    case MOVE(MEM(BINOP(PLUS, CONST(i), e1))):
        il.Append(new assem::OperInstr("STORE M[`s0+" + i + "]<-`s1\n",
        L(), L({c1->Munch(il), e2->Munch(il)}), nullptr));
    case MOVE(MEM(c1), MEM(c2)):
        il.Append(new assem::OperInstr("MOVEM M[`s0]<-M[`s1]\n",
        L(), L({e1->Munch(il), e2->Munch(il)}), nullptr));
}

```

按照我们 IR 的格式生成的代码。前面这些是会生成 operation instruction。

```

void LabelStm::Munch(assem::InstrList &il) {
    il.Append(
        new assem::LabelInstr(temp::LabelFactory::LabelString(label_),
        label_));
}

```

对于 label statement，生成的是 label instruction。

在这个过程中，我们会生成 instruction list，我们用标准模板来生成 instruction list

```
/* assem.h */
namespace assem {
class InstrList {
public:
    void Print(FILE *out, temp::Map *m) const;
    void Append(assem::Instr *instr) { instr_list_.push_back(instr); }
private:
    std::list<Instr *> instr_list_;
};
class Proc {
public:
    std::string prolog_;
    InstrList *body_;
    std::string epilog_;
};
}
```

2021/11/16

2021/11/23

.text		
.type	P, @function	
P:		
subq	\$8, %rsp	• In the translation phase
pushq	%rbp	- tiger uses a virtual frame pointer FP
pushq	%rbx	- each inFrame variable is referred as FP+k
.....		
popq	%rbx	• In code generation phase
popq	%rbp	- Tiger replaces the FP to SP
addq	\$8, %rsp	- each inFrame variable is referred as SP+k+fs
ret		• fs is the size of the frame
.size	P, .-P	- Which can be known only after the register allocation

到现在为止，我们产生 tiger 都使用了 frame pointer，frame pointer 和 stack pointer 差了一个 frame size。frame size 一般来讲是一个整数，

```
f(int n):
{
    intA[n];
}
```

如果在上例中出现了变长的数组，那么我们一定要使用 frame pointer。在 tiger 中是先用了 frame pointer，到最后我们可以去掉它。

No Frame Pointer

- Code gen only generates `SP+L14_framesize`
 - `L14` is assumed as the label of current function to be generated
 - `L14_framesize` will included in the frame `f`
 - `.set L14_framesize 0x20`
 - be generated by `frame::ProEntryExit3`
- If we use the frame pointer, we do not need to the above hacking
- Frame pointer must be used when frame size varied

对于 register 的命名，我们有一个 register manager，它映射了一个 map，把 int 映射到 string。有一些确定的寄存器，比如 stack pointer, return address register, zero register。在 register manager 里，我们还要生成 4 个 list。

Register List

- Four lists of registers
- `specialregs`
 - a list of μ registers to implement "special" registers
- `argregs`
 - a list of μ registers in which to pass outgoing arguments
- `calleesaves`
 - a list of μ registers that the callee must preserve unchanged
- `callersaves`
 - a list of μ registers that the callee may trash

1. 特殊用途的寄存器，return value、stack pointer, etc。
2. 传参寄存器，在 x86-64 中有 6 个
3. callee-saved register
4. Caller-saved register

在产生 call 的时候，call 写过的寄存器都是 caller-saved register。我们做完 instruction selection，就有一个 instruction list。然后我们再做了一个 entryexit2

ProcEntryExit2

frame::ProcEntryExit2

```
/* frame.h */
.....
assem::InstrList *ProcEntryExit2(assem::InstrList *body);
/* xxframe.cc */
temp::TempList *xxRegManager::ReturnSink() {
    temp::TempList *temp_list = CalleeSaves();
    temp_list->Append(StackPointer());
    temp_list->Append(ReturnValue());
    return temp_list;
}
assem::InstrList *ProcEntryExit2(assem::InstrList *body) {
    body->Append(new assem::OperInstr("", new temp::TempList(),
                                         reg_manager->ReturnSink(), nullptr));
    return body;
}
```

Returnsink 其实就是 callee-saved register, return value, stack pointer, 也就是在 function 结束之后还需要用到的值。EntryExit2 其实就是在 body 添加了一条空指令, 提醒到了这里还有这些寄存器是要被用到的。

ProcEntryExit3

frame::ProcEntryExit3

- A scaffold version of frame::ProcEntryExit3
- ```
assem::Proc *ProcEntryExit3(frame::Frame *frame,
 assem::InstrList *body);
{
 char buf[100];
 sprintf(buf,
 "PROCEDURE %s\n",
 temp::LabelFactory::LabelString(frame->Name()).data());
 return new assem::Proc(std::string(buf), body, "END\n");
}
```

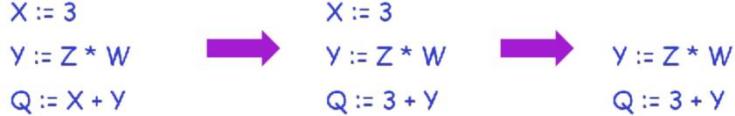
这个函数其实就是产生一个头和产生一个尾, 比如我们 function 是有一个 label, 我们就添加在这里。具体我们要根据汇编来生成这样的一个代码。

下边我们进入到最后一个部分: 寄存器分配

在做 liveness Analysis 的时候, 我们要做全局的流分析。我们再补充一个 global flow analysis 和 constant propagation。

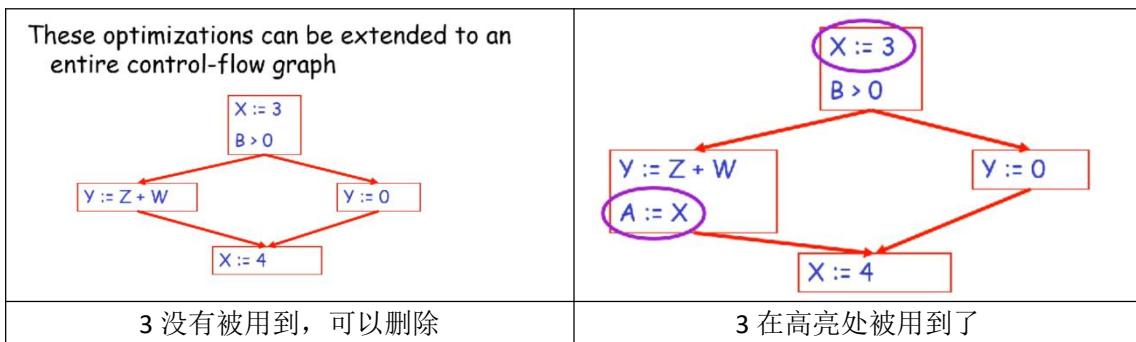
## Global Optimization

其实我们在做寄存器分配的时候，我们只需要知道 liveness analysis，但是在优化的时候，我们需要做常数传播(constant propagation)和死码删除(Dead code elimination)。



3 为什么可以传过来并且删掉？

我们的一个 basic block 实际上是一些 statement 和指令，特征就是单入口和单出口。所以在 control flow graph 中的节点就是一个个 basic block，指向的边就是 basic block 可以跳转到的地方。

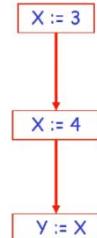


我们做 liveness analysis、常数传播、死码删除都是需要在这个控制流图上分析的。上图中的  $X := 3$  是一个死码，因为 3 无论走哪条路到 4 都没有被用到，那么我们就可以删掉它。

所以我们做这件事情先要保证其正确性。

### Live and Dead

- The first value of  $x$  is dead (never used)
- The second value of  $x$  is live (may be used)
- Liveness is an important concept



也就是我们要判断  $X$  的值在某条语句中是否还是活跃的（之后是否还会被用到。）

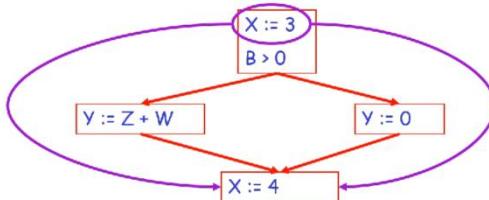
## 变量活跃的条件

变量  $x$  在语句  $s$  中活跃的条件是：存在一个语句  $s'$  用到了  $x$ ，要有一条路径从  $s$  到  $s'$ ，并且在这条路上没有对  $x$  的赋值语句。

## 死码删除

如果  $x$  在语句  $x := \dots$  后立马就不活跃了 (dead), 我们就认为这条赋值语句是死码 (dead code)。死码就可以从程序中删除, 所以在做死码删除的时候我们必须先做活跃性分析 (liveness analysis)。

Example1:



这种情况下  $X := 3$  处就是不活跃的, 因为中间两条语句没有用过  $X$ 。

在做活跃分析的时候, 不活跃就等于所有路上这个性质都不成立。似乎也不复杂, 在我们做全局分析的时候, 麻烦的事情是可能我们存在循环, 导致某些路会反复走。这样我们就需要遍历整个控制流图来做分析。

### Global Analysis

Global optimization tasks share several traits:

- The optimization depends on knowing a property  $X$  at a particular point in program execution
- Proving  $X$  at any point requires knowledge of the entire function body
- It is OK to be conservative. If the optimization requires  $X$  to be true, then want to know either
  - $X$  is definitely true
  - Don't know if  $X$  is true
- It is always safe to say "don't know"

我们活跃分析的目标, 比如说死码删除。编译器在优化的时候采取了比较保守的策略, 当我们不确定某个是死码的时候, 编译器为了正确性是不会删掉它的。

### 活跃分析

我们先讲针对一个变量  $x$  的活跃分析, 在程序的所有点, 这个变量的活跃性的分析。也就是每个点我们需要知道  $x$  在这里是 **true** (活跃) 还是 **false** (不活跃), 最后我们把所有信息分析出来以后, 对于一个赋值  $x := \dots$  我们就可以来看这点上  $x$  是活跃的还是不活跃的, 如果不活跃就可以删掉 (死码删除)。

目标: 对于变量  $x$ , 计算其所有点上的活跃性。

要做这件事情, 我们实际上用了一些规则 **in** 和 **out**。

- The idea is to "push" or "transfer" information from one statement to the next
- For each statement  $s$ , we compute information about the value of  $x$  immediately before and after  $s$

$L_{in}(x, s)$  = value of  $x$  before  $s$

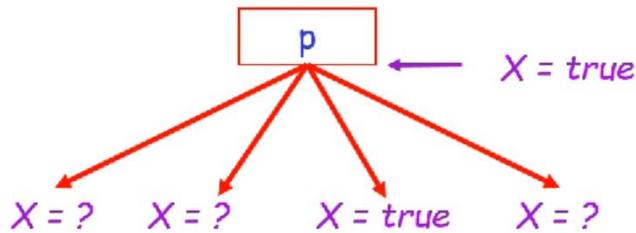
$L_{out}(x, s)$  = value of  $x$  after  $s$

$L_{in}(x, s)$  的含义：在 in 这一点，对于  $x$  是否活跃。

对于多个 statement，我们有一个转移函数。

## Liveness Rule 1

---



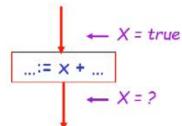
$$L_{out}(x, p) = \vee \{ L_{in}(x, s) \mid s \text{ a successor of } p \}$$

$$L_{out}(x, p) = \vee \{ L_{in}(x, s) \mid s \text{ is a successor of } p \}$$

控制流图中一个节点可能跳转到多个地方去，每个后继节点上都会有  $x$  的 liveness information，true 代表  $x$  在这一点是活跃的，只要有一个点是活跃，那么当前的父节点肯定也是活跃的。

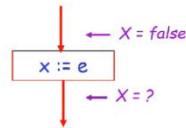
### Liveness Rule 2

---

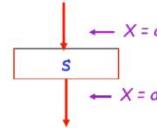


$$L_{in}(x, s) = \text{true if } s \text{ refers to } x \text{ on the rhs}$$

### Liveness Rule 3



### Liveness Rule 4



$$L_{in}(x, x := e) = \text{false} \text{ if } e \text{ does not refer to } x \quad L_{in}(x, s) = L_{out}(x, s) \text{ if } s \text{ does not refer to } x$$

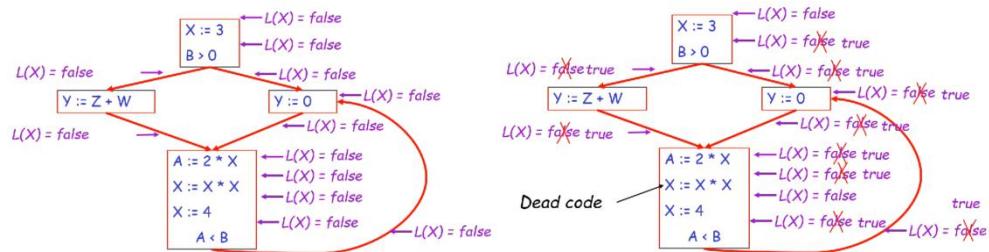
根据这四个规则，我们就可以来计算活跃性分析了。

### Algorithm

1. Let all  $L_{in}(\dots) = \text{false}$  initially
2. Repeat until all statements  $s$  satisfy rules 1-4  
Pick  $s$  where one of 1-4 does not hold and update  
using the appropriate rule

初始化时，每一个点都是 false。

### Another Example



在 4 条规则里，false 可以变 true，但是 true 不能变为 false，对于  $n$  个点的程序，存在  $2n$  个  $L$ ，每次循环我们至少会让一个点的值发生变化。所以这个算法是可终止的。

## Computing Liveness Set

- We just computed the liveness for one variable using  $L(x, s)$
- Now we need to compute the liveness information for each variable at point of a program
- We can use the following set for a statement  $s$ :
  - $\text{Use}[s]$  the set of variables that are used by  $s$
  - $\text{Def}[s]$  the set of variables that are defined in  $s$
  - $\text{In}[s]$  the set of variables that are live on entry to  $s$
  - $\text{Out}[s]$  the set of variables that are live on exit to  $s$

我们可以计算某个点上活跃变量的集合，我们在这里定义了  $\text{use}[s]$ ,  $\text{def}[s]$ ,  $\text{in}[s]$ ,  $\text{out}[s]$ 。  
 $\text{use}$  是说这个 statement 用到了哪些变量（赋值号右边的变量）； $\text{def}$  是赋值号左边的变量。

## Liveness Analysis: 数据流方程

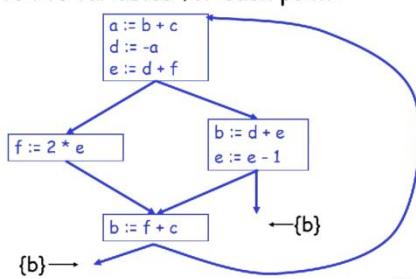
### Computing Liveness Set

- We have the following equations
  - $\text{in}[s] = \text{use}[s] \cup (\text{out}[s] - \text{def}[s])$
  - $\text{out}[s] = \bigcup_{n \in \text{succ}[s]} \text{in}[n]$

活跃分析就是在控制流图上求解控制流方程。

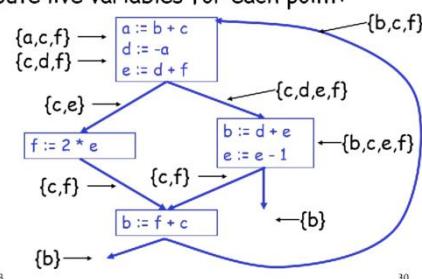
#### Example

- Compute live variables for each point:



#### Example

- Compute live variables for each point:



在程序结束的地方，我们已经知道哪些变量活跃了  $\text{EntryExit}(2)$ 。

这样我们就计算了每个点上哪些点是活跃的，因为每个点上集合的大小总是越来越大的，并且存在点个数的上限，是有限可终止的。

## 常数传播

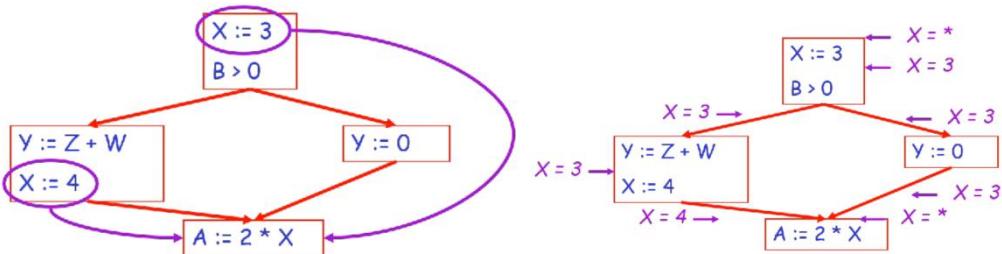
定义：在满足某个条件的情况下，将使用到  $x$  的地方用一个常数  $k$  来代替。

条件：在使用到  $x$  的每条路径上，最终的 assignment 是  $x := k$ 。

为了仔细描述这个问题，我们定义如下的三种  $x$  的情况在每个程序点上（at every program point）。

| value | interpretation                  |
|-------|---------------------------------|
| #     | This statement is not reachable |
| c     | $x = \text{constant } c$        |
| *     | Don't know if $x$ is a constant |

$x$  在每个点都有三个值之一#（还没开始），c（确定这一点是常数）、\*（这个点一定不是常数）

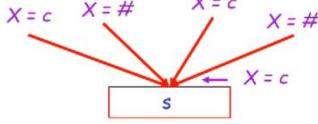
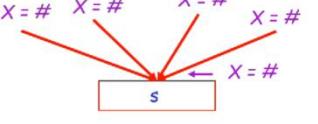
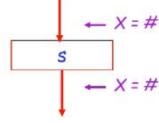
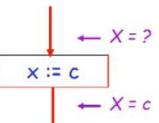
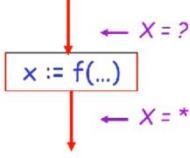
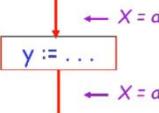


对于每条语句  $s$ ，我们可以如下定义  $x$  在某条语句前后的值。

$C_{in}(x, s)$ ：在  $s$  语句前  $x$  的值。  $C_{out}(x, s)$ ：在  $s$  语句后  $x$  的值。

## 常数传播的八条规则

| Rule 1                                                                                                 | Rule 2                                                                                                                                           |
|--------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>if <math>C_{out}(x, p_i) = *</math> for some <math>i</math>, then <math>C_{in}(x, s) = *</math></p> | <p>If <math>C_{out}(x, p_i) = c</math> and <math>C_{out}(x, p_j) = d</math> and <math>d \neq c</math><br/>then <math>C_{in}(x, s) = *</math></p> |

|                                                                                                                                                                                                                               |                                                                                                                                                                                                             |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Rule 3</b><br> <p>if <math>C_{out}(x, p_i) = c \text{ or } \# \text{ for all } i</math>,<br/>then <math>C_{in}(x, s) = c</math></p>       | <b>Rule 4</b><br> <p>if <math>C_{out}(x, p_i) = \# \text{ for all } i</math>,<br/>then <math>C_{in}(x, s) = \#</math></p> |
| <b>Rule 5</b><br> <p><math>C_{out}(x, s) = \# \text{ if } C_{in}(x, s) = \#</math><br/>and <math>s</math> does not modify <math>x</math></p> | <b>Rule 6</b><br> <p><math>C_{out}(x, x := c) = c \text{ if } c \text{ is a constant}</math></p>                         |
| <b>Rule 7</b><br> <p><math>C_{out}(x, x := f(\dots)) = *</math></p>                                                                        | <b>Rule 8</b><br> <p><math>C_{out}(x, y := \dots) = C_{in}(x, y := \dots) \text{ if } x \neq y</math></p>              |

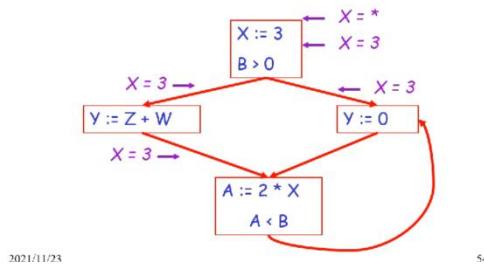
规则 1-4 是根据  $s$  的前驱的  $C_{out}$  来更新  $s$  的  $C_{in}$ ，是沿着 CFG 的边前向传播。而 rule5-8 是根据语句  $s$  的  $C_{in}$  来更新语句  $s$  的  $C_{out}$ 。

## 常数传播算法

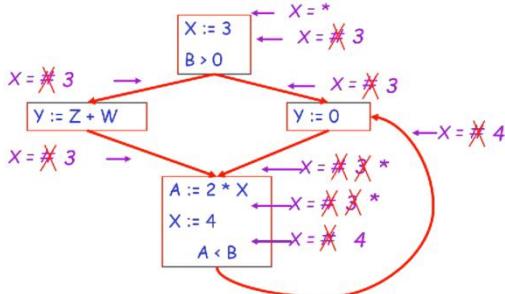
1. 对于程序中的每个入口  $s$ ，我们令  $C_{in}(x, s) = *$ （这个点一定不是常数），其他所有语句  $C_{in}(x, s) = C_{out}(x, s) = \#$ （还没开始）
2. 接下来不断根据规则 1~8 来更新相应的  $C_{in}$  和  $C_{out}$

## The Value #

- To understand why we need #, look at a loop

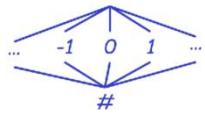


## Another Example



## Orderings

- We can simplify the presentation of the analysis by ordering the values  
 $\# < c < *$
- Drawing a picture with "lower" values drawn lower, we get



## Orderings (Cont.)

- \* is the greatest value, # is the least
  - All constants are in between and incomparable
- Let lub be the least-upper bound in this ordering
- Rules 1-4 can be written using lub:  
 $C_{in}(x, s) = \text{lub} \{ C_{out}(x, p) \mid p \text{ is a predecessor of } s \}$

## 常数传播算法的中止性

因为一条语句的绑定的值只可能从 $\# \rightarrow c \rightarrow *$ , 所以也一定会终止。

**2021/11/26**

我们的 lab5 第二部分是要生成 Assem, 在 lower IR 中和真正汇编的区别就是里面用到的寄存器还是虚的。

在寄存器分配的时候, 就是要把汇编指令中虚的寄存器替换成为真正的物理寄存器。因为通常来说 temporary 比真正的 register 要多, 这时候的原则如下:

我们之前做了 liveness analysis, 当两个变量同时活跃的时候, 只能在两个寄存器中。因为这两个值在后面都是被需要的, 所以我们不能分到同一个物理寄存器中。在这里, 我们这种就称之为两个 variable interfere。

我们接下来看一个例子:

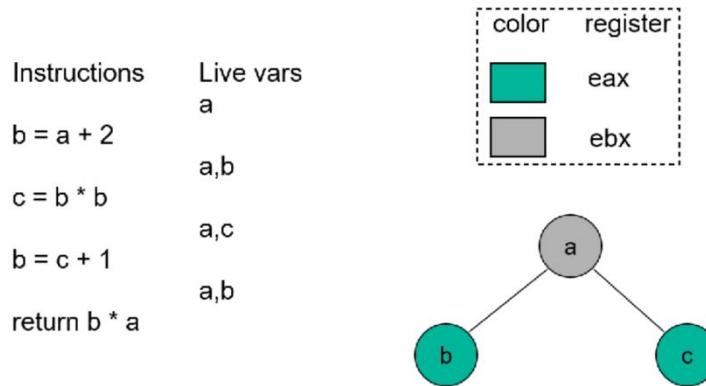
## Discussion

- Consider the statement  $Y := 0$
- To compute whether  $X$  is constant at this point, we need to know whether  $X$  is constant at the two predecessors
  - $X := 3$
  - $A := 2 * X$
- But info for  $A := 2 * X$  depends on its predecessors, including  $Y := 0$ !

## Interference Graph

| Instructions | Live vars |
|--------------|-----------|
|              | a         |
| b = a + 2    | b,a       |
| c = b * b    | a,c       |
| b = c + 1    | b,a       |
| return b * a |           |

在任意一个点，我们都必须有一个活跃信息，如果同时活跃就不能分到同一个寄存器。因为我想让相干的变量不能分到同一个寄存器，我们就根据相干性来构造一个图。



节点是变量，如果两个变量同时活跃就画一条边。这样我们就画出来了相干图，这样我们就把问题转化为了图上的着色问题。以上图为例，如果我们有 2 个寄存器，那么我们就去看这个图是不是 2-可着色的问题，显然是的。

如果某个相干图不是  $k$ -可着色的，那么显然寄存器是不够的，那么有些变量就要放到内存中去。Finding a K-coloring in all situations is an NP-Complete Problem; 求解一张图是几可着色是 NP Hard 问题。所以求解就不太方便，我们只能使用一些启发式算法。

具体地，相干图的构造方法为：对于一条指令  $i$ ，它 define 了变量  $a$ ，并且有一个 live-out 集合： $L_{out} = \{b_1, \dots, b_j\}$ ，如果指令  $i$  并不是一个 move 指令，那么在相干图上连上边  $(a, b_1), \dots, (a, b_j)$ ；如果指令  $i$  是一条 move 指令，记作  $a \leftarrow c$ ，那么我们对满足  $b_i \neq c$  的边

连上  $(a, b_i)$ ，此时的  $a$  和  $c$  可以视为虚线连接（move-related），我们是希望在 coalesce 操作中把这两个节点尽可能合并的。

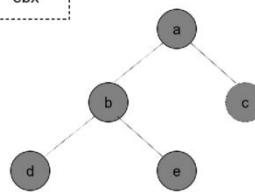
## Kempe's Algorithm

这个启发式算法就是在图中找一些节点，要求节点的度数小于  $k$ ，拿掉这些节点不影响图的可着色性。如果图是  $k$ -可着色的，因为拿掉的节点放进去边小于  $k$ ，所以一定有一个颜

色是可以分配给这个被拿掉的节点。我们用一个栈来记住这个节点被拿掉的顺序。

### Coloring (Assume K=2)

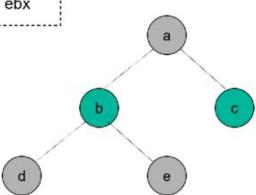
| color | register |
|-------|----------|
| green | eax      |
| gray  | ebx      |



### Coloring (Assume K=2)

| color | register |
|-------|----------|
| green | eax      |
| gray  | ebx      |

stack:  
d  
b  
a  
e  
c

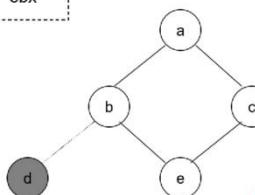


stack:

我们按照 ceabd 的顺序压栈，然后一个个从栈中 pop (顺序为 dbaec) 出来分配颜色。如果 simplify 能把整张图拿掉，那么一定是可以分配出一组颜色的。

### Optimistic Coloring (K=2)

| color | register |
|-------|----------|
| green | eax      |
| gray  | ebx      |



2021/11/26

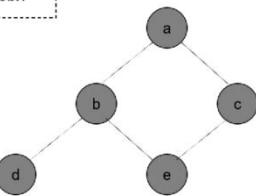
all nodes have  
2 neighbours!

27

| color | register |
|-------|----------|
| green | eax      |
| gray  | ebx      |

stack:  
d

- Continue the simplification
  - Simplification now succeeds: a, e, c

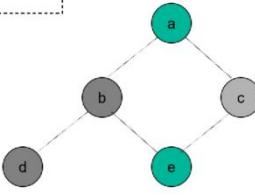


stack:  
c  
e  
a  
b  
d

此时我们就说可能有一个值要放到内存里去。我们先选出 b 拿掉，也进栈，剩下这个图就可以继续 simplify 了，所以我们可以继续拿掉全部压栈。

我们现在一边退栈一边着色，

| color | register |
|-------|----------|
| green | eax      |
| gray  | ebx      |



stack:

b  
d

b 我们刚才说是有问题的，b 放回来的时候不能保证一定能着色，但是当前这种情况，b 放回来正好两个邻居着了相同的颜色，所以我们可以继续做下去。这种情况就是一种 optimistic coloring 的情况。

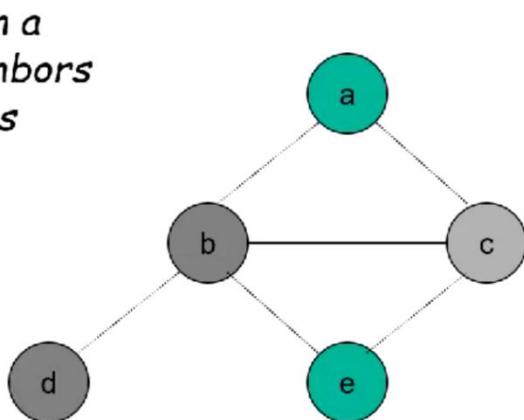
不成功的情况：

## Select: What if the Heuristic Fails?

- What happens if no color can be assigned to a spilled node ?

*when we have to assign a color to b whose neighbors have 2 different colors already*

stack:



跟刚才的图拿走节点顺序是一样的 dbace。给 b 着色的时候，我们发现 b 没有颜色可以选，这时候 b 肯定是要被 spill (寄存器溢出) 的。这种就叫做 actual spill。

## Actual Spill (寄存器溢出)

### Actual Spill

- Step 5 (actual spill): If no color can be assigned to a spilled node
  - Select step stops assigning color and chooses an actual spilled node
  - There are many heuristics that can be used to pick a node
    - not in an inner loop

在选取的时候，我们倾向于选择度数比较大的节点。我们希望这样的计算不在内循环中。如果在最内层循环放到了内存中，那么会有反复的读取内存操作。但是在汇编这个层次找到谁是最内层循环就比较困难了。

## Spilling

---

- Since optimistic coloring failed we must spill temporary  $b$
- We must allocate a memory location as the home of  $b$ 
  - Typically this is in the current stack frame
  - Call this address  $b_a$
- Before each operation that uses  $b$ , insert  $b := \text{load } b_a$
- After each operation that defines  $b$ , insert  $\text{store } b, b_a$

我们总归到最后会选择出一个 temporary  $b$  来进内存。每一次要用它的时候，我们就要把它从内存中拿出来，每次用完了就把计算出来的值存回去。加了这段东西以后，相干图还是在的，但是  $b$  的 liveness 的 range 就变短了，它和别的节点同时活跃就少了。那么相干图的边就变少了，更容易去着色了。

## Recomputing Liveness Information

---

- The new liveness information is almost as before, however
  - Spilling reduces the live range of  $b$
  - And thus reduces its interferences
  - Which result in fewer neighbors in RIG for  $b$

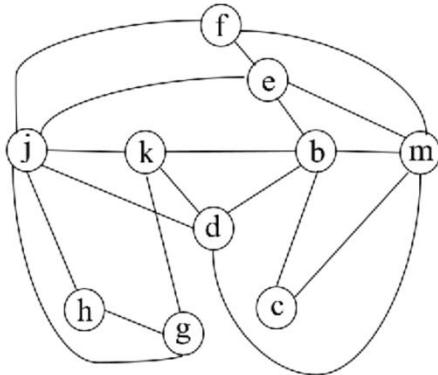
实际上，寄存器分配没那么简单。我们来看如下的这个例子：

## Example(K=4)

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 k j

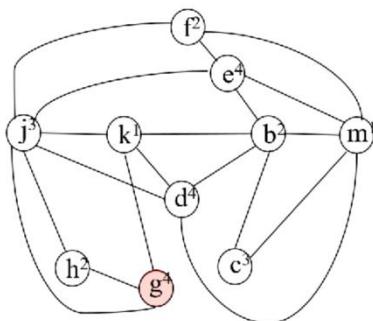


我们按照顺序拿 ghkfjdebcm

## Example(K=4)

Stack

|   |   |
|---|---|
| m | 1 |
| c | 3 |
| b | 3 |
| e | 4 |
| d | 4 |
| j | 3 |
| f | 2 |
| k | 1 |
| h | 2 |
| g | 4 |

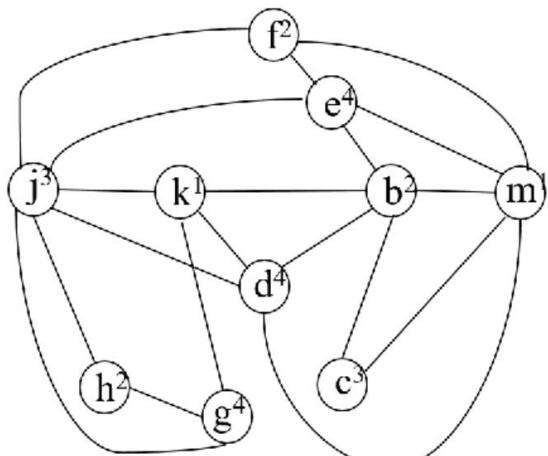


## Example(K=4)

Live in: k j

```
r4 := mem[r3+12]
r2 := r1-1
r2 := r4*r2
r4 := mem[r3+8]
r1 := mem[r3+16]
r2 := mem[r2]
r3 := r4+8
r4 := r3
r1 := r1+4
r3 := r2
```

Live out d k j



我们代回去就得到了这个汇编，注意红色部分是有两种选择的时候随便选了一种颜色值。我们想做的事情是把它们俩赋值给同一个寄存器。分配两个物理寄存器本身没那么错，但是我们想做的是 `r4 := r4, r2:=r2`，这样这两条语句就不要了

因为我们会做 `view-shift`，把物理寄存器赋值给一个虚的寄存器。

$t_1 \leftarrow \%rdi$

$t_2 \leftarrow \%rsi$

Return  $Q(Y) + Q(X)$

实际上，我们发现用到了 2 个 callee-saved register，因为我们要把`%rsi`写到`%rdi`中，所以我们 call 完了第一个 Q，我们需要把临时存的 x 释放到`%rdi`中，并且要把第一个 Q 返回的`%rax`存到 `t3` 中，否则会被第二个 Q 的返回值`%rax`覆盖掉。

此时 `t1` 和 `t3` 就被变成必须的了。

因为我们用的是 `procedureEntry1~3`, 2 我们是在每个 procedure 后贴了一个 live 信息，1 是贴 `view-shift` 信息。我们会发现上面的 `t2` 的这个 `view-shift` 是没用的，所以我们最好在 register allocation 的时候把 `t2` 分配给 `rsi`，这样  $t_2 \leftarrow \%rsi$  这一句就没用了可以删掉。

## Coalescing

---

- For an instruction that defines a variable  $a$ , where the live-out variables are  $b_1, \dots, b_j$ , the way to add interference edges for it is
  - If it is a nonmove instruction, add  $(a, b_1), \dots, (a, b_j)$
  - If it is a move instruction  $a \leftarrow c$ , add  $(a, b_1), \dots, (a, b_j)$ , for any  $b_j$  that is not the same as  $c$

对于定义变量 `a` 的指令，在后面 live-out 的地方，我们有变量 `b1~bj`，我们在加 interference 的时候我们就加进去。我们在 assem 的时候把 instruction 分成了三类 move, op。我们这时候就要判定 `b1` 到 `bj` 有没有一个 `c`，如果有 `c` 的话，我们就不把这条边加进去。

## Coalescing

---

- No edge in a RIG between the src and dest of a move instruction
  - The move instruction can be eliminated
  - The src and dest are coalesced into a new node whose edges are the union of those of the nodes being replaced
- Problem: coalescing can increase the number of interference edges and make a graph uncolorable

但是我们这条边少了以后，我们试图把这条边所连着的两个节点合在一起。但是合完以后会有一个问题，图上的有些节点的度数会变大。这样本来那张图的度数是小于  $k$  的，但是我们合完以后度数大于  $k$  就不可着色了。所以这种情况我们需要避免，也就是我们在判定两个节点是否可以合并的时候，我们要先看是不是影响我们图的继续着色。

我们现在有两个启发式算法

## Briggs 算法

### Heuristic Coalescing

---

- **Briggs:** the coalesced node will have fewer than  $K$  neighbors of significant degree
- **Correctness:**
  - After the simplify phase has removed all the low-degree nodes of the graph
  - The coalesced node has neighbors less than  $K$ , so it can be removed

$a$  和  $b$  合并完的节点 $\{a, b\}$ ，我们看看它的 neighbour，分为两类，一类是度数小于  $k$ ，另一类是度数大于等于  $k$ （称为 significant），我们要求 significant 节点的个数要小于  $k$  个。此时我们可以先把 $<k$  的非 significant neighbour 拿掉，这是肯定可以拿掉了。这样 $\{a, b\}$ 的度数一定小于  $k$ ，所以我们可以把 $\{a, b\}$ 拿掉，剩下的图就是原来的子图了。所以这样原来的子图可以做  $k$  着色，那就可以，否则就不行，所以  $a$  和  $b$  节点的着色没有影响到原来图的可着色性质的判定。

## George 算法

### Heuristic Coalescing

---

- **George:**  $a$  can be coalesced with  $b$  if every neighbor  $t$  of  $a$ :
  - already interferes with  $b$ , or has low-degree ( $< K$ )
- **Correctness:**
  - Suppose  $S$  is the set of all low-degree nodes of  $a$
  - Simplifying the original graph will remove all nodes in  $S$  and the result graph is  $G_1$
  - Simplifying the coalesced graph will remove all nodes in  $S$  and the result graph is  $G_2$
  - $G_2$  is a subgraph of  $G_1$ 
    - node  $ab$  in  $G_2$  corresponds to  $b$  in  $G_1$

我们想把节点  $a$  和节点  $b$  合起来的时候，我们说  $a$  有一些 neighbour， $a$  的每个 neighbour 也分两类，度数小于  $k$  和度数大于等于  $k$ 。度数小于  $k$  的我们不管它，如果大于等于  $k$ ，要求已经和  $b$  之间有一条边了。现在我们合并  $a$  和  $b$  即可。这样合并完  $b$  的节点的 significant neighbour 并没有增加。

我们令  $S$  是  $a$  里的度数小于  $k$  的 neighbour 集合。在没有合并的时候，我们可以把  $S$  先 simplify 掉。剩下的 neighbour 都是  $b$  的了。所以  $a$  和  $b$  合并还是不合并都对图的可着色性没有影响了。

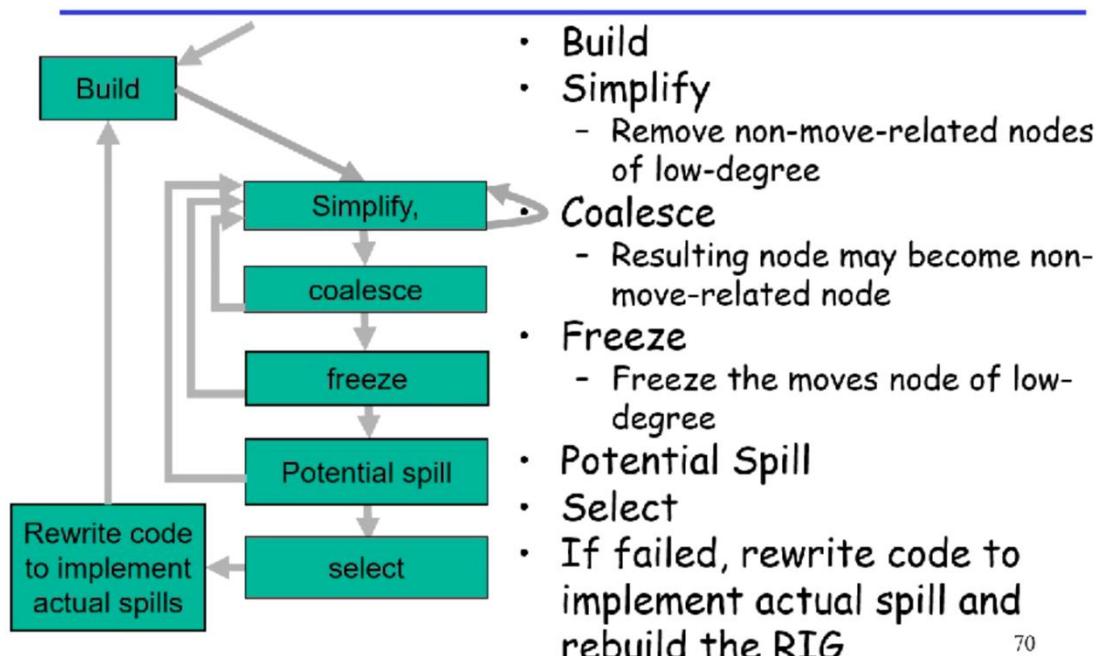
## freeze

- Build
- Simplify
  - Remove non-move-related nodes of low-degree
- Coalesce
  - If resulting node may become non-move-related node, back to simplify
- freeze
  - Look for a move-related node of low-degree
  - Freeze the moves (give up coalesce any more)
  - The frozen node can be simplified

因为我们要合并 a 和 b, 所以这种节点不能 simplify 掉, 所以我们 simplify 是 simplify 掉非 move-related 的节点。Simplify 结束以后, 剩下了 a 和 b 和 significant node, 我们就来看 a 和 b 是不是能合并, 合并完以后就变成了 non-move-related 了, 我们就继续做 simplify。

如果要分配到两个不同的寄存器中, 我们就放弃合并, 变成 non-move related node。

## Overall Algorithm



我们基本上就是这么一个反复迭代的流程。先做 simplify, 能做就做, 但是只能 simplify

non-move-related node。这时候我们尝试合并 a 和 b, 如果可以的话, 那么合完以后{a, b}就变成了 non-move-related node, 继续回去 simplify, 如果没有东西可以 coalesce, 那么我们就去 freeze, 不合并这两个节点了。再上去做 simplify。如果图上还有东西, 所以所有的节点都是 significant node, 这时候我们就要 spill 了, 此时是 potential spill, 标记为可能回内存, 继续回去 simplify, 做到最后图就为空了, stack 就满了。接下来我们就做 select (退栈着色过程), 如果标记为了 potential spill, 到了某个节点可能没有办法着色了, 此时我们就真的去做 actual spill 当前的这个 node, 重新写这个 node 的 load 和 store 信息。此时 liveness 的信息就会发生变化, 所以我们要重新构造 RIG 图。然后再回来做一遍这件事情, 直到最终停下来。肯定会停, 大不了所有变量都进内存。

这两年的考题最后一题都是 register allocation, 基本上都会重新画几遍 RIG 的。

## Example(K=4)

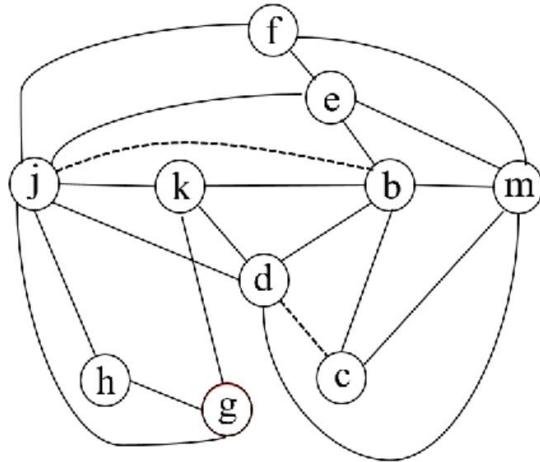
Live in: k j

```

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

```

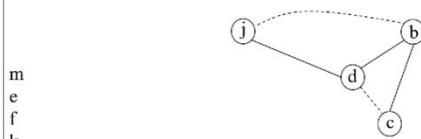
Live out d k j



这样我们就把 dc 和 jb 分到了同一个寄存器里去。在 simplify 的时候要留下 j 和 b 还有 d 和 c。新算法的流程如下

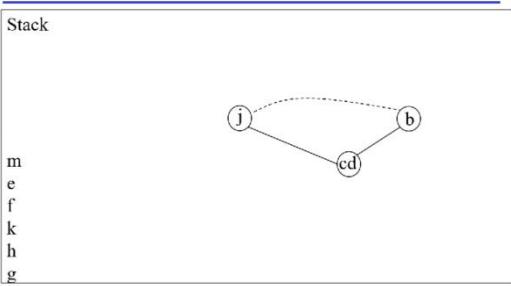
## Coalescing(K=4)

Stack



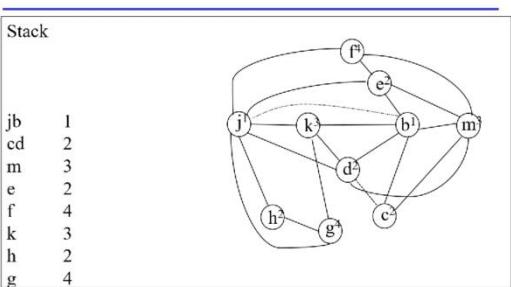
, 我们先选 d 和 c 看看是否可以合并。它们的 neighbour 和 j 都不是 significant node, 所以 d 和 c 可以合并成{c, d}。

### Coalescing(K=4)



此时 cd 就变成了 non-move-related node，可以被拿走了，然后合并 jb 即可。

### Coalescing(K=4)



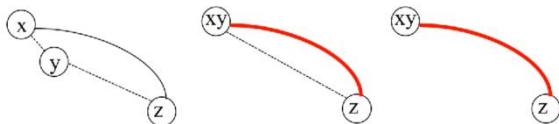
### Example(K=4)

|                |                   |
|----------------|-------------------|
| Live in: k j   | Live in: r3,r1    |
| g := mem[j+12] | r4 := mem[r1+12]  |
| h := k-1       | r2 := r3-1        |
| f := g*h       | r4 := r4*r2       |
| c := mem[j+8]  | r2 := mem[r1+8]   |
| m := mem[j+16] | r3 := mem[r1+16]  |
| b := mem[f]    | r1 := mem[r4]     |
| c := e+8       | r2 := r2+8        |
| d := c         | r3 := r2+4        |
| k := m+4       | Live out r2 r3 r1 |
| j := b         |                   |
| Live out d k j |                   |

其实我们做的事情就是暂缓了 dc 和 jb 的 simplify 操作。

## Constrained Move (受限的 Move 操作)

- Some moves are constrained
- Example
  - After x and y are coalesced
  - xy  $\leftarrow$  z cannot be coalesced further
  - Because of (x, z) are interfered, freeze it



所以我们为了减少一个 register 之间的 move，我们现在多了很多事情出来把寄存器分配这件事情变得比较复杂。

## Precolored Nodes

---

- Some temporaries are pre-assigned to machine registers
  - div instruction on x86/pentium
    - uses %eax; defines %eax, %edx
  - Registers passing parameters such as %rdx, %rsi
- These temporaries are called precolored
  - For any given color, there should be only one precolored node of that color

实际上我们在做 view-shift 的时候， $t_1 \leftarrow \%rdi$ 。在 interference graph 里，节点上肯定有真的寄存器。这是参数导致的真的寄存器出现在 interference graph 中，还有一些情况，比如我们的长整数乘除法，它天生会用到%rdx 和%rax。

## Precolored Nodes

---

- For a K-register machine, there will be K precolored nodes that all interfere with each other
- A machine register used explicitly will have a live range that interferes with any other variables that happened to be live at the same time
- We can neither simplify a graph by removing a precolored node nor spill a precolored node to memory

如果我们机器有 k 个寄存器，我们就有 k 个 precolored node，这是一个 k 个节点的完全图。在 simplify 的时候，不能把 precolored node simplify 掉，我们也不能把 precolored node spill 掉。

## Precolored Nodes

---

- Select and coalesce operations can give an ordinary temporary the same color as a precolored register, as long as they don't interfere
  - A calling convention register can be reused inside a procedure as a temporary variable
  - Precolored node may be coalesced with other non-precolored nodes using conservative coalescing

## Make Precolored Nodes Live Short

---

- The coloring algorithm works by calling
  - simplify, coalesce and spill until only the precolored nodes remains
  - select to color nodes or generate spilling codes
- Precolored nodes do not spill, the front end must keep their live ranges short
  - Generate MOVE instructions to move values to and from precolored nodes
  - A callee-save register can be considered as "defined" at procedure entry and "use" at procedure exit

## Callee-Save register

---

- Introduce move instructions and  $t_{231}$  which can
  - either be coalesced with  $r_7$  if no register pressure
  - or be spilled to memory if register pressure

enter: def( $r_7$ )

...

...

...

exit: use( $r_7$ )

enter: def( $r_7$ )

$t_{231} \leftarrow r_7$

...

...

...

$r_7 \leftarrow t_{231}$

exit: use( $r_7$ )

接下来我们要处理 function call。Callee-saved register 的行为如上图右侧所示，我们希望上来把 callee-saved register 都存到，出去前再给它恢复。Prologue 和 epilogue 在其中有一部分就是要做做的事情。

caller-saved register

## Caller-Save Registers

---

- The CALL instructions in the Assem language have been annotated to *define* (interfere with) all the caller-save registers
- A variable will tend to be allocated to a caller-save register
  - if it is not live across a procedure call
- a variable interferes with all the caller-save register
  - If it is live across a procedure call
- Most variables interfere with new temporaries created for callee-save registers such as  $t_{231}$

我们为什么要 coalesce，就是为了处理 callee, caller 和 view-shift。Call 这条指令是一条 op instruction，后面跟了两个 dest 和 source 还有一个 label。在 destination 里，就是把 caller-saved register 全部放进去。在 function call 的时候，我们假设其一定被写过，除非我们做 inter procedure analysis。所以我们就在那个里把 define 放到这里。

## Put it all Together Example(K=3)

---

|                     |        |                          |
|---------------------|--------|--------------------------|
| int f(int a, int b) | enter: | $c \leftarrow r3$        |
| {                   |        | $a \leftarrow r1$        |
| int d=0;            |        | $b \leftarrow r2$        |
| int e=a;            |        | $d \leftarrow 0$         |
| do {                |        | $e \leftarrow a$         |
| d=d+b;              | loop:  | $d \leftarrow d+b$       |
| e=e-1;              |        | $e \leftarrow e-1$       |
| } while (e>0);      |        | if $e > 0$ goto loop     |
| return d;           |        | $r1 \leftarrow d$        |
| }                   |        | $r3 \leftarrow c$        |
|                     |        | return (r1, r3 live out) |

我们看到  $r1$  和  $r2$  是传参的， $r3$  是一个 callee-saved register。 $r1$  又作为返回值的寄存器来使用。

```

/* codegen.h */
namespace cg {
class CodeGen {
public:
 CodeGen(frame::Frame *frame, std::unique_ptr<canon::Traces>
traces) : frame_(frame), traces_(std::move(traces)) {}
 void Codegen();
private:
 frame::Frame *frame_;
 std::unique_ptr<canon::Traces> traces_;
 std::unique_ptr<assem::InstrList> assem_instr_;
};
}

```

第二部分就是 code generation 我们要做的事情。第一部分做完以后就是一个 trace。有了 frame tiling 完就是一个 instruction list。

```

/* codegen.cc */
namespace cg {
void CodeGen::Codegen() {
 auto *list = new assem::InstrList();
 for (auto stm : traces_->GetStmList()->GetList())
 stm->Munch(*list);

 assem_instr_ =
 std::make_unique<AssemInstr>(frame::ProcEntryExit2(list));
}
}

```

我们把 trace 里的一个个 statement 来做 tiling，然后我们合到一起来做 ProcEntryExit2。生成 shift-of-view、生成头尾，2 是做 register allocation 来用的。

## 2021/11/30

上节课基本上讲了寄存器分配的原理，我们以下面这个例子为例。

R1 和 R2 是传参的寄存器，R3 是 callee-saved register

## Put it all Together Example(K=3)

```
int f(int a, int b)
{
 int d=0;
 int e=a;
 do {
 d=d+b;
 e=e-1;
 } while (e>0) ;
 return d;
}
```

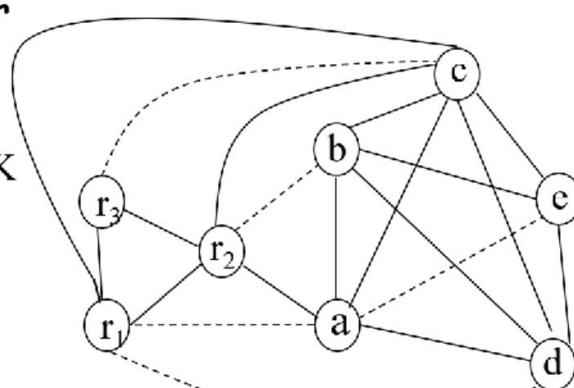
|        |                               |
|--------|-------------------------------|
| enter: | $c \leftarrow r_3$            |
|        | $a \leftarrow r_1$            |
|        | $b \leftarrow r_2$            |
|        | $d \leftarrow 0$              |
|        | $e \leftarrow a$              |
| loop:  | $d \leftarrow d+b$            |
|        | $e \leftarrow e-1$            |
|        | if $e > 0$ goto loop          |
|        | $r_1 \leftarrow d$            |
|        | $r_3 \leftarrow c$            |
|        | return ( $r_1, r_3$ live out) |

在结束的时候  $r_1$  和  $r_3$  是活跃的，这就是 procedureentryexit2 干的事情。大家在生成 instruction selection 最后一步会加一条 live out 的 register。我们从最后开始一点点往上走，做 liveness analysis 从而构造 interference graph。如果两个变量同时活跃，那么在 interference graph 就有一条边。

$r_1, r_2, r_3$  是 precolored register，构成了一个完全图。在 goto 后面  $c$  和  $d$  就同时活跃了。如果我们的死码没有做死码删除，那么  $f < -1$  是要放一个寄存器去存的，所以  $f$  和后面的变量也会有 interference graph。

## Put it all Together Example(K=3)

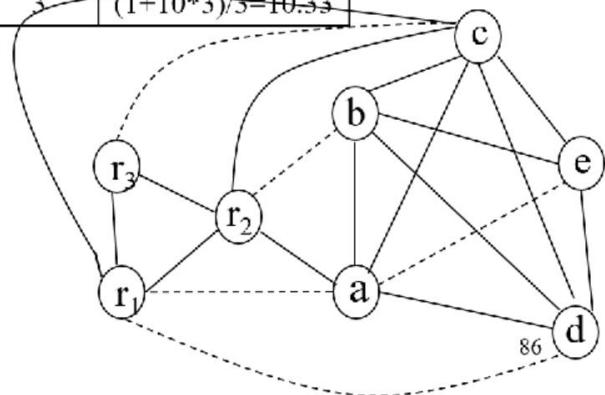
- No way to simplify or freeze
  - All non-precolored nodes have degree  $\geq K$
- No way to coalesce
  - No less than  $K$  significant-degree nodes for coalesced nodes
- The only way is spilling



3 条虚线对应  $r_3$  和  $c$  (有一条赋值)， $r_2$  和  $b$ ， $r_1$  和  $d$ 。所以  $dbe$  是在内循环中出现的，

## Spilling Criteria

| Node | Use+Def<br>Outside loop | Use+Def<br>inside loop | Degree | Spill priority     |
|------|-------------------------|------------------------|--------|--------------------|
| a    | 2                       | 0                      | 4      | $(2+10*0)/4=0.5$   |
| b    | 1                       | 1                      | 4      | $(1+10*1)/4=2.75$  |
| c    | 2                       | 0                      | 6      | $(2+10*0)/6=0.33$  |
| d    | 2                       | 2                      | 4      | $(2+10*2)/4=5.5$   |
| e    | 1                       | 3                      | 3      | $(1+10*3)/3=10.33$ |

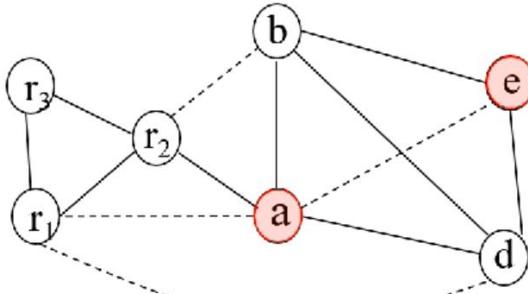


2021/11/30

变量进内存，出现的次数越多，那么访存次数也会越多，我们优先考虑内循环出现的尽量不要 spill 掉。外循环我们就要看出现的次数。我们对内循环的次数 \* 10 来放大这个因素。这样 spill priority 的时候，我们就优先拿掉。大家在做 lab 的时候就随机选一个就行了。

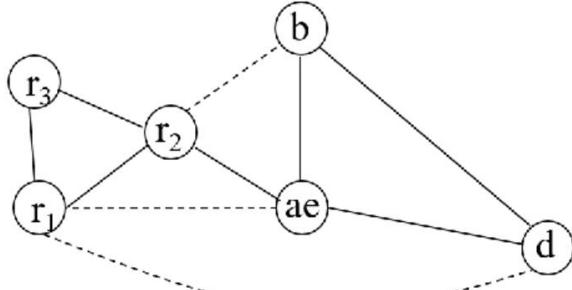
所以我们先拿掉 c，标记为 potential spilled 的点。

- *Coalesce a and e*
  - Only use briggs



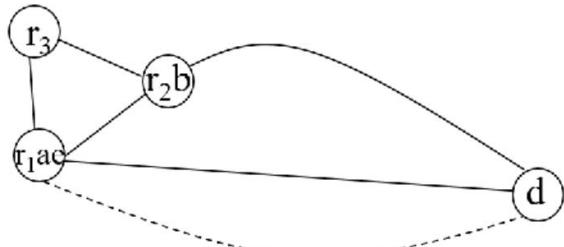
现在剩余的几个点都是 move-related，不能去做 simplify 了。我们只能去做 coalesce。Briggs 可以用来作为非 temporary register。它合并完以后，significant node 会小于 k。

- We can only coalesce with pre-colored nodes
  - Only use George
  - Merge normal node to pre-colored node
  - b & r2 and ae & r1
- r1 and d cannot be coalesced because it will introduce a significant-degree node ae to r1



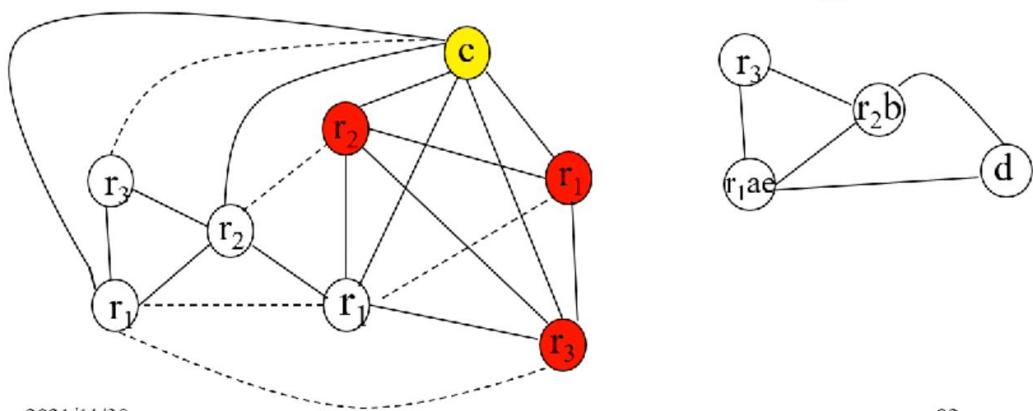
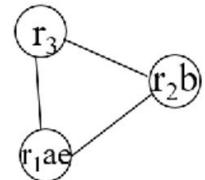
Ae 合并完以后，只有 r2 是 significant node。下面再要合并的时候都是 temporary 和 precolored register，此时我们使用的是 George 方法，我们得出结论 r2 和 b 可以合并，要把 b 合并进 r2 里，ae 也可以合入 r1.但是 d 和 r1 不能合，因为 ae 的 degree = 3，所以不是 r1 的 neighbour，合入以后 r1 会多一个 neighbour。

- Now d becomes constrained node



此时 d 和 r1ae 就叫做 constraint move，我们就可以去掉这条边，也就是 freeze 掉。接下来我们就可以拿掉，然后把 d 拿掉。

- We pick d and give it only color r3
- Now a&e are r1, b is r2
- c must be spilled



2021/11/30

92

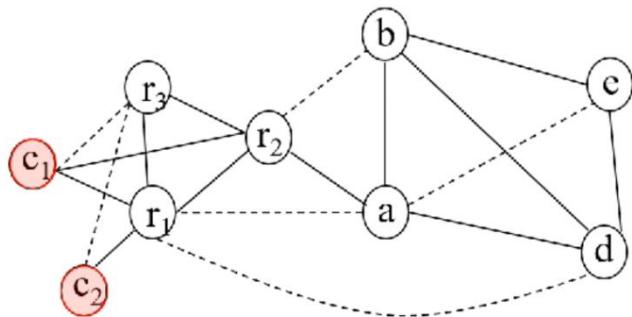
我们发现 c 和 r1, r2, r3 都有 neighbour，此时 c 就必须真的 spill 掉。因为 r3 是一个 callee-saved register 一旦我们中间用到 r3 的话，必然要 push-pop 一次。

```

enter: c1 ← r3
 M[cloc] ← c1
 a ← r1
 b ← r2
 d ← 0
 e ← a
 loop: d ← d+b
 c ← c-1
 if c>0 goto loop
 r1 ← d
 c2 ← M[cloc]
 r3 ← c2
 return (r1, r3 live out)

```

2021/11/30

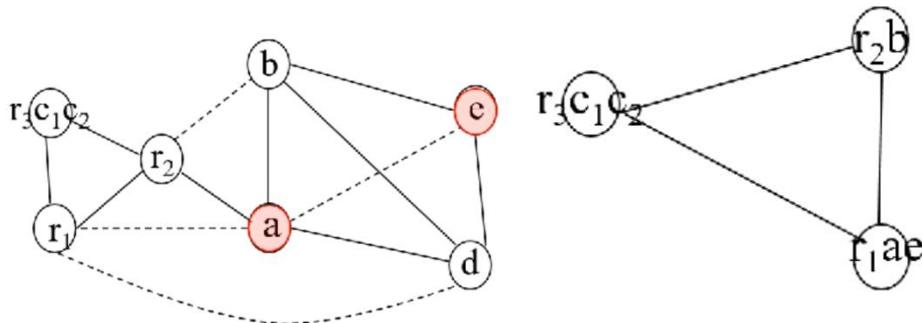


*c is live only*

- Between a store  $c$ ,  $c_a$  and the preceding instruction
- Between a  $c := \text{load } c_a$  and the next instruction

94

再要用  $c$  的时候，就去从 memory 里拿出来，这样  $c_1$  和  $c_2$  的 live range 就特别短。新增的 2 个节点，度数为 2 和 1 很低。通过 spill 方法，这个图更加容易着色了。



## Saving Spilling Spaces

- There may be many temporaries to be spilled
  - The number of hard registers are small such as IA32
  - Advanced analysis techniques introduce more temporaries such as SSA
- Moves between spilled temporaries are expensive, and waste registers
  - We can coalesced move related temporaries without restriction
  - Because there unlimited number of colors in memory

我们在 spill 的时候，每 spill 一个变量，我们就在内存中增加一个空间去放这个变量。如果 a 和 e 都被 spill 了，在上例中就会导致 memory 中来回 copy。所以 coalesce 也可以对于 spilled node 来做。这样减少 memory 之间的 move。如果有比较多的 temporary 被 spill 了之后，来回的内存 move 是很昂贵的，在 x86 中，必须先 load 到 register，再 move 到对应的内存里去，所以会浪费寄存器。因此，我们希望尽量地去 coalesce，尽量把 a 和 e 放到同一个 memory 里去。

## Saving Spilling Spaces

---

- Build interference graph for spilled nodes
- While there is any pair of non-interfering spilled nodes connected by a move instruction, coalesce them
- Use simplify and select to color the graph
  - Simplify picks the lowest-degree node
  - Select picks the first available color
  - No limit on the number of colors

所以，我们要对于 spilled node 再去做一次 interference graph，如果是赋值就是一条虚线。Simplify 比较容易，因为 memory 是无限的，不需要关心是 k-可着色的。

## Saving Spilling Spaces

---

- When to do it?
  - Before generating the spill instructions
- What to do after it?
  - Regenerating the RIG and do coloring again
- Why?
  - Avoid creating fetch-store sequences for coalesced moves of spilled nodes
  - Less interference after coalescing

真的要产生 spill instruction 的时候，在生成指令之前，我们需要给这些变量分配空间。

## 在 Tiger 中寄存器分配的实现

前面是 lab6 的原理，后面是 lab6 的实现。

### Graph

---

- Graph abstract data type
- graph::Graph()
  - Create an empty directed graph
- g->NewNode(x)
  - Make a new node within graph g
  - x is any extra information "attached" to the new node
- g->AddEdge(n,m)
  - Create a direct edge from n to m
  - n->Succ()->Append(m), m->Pred()->Append(n)
  - m->Adj()=m->Succ() ∪ m->Pred(m)

首先，我们有 graph 这个 class。Control flow graph 是有向图，而 interference graph 是无向图。给定一个图 G 的时候，我们可以在图上增加一个节点。节点会和一些信息关联起来，代表了 basic-block、code、temporary、precolored register。在构造节点的时候，信息就会 attach 到节点上。

### Attach additional information to nodes

---

- Each node represents some thing
  - An instruction in a program
  - Dataflow information
  - A variable in a program
- Two ways to map from node to things
  - Put things in node directly (via a pointer)
    - n=g->NewNode(x)
    - n->Nodeinfo() retrieves x
  - Using graph::Table = tab::Table<Node<T>, ValueType>
    - mytable->Enter(n, y)
    - Associates information y to node n in a mapping mytable

在 control flow graph 中，节点代表了一些指令，每个节点之后，我们还需要 liveout information，这种我们该怎么样在图上画呢？可以直接绑定到节点上，节点维护一个指针，此外还可以用 table 来维护。

## Control Flow Graph

---

- class Instr {  
public:  
    [[nodiscard]] virtual tempList \*Def() const = 0;  
    [[nodiscard]] virtual tempList \*Use() const = 0;  
};
  - def information is obtained from the dst fields of the instruction
  - use information is obtained from the src fields of the instruction
- typeid(\*instr) == typeid(MoveInstr)
  - check if the instruction is a move instruction

这个 Instr 类里对每条 instruction 构造一个 def 和 use, 实际上就是 destination 和 source。

有了图的基本结构后，我们就来构造 CFG 和 RIG

## Control Flow Graph

### Control Flow Graph

---

```
class FlowGraphFactory {
public:
 void AssemFlowGraph();
 graph::Graph<assem::Instr> *GetFlowGraph() { return flowgraph_; }
```

```
private:
 assem::InstrList *instr_list_;
 graph::Graph<assem::Instr> *flowgraph_;
 tab::Table<tcmpl::Label, graph::Node<assem::Instr>> *label_map_;
};
```

- AssemFlowGraph() will construct the flow graph and store into flowgraph\_
  - Info of each graph::Node is actually a pointer to an assem::Instr
  - jump fields of the instrs are used to in creating control flow edges

map 就是用来记录关联信息的。

## Register Allocation

---

- Takes a flow graph produce a live::LiveGraph
  - An interference graph
    - Using liveness information at the exit of each node
  - A list of node-pairs representing MOVE instructions
- If a variable is defined but never used, we still need to generate interference edge from this variable because this instruction needs to be executed

在每个节点退出的地方有一个 liveness information。MOVE 之间会有一条虚线。

## Register Allocation

---

- Two ways to represent edges
  - Adjacent list
    - Only for normal temporaries
    - Because the adjacent list for machine registers may be too large
    - Machine registers cannot be selected
    - We only coalesce temporary node to machine registers using George method
  - Bitmap
    - Both for temporaries and machine registers
- Move-related node
  - associated with a count number of moves
  - easy to maintain and test if it is no longer move-related

2021/11/30

R

bitmap 就是一个  $n \times n$  的矩阵，如果有 1 就代表有一条边。在我们书上实现的时候，邻接表只有 temporary，而 bitmap 既有 temporary，又有 register。节点上会关联到一个 move\_count 如果其不为 0，那它就是一个 move-related node。还需要记录下每个节点的 degree。

```

procedure Main():
 LivenessAnalysis()
 Build()
 MakeWorkList()
 repeat
 if simplifyWorklist != {} then Simplify()
 else if worklistMoves != {} then Coalesce()
 else if freezeWorklist != {} then Freeze()
 else if spillWorklist != {} then SelectSpill()
 until simlifyWorklist = {} && worklistMove = {} && freezeWorklist = {} && spillWorklist = {}
 AssignColor()
 if spilledNodes != {} then
 RewritePrograms(spilledNodes)
 Main()

```

其中

simplifyWorklist: 度数  $< k$  的非 move-related 节点列表。

worklistMoves: 可以做 coalesce 的 move 指令集合。

freezeWorklist: 度数  $< k$  的 move-related 节点列表。

spillWorklist: 度数  $\geq k$  的节点列表。

spillNodes: 被 spill 的节点列表。

这就是一个总体的伪代码。小于  $k$  个节点,要么是可以 simplify 的,要么是可以做 coalesce 的,剩下的就是 $\geq k$  的 degree 的节点,就放到 spillworklist。

# Program Code

---

```
procedure Build()
 forall b ∈ blocks in program
 let live = liveOut(b)
 forall I ∈ instruction(b) in reverse order
 if isMoveInstruction(I) then
 live ← live \ use(I)
 forall n ∈ def(I) ∪ use(I)
 moveList[n] ← moveList[n] ∪ {I}
 workListMoves ← workListMoves ∪ {I}
 live ← live ∪ def(I)
 forall d ∈ def(I)
 forall l ∈ live
 AddEdge(l, d)
 live ← use(I) ∪ (live \ def(I))
```

Basic\_block 中的指令逆序来看，拿到一条指令先看看是不是 move 指令，如果是 move 那么就有一个 use 和有一个 def。所以每个变量要和 live 中的每一个节点画一条边。

# Program Code

---

```
procedure AddEdge(u, v)
 if ((u, v) ∉ adjSet) ∧ (u ≠ v) then
 adjSet ← adjSet ∪ {(u, v), (v, u)}
 if u ∉ precolored then
 adjList[u] ← adjList[u] ∪ {v}
 degree[u] += 1
 if v ∉ precolored then
 adjList[v] ← adjList[v] ∪ {u}
 degree[v] += 1
```

- adjSet is a bit-matrix

我们先要判定这两个节点不一样，并且这条边不存在。

```

procedure MakeWorkList()
 forall n ∈ initial
 initial ← initial \ {n}
 if degree[n] ≥ K then
 spillWorkList ← spillWorkList ∪ {n}
 else if MoveRelated(n) then
 freezeWorkList ← freezeWorkList ∪ {n}
 else
 simplifyWorkList ← simplifyWorkList ∪ {n}

```

Node work-list, sets, stacks

- **initial**
  - Temporary registers, not precolored and not yet processed

我们根据图中的节点的度数以及是否 move-related 来构建这几个 list。

```

function Adjacent(n)
 adjList[n] \ (selectStack ∪ coalescedNodes)
function NodeMoves(n)
 moveList[n] ∩ (activeMoves ∪ worklistMoves)
function MoveRelated(n)
 NodeMoves(n) ≠ {}

```

**procedure** Simplify()

```

let n ∈ simplifyWorkList
 simplifyWorkList ← simplifyWorkList \ {n}
 push(n, selectStack)
 forall m ∈ Adjacent(n)
 DecrementDegree(m)

```

处理过的 move-related 我们就要去掉。

```

procedure Decrement(m)
 let d = degree[m]
 degree[m] \leftarrow d - 1
 if d = K then
 EnableMoves(m \cup Adjcent(m))
 spillWorkList \leftarrow spillWorkList \ {m}
 if MoveRelated(m) then
 freezeWorkList \leftarrow freezeWorkList \cup {n}
 else
 simplifyWorkList \leftarrow simplifyWorkList \cup {n}
procedure EnableMoves(nodes)
 forall n \in nodes
 forall I \in NodeMoves(n)
 if I \in activeMoves then
 activeMoves \leftarrow activeMoves \ {I}
 workListMoves \leftarrow workListMoves \cup {I}

```

2021/11/30

节点的度数减一之后，性质可能发生变化，可能就变成一个好节点了。

```

procedure Coalesce()
 let m(-copy(x,y)) \in workListMoves
 x \leftarrow GetAlias(x)
 y \leftarrow GetAlias(y)
 if y \in precolored then
 let (u,v) = (y, x)
 else
 let (u,v) = (x, y)
 workListMoves \leftarrow workListMoves \ {m}
 if (u = v) then
 coalescedMoves \leftarrow coalescedMoves \cup {m}
 AddWorkList(u)
 else if v \in precolored \vee (u,v) \in adjSet then
 constrainedMoves \leftarrow constrainedMoves \cup {m}
 AddWorkList(u)
 else AddWorkList(v)

```

2021/11/30

## 2021/12/03

我们上节课开始讲了 **Coloring** 的实现，数据结构之前提过了，主要是一个图，图的话是使用邻接表来做，这里面还用了一个 **bitmap**，在实现 **register allocation** 的时候，里面使用了大量的列表，主要是 **list** 和 **set**。**List** 第一个就是把节点给分类了，这个节点分类呢，是按照我们度数的大小来分，分成了 **significant node**（度数  $\geq K$ ）和 **non-significant node**（度数  $< K$ ），度数  $< K$  的节点又分成两类。

**SpilledWorkList:** 里面放置了度数  $\geq k$  的节点，也就代表这些节点不能简单地从图上拿走。因为我们的 **temporary allocation** 算法说的是，如果节点的度数  $< K$ ，那么可以简单地从 **interference graph**（相干图）上拿掉，拿掉以后不影响我们最终着色的结果。

度数  $< K$  的节点又要分成可以直接从图上拿掉的节点和 **move-related node**。对于 **move-related node** 来说，就是一条 **move instruction** 把两个节点联系在一起，比如说  $x \leftarrow y$ ，此时  $x$  和  $y$  都不能从图上拿掉。那么什么时候可以拿掉呢？如果  $x$  和  $y$  可以合并(**coalesce**)，那么我们可以把  $x$  和  $y$  合并成一个节点后，再从图上拿掉；还有一种情况是，它们不能合并，那么我们要把 **move-related** 的虚线重新加回实线，让  $x$  和  $y$  变成相干的，这个就是 **freeze** 操作。

所以我们就把度数  $< K$  且 **non-move-related** 的节点放入 **SimplifyWorkList** 中，而度数  $< K$  的 **move-related** 的节点放入 **FreezeWorkList** 中。那我们整个寄存器分配，上来先做活跃分析，然后构造 **interference graph**。在构造 **interference graph** 的时候，一个是根据变量的活跃信息来构造边，不过这里有一点变化，就是 **move instruction** 导致的相干性我们要通过虚线来连接。

```
procedure Main():
 LivenessAnalysis()
 Build()
 MakeWorkList()
 repeat
 if simplifyWorklist != {} then Simplify()
 else if worklistMoves != {} then Coalesce()
 else if freezeWorklist != {} then Freeze()
 else if spillWorklist != {} then SelectSpill()
 until simlifyWorklist = {} && worklistMove = {} && freezeWorklist = {} && spillWorklist = {}
 AssignColor()
 if spilledNodes != {} then
 RewritePrograms(spilledNodes)
 Main()
```

其中

simplifyWorklist: 度数  $< K$  的非 move-related 节点列表。

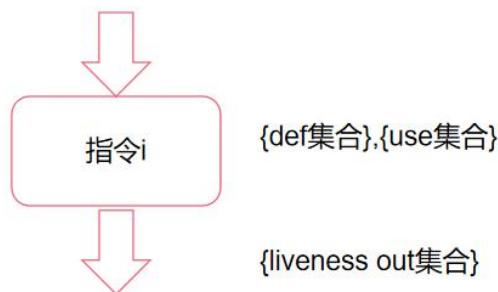
worklistMoves: 可以做 coalesce 的 move 指令集合。

freezeWorklist: 度数  $< K$  的 move-related 节点列表。

spillWorklist: 度数  $\geq K$  的节点列表。

spillNodes: 被 spill 的节点列表。

我们重温一下构造相干图的方法: 对于 instruction  $i$ , 我们的 liveness analysis 是 backward analysis (从程序结束的时候的变量活跃信息来倒退前面的每一条 instruction 的 live-in 和 live-out information)。 $i$  这个 instruction 有 def 集合和 liveness out 集合。



我们要做的就是对  $\{def\} \times (\{out\} / \{use\})$  这个笛卡尔积在相干图中连上边。如上就是 build 操作做的事情。

接下来的 makeworklist 其实就是, 我们已经得到相干图了, 接下来按照节点的度数以及是否是 move-related node 去分类。

```
procedure Build()
 forall b ∈ blocks in program
 let live = liveOut(b)
 forall I ∈ instruction(b) in reverse order
 if isMoveInstruction(I) then
 live ← live \ use(I)
 forall n ∈ def(I) ∪ use(I)
 moveList[n] ← moveList[n] ∪ {I}
 workListMoves ← workListMoves ∪ {I}
 live ← live ∪ def(I)
 forall d ∈ def(I)
 forall l ∈ live
 AddEdge(l,d)
```

$$live \leftarrow use(I) \cup (live \setminus def(I))$$

Build 函数中，在处理 move 指令时，我们需要特殊处理一下，比如说存在如下的两条 move 指令：

$$\begin{aligned} K &: x \leftarrow z \\ I &: x \leftarrow y \end{aligned}$$

那么我们从后向前遍历的时候，就要把指令 I 和指令 K 加入到 moveList(x) 中，表明这是和变量 x 相关的 move 指令。并且要把 move 指令加入到 workListMoves 里去，表明这些 move 指令可以准备做 coalesce 的。如果做完 coalesce，在 freezeWorklist 还存在节点，那么我们就做 freeze，也就是把虚线变成实线。

到目前为止我们的准备工作已经做好了（把节点和指令都分类完毕了），接下来我们进循环正式开始着色流程，注意到每次循环我们只处理一个节点，按照 Simplify, Coalesce, Freeze, SelectSpill 的优先级去做。注意到这里的 SelectSpill 的结果是 potential spill，也就是当我们退栈放回来的时候，是有可能能够着上色的。

```
procedure AddEdge(u, v)
if ((u, v) ∈ adjSet) ∧ (u ≠ v) then
 adjSet ← adjSet ∪ {(u, v), (v, u)}
 if u ∉ precolored then
 adjList[u] ← adjList[u] ∪ {v}
 degree[u] += 1
 if v ∉ precolored then
 adjList[v] ← adjList[v] ∪ {u}
 degree[v] += 1
```

其中 adjSet 是一个元素取值为 0 或 1 的矩阵，如果  $adjSet(i, j) = 1$ ，那么就说明在相干图上 i 和 j 有一条边。而 adjList 其实就是邻接表，也就是每个节点都有一个 head，如果有一条新的边连出去，就加载 head 对应的 list 的末尾。并且，因为我们在 adjList 中不加入 machine register，只加入 temporary register，所以我们在加入 adjList 的时候，需要判定一下 u 和 v 是否是 precolored（也就是 machine register）。注意到 machine(hard) register 的颜色是固定的，是不能从图上被拿掉的。所以我们把 precolored node 都当做 significant node 来看待。

```

procedure MakeWorkList()
 forall n ∈ initial
 initial ← initial \ {n}
 if degree[n] ≥ K then
 spillWorkList ← spillWorkList ∪ {n}
 else if MoveRelated(n) then
 freezeWorkList ← freezeWorkList ∪ {n}
 else
 simplifyWorkList ← simplifyWorkList ∪ {n}

```

其实就是对节点分类。

```

function Adjacent(n)
adjList[n] \ (selectStack ∪ coalescedNodes)

```

Adjacent 这个函数说的是，我们希望找到 n 在图上的邻居是谁，因为我们在一次过程中不再会删除图上的内容，所以我们对于已经进栈的元素要手动剔除掉。还有一个需要考虑的就是合并的元素，比如说  $x \leftarrow y$  这条 move 指令，最终我们把 x 和 y 合并成一个节点了，我们不会说在图上把 x 和 y 删除，然后新增一个{x,y}节点，因为这会让我们重新构建 bitmap 和 adjacent list。所以我们使用 x 来代表 y，而 y 就加入到 coalescedNodes 中，在所有操作中被手动剔除即可。

```

function NodeMoves(n)
moveList[n] ∩ (activeMoves ∪ worklistMoves)

```

这个函数就是询问节点 n, 是否还有 move 相关的指令。因为我们过程中会做一些 freeze, 做了 freeze 之后，某些 move 指令就会被拿掉。activeMoves 就是那些还没有准备好做 coalesce 的 move 指令。

```

function MoveRelated(n)
NodeMoves(n) ≠ {}

```

如果某个节点没有 move 相关的指令了，那么它就是一个可以 simplify 的节点，不能再放在 freezeWorklist 中了。

```

procedure Simplify()
 let n ∈ simplifyWorkList
 simplifyWorkList ← simplifyWorkList \ {n}
 push(n, selectStack)
 forall m ∈ Adjacent(n)
 DecrementDegree(m)

```

因为 `simplifyWorkList` 中都是 non-move-related 的度数  $< K$  的节点，我们可以从中拿出来进栈然后减掉对应的度数。

```

procedure Decrement(m)
 let d = degree[m]
 degree[m] ← d - 1
 if d = K then
 EnableMoves(m ∪ Adjcent(m))
 spillWorkList ← spillWorkList \ {m}
 if MoveRelated(m) then
 freezeWorkList ← freezeWorkList ∪ {n}
 else
 simplifyWorkList ← simplifyWorkList ∪ {n}
procedure EnableMoves(nodes)
 forall n ∈ nodes
 forall l ∈ NodeMoves(n)
 if l ∈ activeMoves then
 activeMoves ← activeMoves \ {l}
 workListMoves ← workListMoves ∪ {l}

```

2021/12/3

度数减一可能会引发一系列的事情，因为如果度数减一之前是度数  $\geq K$  的节点，那么度数减一之后就会变成一个好节点。然后我们根据其是否是 move-related，把 `spillWorkList` 中元素放进 `freezeWorkList` 或者 `simplifyWorkList`，注意到成为一个 non significant node 之后，此节点及其邻居如果存在 move-related 关系，可能可以再次做 coalesce，因为无论是 Briggs 算法还是 George 算法都是会根据邻居的节点度数  $\geq K$  类似的条件来判断的，所以我们要把这些节点从 `activeMoves` 移动回 `workListMoves`。

```

procedure Coalesce()

let m(x, y) ∈ workListMoves

 x ← GetAlias(x)

```

```

 $y \leftarrow GetAlias(y)$

if $y \in precolored$ then //保证第一个元素是 precolored register (如果有的话)

 let $(u,v) = (y,x)$
 else
 let $(u,v) = (x,y)$

 $workListMoves \leftarrow workListMoves \setminus \{m\}$

 if $(u=v)$ then //u 和 v 已经被合并到一个寄存器里的情况

 $coalescedMoves \leftarrow coalescedMoves \setminus \{m\}$

 AddWorkList(u)

 else if $v \in precolored \vee (u,v) \in adjSet$ then //u 和 v 有一条边 (实线) 又有虚线

 的情况, 我们就 freeze, 保留实线, 如果 v 是 precolored, 根据前面的, u 也是 precolored, 显然相干; 或者就是 u, v 因为某种意义存在了实线的情况

 $constrainedMoves \leftarrow constrainedMoves \cup \{m\}$

 AddWorkList(u)
 AddWorkList(v)

 else if $u \in precolored \wedge (\forall t \in Adjacent(v), OK(t,u)) \vee$
 $u \notin precolored \wedge Conservative(Adjacent(v) \cup Adjacent(u))$ then

//第一种情况: u 是 precolored, 使用 George 算法, 对于 v 的度数 $\geq K$ 的邻居, George 条件要求必须也是 u 的邻居。
//第二种情况: u 不是 precolored, 使用 Briggs 算法, 我们考察合并完的节点的度数 $\geq K$ 的邻居的个数是不是小于 K, 如果确实小于 K, 那么就满足 Briggs 条件。

 $coalescedMoves \leftarrow coalescedMoves \cup \{m\}$

 Combine(u,v)
 AddWorkList(u)

else

 $activeMoves \leftarrow activeMoves \cup \{m\}$

```

**constrainedMoves:** src 和 dst 相干的 move 指令。

**coalescedMoves:** 被合并的 move 指令

我们从  $workListMoves$  中拿出一条指令, 它一定是  $x \leftarrow y$  的形式, 但是 x 和 y 此时可能已经被合并了, 那么我们需要找到代表 x 和代表 y 的节点, 就是通过如下的这个  $GetAlias$  函数:

```

function GetAlias(n)
 if $n \in coalescedNodes$ then

```

```
GetAlias(alias[n])
```

```
else n
```

我们是要递归地从 alias 中得到某个被合并节点的代表节点的编号。当 move(u,v) 被合并的时候，我们就令 alias[v]=u。

到最后，如果既不满足 Briggs 条件也不满足 George 条件，我们就把这条指令放进 activeMoves (当前 setting 已经被考虑过不能做 coalesce 的情况)。只有当我们合并的节点以及其邻居的状态发生了变化的时候 (如：度数  $\geq K$  变为了度数  $< K$ )，那么我们再把对应的指令从 activeMoves 移动回 workListMoves，再考虑新的 setting 能不能做 coalesce。只有这样，我们的算法才能够终止，如果我们没有 activeMoves 这个集合的逻辑，那么我们循环每次都是 coalesce 失败，并且导致算法陷入死循环。

```
procedure addWorkList(u)
```

```
if $u \notin \text{precolored} \wedge \neg (\text{MoveRelated}(u)) \wedge \text{degree}[u] < K$ then
```

```
freezeWorkList $\leftarrow \text{freezeWorkList} \setminus \{u\}$
```

```
simplifyWorkList $\leftarrow \text{simplifyWorkList} \cup \{u\}$
```

在合并时， $u$  还是在 freezeWorkList 中的，如果它是 precolored，那么我们不去管它；如果它还存在 move-related 关系，那么它还是在 freezeWorkList 中；如果它既不是 precolored，也没有 move-related 关系，并且度数  $< K$ ，那么合并完的这个节点就是一个普通节点，加入到 simplifyWorkList 即可。

```
function OK(t, r)
```

```
return $\text{degree}[t] < K \vee t \in \text{precolored} \vee (t, r) \in \text{adjSet}$
```

George 算法的辅助函数：也就是合并  $r$  和  $v$  的时候，对  $v$  的邻居  $t$  考察，要么其度数  $< K$ ，要么度数  $\geq K$  且同时也是  $r$  的邻居。当然  $t$  如果是物理寄存器，那么  $r$  在我们的算法中也是物理寄存器，它俩显然也是有一条边的。

```
function Conservertive(nodes)
```

```
let k = 0
```

```
forall $n \in \text{nodes}$
```

```
if $\text{degree}[n] \geq K$ then
```

```
 $k \leftarrow k + 1$
```

```
return (k < K)
```

Briggs 条件判断，也就是合并完的节点的邻居集合中，度数  $\geq K$  的邻居的个数是否小于  $K$ 。

```

procedure Combine(u,v)
 if $v \in \text{freezeWorkList}$ then
 $\text{freezeWorkList} \leftarrow \text{freezeWorkList} \setminus \{v\}$
 else
 $\text{spillWorkList} \leftarrow \text{spillWorkList} \setminus \{v\}$
 $\text{coalescedNodes} \leftarrow \text{coalescedNodes} \cup \{v\}$
 $\text{alias}[v] \leftarrow u$

```

```

procedure Combine(u,v)
 if $v \in \text{freezeWorkList}$ then
 $\text{freezeWorkList} \leftarrow \text{freezeWorkList} \setminus \{v\}$
 else
 $\text{spillWorkList} \leftarrow \text{spillWorkList} \setminus \{v\}$
 $\text{coalescedNodes} \leftarrow \text{coalescedNodes} \cup \{v\}$
 $\text{alias}[v] \leftarrow u$
 $\text{moveList}[u] \leftarrow \text{moveList}[u] \cup \text{moveList}[v]$
 EnableMoves(v)
 forall t $\in \text{Adjacent}(v)$
 addEdge(t, u)
 DecrementDegree(t)
 if $\text{degree}[u] \geq K \wedge u \in \text{freezeWorkList}$ then
 $\text{freezeWorkList} \leftarrow \text{freezeWorkList} \setminus \{u\}$
 $\text{spillWorkList} \leftarrow \text{spillWorkList} \cup \{u\}$

```

我们要把  $v$  合并到  $u$  里去，显然  $v$  只可能存在于  $\text{freezeWorkList}$  或者  $\text{spillWorkList}$  中，我们从中把  $v$  去掉加到  $\text{coalescedNodes}$  集合中去。因为使用  $u$  来代表  $v$ ，我们需要设置  $\text{alias}[v]=u$ ，并且  $u$  的  $\text{moveList}$  需要继承  $v$  的  $\text{moveList}$ 。因为此时  $v$  的状态发生了变化，之前不能做的 move 指令的 coalesce 可能可以做了，所以我们还需要  $\text{EnableMoves}(v)$ 。最后，我们需要把  $v$  的邻居全部连到  $u$  上，并且把  $u$  根据其度数放到对应的集合中去。

```

procedure Freeze()

```

```

let $u \in freezeWorkList$

 $freezeWorkList \leftarrow freezeWorkList \setminus \{u\}$

 $simplifyWorkList \leftarrow simplifyWorkList \cup \{u\}$

FreezeMoves(u)

```

当 `simplify` 和 `coalesce` 都不能做了以后，那么我们就要把 move-related 的 `freezeWorkList` 中的节点真的 `freeze` 掉。

```

procedure FreezeMoves(u)
 forall $m(x, y) \in NodeMoves(u)$
 if GetAlias(y)=GetAlias(u) then
 $v \leftarrow GetAlias(x)$
 else
 $v \leftarrow GetAlias(y)$
 $activeMoves \leftarrow activeMoves \setminus \{m\}$
 $frozenMoves \leftarrow frozenMoves \cup \{m\}$
 if $NodeMoves(v) = \{\} \wedge \text{degree}[v] < K$ then
 $freezeWorkList \leftarrow freezeWorkList \setminus \{v\}$
 $simplifyWorkList \leftarrow simplifyWorkList \cup \{v\}$

```

```

procedure SelectSpill()
 let $u \in spillWorkList$ /* Should use heuristic algorithms */
 $spillWorkList \leftarrow spillWorkList \setminus \{m\}$
 $simplifyWorkList \leftarrow simplifyWorkList \cup \{m\}$
 FreezeMoves(m)

```

如果都干不了了，我们只能强行选一个放到 `simplifyWorkList` 中，并且去掉 move-related 关系。

```

procedure AssignColor()
 while SelectStack not empty
 let n = pop(SelectStack)
 okColors $\leftarrow [0, \dots, K-1]$
 forall w \in adjList[n]
 if GetAlias[w] \in (ColoredNodes \cup precolored) then
 okColors \leftarrow okColors \setminus {color[GetAlias(w)]}
 if okColor = {} then
 spillWorkList \leftarrow spillWorkList \cup {n}
 else
 coloredNodes \leftarrow coloredNodes \cup {n}
 let c \in okColor
 color[n] \leftarrow c

```

最终我们一个个退栈。当前退栈退出  $n$ , 我们就来看  $n$  的邻居的颜色。我们把对应的颜色从  $okColor$  中去掉, 如果不剩颜色了, 那么这个  $n$  就不能被着色; 否则就随便选一个颜色着色。

**procedure** RewriteProgram()

```

Allocate memory locations for each $v \in$ spilledNodes
Create a new temporary v_i for each definition and each use
In the program (instructions), insert a store after each
definition of a v_i , a fetch before each use of a v_i
Put All the v_i into a set newTemps
spilledNode $\leftarrow \{\}$
initial \leftarrow coloredNodes \cup coalescedNodes \cup newTemp
coloredNodes $\leftarrow \{\}$
coalescedNodes $\leftarrow \{\}$

```

真的要被 spill 的, 我们就重新写代码, 把对应的位置做 load and store, 当然这里也可以重新做 coalesce。

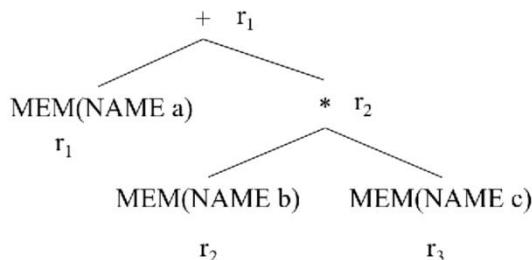
## 在 Tree 上做寄存器分配

下面呢, 我们的书上还提了一点, 就是我们能不能直接在 tree 上做 register allocation。

## Register Allocation for Trees

```
procedure SimpleAlloc(t)
 for each nontrivial tile u that is a child of t
 SimpleAlloc(u)
 for each nontrivial tile u that is a child of t
 n ← n - 1
 n ← n + 1
 assign r_n to hold the value at the root of t
• Assume we have enough number of registers
• Still we want to minimize the number of registers to be used
• The above code is only sub-optimal
 - n is a global initialize as 0
```

在 tree 上不太好做 spill，但是它可以计算某一棵树需要几个寄存器。这时候我们就有 一个全局变量 n，初始化为 0.



我们从根这个地方开始做 tiling，就会分出一些子节点。每个子节点我们递归地算一下需要 几个寄存器。因为在根节点计算的时候，子树的寄存器的值不会被别人改变，所以我们可以 复用这个寄存器。

# Sethi-Ullman Labeling Algorithm

---

```
procedure Label(t)
 for each tile u that is a child of t
 Label(u)
 if t is trivial then need[t] ← 0
 else if t has two children, u_{left} and u_{right} then
 if $need[u_{left}] = need[u_{right}]$ then
 $need[t] \leftarrow 1 + need[u_{left}]$
 else $need[t] \leftarrow \max(1, need[u_{left}], need[u_{right}])$
 else if t has one child, u then
 $need[t] \leftarrow \max(1, need[u])$
 else if t has no children then
 $need[t] \leftarrow 1$
```

所以我们先去做一个 label，先扫描一遍看看每个子树要多少个寄存器。然后我们真的分寄存器的时候，我们看看左右要多少个寄存器。

**2021/12/07**

到寄存器分配基本上就是一个完整的编译器了。

虽然我们经常认为垃圾回收是在 java 这种比较安全的语言中去做的，但是在 C 和 C++ 中也有实现垃圾回收的项目。总的来说，垃圾回收的前提是我们有一个堆，在 tiger 中用到堆的情况就是我们的数组分配还有我们的 record 分配。所谓的 dynamic allocation，栈上的分配我们认为是比较 static 的，因为栈上分配的生命周期通常是和我们的 frame 的生命周期一样的。而堆上分配的变量的生命周期是相对独立于执行流的，frame 的退出可能会导致堆上的变量的一些引用没有了，堆上的不再被引用的变量（not reachable）其实就是垃圾。

假设一个 c 的代码只做 malloc 不做 free，我们不断 push 和 pop 新的 frame，这样很多对堆上的引用就没有了。在 c 和 c++ 里面，我们认为这是一个 memory leak，因为这块内存就不能再被使用到了，导致程序最终 out of memory。在有运行时的一些语言，比如 java/Go/C#，提供了垃圾回收机制，其实就是找到垃圾对象然后把堆的情况释放出来。

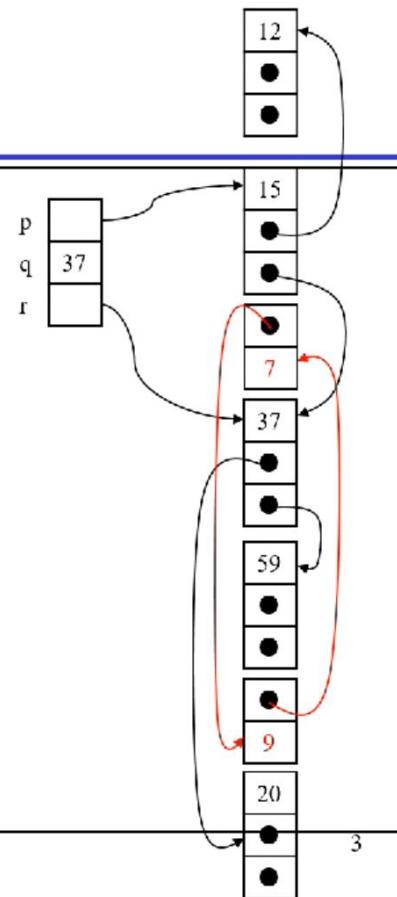
对控制流的静态分析是可以规约到停机问题的，我们不能很清楚地知道每个对象的 life cycle，所以对垃圾对象的判断是通过 reachable 来判断的。

比如我们栈上有一个 a，引用了堆上的 b。比如说我们用完 a 之后再也不使用 a 了，此时 b 其实已经是个垃圾了。但是在静态分析的时候限制比较多，比如说有一些分支用到了 b 而有些分支没有用到 b。这样子就不太好判断。所以垃圾回收机制主要还是使用可达性，也就是只要有引用指向 b，我们就认为是活着的。如果此时 a 被另一个指向堆上的变量 d 的 c 覆盖了，那么根据可达性我们就知道没有人再指向 b 了。

## Example

```
let
 type list = {link: list, key: int}
 type tree = {key: int, left: tree, right: tree}
 function maketree() =
 function showtree() =
in
 let var x := list{link=nil, key=7}
 var y := list{link=x, key=9}
 in x.link := y
 end
 lct var p := maketree()
 var r := p.right
 var q := r.key
 in garbage collection here
 showtree(r)
 end
end
```

2021/12/7



这里有一个例子。注意我们暂时认为垃圾回收的点是一些程序中的定点，我们先不考虑怎么触发垃圾回收机制。这个时候 `x` 和 `y` 已经没有用了。在这里活跃的变量只有 `pqr` 三个变量，而红色标注出来的其实就是我们的 `x` 和 `y`，此时已经没有指针指向他们了。

后面我们来讲一些具体的垃圾回收算法。

## Mark and Sweep

### Mark and Sweep

- **Directed graph**
  - nodes: program variables and heap-allocated records
  - edges: pointers
- **Mark**
  - Search the graph from the roots (program variables)
  - Mark all the nodes searched
- **Sweep**
  - Sweep the entire heap
  - Reclaim nodes not marked (put them into a freelist)
  - Unmark marked nodes
- **Program resumes execution after GC**
  - Allocates records from the freelist
  - Do GC again when freelist is empty

通常我们会建模成为 graph，这个图里的节点要么是程序变量，要么是堆上的 record，而边就是引用关系。Mark and sweep 分为两步。Mark 就是图遍历的过程，把从 root（栈上的程序变量）开始遍历做可达性分析。而 sweep 就是把堆扫描一遍，因为我们已经标记过对象了，我们认为没有被标记的对象就是垃圾对象，我们回收掉这些对象，并且被标记过的对象回重新取消标记，以便下一次 mark 操作。

我们还有一个 freelist 作为空闲列表，我们每次要分配一个新的 record 的时候就会从空闲列表中分为空间，如果空闲列表为空，就会触发垃圾回收。

```
function DFS(x)
 if x is a pointer and points to record y:
 if record y is not marked:
 mark y
 for each field f_i of record y:
 DFS($y.f_i$)
```

Mark 阶段:

for each root v:  
    DFS(v)

Sweep 阶段:

$p \leftarrow$  last address in heap  
while  $p <$  last address in heap:

```

If record p is marked:
 unmark p
else:
 let f_i be the first field in p
 $p.f_1 \leftarrow \text{freelist}$
 $\text{freelist} \leftarrow p$
 $p \leftarrow p + (\text{size of record } p)$

```

就是一个 DFS 的图遍历。在 sweep 的时候，如果一个 record  $p$  已经是垃圾了，我们可以把 freelist 的指针放在  $p$  的第一个 field 中。这样我们就不用专门生成一个 freelist 来存储信息了。

## Mark And Sweep 的开销

### Cost of garbage collection

- Time of GC
  - Mark: Time proportional to the amount of reachable data  $R$
  - Sweep: Proportional to the size of the heap  $H$
  - Total time:  $c_1R + c_2H$
- GC replenish the freelist with  $H-R$  words
- Amortized cost  $\frac{c_1R+c_2H}{H-R}$ 
  - If  $R$  is close to  $H$ , the cost is very higher
  - If  $R/H$  is larger than 0.5, the collector should increase  $H$

首先，mark 的执行时间和活着的 record 的数量相关。而 sweep 是全堆扫描，所以 sweep 时间是和堆的大小有关的。

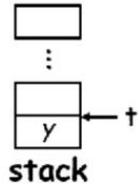
GC 的效果就是我们把 freelist 重新填满了，可以填充的数量是  $H-R$  的大小。如果  $R$  和  $H$  差不多大，那么就是堆上活跃的 record 基本上快填满堆了，所以我们的总的 cost 很大但是并没有回收掉很多内容。此时可能我们不应该做垃圾回收，而是扩展这个堆，或者抛一些 out-of-memory 的异常。

如果  $R/H$  大于 0.5，其实垃圾回收的效用就比较差了，也是建议让回收器去增加堆的大小。

## Explicit Stack

面临的问题：DFS 会创建太多的栈帧。

我们先讲 DFS 的问题，它是通过递归函数实现的，每多一层的访问都会创建一个新的 stack frame，这样开销是比较高的。举个极端的例子来说，我们有一个长度为一百万的链表，我们就要一直 DFS 下去，这样我们生成的 frame 就会有很多。当我们做图遍历的时候，如果我们结构整体是一条链，那么我们的节点的遍历深度就会很深。



解决的办法就是不要使用原先的栈帧，而是使用自己实现的 `vector` 替代原先的栈。函数逻辑如下：

```
function DFS(x)
 if x is a pointer to record y which is not marked
 mark y
 t ← 1
 stack[t] ← y //把深搜的起点加入

 while t > 0:
 y ← stack[t]; t ← t - 1 //取出栈顶元素

 for each field fi of record y
 if y.fi is a pointer to record z which is not marked:
 //指向了一个没有标记过的 record
 mark z
 t ← t + 1; y ← stack[t] //加入栈中
```

因为我们并不知道栈的深度可能是多少。我们把栈顶的 field 拿出来遍历一遍。这个和原先的算法没有什么差别，但是使用了 `explicit` 的栈来代替了 `c++` 系统的栈帧。现在我们每一个 `record` 只需要一个 `entry` 来保存它的 `port`，就可以达到节省内存的目的。还有，我们给栈分配的内存不会很多，通常只是 `MB` 级别的。而当我们使用 `vector` 的时候，就没有大小的限制了。

注意到这部分并不是什么编译原理限定的内容，这在 OI/ACM 的一些特定场景下，是一个很常见的优化方式。

## Pointer Reversal

后面还有一个 pointer reversal 的优化。这个方法的本质就是说，对于每个对象来说，一旦访问了这个 field 并且把这个 field 放到了栈上，后面是通过栈来访问这个 reference 的。

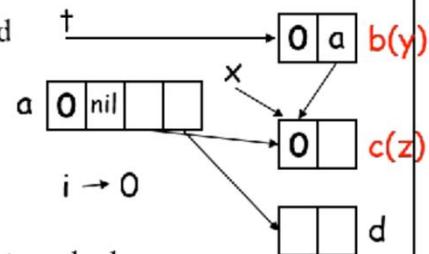
比如原本是  $x$  指向  $y$ ，接下来我们会把  $x$  的指针放到栈上，接下来我们就直接通过栈上的引用访问  $y$  的值。这样子  $x$  就可以用来存放一系列别的东西。

结论：在 pointer reversal 之后，我们就没有栈了，使用了一系列 record 的指针来把栈的信息嵌在里面。这个有点像书上介绍 symbol table 的一些反向指针。

$\text{done}[y]$  是表示  $y$  有没有做完，也可以作为 mark 过程来看。它的作用就是来判断  $y$  里面的三个 field 有没有处理完。当我们处理完了以后  $\text{done}[y]=3$ .

## Pointer reversal

```
function DFS(x)
 if x is a pointer to record y which is not marked
 t ← nil; mark y; done[y] ← 0
 while true
 i ← done[y]
 if i < # of fields in record y
 x ← y.fi
 if x is a pointer to record z which is not marked
 y.fi ← t; t ← y; y ← z; mark y; done[y] ← 0
 else
 done[y] ← i + 1
 else
 z ← y; y ← t
 if y = nil then return
 i ← done[y]; t ← y.fi; y.fi ← z; done[y] ← i+1
```



这个方法可以节省内存开销，本来我们还需要在内存中额外一个栈去存，现在我们只需要把指针都存在 field 里面就可以了。现在的垃圾回收算法通常不会使用 field 来存，比较麻烦，并且可能多线程不好处理，并且栈的占用可能并不是很大，没必要这样做。

## Freelist

方法就是类似 malloc，把 list 分成好几种，比如小于 128B，小于 512B，小于 4K 等，分几个类，当我们想 malloc 1K 的时候，就可以从 512~4K 去找，我们的目标就是减少 freelist 的时间，避免扫描全部的 freelist。

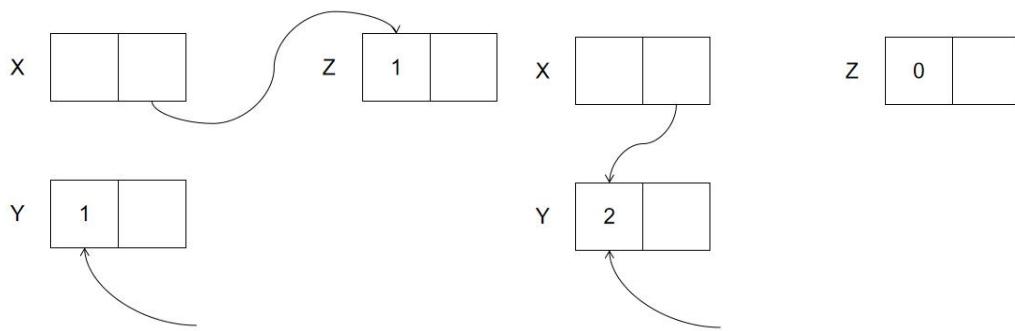
总结：算法由 mark 和 sweep 组成，mark 时间和活着的 record 的数量相关。而 sweep

是全堆扫描，所以 sweep 时间是和堆的大小有关的。值得注意的是用 DFS 完成的，后续有一些算法会使用 BFS。

## Reference Count

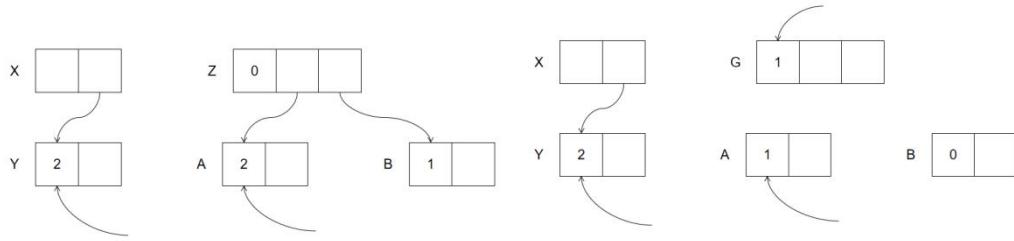
第二个算法是一个理论很完美，但是操作起来很多问题的算法。思路就是对每个 reference 记录一个引用，比如在文件系统中多个进程引用了一个文件。我们可以通过 reference counting 记录到底有几个进程引用了这个文件，直到其为 0 就说明没有人引用了，我们就可以删掉了。还有一个例子就是 virtual memory 引用了物理页的情况，当引用没了以后就可以释放这个物理页了。所以在垃圾回收中可以使用类似的方法，我们给每一个 record 加一个 field 叫做 counter，也就是有多少个引用了我。

- Reference count
  - An extra field in a record to keep how many pointers point to the record
- y is a record
  - Y is stored into x.f<sub>i</sub>, reference count of y is increased
  - The record z originally stored in x.f<sub>i</sub> is decreased
  - If the reference count of z is 0, z is put into the freelist
  - Defer reducing the reference count of z.f<sub>i</sub> when z is removed from the freelist (allocator)
    - Avoid batched, recursive decrement operations



上左图为原先的情况，右图为  $X.f_1 \leftarrow Y$  的情况，引用从 Z 改变为 Y 了以后，原引用-1，新引用+1。如果引用的 counter 为 0 了，那么我们就加到 freelist 中，所以在上例中，对象 Z 需要被回收放到 freelist 中。

## Defer reducing the reference count



在上例中，Z 被加入到 freelist 以后，我们暂时不修改它的 field 引用：A 和 B 的 reference count，而是当 Z 被 freelist 拿出来赋值为新的 G 的时候，我们才去修改并检测 A 和 B 是否需要回收。

为什么要做出这个 defer 的实现呢？因为在我们之前做 mark 和 swap 的时候，我们会把全堆做扫描，而 reference count 只需要看对象身上的元数据就知道它的死活。所以这个算法可以比较好的实现对时延要求比较高的垃圾回收算法。如果是一个很复杂的树对象死了，根节点一死可能所有节点都变成了 0 了，如果不 defer 的话，可能回收要占据很长时间。所以 defer 的优化，每次只回收一层，把过程拆分的很细，这样对于应用的影响比较小，分配和回收都很快地完成。

算法的优点：这个算法设计简单、实现简单、可以很快回收。

算法的缺点：我们不会把事实上已经死的对象很快地回收掉，这被称为浮动垃圾，可能会造成内存的占用量高一些。

## RC 的性能开销高

算法问题 1：性能开销高，每次我们对引用的写操作就要多写很多指令。如果是多线程去做的话，+1 和 -1 还需要用原子指令保护起来。

比如我们想赋值  $x.f_i \leftarrow p$

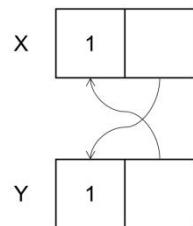
```
z ← x.fi
c ← z.count
c ← c - 1
z.count ← c
if c == 0:
 call putOnFreelist
x.fi ← p
c ← p.count
c ← c + 1
p.count ← c
```

场景 2: Swift 语言用的是 reference count 作为垃圾回收, 这个开销可能在极端情况下有 40%~50% 的应用占用。

## RC 不能解决相互引用的问题

第二个问题就是解决 cycle 问题, 这是 reference count 的死穴。也是因为这个原因导致不能应用在大型系统中。

最极端的例子就是两个对象相互引用的情况, 如下图所示:



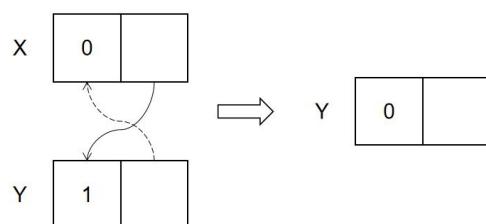
reference count 算法那就不能回收这种对象。

解决方法分为两种:

解决方法 1: Swift 认为这个需要靠程序员去打破这种 cycle, 可以通过 weak reference 的机制或者通过软件工程的模块化去避免相互引用。

提供了 weak reference: 也就是在内存压力大的时候, 我们可以忽略一些引用(弱引用)。

也就是在内存压力大的时候, 弱引用的虚线就无效了, 也就是 x 的 reference count 会变成 0, 然后 x 被回收, 然后 y 的 reference count 也变成 0, 这样 x 和 y 都可以被回收了。



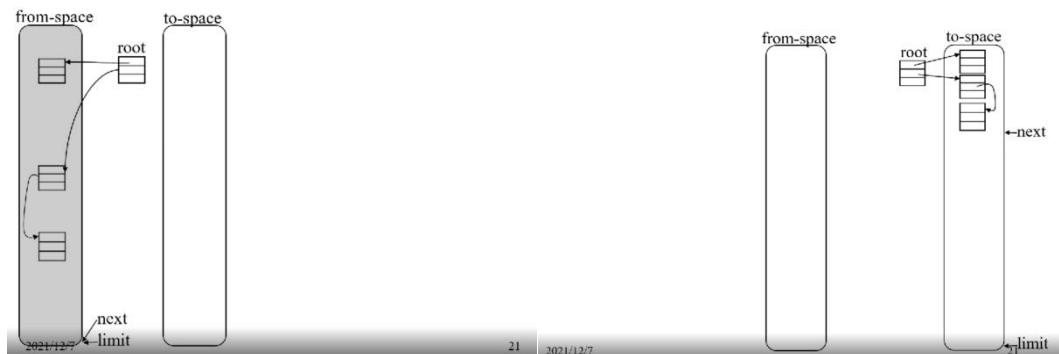
这个比较适合用在 read-only cache 里面, 因为 cache 拿到以后是为了加速我们本地的访问, 如果能 hit 就用本地的, 如果不能 hit, 我们就从网上拉这个数据, 所以内存压力大的时候可以回收内存中的 cache, 无非是再从网上拉一份这个数据。

解决方法 2: 一段时间内如果 reference count 回收的不够多, 再调用 mark and sweep 算法作为 backup。

## Copy 算法

copy 算法和 mark sweep 有点相似, 它也是基于标记的, 从 root 出发它也会找到 3 个活着的

对象。



它一定有一个 space 是空的用来接收我们的对象，from-space 和 to-space 是可以相互转化的。copy 以后我们对象就更加紧凑了。

这个算法的核心好处就是分配快，所有的数据都被 copy 到顶端去了，后面的都是 freespace，所以我们只需要一个 next 指针就划分出了已经使用的区域和 free 的区域。

- **Split the heap into two parts**
  - from-space, to-space
- **Without sweep**
  - Copy the reachable nodes from from-space to to-space with a compact way
  - All the reachable nodes are around the beginning of to-space
  - Rest part of the to-space are freed
  - When next reaches the limit
    - Switch the roles of from-space and to-space

---

所以分配的事情如果大家考虑的比较多，就可以考虑使用这个算法。

## Cheney's Algorithm

这边给了一个 Cheney's Algorithm，它提出了一个基于 BFS 的 copy 算法，核心就是这个 forward 算法。

```
function Forward(p)
 if p is a pointer to from-space then
 if p.f1 points to to-space then
 //如果已经被别人 copy 走了，我们不需要对同一个对象再 copy 一次
 return p.f1
 else
```

```

for each field f_i of (*p)

 $next.f_i \leftarrow p.f_i$ //注意此时 next.fi 里的指针还是指向 from-space 的

 $p.f_1 \leftarrow next$ //from 的第一个 field 指向 to-space

 $next \leftarrow next + size\ of\ record\ p$

 return $p.f_i$

else

 return p

function Main()

 $scan \leftarrow next \leftarrow beginning\ of\ to-space$

 for each root r

 $r \leftarrow Forward(r)$

 while scan < next

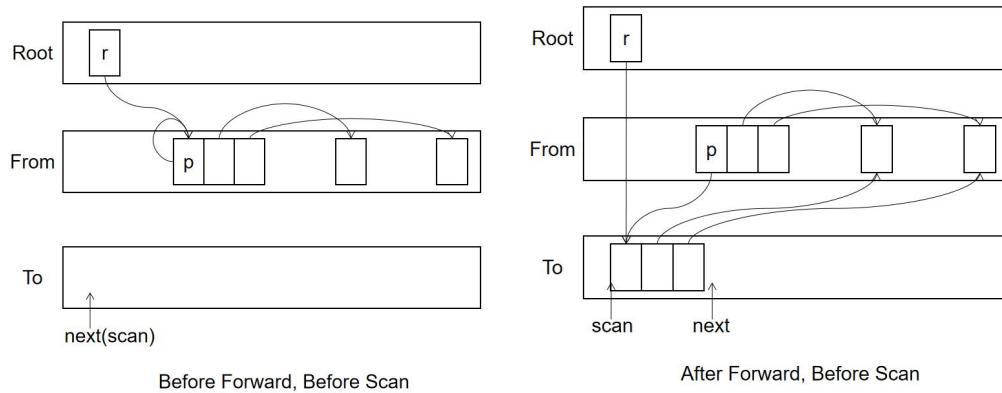
 for each field f_i at scan

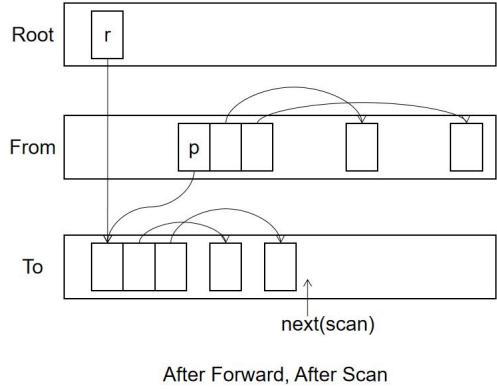
 $scan.f_i \leftarrow Forward(scan.f_i)$

 $scan \leftarrow scan + size\ of\ record\ at\ scan$

```

在这个算法中每个 record 的第一个 field 就是垃圾回收专用的，一定会指向 copy 后的位置，它保证了不会把一个对象 copy 到两个地方的地方。





因为 Forward 只是把根直接连接的一些 record 放到了 To-Space 中，而引用的 field 的指针还是 From-space 中的指针，所以 Scan 就是把剩余的引用到的所有对象放到 To-space 中。当  $scan=next$  之后，算法就停止了，因为我们把所有可达的对象全部复制到了 To-space 中。

所以我们其实是一层层 field copy 的，所以是按照广度优先搜索的思路去做的。广搜的好处就是把 to-space 天然地用成了队列。

坏处就是局部性比较差，比如一棵树的 record，会破坏子树的局部性导致性能下降。

## 2021/12/10

我们继续讲 Cheney 算法，它是一个基于 BFS 的 copy 算法，对象进来了以后，它是一层一层 copy 的。如果我们有一棵树，它是按照树的层次遍历顺序 copy 的，这会导致子树的 locality 并不好，所以 BFS 会导致局部性下降。

所以就有人提出可以优化这个算法。

```

function Forward(p)
 if p points to from-space then
 if p.f1 points to to-space then
 return p.f1
 else
 chase(p)
 return p.f1
 else
 return p

function Chase(p)
 while (p != nil)
 r ← nil
 for each field fi of record p
 if fi points to to-space then
 r ← r + fi
 else if fi points to from-space then
 r ← r + Forward(fi)
 else
 r ← r + fi
 if r != nil then
 p.fi ← r

```

```
 $next.f_i \leftarrow p.f_i$
```

**if**  $next.f_i$  points to from-space and  $next.f_i.f_1$  does not points to to-space **then**

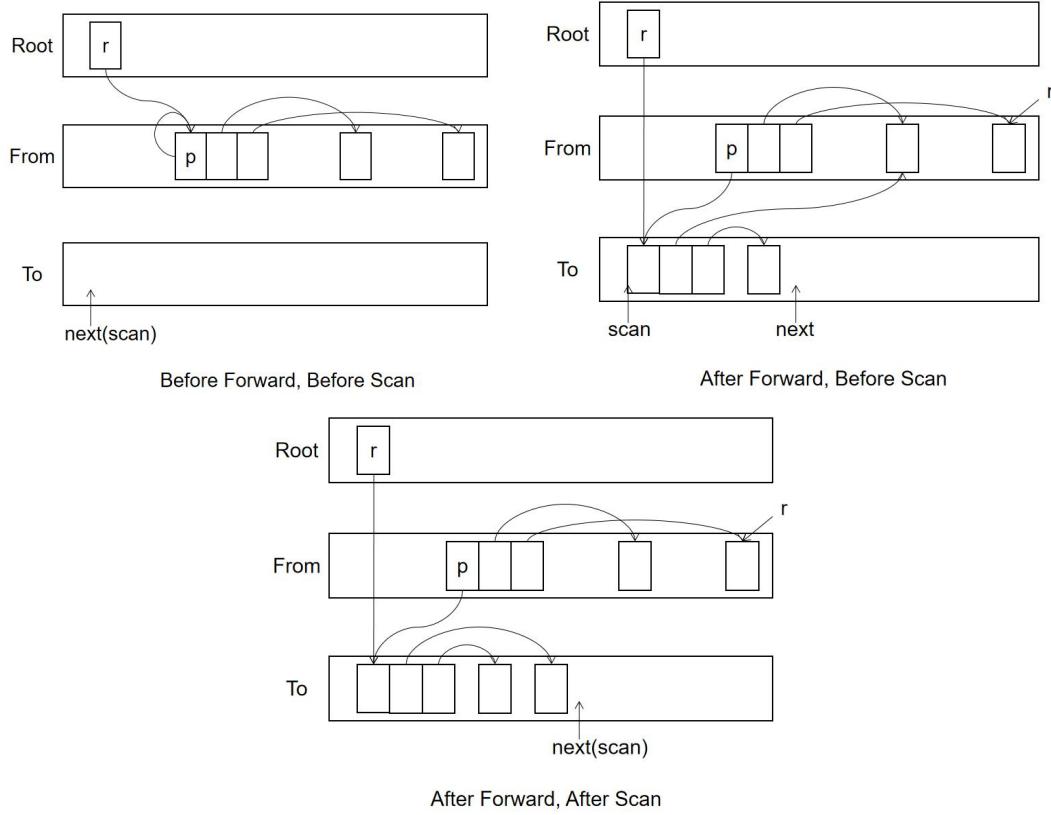
```
 $r \leftarrow next.f_i$
```

```
 $p.f_1 \leftarrow next$
```

```
 $p \leftarrow r$
```

```
 $next \leftarrow next + size\ of\ record\ p$
```

所以就有一个半深搜算法，没有改变本质，但是加了一个 Chase 函数。把  $p$  copy 完以后，我们不马上返回，我们去看  $p$  的 record。



我们发现  $r$  没有一个 forwarding pointer 指向 to-space，也就是  $r$  还没有被 copy。此时我们把  $r$  记录下来。所以  $r$  记录的就是 record 里最后一个没有被 copy 的 reference。

所以半深搜会把 record 中的一个 field 一路最终下去 copy，局部性会比原先的 BFS 好一点。

## Copy Collection 的开销

### Overhead for Copying Collection

---

- The overhead is directly related to live record size ( $R$ ) instead of the whole heap size ( $H$ )
  - Say  $c_3R$  ( $c_3$  is approximately equal to 10)
- The amortized cost:  $\frac{c_3R}{\frac{H}{2} - R}$ 
  - When  $H = 4R$ , the cost is  $c_3$
  - Can we further reduce the cost? - Generational!

因为我们 copy 只 copy 活对象，所以总开销和活着的 record size 相关。书上认为  $c3=10$ 。在我们使用的时候，我们永远只能使用一半的空间（另一半的空间要作为 copy 的目的地），所以一半的空间被浪费掉了。进一步降低 cost 需要分代 GC。

## Generational Garbage Collection

### Generational Collection

---

- Observations
  - Newly created objects are likely to die soon
  - Object will probably survive for more collections if still reachable after many collections
- Collector concentrates on "young data"
- The heap is divided into generations  $G_0, G_1, G_2, \dots$
- The collector just collects  $G_0$ 
  - Roots are not just program variables
  - But also pointers in  $G_1, G_2, \dots$  that points to  $G_0$

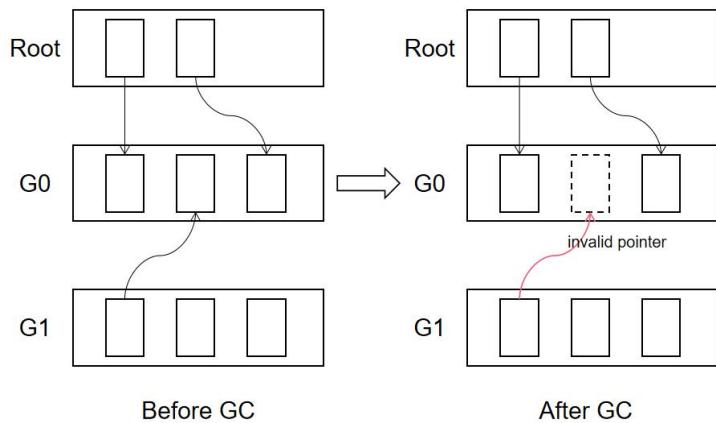
即大多数新创建的数据是很快会死的，比如面向对象语言中会在函数中分配很多对象，

然后在函数结束就立马消亡的，而存活很久的对象，如网站的一些全局配置对象，有时候应用就会查询全局配置，这些就是一直存活的对象。

所以对象和对象之间的寿命是不一样的，所以这个算法本身就是把这些对象分配，它比较关心年轻的对象。回到之前的 **copy GC** 中，如果我们对那些长寿的对象反复地在 **from-space** 和 **to-space** 来回 **copy**，就非常浪费性能。

所以算法分代的意思就是把堆进行分区，一个比较 **general** 的想法就是把堆分成很多代，年龄从小到大依次是  $G_0, G_1, G_2, \dots$ 。主流实现就是年轻代  $G_0$  和老年代  $G_1$ 。分代 GC 大部分时间只回收  $G_0$ ，也就是最年轻的代，它也会负责对象的分配。

这里有一个问题就是，我们考察如下一个场景：



倘若我们只通过 **Root** 的引用来判断  $G_0$ ，那么我们会错误地把  $G_1$  引用到的变量回收掉，导致  $G_1$  中的对象的指针非法。所以当我们回收的时候，我们的出发点是  $Root, G_1, G_2, \dots$  中的所有 record。

分代 GC 只是一个架构，并不限定具体使用什么算法，我们也可以在  $G_0$  内部分成 **from-space** 和 **to-space**，只是说过一段时间我们可以把年龄足够大的 **copy** 到  $G_1$  中。

## Generational Collection

---

- It is rare that pointers in  $G_1, G_2, \dots$  point to  $G_0$ 
  - An object  $a$  is created, its fields are created immediately which point to objects  $b$  and  $c$  which are already created
  - An object  $a$  in old generation whose field points to young generation may be the field is updated by latter which is a rare case
- To avoid all of  $G_1, G_2, \dots$  for roots of  $G_0$ 
  - Compilers must let the program remember where there are pointers from old objects to new ones

不过根据统计来说，从老年代往年轻代的引用这种情况下相对是比较小的，因为在创建对象的时候，正常情况下它的 field 会指向更老的情况。而在对象创建后使用一个年轻代的对象更新 field 的情况相对比较少。

为了避免对老年代做变量扫描，我们需要记录下来老年代指向  $G_0$  的 reference，所以编译器会要记录下来这种情况。

## Ways of Remembering

---

- Remembered List
  - After each update store of the form  $b.f \leftarrow a$
  - Put  $b$  into a vector updated objects
- Remembered Set
  - Like remembered list, but uses a bit within object  $b$  to record that  $b$  is already in the vector
  - To avoid duplicate references to  $b$  in the vector

### 1. Remembered List

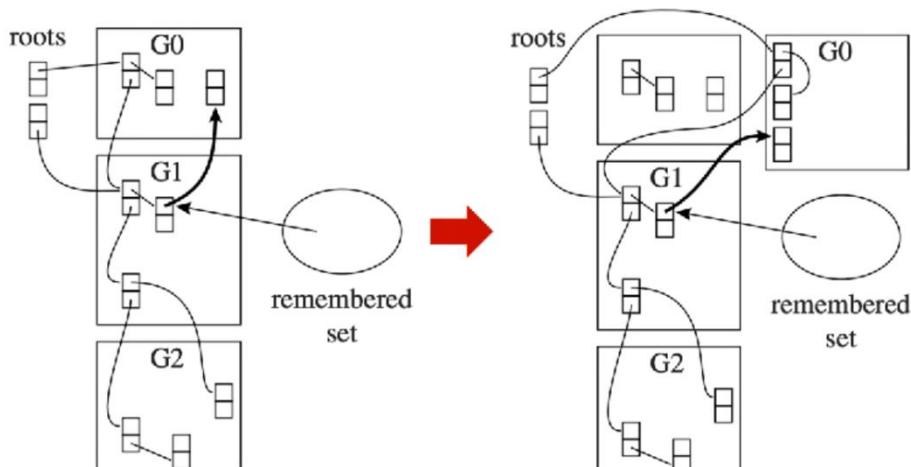
最早的方法就是使用一个列表来记录，每次我们来更新一个引用的时候，如  $b.f \leftarrow a$ ，并且  $b$  是老年代的，而  $a$  是年轻代的，那么我们就把  $b$  记录进一个列表里。但是这种实现的问题可能就是  $b$  会被重复加到列表中。

### 2. Remembered Set

为了避免重复的情况，我们可以在对象中放一个 bit 来记录它是否在 List 中。

## An Example of Remember Set

- Remember sets mark all inter-gen references

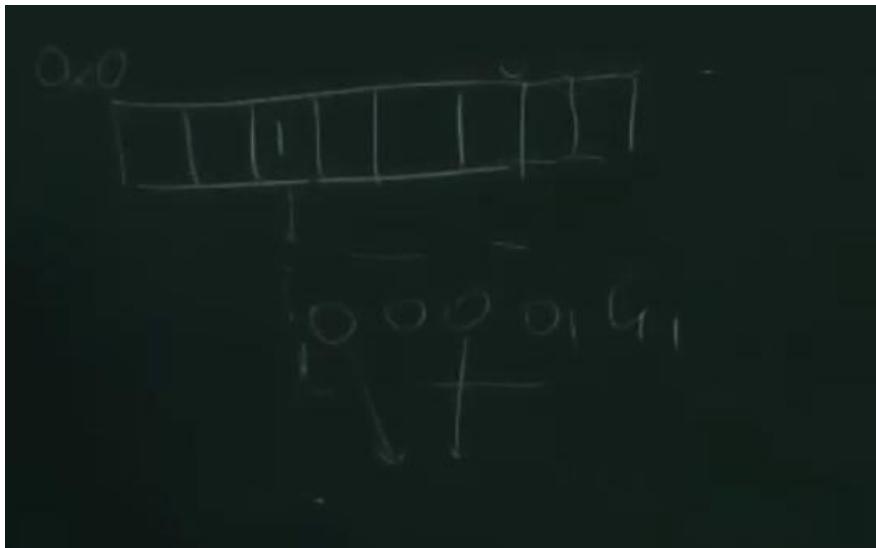


### 3. Card Marking

思路是把内存分成很多个逻辑上的 card，每一个 card 包含 512Byte

- Card Marking

- Divide memory into logical "cards" of size  $2^k$  bytes
- An object can
  - occupy part of a card or
  - start in the middle of one card and continue onto the next
- Whenever address  $b$  is updated, the card containing that address is marked
- There is an array of bytes that serves marks
  - The byte index can be found by shifting address  $b$  right by  $k$  bits



一个 card 对应很多对象，一旦置为 1，我们在 card 中遍历一遍，是折中的方法。

- **Page Marking**
  - Like card marking, but  $2^k$  is the page size
  - Update an object in a page makes the page dirty
  - User level program can protect each page write-protected after each GC
  - When a write protect violation occurs, user lever fault handler can record the dirtiness and unprotects the page

可以很容易地对应成页表。一旦更新过了，page 就是 dirty 的。我们可以主动地把页标志成 read-only 的，这个 syscall 是 mprotect。如果有人去写它就会发现 page fault，然后用户可以自己注册一个 page fault handler 来记录 page 的 dirtyness。这个就是转换成了硬件的实现。

## Generational Collection

---

- When garbage collection begins
  - The remembered set tells which objects of the old generation can possibly contain pointers in  $G_0$
- After several collections of  $G_0$ :
  - Generation  $G_1$  may have accumulated a significant amount of garbage that should be collected
  - It is better to collect  $G_0$  and  $G_1$  together

Generational Collection 的整体的 workflow: 我们开始的时候只要知道哪些对象拥有 G<sub>0</sub> 的指针。因为引用可以改变，所以只是可能拥有 reference。越往上，越老的代，回收的频率越低。

## Generational Collection

---

- When garbage collection begins
- After several collections of G<sub>0</sub>,
- Each old generation should be exponentially bigger than the previous one
- An object should be promoted from G<sub>i</sub> to G<sub>i+1</sub>
  - When it survives two or three collections of G<sub>i</sub>

当一个对象经历了多次回收还存活以后，我们就让它成为更老的一代，具体的值取决于应用。

## Cost of Generational Collection

---

- Review - the amortized cost:  $\frac{c_3 R}{H - R}$ 
  - No H/2, as we do not need a from-to design
- The live object size is much smaller for younger generation
  - E.g. it is common when H > 10R
  - The amortized cost is about c<sub>3</sub>/9 (1.1)

假设大部分刚创建的对象都很快消亡了，那么相应的开销是 c<sub>3</sub>/9，而刚才 Copy Collection 的开销是 c<sub>3</sub>。

所以分代就是分开管理不同生命周期的对象，降低了存活对象的比例，也降低了开销。

- What about the older generations?
  - The cost is still high
  - But the frequency is much lower

我们之前提到的回收方法都是停顿式回收，以 copy collection 为例，也就是当我们要 allocate 一个 record 的时候，我们发现 from-space 满了，那么应用程序就要阻塞在那里等待我们回收算法执行完毕释放出空间来。

停顿式垃圾回收算法的好处就是，在回收的时候我们占据了所有计算资源，所以性能更高，并且应用程序阻塞在那里，我们不需要考虑和应用程序的同步，所以正确性比较容易保证。

但是垃圾回收的占据时间可能比较长，比如 10+G 的堆可能就要几秒十几秒，这对于交互式应用是很不友好的。

## Incremental Collection (增量回收)

### Incremental Collection

- Prior collection algorithms require applications to pause
  - So GC threads can move objects correctly
- However, the pause time might be long
  - Especially for a whole-heap collection ( $G_1 + G_0$ )
  - Undesirable for interactive/real-time programs
- That's why we need an "incremental" collection

可以想到，增量回收是每次回收一点，和应用交叉执行的，所以我们定义两个角色：

1. 回收器：负责垃圾回收
2. 修改者（mutator，其实就是我们的应用）：我们认为应用的事情就是不断在修改数据的 reference graph。

在这个基础上我们提出 incremental collector 和 concurrent collector。

增量垃圾回收就是应用需要的时候触发；而并行垃圾回收可能就是堆到一定程度了以后会和应用并行执行，此时应用可能在任何状态。

## Logic Model: Tri-color Records

- A record must have one of the following three colors:
  - White: not visited by GC
  - Grey: visited by GC, but its children are not
  - Black: visited by GC, so as its children

三色模型就是把 record 的状态分为三种，分别是黑白灰。

白色：对象还没有被 GC 访问过。

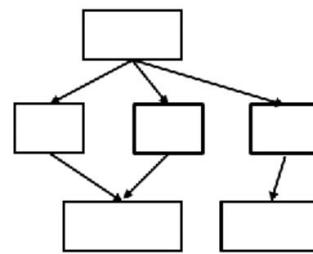
灰色：对象被 GC 访问过，但是它的儿子还没有被访问过。

黑色：对象和它的儿子节点都被 GC 访问过了。

## Tricolor Marking: The Basic Model for Marking

- Tricolor generalizes the marking phase of GC
  - All records are white before GC
  - When a record is visited, marking it grey and examine its (reachable) children
  - Marking it black, go processing more grey records

```
while there are any grey objects
 select a grey record p
 for each field fi of p
 if record p.fi is white
 color record p.fi grey
 color record p black
```



27

思路就是把之前的标记算法（mark & sweep、copy）做进一步的抽象，通过三色的方式表达。

## Tricolor Marking: The Basic Model for Marking

- Tricolor generalizes the marking phase of GC
  - Different algorithms only have different UDFs during marking

```
while there are any grey objects
 select a grey record p
 for each field f_i of p
 if record $p.f_i$ is white
 color record $p.f_i$ grey
 color record p black
```

32

inserting into stack

removing from stack

## Invariants of Tricolor Marking

- Invariant 1: No black object points to white
  - This suggests black objects are not completely processed



- Invariant 2: Every grey is on the collector's data structure (stack, queue, etc.)
  - Or an under-processed object is missing

三色算法保持正确性需要保证两个特质：

1. 不能有黑色对象指向白色对象，我们只会出现黑->灰、黑->黑、灰->白的情况。
2. 所有的灰色对象都应该在 collector 的管辖范围内，我们可以通过栈、队列等记录下来。

## Incremental Ways to Preserve Invariants

- Basic idea: instrumenting read or write operations (or known as barriers)
- Method 1 (Dijkstra, Lamport et al.)
  - When **storing** a white pointer  $a$  into a black object  $b$ , coloring  $a$  grey



GC process guarantees that these two properties are not violated. Because at this time the application is also modifying data, it is possible to encounter a situation where a black block is引用 a white block. To solve this problem, we need to add write operation hooks, that is, when we want to let a black object's field be a white object, we color the white object grey.

- Method 2 (Steele)
  - When **storing** a white pointer  $a$  into a black object  $b$ , coloring  $b$  grey



- Method 3 (Boehm, Demers, Shenker)
  - Marking black pages as read-only. Page faults mark objects grey and the page writable



We can find a common thought in garbage collection is through virtual pages management, because garbage collection關注的事情无非就是某一块有没有写、有没有读，所以这和虚拟页的想法是一致的。这个方法就是以页为单位来管辖，一旦发生一个 page fault，我们把这个 page 中的所有对象全部重新标记为灰色进到队列中重新处理，所以这个开销就比较高。

- Method 4 (Baker)
  - When **reading** a pointer  $b$  to a white object, coloring  $b$  grey
    - A mutator can never have a pointer to white objects



我们写的时候运行黑指向白的情况，它认为只有应用观察到的时候才会有问题，所以当应用读到这个对象的时候才把  $b$  染色成灰色，保证了应用永远不会从黑色对象中拿到一个对白色对象的引用。

- Method 5 (Appel, Ellis, Li)
  - When **reading** a pointer  $b$  to a non-black object, a page fault is triggered
    - A mutator can never have a pointer to non-black objects



变成了虚拟内存的思路，所以我们要把页标记成不可读。

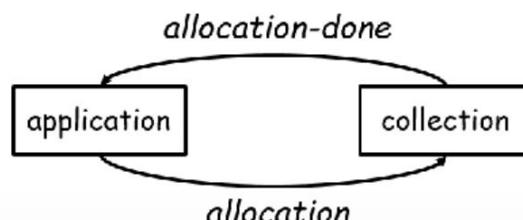
## Baker's Algorithm

### Baker's Algorithm

- An incremental collection algorithm atop Cheney's
  - Review: Cheney's copying algorithm
    - Dual spaces (from/to), copy-based, forward pointers

```
scan ← next ← beginning of to-space
for each root r
 $r \leftarrow \text{Forward}(r)$
while scan < next
 for each field f_i at scan
 scan. $f_i \leftarrow \text{Forward}(\text{scan}, f_i)$
 scan ← scan + size of record at scan
```

- Single threaded: one thread for both application and GC
  - Do some GC work when allocation happens



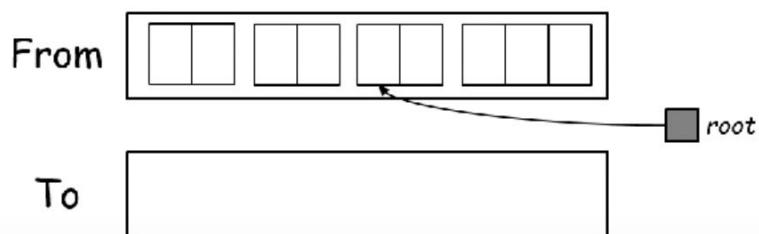
2021/12/10

40

当我们进入 allocation 时，我们也要进入回收，当 allocation 完毕的时候，我们回收也结束。有些时候分配没有失败的时候，我们也要回收，这就是和之前的停顿式回收算法不同的地方。

## Baker's Algorithm: Triggering

- The heap still contains from/to spaces
  - From space: served for allocation
  - To space: served for copy destination
- Collection starts when from space is full
  - Detected by an allocation request



2021/12/10

41

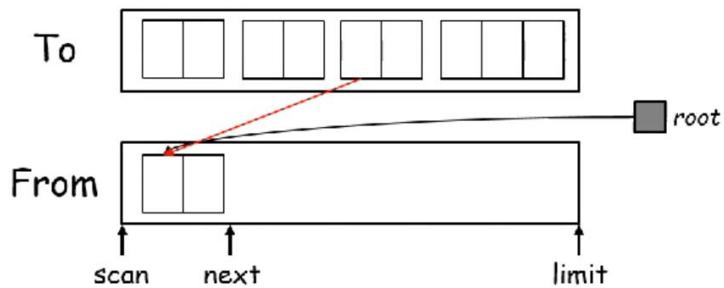
首先 Baker 也延续了 Cheney 的设计，也是分成了 From-space 和 To-space。From-space 接收分配的对象，而 To-space 接收拷贝的对象。

当 From-space 满的时候，这个算法开始。

第一步就是 flipping，把两个区的角色对调，这一步 Cheney 是在垃圾回收结束后改变的。

### Step 1: Flipping

- The role of from/to space switches
  - The original to-space now accepts new objects
- Forwarding all roots
  - Copying the referred object to the new space
  - Do not forget the forwarding pointer

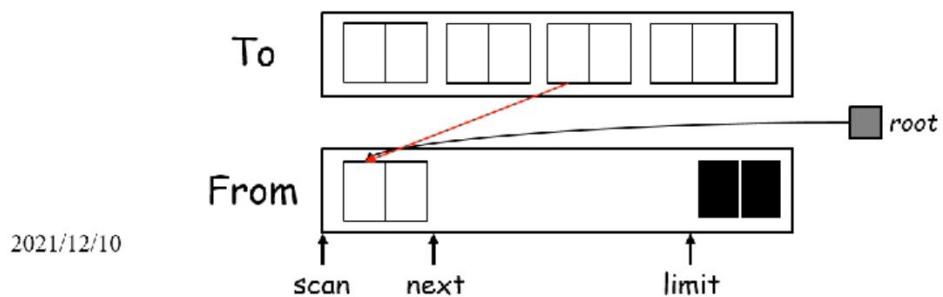


2021/12/10

43

## Step 2: Allocation during GC

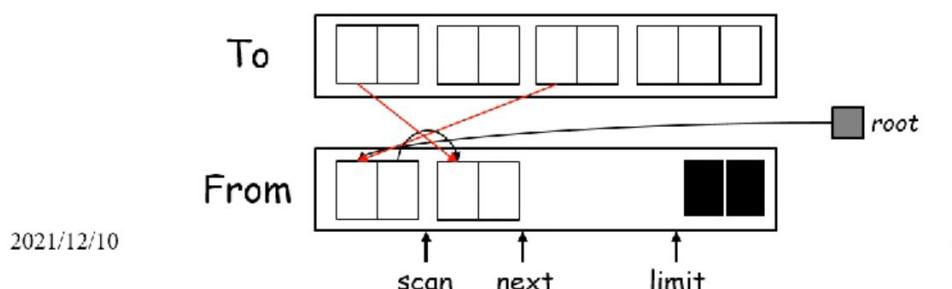
- Objects are allocated from the **bottom**
  - Allocating by minus the *limit* pointer
  - Preserving locality for live objects
- New objects do not need special handling
  - They have not created any useful references
  - Can treat them black



因为是 allocation 的时候触发，所以我们还是要分配出来空间给应用的，它想出来的办法就是从尾部开始分配。而前面的空间就用来垃圾回收。这样就不会把活对象和新对象混在一起，有一个局部性的好处。我们可以认为刚创建的没有任何引用的对象可以直接认为是黑色的。

## Step 3: Incremental Scanning

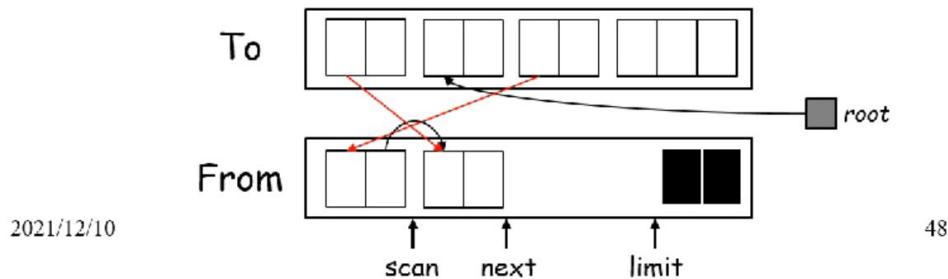
- Still scanning objects, but not all
  - If allocating N bytes, then scanning at least N bytes to catch up with allocation
- Copy more objects to the new space
  - Also: remember the forwarding pointer



如果我们 allocate N 个 byte，那么我们至少就要 scan N 个 byte，让回收能够追上分配的速度。

## Step 4: Loading References

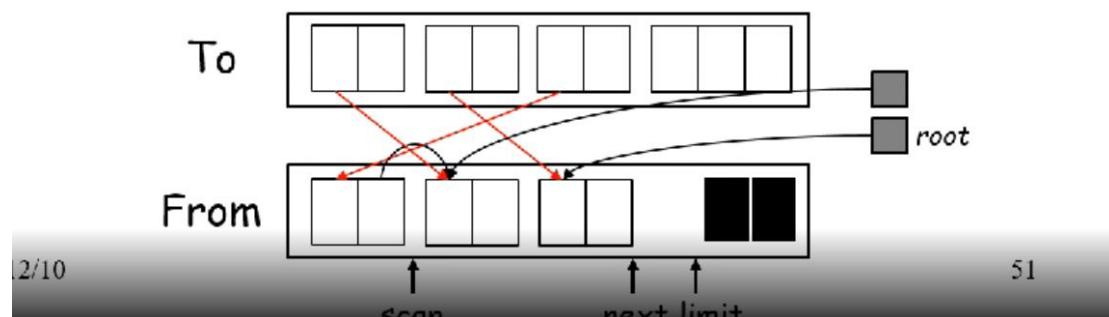
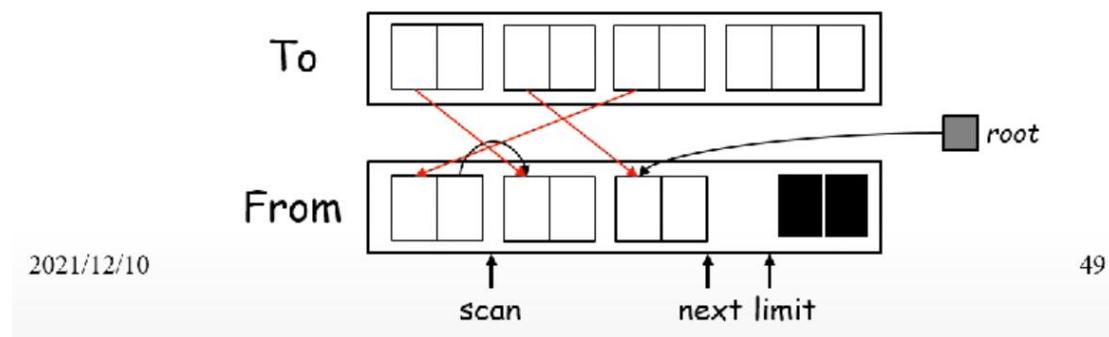
- Mutators create new roots by loading a reference
  - Invariant: no reference should be pointed to To-space!
  - Detected by instrumenting read instructions



此时这一步的回收基本上完成了，下一步我们就是要让 mutator 去执行，交还给应用去执行，应用就会加载一些 reference 进来。我们的最终目标是在 GC 之后，所有对象都在

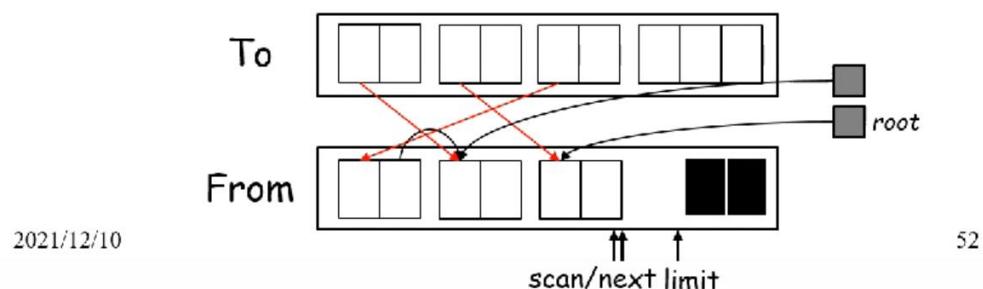
From-space, 让 To-space 完成下一轮的 GC, 所以此时加载的 reference, 我们不能指向 To-space。我们要检查所有的引用, 这个方法就是插桩 read 操作来完成。也就是对于加载的执行 To-space 的 reference, 我们立马做 forwarding, 把它 copy 到 From-space。

- Solution: forwarding immediately



## When does It Finish?

- When scan finally reaches next
  - Triggered again when the from space is full
- Can collection catches up with allocation?
  - Yes when  $R < H/4$ 
    - The live size is smaller than the half of from-space
    - When finished, allocation is also smaller than  $H/4$



当 scan 和 next 重合的时候，GC 就结束了，此时所有对象都 copy 完了。当活着的对象比较少的时候，就可以使用这个算法。

## Pros and Cons for Incremental Collection

---

- Pros: low latency
  - Avoid a large number of copied objects at a time
  - Suitable for interactive/real-time applications
- Cons: large performance overhead
  - For Baker's: adding two operations for each read
    - One for compare, one for jump
  - The overhead can be more than 20%
    - Does not consider issues like locality

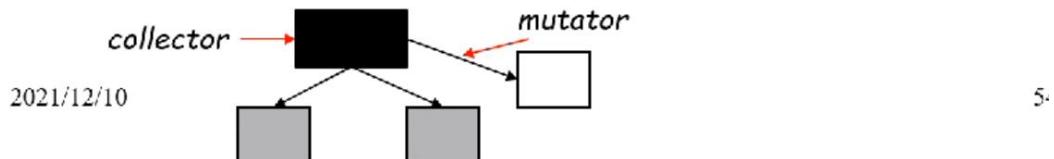
每次垃圾回收我们都可以很快完成，我们每次只需要 copy 一部分就返回给应用了，对于实时性较高的应用是比较好的。缺点就是 overhead 比较高，我们要对每个读操作插桩，比较当前对象的 forwarding pointer 是否指向自己（有没有 copy），包含了内存访问和分支操作。

最后稍微讲一下并行回收，算法基本上思路一样

## Some Words on Concurrent Collection

---

- The algorithms are nearly the same
  - Read/write instrumentation (barriers)
- But synchronizations are required
  - A simple example to violate the invariant
    - Collector: finish scanning all references of a
    - Mutator: add a reference to white object b
    - Collector: color a black
    - A black → white reference occurs!



麻烦的地方是同步指令的加入，因为这是两个线程，应用和回收器可能在任何时候修改和 copy 对象。可能回收器刚刚完成扫描，把当前对象标黑了，此时应用立马把一个引用修改到了白色对象上。因为很多行为不是原子的，很容易出现这种情况。

解决方法也很多，最简单的操作就是加锁/原子指令/虚拟内存的 page fault。

## Some Words on Concurrent Collection

---

- The algorithms are nearly the same
  - Read/write instrumentation (barriers)
- But synchronizations are required
  - Many methods can be used
    - Locks/atomic instructions
    - Virtual memory (page fault)
    - Other hardware-supported features

还有一些硬件支持的 feature，比如硬件事务内存等。总之就要考虑更多同步方面的事情。

2021/12/14

## GC 的实现

我们需要让编译器产生一些 GC 相关的信息，然后让 GC 去处理。

### What is required for compiler to support GC?

---

- *Code for record allocation*
- *Data layout of records*
- *Describing roots for each GC*
- *Read/write barriers*

首先是 *allocation* 的生成，需要从堆上分配。GC 需要知道每个 *record* 对应的 *field* 是不是指针；第三个就是 *Root* 是怎么判断出来的，我们需要从根对象出发找到存活的对象，最后一个就是 *barrier*，也就是上节课讲到的并行/增量 GC 中对读和写可能需要一些插桩，来维持并行/增量 GC 的不变量。

*Allocation* 为什么重要呢？教材上做了一个分析：对于内存密集型应用（图相关算法、机器学习型算法），我们会访问很多内存节点，还有函数式语言（不太鼓励更新）。

在这种场景下，分配是比较的，假如我们每 7 条指令就有 1 条 *memory store* 的话，每一次 *memory store* 都会对应一个 *allocation*，相当于把七分之一的时间都放在了 *allocation* 上。

我们看一下 *copy allocation* 的步骤。

## Steps in Allocation (for Copying Collection)

---

1. Call the allocate function (allocateRecord)
2. Test  $next + N < limit$ ? (test succeeds)
3. Move  $next$  into  $result$
4. Clear  $M[next], M[next + 1], \dots, M[next+N-1]$
5.  $next \leftarrow next + N$
6. Return from the allocate function

1 和 6 就是分配+返回。

## Overhead Analysis & Elimination

---

1. Call the allocate function (allocateRecord)
  2. Test  $next + N < limit$ ? (test succeeds)
  3. Move  $next$  into  $result$
  4. Clear  $M[next], M[next + 1], \dots, M[next+N-1]$ 
    - Can be directly filled with user data (**safe?**)
  5.  $next \leftarrow next + N$
  6. Return from the allocate function
- B. Store useful values into the record**

我们来看一下怎么优化，对于 1 和 6 我们可以 inline 一下。

## Optimization Results: Two Steps left

---

1. Test  $\text{next} + N < \text{limit}$ ? (test succeeds)
  2.  $\text{next} \leftarrow \text{next} + N$
- 
- Overall overhead: about 4 instructions
    - Load N; next + N; cmp; store next
  - Potential optimizations: combining multiple allocation requests together
    - Allocate A, Allocate B  $\rightarrow$  Allocate A+B

所以 overhead 不会很高，通常为 4 个 instruction。我们在做后端优化的时候，如果发现在一个 basic\_block 中既有 allocate A 也有 allocate B，我们可以简单地合并成 allocate A+B。

## Data Layout Description

# Data Layout Description

- The collector needs to handle records of different types
  - Different length: used when adding scan
  - Field type: used by Forward
    - Only pointers need to be processed

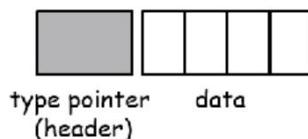
```
scan ← next ← beginning of to-space
for each root r
 $r \leftarrow \text{Forward}(r)$
while scan < next
 for each field f_i at scan
 scan. $f_i \leftarrow \text{Forward}(\text{scan}.f_i)$
 scan ← scan + size of record at scan
```

2021/12/14

第二部分就是怎么描述 data layout。因为我们的 GC 需要 record 的 size 的大小，并且在 Forward 函数中，我们的 GC 需要判断一个 record 的 field 是否是一个 pointer。所以这些都需要我们的 data layout descriptor 来描述。

# Data Layout Description

- The collector needs to handle records of different types
- A simple solution: add a descriptor for every record!
  - For OO languages: no extra overhead
    - objects need type descriptors (introduced later)
  - For Statically typed language: one-word overhead



2021/12/14

12

一个简单的方法就是给每个 record 都添加一个 descriptor，这个在面向对象中也是很常见的。比如说 C++ 中的虚表来查询最终函数对应到的是哪个对象的成员函数。这个我们就可以认为是 descriptor 的一种实现。

所以对于面向对象语言，它总是要生成这些东西的，所以没有什么额外的开销。

而对于 statically typed language，就需要一个 word 的额外开销。因为在编译的时候，我们是可以通过 padding 等需求计算出我们的 malloc 的大小的，而 GC 是在运行时操作的，已经丢失了编译时的信息，所以我们需要这个 field 来额外维护这个信息。

## Descriptor Implementation

---

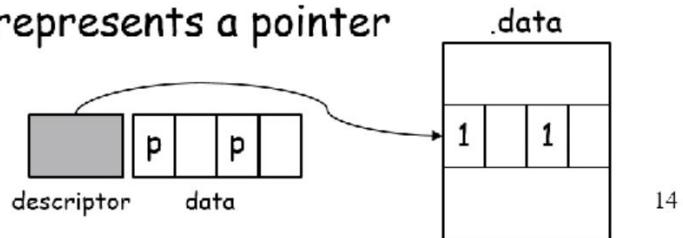
- To implement GC in Tiger, record descriptors are required
  - Then GC knows how to handle different types
- A simple design: using a string
  - `int *alloc_record(int) -> int *alloc_record(string)`
  - String length: record length
  - Each byte: 1 represents a pointer

首先原本的 alloc\_record 就是传入一个 int 进去，因为我们现在的 tiger 都是一些 64 位的数据结构。我们现在参数传一个 string 进去，string 的长度就是 record 的长度，而每个 byte 就表示是否是一个 pointer。（如 P 表示 pointer，而 N 表示非 pointer，或 1 表示是 pointer 而 0 表示是非 pointer）

## Descriptor Implementation

- To implement GC in Tiger, record descriptors are required
  - Then GC knows how to handle different types
- A simple design: using a string
  - `int *alloc_record(int)` -> `int *alloc_record(string)`
  - String length: record length
  - Each byte: 1 represents a pointer

2021/12/14



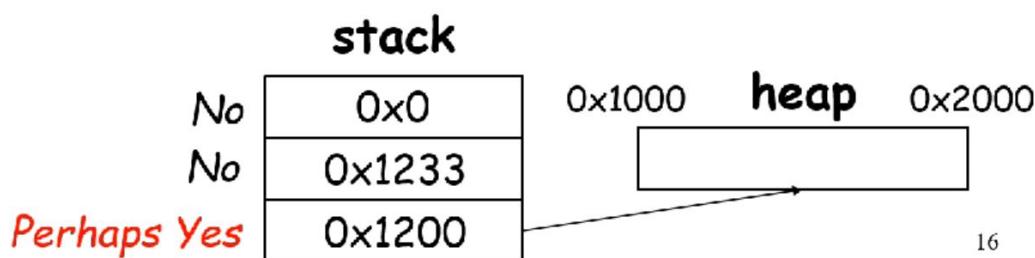
14

注意这个 string 放到堆上会很麻烦，因为这个信息我们是在 type-checking 之后我们就知道的，所以我们可以把这个 string 放进.data 区域。

对于 array 的情况，比如长度为 100，array 元素为 record 的情况，我们可以用 100 个 1 来描述，也可以自己设计一套解码机制来压缩长度。

## Root Description

- GC starts from roots for tracing
  - But where are them?
- A straightforward design: guess!
  - Scanning stacks/registers
  - Finding all looking like pointers



16

怎么去描述根对象？在 trace GC 中，都是从根对象出发去 trace 这些活着的对象。在运行时，栈上会不断生成指向堆的指针，除了栈以外还有寄存器可能包含 root。我们知道往哪里找，但是我们不知道哪些是指向堆的。

早期的时候，我们就把栈和寄存器全部扫一遍，全部试试看能不能指向堆。比如 0x0 不在堆的范围中，0x1233 没有对齐肯定不是，0x1200 符合条件，可能是指向堆的指针，那我们可以进去看一下是不是一个 record。这样做的时候，当我们猜到在这里面以后，我们应该找到对应的 record 头。

这种基于猜测的方法就叫做 approximate GC。

它的缺点是：运行时的一些 integer 可能被错误地认为是 pointer，导致死的 record 被标记为活的 record 继续存活。

## Exact Root Description

怎么建立 exact root description 呢？方法就是建立一个 pointer map，在编译期间生成一些和指针相关的 description，因为编译器在编译期间，type-checking 后，我们知道所有变量及其类型是什么。

# Pointer Maps for Tiger: Contents

- A pointer map should consist:
  - Pointers on stack
  - Pointers in callee-saved registers
  - GC threads use those pointers to traverse
- Where should we insert a pointer map?
  - It depends on when GC is triggered
    - When allocation: inserting before alloc\_record

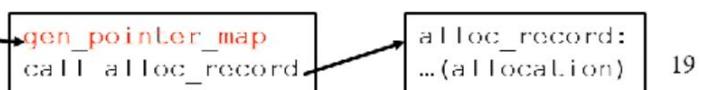
2021/12/14



它的内容包含栈上的内容和 callee-saved register。我们在调用 alloc\_record 的时候插入一个 pointer map 的生成。

- When allocation: inserting before alloc\_record
- For recursive invocation: insert for all function calls

2021/12/14



为了处理嵌套调用的情况，我们对于每一个函数调用都插入一个 pointer map。

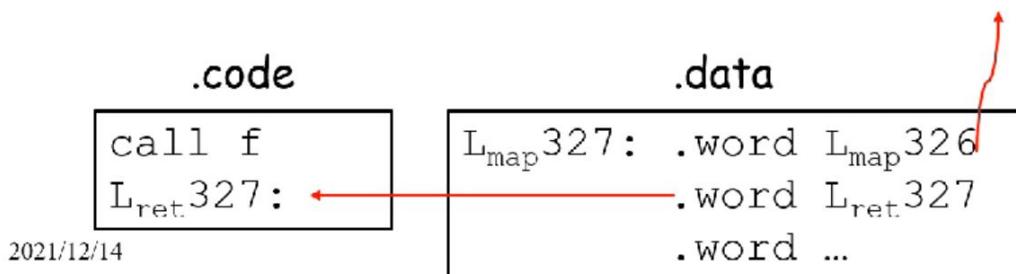
## Pointer Maps for Tiger: Information Passing

- How to know which are pointers?
  - We knew them, but we lost them during compilation
  - For example: we know them in AST (type checking)
- We need to pass this information downwards
  - Temp: mark if it is a pointer in `temp_map`
  - Stack slot: mark if it contains a pointer
  - Register allocation: passing temps to registers

在寄存器分配的时候，那些溢出的变量我们也可以知道它是否是一个 pointer。

## Pointer Maps for Tiger: Storage

- The contents of pointer maps can be totally determined during compilation
  - Each call site has exactly one pointer map
- We can store pointer maps in `.data` section
  - Index with labels, linked to the next map



那么怎么存呢？因为 pointer map 是编译器决定的，所以我们每个 call site 只会生成一个 pointer map。既然是静态的生成，我们就可以把它放到 `.data` 段。我们可以看到每个 pointer map 的元素如上右图组织，首先第一个值是指向上一个 label 的位置，也就是链表的形式把整个 pointer map 连起来，第二个指向的是 call 后的 return address 的 label，唯一标识了这个 call。后面就可以放一系列的寄存器的 pointer 信息和栈上的 pointer 信息。

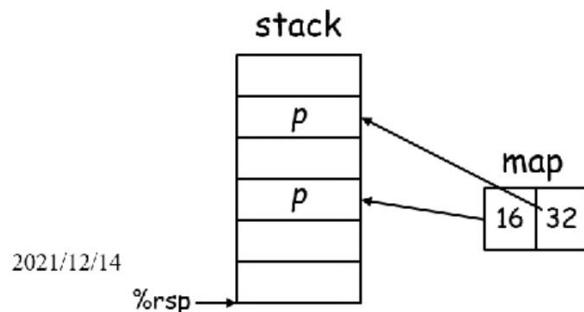
| .code                       | .data                                       |
|-----------------------------|---------------------------------------------|
| <pre>call f L_ret327:</pre> | <pre>.global L_head ... L_map327: ...</pre> |
| 2021/12/14                  |                                             |

在运行时的时候，我们有一个 call，拿到其返回地址以后，我们就在.data 端从链表头开始搜索，每次找的时候比对第二个元素是否相同。所以我们用.global 来标识链表头在哪。

## Pointer Maps for Tiger: Structure

---

- Stack variable: marking offsets relative to stack frame pointers
  - Pointer values can be found on the stack address
- What about registers?

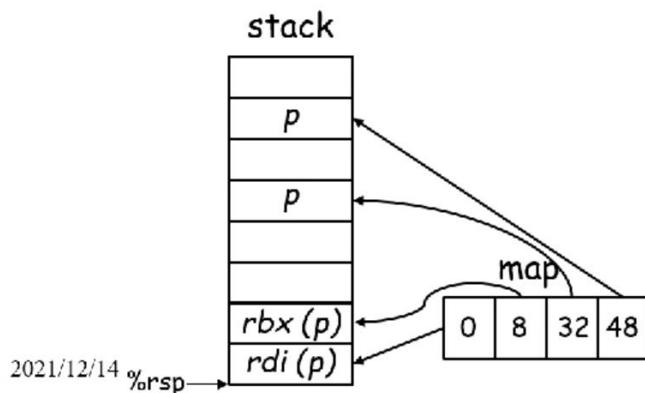


23

那么我们栈上的内容怎么存值呢？我们只需要存栈的偏移量即可。我们通过%rsp 加上这个 offset 来找到这个偏移量。

## Pointer Maps for Tiger: Structure

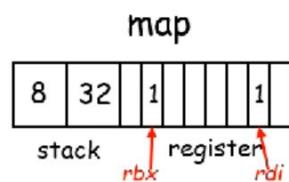
- A naive method: push callee-saved registers before invocation
  - Storing pointers is enough
  - Maps only need to remember the offsets on stack



寄存器的存放的一个简单的方法就是我们把 callee-saved register 全部 push 到栈上，然后复用之前的这个偏移量。

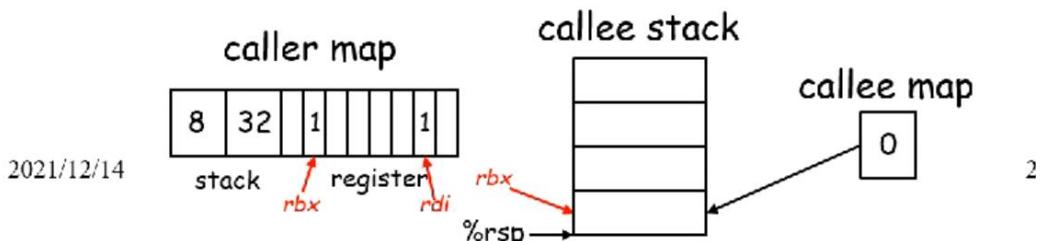
## Pointer Maps for Tiger: Structure

- An advanced method: only marks if each register contains a pointer
  - A simple bitmap (each bit for a register)
  - The value is not stored in the map



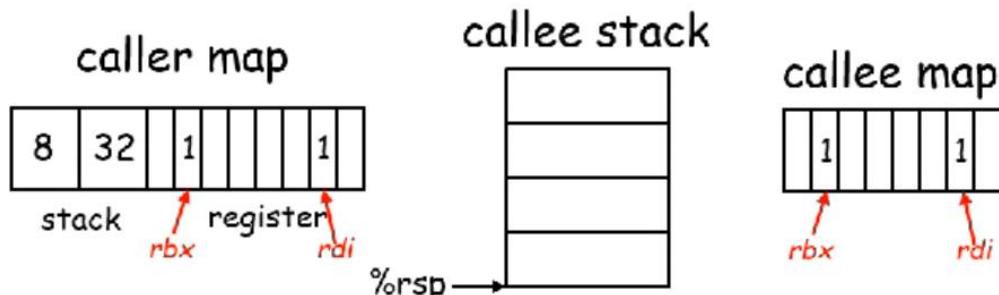
## Pointer Maps for Tiger: Structure

- An advanced method: only marks if each register contains a pointer
  - A simple bitmap (each bit for a register)
  - The value is not stored in the map
- Co-generate with callees to find values
  - Callee tracks types of those pointers
    - Spilling: enter them into callee map



此时我们就不能单纯考虑自己了，因为 caller 中的 rbx 可能在 callee 中被 spill 到了栈上，在这种情况下我们就必须维护住 rbx 是个指针的信息。所以 caller 的信息要传递给 callee, callee 拿到这些信息要生成对应的 callee map。

在这样的结构下做 GC 是没有意义的，因为 caller map 中的 1 并不代表它在寄存器中，我们需要通过信息的传递不断地去寻找。



简单函数的话直接把 0 和 1 往后传即可。

- What about the uppermost function?
  - i.e., calling to the allocation function (in C)
  - Go back to the naive solution (pushing into stack)

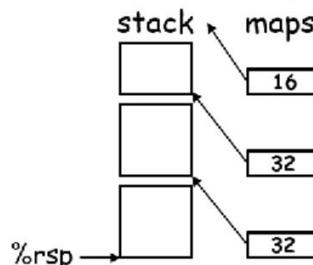
这样传总得有一个尽头，到达 uppermost function 的时候，我们重新调用 naive 算法来把对应的寄存器 push 到栈上。

## Pointer Maps for Tiger: Scanning

- All stack frames have their own pointers to heap
  - GC should recursively scan them to find all
- Solution: also put the frame size on maps
  - Frame size is known at compilation time
  - Decreasing the size to find the previous frame

2021/12/14

29



我们的栈是多个 frame 组成的, 所以 GC 不得不去递归地扫描这些 frame, 因为每个 frame 都有自己的 pointer。

我们可以把 frame\_size 也放到 pointer map 里, 这样我们就可以不断对%rsp + frame\_size 从而得到上一个 frame 在哪。

- What about the last one?
  - Get %rbp in the allocRecord frame and scan upwards

# Derived Pointers

---

- Detecting roots might not be enough
  - We may have pointers to internal structures of record/array
- An example: load  $a[i-2000]$ 
  - Can be represented as  $M[a-2000+i]$
  - $t_1$  is not a pointer to an object
    - It may point to an integer (for array of int)
    - GC cannot start from it for traversal

2021/12/14

31

```
t1 <- a - 2000
t2 <- t1 + i
t3 <- M[t2]
```

这里有一个比较特殊的情况，在 guess 算法中发现了一个指向 record 中间的情况。这种情况下就是 pointer 指向数组的中间或者 record 的中间的一个 field。此时我们是没有它的信息的，我们不能盲目地往后继续扫描。这种就叫做 derived pointer（派生指针）。在这种情况下，单单从 root 开始找是不够的。

## Handling Derived Pointers

---

- Derived pointers should keep its base alive
  - Even though its base is no longer reachable
- Implementation in Tiger
  - Marking pointers as "derived" in pointer maps
    - Maintaining the base for each pointer

2021/12/14

33

```
a is dead! ←
rax <- a
rax <- rax - 2000
rcx <- rax + i
rdx <- M[rcx]
```

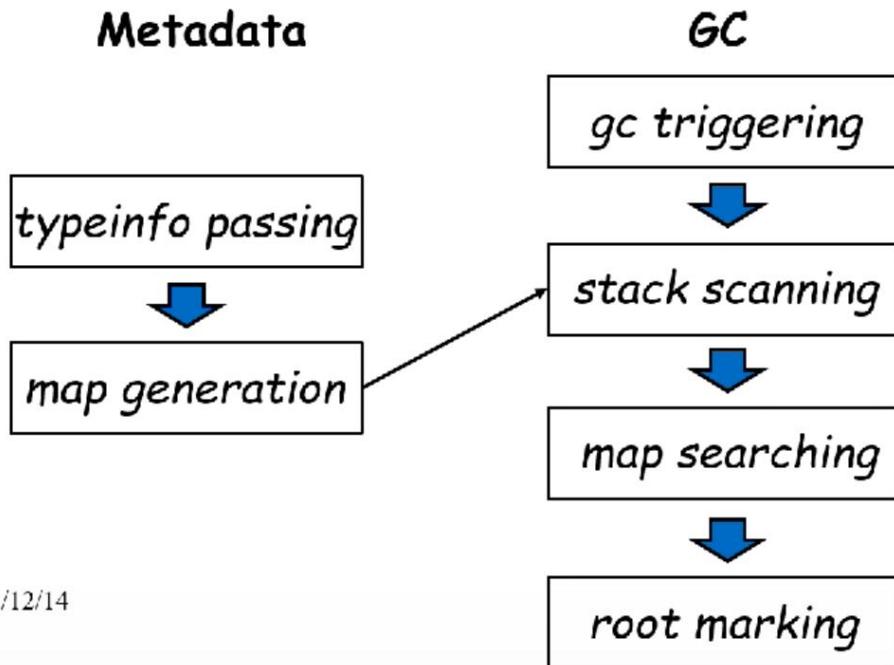
此时栈上已经没有  $a$  的指针了，但我们做 GC 的时候不能认为  $a$  死了。我们要认为数组的生命周期和其中每个 field 的生命周期是一样的，我们持有数组  $a$  的 derived pointer 的时候，

不能认为 a 死了。

这在编译期间也是比较容易的，我们可以判断出派生指针，然后记录下来它的 base pointer。

## Putting It All Together: Lab7

- The most difficult part is root marking



这里就讲完了除了 GC 本身要做的事情。GC 的实现麻烦在元数据生成的过程中，也就是把 AST 中的变量类型往后传到最后寄存器分配。而在 GC 阶段，触发就是在 allocation 的时候，扫描栈就是通过%rsp 不断加上 frame\_size 得到，然后我们在.data 中的 pointer map 中找到对应的 frame 的 pointer map，把对应位置上的栈或寄存器标记为 root，最后有了 root marking 以后就是做垃圾回收了。

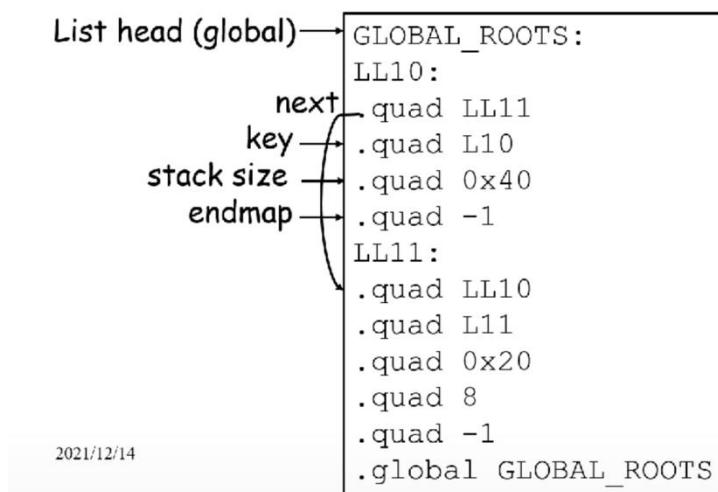
## A Little Words on Collection

---

- We do not restrict the algorithm for collection
  - Sweep and copying are both OK
  - Marking is a must (no reference counting!)
- The result should be **exact**
  - i.e., all unreachable objects should be reclaimed

### Examples of register maps in .data (x64)

---



GC 的时候就通过 `GLOBAL_ROOTS` 来扫描整个 pointer map 的列表。`key` 对应的就是 call site 处的 return address。注意这里助教的实现是一个循环的链表。

## Modern GC in Java

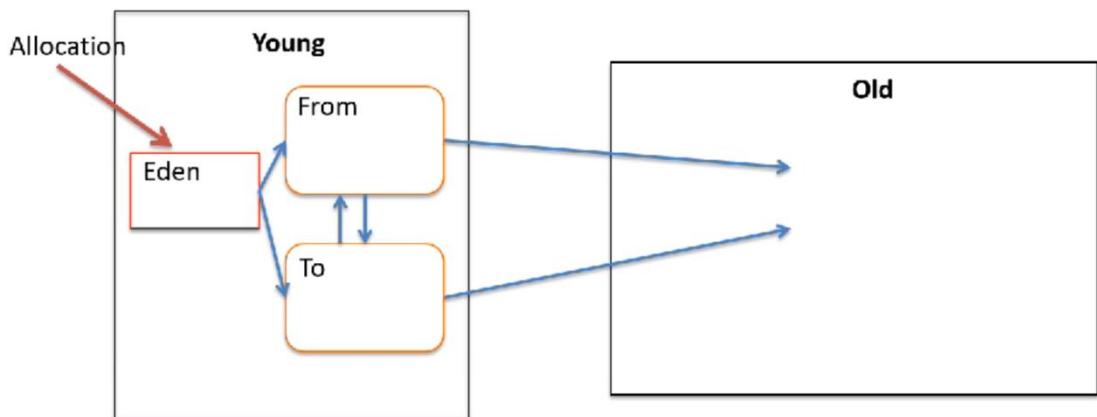
最后我们介绍 java 中的现代 GC，这个算法是一个停顿算法，所以主要覆盖的是分代算法和 copy 算法。

## Parallel Scavenge

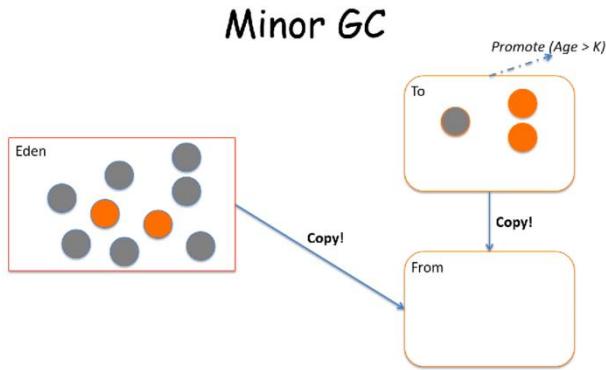
- Generation GC
  - Stop-Copy-Compact
- Different algorithms for old/young
  - Minor GC: From-to copy
  - Major GC: In-place copy

parallel scavenge 是 java 8 之前的默认 GC 算法，近几年 java 比较重视 GC，出了很多个新的。

这个算法是一个分代算法，回收过程分为年轻代回收 (from-to copy) 和年老代回收 (全堆回收、就地 copy)

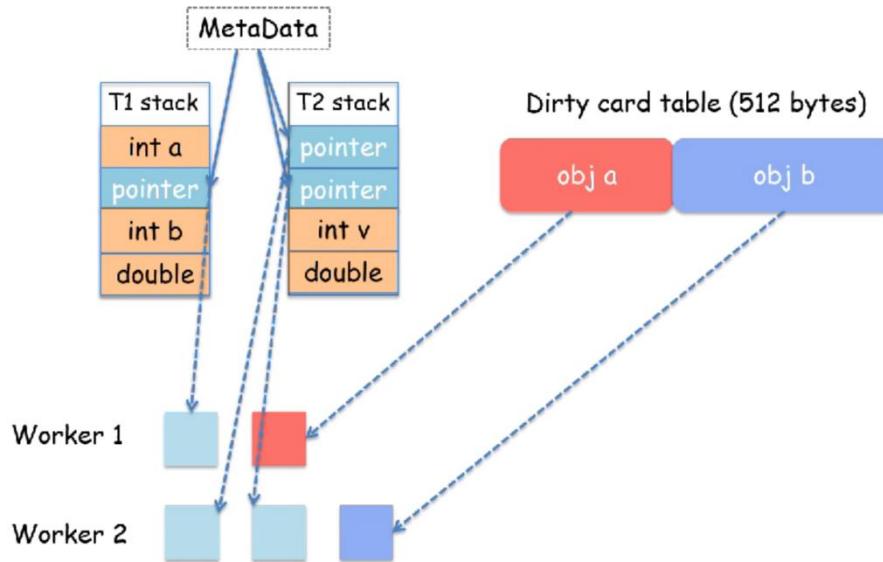


年轻代专门留出来一块分配的区域，from-to 存的是相对年纪大的对象，而 Eden 区大部分年龄都是 0。

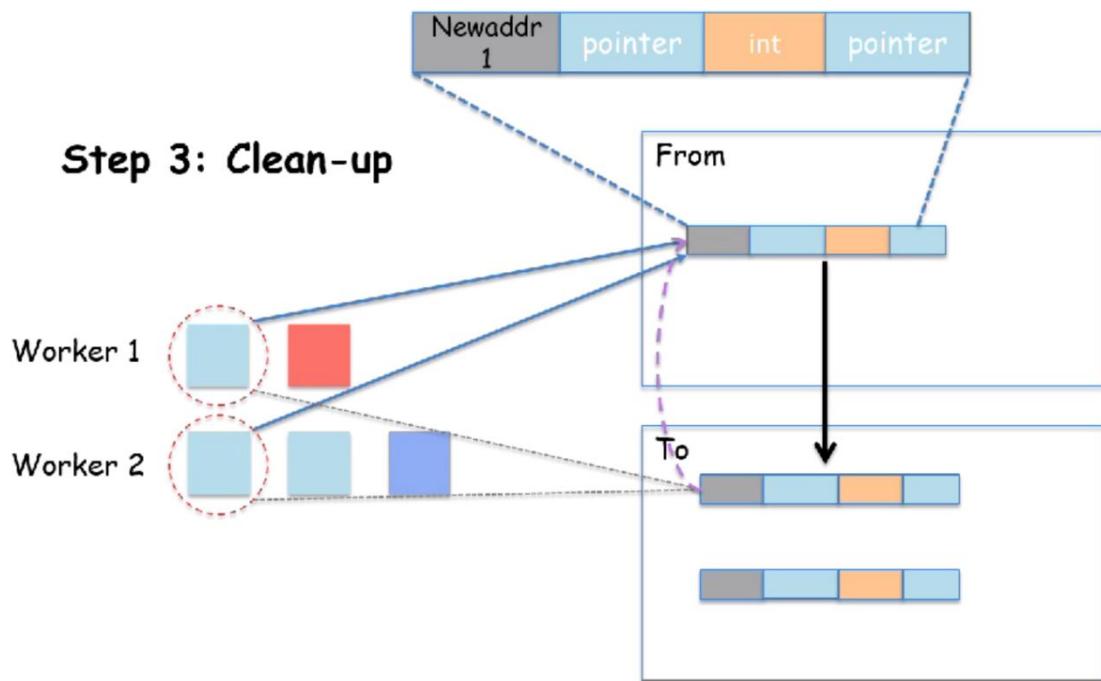


所以 minor GC 在现代的垃圾回收中，就需要使用多核 GC 来实现。遇到的挑战分为三部分：

1. root assignment to GC threads
2. Copy race
3. work-stealing

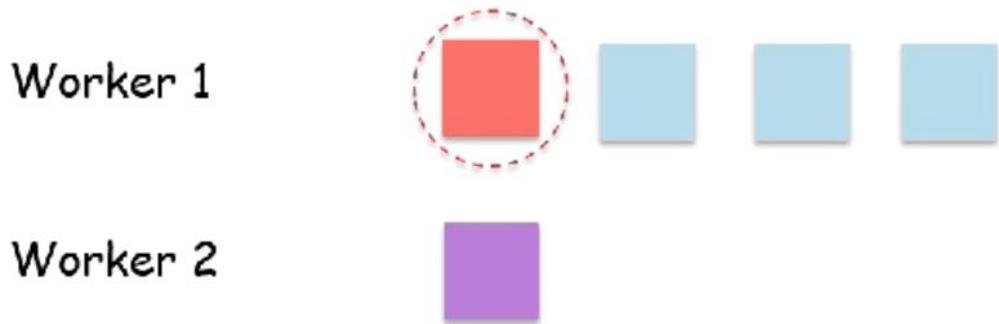


## Copy race



这个 header 包含了很多 OO 中的信息。对于两个 worker 都 copy 了一个对象到 to-space 的情况，我们把设置 From-space 中的 forwarding pointer 认为是一个原子的操作，worker2 想置这一个 field 的时候发现 worker1 已经置上了，那么它就把自己的指针指向 worker1 的 copy 位置，并且把自己 copy 出来的对象删除掉。

work stealing



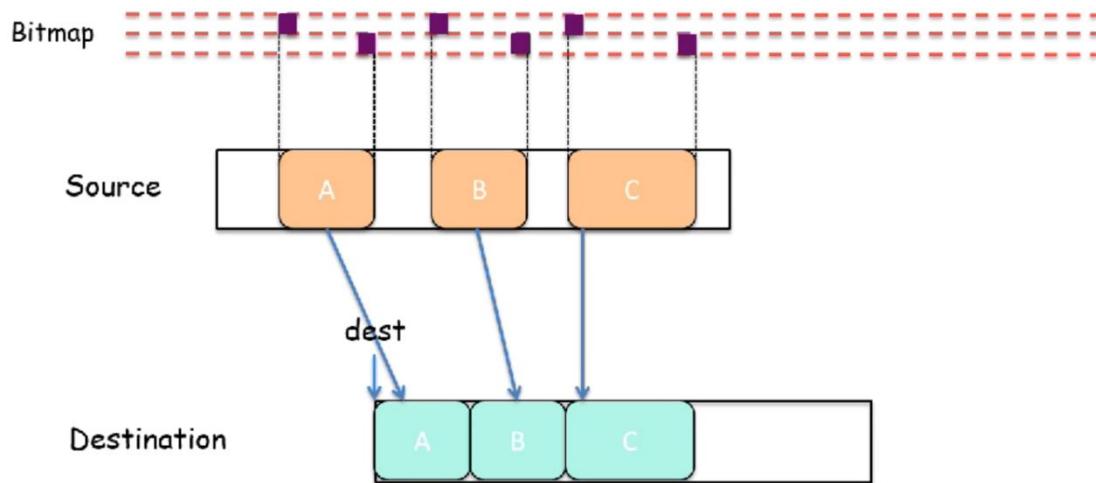
很有可能出现任务分配不平衡的情况，当我们发现 worker1 的任务太多的时候，我们 worker2 就可以从 worker1 的末尾去偷任务，也就是实现为一个双向队列。

## Full (Major) GC

- **Marking**
  - *Mark all live objects*
- **Summary**
  - *Calculate new address for live objects*
- **Compact**
  - *Move objects and update references*

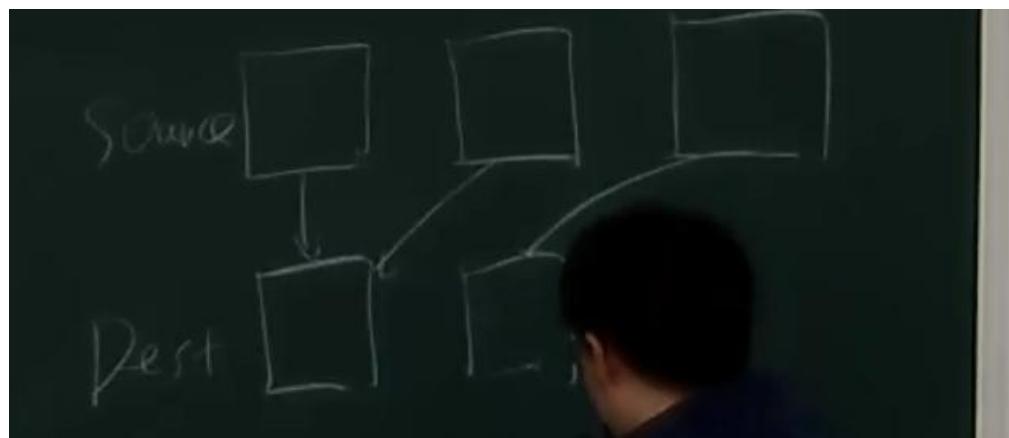
在全堆 GC 的时候，内存比较紧张，不能做 from-to copy，我们只能做 in-place copy。其实就是在自己的区域中做 copy，相对来说效率没那么高。在 summary 的时候我们可以准确地算出所有存活对象的新地址，算出来以后 GC 就变成了一个确定的算法，这样多核处理就没有什么压力了。

# Summary & Compact



两个 bitmap 表示对象的开始和结尾。我们规定 destination 一定在另一个区域去接收这些对象。我们只需要依次找到，copy 进来就可以了。

但是我们的 compaction 的区域是非常不充足的。



这样就出现了依赖问题，第二块需要等待先全部 copy 走以后，source3 才可以 copy 到 destination2 中。

**2021/12/17**

函数式语言教材里会将两个基础的概念：

1. 函数本身不再像是面向过程中使用的 function。

这个 function 还是有很大差距的。我们在接触计算机前就知道了  $y=f(x)$ ，数学上的函数就是两个集合之前的映射。在计算机中对这类函数的叫法叫做 **equational reasoning**，也就是不管在什么时候传入相同的参数  $x$ ，那么  $f(x)$  永远是相等的。

我们回想一下 **imperative programming**，很容易不是这个结果。比如引入一个全局变量。比如  $f(a)=a+b$ 。

```
var b:=10;
let f(a) = {
 b=b+1;
 return a+b;
}
```

每次进入函数的时候，全局变量 **b** 的值会改变，导致每次传入相同的 **a** 但是返回值变了。这个改动的全局状态就导致我们函数的返回值不相同，这就叫做 side-effect。

所以我们如果要追求函数式的话，它会要求你不修改全局状态（如：堆里的状态）。要设计纯正的 functional programming language 限制是比较的多。

还有一个重要的特性就是高阶函数（把函数作为参数传入），通常函数式语言应当支持这一点。应当可以支持传入参数作为参数/返回一个函数作为返回值。我们可以认为普通的函数是一阶的，而把函数作为参数传入是二阶的。

它会带来一些复杂的东西，比如内部函数会使用外部的变量，通常来说高阶函数都会有这个特点。那问题就来了：到底哪个才是函数式语言的本质呢？

## What is the essence of functional programming?

- Equational reasoning or higher-order functions?
  - No clear agreement about the answer
- Functional programming can be in different flavors
  - Fun-Tiger: Tiger + higher-order functions
  - PureFun-Tiger: Fun-Tiger + no-side-effect
  - Lazy-Tiger: Fun-Tiger + lazy evaluation

所以 high-order function 通常是我们写程序的时候碰到的东西，没有这个特性我们就感觉不到我们在用函数式。Equational reasoning 我们不一定感受得到，但是一定要尊崇这个定义才算做函数式。

所以我们在这里介绍 3 种不同的函数式语言。

Fun-Tiger: Tiger + higher-order functions，但是 C/C++ 也有函数指针，我们通常认为 C 是一个 imperative 的语言，还不算做函数式。

PureFun-Tiger: Fun-Tiger + no-side-effect，在 Fun-Tiger 的基础上保证了 equational reasoning 的部分。

Lazy-Tiger: Fun-Tiger + lazy evaluation，传入的函数的好处是不用立即返回值，最后我们讲延迟计算的语言。

## Fun-tiger

我们先讲 Fun-Tiger，我们需要加入一些 grammar rules，让它支持函数式。首先要加入箭头（marker）。

复习一下在语法翻译中，有翻译方式。

$\text{ty} \rightarrow \text{ty} \rightarrow \text{ty}$  (左边的箭头就是 parsing 的箭头，而右边的箭头代表返回值是一个 ty)

$\rightarrow (\text{ty}, \{, \text{ty}\}) \rightarrow \text{ty}$  (多参数的 ty，返回一个 ty)

$\rightarrow () \rightarrow \text{ty}$  (没有参数的 ty)

## Grammar rules for Fun-Tiger

---

- First add function types to Tiger

- "arrows" are markers for function types
- Left: parameters; right: return value

|                                                                 |                       |
|-----------------------------------------------------------------|-----------------------|
| $\text{ty} \rightarrow \text{ty} \rightarrow \text{ty}$         | (single parameter)    |
| $\rightarrow (\text{ty} \{, \text{ty}\}) \rightarrow \text{ty}$ | (multiple parameters) |
| $\rightarrow () \rightarrow \text{ty}$                          | (no parameter)        |

- The arrow operator is right-associative

- What is the following type means?
  - $(\text{int} \rightarrow \text{int}) \rightarrow \text{int} \rightarrow \text{int}$

除了我们要加入类型相关的内容以外，我们还需要加入 call expression 相关的东西。

## Grammar rules for Fun-Tiger

---

- The CALL expression should also be extended

- Original: only ID (name) can be used for calling
- Now: arbitrary expressions (examples later)
- What if expressions are not functions? (solved by semantic analysis)

originally ID

|                                                                   |
|-------------------------------------------------------------------|
| $\text{exp} \rightarrow \text{exp} (\text{exp} \{, \text{exp}\})$ |
| $\text{exp} \rightarrow \text{exp} ()$                            |

我们引入了高阶函数后，会有一些比如说。比如说 `var x = g(y);` 此时 `x` 的类型是一个函数；又比如拿函数返回值直接调用一个函数。现在任意的返回值都有可能是一个函数，所以现在要把 `call` 的返回值从原先的 `ID` 调整为一个 `expression`。但是现在把这个条件放太宽了，语法这里是可以通过的，具体检查下放到语义分析。

下面有一个例子：变量的名字就是它的意义。

```
let
 type intfun = int -> int //参数是 int，返回值 int 的函数类型。
 function add(n: int): intfun = //函数作为返回值类型
 let function h(m: int): int = n + m
 in h //返回了 h 这个函数本身
 end
 var addFive : intfun := add(5) //定义了一些 intfun 的实例
 var addSeven : intfun := add(7)
 var twenty := addFive(15) //使用函数实例真正获得 int 值，这里是 20
 var twentyTwo := addSeven(15) //使用函数实例真正获得 int 值，22
 function twice(f: intfun) : intfun =
 let function g(x: int) : int = f(f(x)) //把传入的 intfun f 调用 2 遍
 in g
 end
 var addTen : intfun := twice(addFive) // f(x)=x+5, f(f(x))=x+10
 var seventeen := twice(add(5))(7) // f(f(7)) = 17
 var addTwentyFour := twice(twice(add(6))) // f(x)=x+6, f(f(x))=x+12, f(f(f(x)))=x+24
in addTwentyFour(seventeen)
end
```

这似乎是把计算之间的中间过程做了一个 `snapshot` 存了下来。这个就把之前我们函数的表达能力拓宽了。

这里就讲完了函数基础。

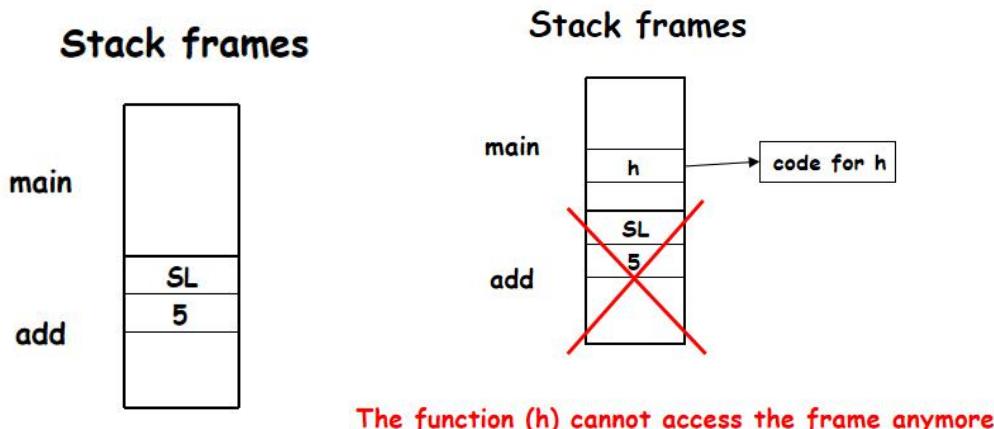
## 闭包 (Closure)

### Why higher-order is more interesting with nested?

- Consider a language without nested functions
  - We can use function pointers
    - Like C: `void (*fun)()`
  - Invocation is achieved by calling to the address
    - X86: `call *%rax`
- However, it doesn't work for nested functions
  - A child function can live beyond the outer ones
  - How can it access the variables in outer frames?

C 语言是没有函数嵌套的，不支持这种语法。C 语言的好处就是接近底层的东西不需要有太多的 feature，函数指针具体到 x86 上就是一个间接的调用。虽然它有 high-order function，但是处理起来很简单。但是 nested function 就比较复杂，child function 可以在调用完存活，所以里面的状态是不能删除掉的。

比如之前的 `var addFive : intfun := add(5)`，我们不考虑寄存器传值，把 static link 和 5 都放在栈上。然后 add 做完返回 h 存到 main 变量的 addFive 中。因为调用的时候会直接调用函数变量的地址的代码。

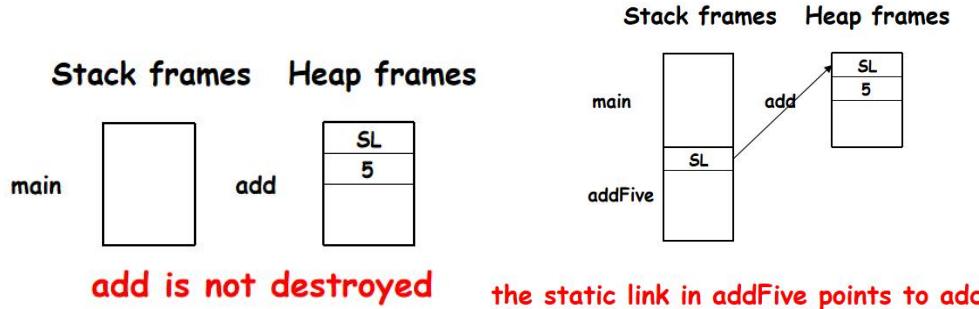


关键问题是 add 退出了，它的栈帧就不复存在了。所以在这个地方有一个问题，h 是有一个储值 5 的，但是我们现在只有 h 的代码，这是不足以我们去调用的，因为 5 原本是存在 add 里的，而不是存在 h 里的。所以 5 不能在 add 退出时就删掉，5 应该和 h 的 lifecycle 是一致的。

所以我们要引入一个新的机制来存储 5，也就是 closure。

Closure 的定义就包含了 function code 的 pointer 以及如何访问 non-local variable (or environment)。Closure 是一个复杂结构，因为又有 code 和 data。因为 data 并不在运行时栈上。

它的实现方法有很多种，比如我们在编译的时候有一个全局变量 m，在构造的时候把类传进去。但在 tiger 中大家比较熟悉 static link，所以这里我们还是使用 static link 来做。



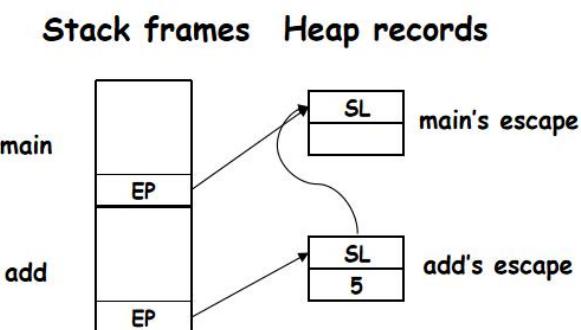
首先一个修改是 activation record 我们是放在堆上了。所以 Add 退出以后，其 frame 不会再被销毁掉。这里有一个问题就是说，栈上的 frame 通过 push 和 pop 就自动回收了，但是堆上的 frame 不会自动回收，我们需要 GC 去回收它们。

在上例中 heap frame 和 h 的 lifecycle 一致的。

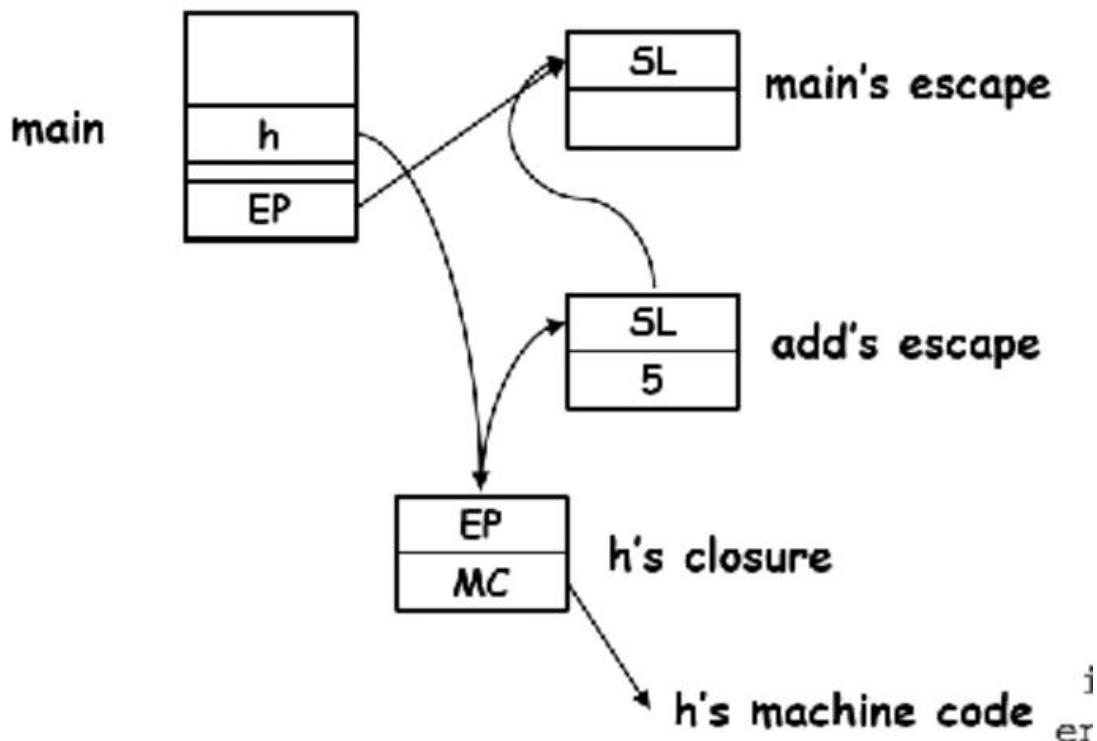
如果我们只有一个指针的话，就可以用简单的 reference count，但是如果 heap frame 中还有一些复杂的连接，可能就不太方便。

堆分配 frame 的缺点就是 frame 可能比较大，对于一个复杂函数可能要把整个栈帧都放到堆上去会导致内存占用变多。我们希望只存放一些 necessary information，也就是我们只需要存放逃逸变量就可以了。

为什么是逃逸变量呢？因为下次我们使用函数实例（addFive）的时候，其实我们看 h 本身之外的逃逸变量，所以其实也就是 static link 和 n。所以我们只需要 escape variable 放到堆上的 frame 里去。所以我们栈帧现在分成了两份，一个是原来正常的栈帧，所有的 escape 和 staticlink 就全部放到 堆上去了。我们使用 escape pointer (EP) 来指向堆上的 frame。

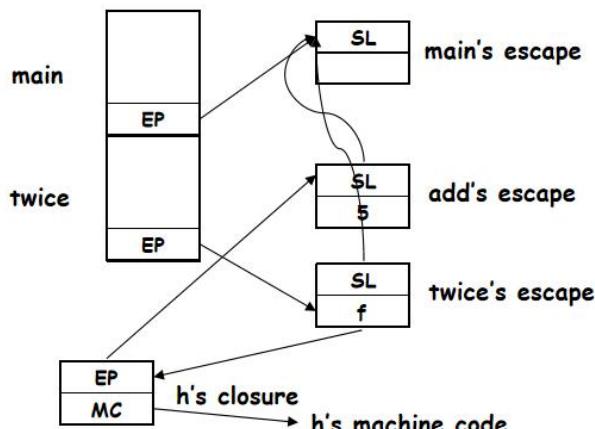


## Stack frames    Heap records



在这样一个架构下，Add 退出的时候不会影响堆上的结构。因为在调用 Add 的时候也会创建堆上的对应逃逸变量的帧，这个是不会随着 add 的退出而回收的。

## Stack frames    Heap records



所以看到这就是 closure 很明显的展示，既指向了代码，也指向了 escape record。所以我们看 `twice` 的时候，传入的 `f` 本身是 escape 的变量，我们必须把 `f` 保存起来，而 `f` 传入的是 `addFive`，所以 `f` 要执行刚才生成的 Closure。这样的话，通过 escape pointer 和 static link 的配合，就可以实现 closure 方法。当然这部分就不需要大家实现了，但是不排除期末会考这样的题目。

这个其实就是在 static link 上划分成两部分去做。

## Modifications to the Tiger compiler

---

- Adding EP to the frame structure
  - All static-link-related operations now rely on EP (not FP)
  - EP itself is a non-escaping local temporary
    - Other frames do not directly access it
- Formals/locals creation should be modified
  - Offsets should be based on EP
  - Escaped variables should be allocated on heap
    - In procEntryExit1

EP 应该是需要一个固定的 slot，现在所有和 static link 相关的东西都需要通过 EP 指向的地方去找。EP 本身是不 escape 的，否则会导致死循环。如果 EP 是固定位置的话，也可以通过对 EP 加减 offset 来得到栈上的位置。

这就讲完了 closure，它的核心就是 code 和 data，有时候光有代码是不足以执行的，我们把数据和代码放在一起变成一个更大的抽象就是 closure。

这就讲完了 Fun-Tiger，它是一个非常 minimize 的函数式语言。

## PureFun-Tiger

如果我们想实现 equational reasoning，Fun-Tiger 是不够的。因为如果在 Fun-Tiger 的高阶函数中修改了 escape 变量，那么就可能导致 side-effect。

所以纯函数语言要求在可观测的范围内，我们不能修改函数的状态。这种状态就做不可变形。

Scala 是基于 JVM 的函数式语言，第一节课讲的就是不可变型。

## Immutable Variables (不可变形)

要实现纯函数语言，除初始化以外不能给变量赋值；不能给堆分配的 record 赋值。

最后一个问题是比较 tricky 的问题，external function 不能有可见的影响，比如 print, flush, getchar, exit 等。函数式希望的是 getchar() 不管被怎么调用都是一样的值，但是 getchar() 每次返回的都是用户输入的新的值，所以函数式也要解决 external function 的 side-effect。

其实就是返回值会变的情况，我们返回 void 即可，来避免可见的 side-effect。

不能对新的变量赋值，我们只能使用 copy-on-write。

例子：二叉搜索树

## imperative

```

type key = string
type binding = int
type tree = {key: key,
 binding: binding,
 left: tree,
 right: tree}

function look(t: tree, k: key)
 : binding =
if k < t.key
 then look(t.left,k)
else if k > t.key
 then look(t.right,k)
else t.binding

```

## functional

```

type key = string
type binding = int
type tree = {key: key,
 binding: binding,
 left: tree,
 right: tree}

function look(t: tree, k: key)
 : binding =
if k < t.key
 then look(t.left,k)
else if k > t.key
 then look(t.right,k)
else t.binding

```

This part is the same!

34

```

function enter(t: tree, k: key,
 b: binding) =
if k < t.key
 then if t.left=nil
 then t.left := tree{key=k,
 binding=b,
 left=nil,
 right=nil}
 else enter(t.left,k,b)
else if k > t.key
 then if t.right=nil
 then t.right := tree{key=k,
 binding=b,
 left=nil,
 right=nil}
 else enter(t.right,k,b)
else t.binding := b

```

**direct assignment**

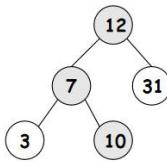
```

function enter(t: tree, k: key,
 b: binding) : tree =
if k < t.key
 then tree{key=t.key,
 binding=t.binding,
 left=enter(t.left,k,b),
 right=t.right}
else if k > t.key
 then tree{key=t.key,
 binding=t.binding,
 left=t.left,
 right=enter(t.right,k,b)}
else tree{key=t.key,
 binding=b,
 left=t.left,
 right=t.right}

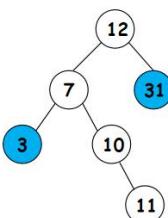
```

**reconstruction**

## origin



## returned



Unused nodes can be deleted  
Shared nodes are kept

在函数式中，我们一个 enter 修改了 4 个节点，开销还是比较高的。

## Continuation-based I/O

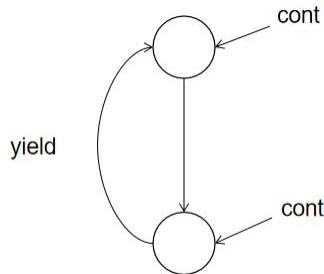
### Recap: the prohibited behaviors

- Assignments to vars
  - Assignments to heap records
  - Calls to external functions
- ?      reconstruction
- 
- External functions are usually I/O related
    - E.g. print, getchar
    - How to make them 'functional'?
    - Solution: continuation-based I/O

所以对于变量和堆上的 record 的不能赋值的限制，我们可以通过重构来做，那么对外部函数的调用怎么办呢？

尤其是 IO 相关的外部函数去做函数式是比较麻烦的。解决方法就是：continuation-based I/O.

continuation 就是把一个程序的控制流（control state）用一个抽象的方式封装起来了。它有点像进程、线程中的上下文的概念。有人把它比喻成时空穿越的一个点，我们在某一个时空的状态下把状态保存下来，之后我们可以切换回去继续执行。



比如我们原本在这个状态执行，我们把这个状态用一个 cont 记录下来，继续执行下去，之后我们可以通过一个方法（如协程中的 yield）跳转回去再执行一次。在 functional 中，限制比较多，回去的状态还是原来的状态。

在我们这里 continuation 怎么用呢？其实我们就是生成一个固定的 return，我们之前可能用的是 void，但是 void 不是一个很好的 return value，我们这里就用了 answer。它也没有什么特定的意义，就是用来做特定返回值的类型。这样我们的函数就永远返回一个值了，这样我们就把 side-effect 去掉了。因为我们返回的可见的值永远就是相同的值了。

那对我们 Pure-Func tiger 呢，我们需要加一个 answer 类型，我们就可以理解为之前的 void，没什么特殊的含义。

```

type answer
type stringConsumer = string -> answer
type cont = () -> answer

function getchar(c: stringConsumer) : answer
function print(s: string, c: cont) : answer
function flush(c: cont) : answer
function exit() : answer

```

它只是表示了程序的返回值，无论什么时候调用这些函数，返回值都是一样的。

除了 `answer` 还有一个很重要的 `consumer` 类型，它是来消费 `input` 的。所以 `consumer` 它是一个函数，传入参数类型是 `string`，返回类型为 `answer`，把处理 `input` 的逻辑包到这个函数里，然后传递给 `getchar` 去处理。

所以现在的 `getchar` 就像一个工具人一样，它受命去调用这个函数，调用完返回 `answer` 就可以了，至于参数怎么存、怎么传都是 `stringConsumer` 自己决定的。因为我们规定是函数没有状态，而不是传入的参数没有状态，所以它把函数里可能有的东西都丢给了参数去做，这样函数保证了自己是无状态的，可以通过让参数去做这些事情。

所以当我们发现有些逻辑不能实现完全的函数式的话，我们就把它包成参数传进来，作为函数外部的一部分来处理。

当然还有一个是 `continuation` 类型，也就是我们调用一下返回一个 `answer`。它就是来生成 `return value` 的。

`print` 现在也是会输出一个 `string`，输出完以后调用一下 `cont` 类型返回一个 `answer`。`exit` 其实最简单，我们不做什么事情直接返回 `answer` 就行了。

这里我们就来讲一个比较复杂的例子：

```

let
type intConsumer = int -> answer
function isDigit(s : string) : int =
 ord(s)>=ord("0") & ord(s)<=ord("9")

function [getInt]done: intConsumer) =
let function nextDigit(accum: int) =
 let function eatChar(dig: string) =
 if isDigit(dig)
 then nextDigit(accum*10+ord(dig))
 else done(accum)
 in getchar(eatChar)
 end
in nextDigit(0)
end

function [putInt](i: int, c: cont) =
if i=0 then c()
else let var rest := i/10
 var dig := i - rest * 10
 function doDigit() = print(chr(dig), c)
 in putInt(rest, doDigit)
 end

function factorial(i: int) : int =
if i=0 then 1 else i * factorial(i-1)

function loop(i) =
if i > 12 then exit()
else let function next() = getInt(loop)
 in putInt(factorial(i), next)
 end
in
 getInt(loop)
end

```

首先我们传入的是 `int`，所以有一个 `intConsumer` 传入 `int` 返回 `answer`。然后有两个 IO 相关的函数 `getInt` 和 `putInt`。

首先是 `getInt`，它里面声明了 `nextDigit` 是拿下一个 `digit`。而 `nextDigit` 声明了 `eatChar`，里面调用了 `getchar`。`eatChar` 就是声明了一个 `digit`，返回要么是 `nextDigit`，要么是 `done`。`done` 就代表读到了非数字的字符，我们就进到 `loop` 中去执行。

在 `loop` 中，如果 `i` 超过了 12，那么就会调用 `exit()` 退出，否则就会调用 `putInt`。而 `putInt` 是接受 `next` 这个 `continuation`。`next` `continuation` 会不断地往里传。首先 `putInt` 的任务是输出这个传入的 `integer`，如果 `i = 0`，它就会调用 `continuation`，它就去继续做左边的 `getInt` 去了，

否则它就要去 print 这个字符。它继续把 i 分解成更小的数字，然后用 doDigit 作为 continuation 继续往下递归去 putInt。

我们假设最开始的传入是 putInt(24, getInt(loop)) 的情况：

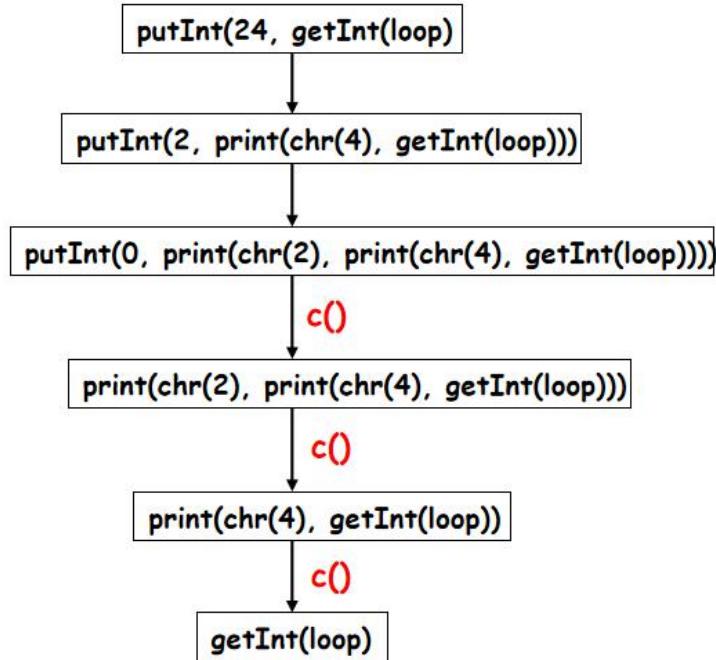


图 1 输入  $i = 24$  的控制流

这个程序不断地 `getint` 和 `putint`，原因它用 `continuation` 把 `print` 给串了起来。通过 `continuation` 的连续调用直接输出了。所以我们可以认为 `continuation` 是把控制流保存起来了。可以看到这个已经像是函数调用的一个关系了。这样一个 `continuation` 表示的东西就是先要调用 `print` 输出 2，再调用 `print` 输出 4，最后调用 `getInt`。并且这个 `continuation` 还可以不执行，等到我们调用的时候再一个个执行。其实就是生成一个函数调用链再执行这个函数调用链的过程。

对于 `putInt` 这个函数本身来说，它不产生什么 `side-effect`，每次调用以后都只返回一个 `answer`。

## 2021/12/21

前面讲完 `copy-on-write` 和 `continuation` 以后，我们就基本上知道了 Pure-Fun Tiger 是怎么实现的。如果 Tiger 要实现这个要改什么东西，比如我们要添加上节课提到的 `answer`、`consumer`、`continuation` 类型，包括说我们的一个过程现在返回的是 `answer`。

最后是 `assignment`、`for`、`while`、`compound statement (seqEXP)` 要删除。

`assignment` 比较好理解，它原先是对变量赋值，现在我们只允许对变量的初始化，我们就只需要保留 `let` 就可以了，`for loop` 是因为我们不能做 `i++` 了；而 `while` 循环是因为我们在循环条件中往往是判断 `i` 是否等于某个值，当我们 `i` 不能修改的时候，这个条件只可能永真或永假。要实现循环，我们需要通过递归的形式去实现。而在 `compound statement` 中，

我们说前面是一个赋值语句，后面可以加分号。这个也可以被删掉。

从语言上来说，pure-fun tiger 相对于 tiger 修改的不是特别多。因为高阶函数对 tiger 修改是比较的，而 equational reasoning 更多的影响是内部的，而不是外部的。所以整体语言变化不是很大，很多优化方法，比如寄存器分配算法，还是可以正常使用的。

当我们写 pure-fun 的时候，控制流会复杂很多，原先只传变量，现在我们可以传 function call 进来。

引入函数变量了以后，我们就很难知道函数的信息，可能就只能做一些保守的优化了。

### Pure functional language optimizations

- Are there any specific languages for pure functional languages?
  - The answer is YES, by using equational reasoning
- Consider the following example:

```
type recrd = {a: ..., b: ...}
a1 (immutable) -> 5
b1 (immutable) -> 7
r (immutable) -> {a=5, b=7}
y <- 5+7 = 12 (constant propagation)
var a1 := 5
var b1 := 7
var r := recrd{a := a1, b := b1}
var x := f(r)
var y := r.a + r.b
```

好处的话扩大了常数传播的范围，因为这些值都是常数。

## inline expansion

我们就主要讲一下，怎么去优化 pure-fun tiger。我们直接看如下的一个例子：

### The example used in this part

```
let
 type list = {head: int, tail: list}
 type observeInt = (int, cont) -> answer

 function doList(f: observeInt, l: list, c: cont) =
 if l=nil then c()
 else let function doRest() = doList(f, l.tail, c)
 in f(l.head, doRest)
 end

 function double(j: int) : int = j+j

print twice: i, i+i
suppose list = {1,2}
output: 1,2,2,4

function printDouble(i: int, c: cont) =
 let function again() => [putInt(double(i),c)
 in putInt(i, again)]
 end

function printTable(l: list, c: cont) =
 doList(printDouble, l, c)

var mylist := ...
in printTable(mylist, exit)
end
```

这个函数就是对于一个 list，对于其中的每个元素，我们都输出元素和元素 \* 2。如果我们用 imperative style 来写，我们只需要一个 for 循环就可以了。但是函数式里面没有循环，

所以我们用的都是递归的方式来做了，比如我们的 `doList`。导致控制流比较混乱，功能也被分散到了各个函数去做。

我们先讲内联的思路先做一层优化。不知道大家有没有听说过这个词，也就是把函数体直接嵌入到我们调用的地方。

### Yet another important optimization

- Observation: functional programs tend to use many small functions
  - Loops are not used; replaced by recursive functions
  - Too many invocations cause performance overhead
- Solution: inlining function invocations
  - Replacing a function call with a copy of a function body
  - The resulting program can be as efficient as a imperative one

比如说函数 `f` 调用函数 `g` 的时候，我们直接把 `g` 的 `body` 里的内容放到 `f` 调用 `g` 的位置，这样就没有 `call` 这个动作了。比如说 c 和 c++ 中可以添加 `inline` 关键字，让编译器帮我们优化。

好处就是 `call` 被去掉了，省掉了一次 `call-return` 的开销。通过这个优化我们就可以接近 `imperative` 的性能。

这里要注意的一点就是 `variable` 的 `capture`。也就是可能出现外面函数变量和里面函数变量名字一样的问题。

```
let var x := 5
 function g(y: int): int =
 y+x
 function f(x: int): int =
 g(1)+x
 in f(2)+x
end
```

## Note: avoiding variable capture

- Outer variables can have the same name with inners
- Suppose we want to inline  $g(1)$  in  $f$

```
let var x := 5
 function g(y: int): int =
 y+x
 function f(x: int): int =
 g(1)+x
 in f(2)+x
end
```

## Note: avoiding variable capture

- Outer variables can have the same name with inners
- Suppose we want to inline  $g(1)$  in  $f$ 
  - The result is  $1+x+x$
  - After inlining:  $f(2) = 1+2+2=5$  (before is  $1+5+2=8!$ )
  - Key reason: two "x"s are different!

```
let var x := 5
 function g(y: int): int =
 y+x
 function f(x: int): int =
 1+x+x
 in f(2)+x
end
```

19

在做内联的时候，其实我们不会再回去检查这件事情了，所以这就会出现不同命名体系的冲突。一种方法就是重新命名冲突的参数，还有一种方法更旁边一些，我们可以在 IR 翻译的时候给每个变量翻译成不同的名字。

## Algorithm for inline expansion

(a) When the actual parameters are simple variables  $i_1, \dots, i_n$ .

Within the scope of:

function body  
function  $f(a_1, \dots, a_n) = B$

the expression

$f(i_1, \dots, i_n)$

rewrites to

$B[a_1 \mapsto i_1, \dots, a_n \mapsto i_n]$

(b) When the actual parameters are non-trivial expressions, not just variables.

Within the scope of:

function  $f(a_1, \dots, a_n) = B$

the expression

$f(E_1, \dots, E_n)$

rewrites to

```
let var $i_1 := E_1$
 :
 var $i_n := E_n$
in $B[a_1 \mapsto i_1, \dots, a_n \mapsto i_n]$
end
```

where  $i_1, \dots, i_n$  are previously unused names.

所以我们其实就把 function body 中的变量替换掉即可。如果实参是一套 expression 的话，我们也可以定义出一套常数，每个值为 expression 算出来的结果替换掉它们。

rewrite 的坏处就是我们最好在 AST 层去做。我们就要先加一个 parse 把变量名字去掉。

## Back to the original example

- Suppose we want to inline the  $double(i)$  invocation

```
...
function double(j: int): int = j+j
function printDouble(i: int, c: cont) =
 let function again() = putInt(double(i), c)
 in putInt(i, again)
end
...
```

## Back to the original example

- Suppose we want to inline the  $double(i)$  invocation
  - Following (a): replace  $i$  in  $j+j \rightarrow i+i$

```
...
function double(j: int): int = j+j
function printDouble(i: int, c: cont) =
 let function again() = putInt(i+i, c)
 in putInt(i, again)
end
...
```

我们只需要在函数体里把  $j+j$  换成  $i+i$  就行了。

### A slightly different example

- The invocation now becomes `double(g(i))`
  - We do not want to inline the `g` function
  - Following (a): replace `g(i)` in `j+j`

```
...
function double(j: int): int = j+j
function printDouble(i: int, c: cont) =
 let function again() = putInt((g(i)+g(i)), c)
 in putInt(i, again)
end
function g(x: Int) = ...
...
```

如果我们函数有副作用，可能就会使结果不一样。所以在 imperative 中不一定能做这样的优化，但是在 functional 中是可以这样做的。

### Dead function elimination

- Similar to dead code elimination (DCE)
  - Useless functions can be removed
  - The previous example: `double()` can be removed

```
...
function double(j: int): int = j+j
function printDouble(i: int, c: cont) =
 let function again() = putInt(i+i, c)
 in putInt(i, again)
end
...
```

一旦 inline 这个函数了，没有人在使用它了，我们就可以把它删掉，从而减少我们汇编的长度。

### Inlining recursive functions

- Following (a), we can inline `doList` in `printTable`

```
...
function doList(f: observeInt, l: list, c: cont) =
 if l = nil then c()
 else let function doRest() = doList(f, l.tail, c)
 in f(l.head, doRest)
 end
function printTable(l: list, c: cont) =
 doList(printDouble, l, c)
...
```

递归函数可能就复杂一点。

## Inlining recursive functions

- Following (a), we can inline doList in printTable
  - But invocations to doList still exist
  - We only inline "the first iteration"

```
...
function printTable(l: list, c: cont) =
 if l = nil then c()
 else let function doRest() =
 doList(printDouble, l.tail, c)
 in printDouble(l.head, doRest)
 end
...
...
```

简单来说就是展开一层，把 doList 的逻辑全部放到 printTable 中来。但是我们没有办法做 function elimination。这样 doRest 就被重复地声明了，所以这里的内联只减少了一次调用。后面的 doRest 的调用还是走的原本的 doList 函数，所以我们希望可以内联的更多。

这里也提供了一个办法，就是把函数做了一个简单的转换。

## How to entirely inline recursive functions?

- Simply inline them does not work
  - They can have infinite iterations
  - We need to refactor/customize them
- Algorithm: loop-preheader transformation
  - Key idea: split the function into nested two
  - The outer one serves as a "hook" (or prelude)

```
function f(a1, ..., an) =
 B
```

→

```
function f(a'1, ..., a'n) =
 let function f'(a1, ..., an) =
 B[f ↦ f']
 in f'(a'1, ..., a'n)
 end
```

简单来说就是把这个函数分成 nested 的两个函数。

## How does this algorithm work?

- Still use doList as an example

```
function doList(f: observeInt, l: list, c: cont) =
 if l = nil then c()
 else let function doRest() = doList(f, l.tail, c)
 in f(l.head, doRest)
 end
```



```
function doList(fX: observeInt, lX: list, cX: cont) =
 let function doListX(f: observeInt, l: list, c: cont) =
 if l = nil then c()
 else let function doRest() = doListX(f, l.tail, c)
 in f(l.head, doRest)
 end
 in doListX(fX, lX, cX)
end
```

33

变换之后的 doListX 其实就是变化之前的 doList。

## Inline expand after transformation

- Now if printTable inlines doList:
  - Following (a)
    - Note: the formal parameters do not need to change

```
...
function printTable(l: list, c: cont) =
 let function doListX(f: observeInt, l: list, c: cont) =
 if l = nil then c()
 else let function doRest() = doListX(f, l.tail, c)
 in f(l.head, doRest)
 end
 in doListX(printDouble, l, c)
...
```

这样以后我们再去 inline 就容易了。这样我们就把整个 doList 全都 inline 进来了。

我们还发现 doListX 一直在调用 printDouble，一直在 doListX 里传递这个 f。

## Inline expand after transformation

- Is it good enough?
  - Perhaps more potential optimizations
  - E.g., f is always printDouble in doListX

```
...
function printTable(l: list, c: cont) =
 let function doListX(f: observeInt, l: list, c: cont) =
 if l = nil then c()
 else let function doRest() = doListX(f, l.tail, c)
 in f(l.head, doRest)
 end
 in doListX(printDouble, l, c)
...
...
```

我们是否能让 printDouble 也不要传来传去，因为传递参数也是有开销的。参数如果放栈上就更麻烦了，要压栈。这就是 loop-invariant argument。

## Loop-invariant arguments

- doListX has two unchanged arguments in every calls
  - f (always fX); c (always cX)
  - They are **invariants** during recursive invocations

```
function doList(fX: observeInt, lX: list, cX: cont) =
 let function doListX(f: observeInt, l: list, c: cont) =
 if l = nil then c()
 else let function doRest() = doListX(f, l.tail, c)
 in f(l.head, doRest)
 end
 in doListX(fX, lX, cX)
end
```

在 ICS 中，我们有提到过 `for (int i = 0; i < strlen(s); i++)`，这个 `strlen` 会成为一个性能瓶颈，我们需要提取到外面来。这里的话，编译器也是可以做这种优化的，我们知道每次调用都是一样的结果，我们就可以把它往外移。因为在函数式中我们没有循环了，所以这里的循环不变量其实就是递归不变量。

## Loop-invariant hoisting

Algorithm: if every use of  $f'$  within  $B$  is of the form  $f'(E_1, \dots, E_{i-1}, a_i, E_{i+1}, \dots, E_n)$  such that the  $i$ th argument is **always**  $a_i$ , then rewrite

```
function f(a'_1, ..., a'_n) =
let function f'(a_1, ..., a_n) = B
in f'(a'_1, ..., a'_n)
end → function f(a'_1, ..., a'_{i-1}, a_i, a'_{i+1}, ..., a'_n) =
let function f'(a_1, ..., a_{i-1}, a_i, a_{i+1}, ..., a_n) = B
in f'(a'_1, ..., a'_{i-1}, a'_{i+1}, ..., a'_n)
end
```

moved out

where every call  $f'(E_1, \dots, E_{i-1}, a_i, E_{i+1}, \dots, E_n)$  within  $B$  is rewritten as  $f'(E_1, \dots, E_{i-1}, E_{i+1}, \dots, E_n)$

所以就有 hoisting 操作，也就是可以把内部参数往外拉，这样  $f'$  就少一个参数。

### Loop-invariant hoisting in our example

- Moving  $f$  and  $c$  out
  - Replacing the original name:  $fX$  and  $cX$

```
function doList(fX: observeInt, lX: list, cX: cont) =
let function doListX(f: observeInt, l: list, c: cont) =
if l = nil then c()
else let function doRest() = doListX(f, l.tail, c)
in f(l.head, doRest)
end
in doListX(fX, lX, cX)
end
```

### Loop-invariant hoisting in our example

- Moving  $f$  and  $c$  out
  - Replacing the original name:  $fX$  and  $cX$
  - Now  $doListX$  has only one parameter ( $l$ )

```
function doList(f: observeInt, lX: list, c: cont) =
let function doListX(l: list) =
if l = nil then c()
else let function doRest() = doListX(l.tail)
in f(l.head, doRest)
end
in doListX(lX)
end
```

这样我们的内联就可以更近一步。在我们内联之后就不用传递重复的信息了。

## Loop-invariant hoisting in our example

- Moving f and c out
  - Replacing the original name: fX and cX
  - Now doListX has only one parameter (l)
  - The inlined result also changes

```
...
function printTable(l: list, c: cont) =
let function doListX(l: list) =
 if l = nil then c()
 else let function doRest() = doListX(l.tail)
 in printDouble(l.head, doRest)
 end
 in doListX(l)
...

```

41

每次 inline 之后，我们可能会找到更多的优化机会。之前的情况里，doListX 里调用的是 f，我们没办法做进一步的优化。现在我们有了 loop-invariant hoisting 了以后，我们发现中间永远调用的是 printDouble。我们就可以进一步内联展开。

## Cascading inlining

- The result for inline expanding
  - Can we do more inlining? (sure, consider putInt)

```
...
function printTable(l: list, c: cont) =
let function doListX(l: list) =
 if l = nil then c()
 else let function doRest() = doListX(l.tail)
 in let var i := l.head
 in let function again() = putInt(i+i, doRest)
 in putInt(i, again)
 end
 end
 end
 in doListX(l)
...

```

43

putInt 也可以继续 inline，随着这个 inline，函数体会越来越大，但是 call 的次数会少了。

## Unnesting Let

最后一个优化点是 let。在我们不断优化过后，可能有一系列嵌套的 let。

## Another opportunity: unnesting lets

- Nested lets can be simplified

```
let dec1 in let dec2 in exp end end = let dec1 dec2 in exp end
```

- In our example: three nested lets

```
function printTable(l: list, c: cont) =
 let function doListX(l: list) =
 if l = nil then c()
 else let function doRest() = doListX(l.tail)
 in let var i:= l.head
 in let function again() = putInt(i+i,doRest)
 in putInt(i,again)
 end
 end
 end
 in doListX(l)
end
```

44

我们简单做一个优化。

## Another opportunity: unnesting lets

- Nested lets can be simplified

```
let dec1 in let dec2 in exp end end = let dec1 dec2 in exp end
```

- The optimized version (neat in source code)

```
function printTable(l: list, c: cont) =
 let function doListX(l: list) =
 if l = nil then c()
 else let function doRest() = doListX(l.tail)
 var i:= l.head
 function again() = putInt(i+i,doRest)
 in putInt(i,again)
 end
 in doListX(l)
 end
```

45

好处就是把 scope 减少了，现在我们可以把它们放到一个 scope 里去。

## Tradeoff for inline expansion

- Inline expansion can improve the performance
  - Call instructions are removed
  - Less register saving/restoring
- However, it makes the program bigger
  - A function body might be copied in multiple callers
  - Can cause code explosion if handled incorrectly

最后再介绍一下 inline expansion 的 tradeoff，它的核心目的是优化性能，第二个目的是减少编译出来的代码大小。如果我们无休止地进行内联的话，我们的代码量可能迅速增长。那么怎么去控制这件事情呢？这边提供了几种方法。

## **Heuristics to control inlining**

---

- Expand only call sites frequently executed
  - The most cost-effective
  - Frequency can be determined by static analysis or dynamic profiling
  
- Expand functions with small bodies
  - Some functions (getter/setter) are even smaller than the call-related instruction sequence
  - A smart compiler automatically inlines them

哪些频繁执行，哪些就内联。静态分析有可能也能分析 frequency，也就是数一数一个函数被多少函数调用，但是这个可能不太精准。

第二种主要就是静态的，比如 getter/setter 这种简单函数就很容易被优化掉。

## **Heuristics to control inlining**

---

- Expand only call sites frequently executed
  
- Expand functions with small bodies
  
- Expand functions called only once
  - The original body can be removed by DCE
  - The program size may not increase

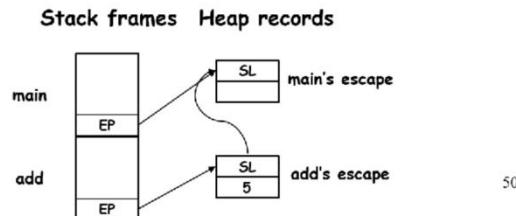
如果一个函数只被调用一次，那么我们就可以把它消除掉，可以让我们的程序更加简洁。

## **Closure Conversion**

之后就是讲 closure conversion。

## Recap: what is a closure?

- A closure combines
  - A function pointer (to its machine code)
  - A means of accessing non-local (free) variables
- We use heap-allocated records to implement closures



50

高阶函数就需要 closure 提供环境信息使用。closure 就会比一般的函数要复杂。

### **Closures are complicated than normal functions**

- It contains not only code but also data
  - Making it harder for backend analysis
- So we need a closure conversion phase
  - So none of the functions appears to access free variables
  - Basic idea: turning free variable access into formal parameter access

原则就是我们尽量不想要有堆上的变量访问，而是把它们作为参数的访问。

## Formal rewriting rules

---

- Given a function  $f(a_1, \dots, a_n) = B$  at nesting depth  $d$  with escaping variables  $x_1, x_2, \dots, x_n$  and non-escaping variables  $y_1, y_2, \dots, y_n$ ; rewriting into:

$f(a_0, a_1, \dots, a_n) = \text{let var } r := \{a_0, x_1, x_2, \dots, x_n\} \text{ in } B' \text{ end}$

- $a_0$ : static link
- $r$ : a record containing all escaping variables and static link
  - Acting as the static link when calling functions of depth  $d+1$

so

所以 rewriting rule 如上图所示, 首先 non-escaping 的变量不需要变, 我们去定义一个 record, 成员包含了 SL 和其余的逃离变量。在定义了这个之后再去调用原本的 body。之后我们所有对于 escape 的调用就不需要从堆上找了, 我们直接访问这个 record 即可。当我们想要找更早嵌套层次中所定义的变量的时候, 我们就可以从  $a_0$  中去访问。

### What about functions in the parameters?

---

- They were represented as closures before
  - We create records on heap to store them
- After closure conversion, we split them into two
  - One parameter as the code pointer
  - Another as the converted static link
    - Containing escaping variables

如果参数里有函数呢? 这些函数可能也是 closure, 我们不需要太担心, 因为 rewrite 是对所有函数做 rewrite, 所以函数都有 escape record。

## An example: printTable (before conversion)

```
function printTable(l: list, c; cont) =
 let function doListX(l: list) =
 if l=nil then c()
 else let function doRest() =
 doListX(l.tail)
 var i := l.head
 function again() =
 putInt(i+i, doRest)
 in putInt(i, again)
 end
 in doListX(l)
end
```

这些函数因为有 escape 变量，我们都要做成 closure，就会复杂很多。

## An example: printTable (after conversion)

```
type mainLink = { ... }
type printTableLink = {SL: mainLink, cFunc: cont, cSL: ?}
type cont = ? -> answer
type doListXLink1 = {SL: printTableLink, l: list}
type doListXLink2 = {SL: doListXLink1, i: int,
 doRestFunc: cont, doRestSL: doListXLink1}
split into two
function printTable(SL: mainLink, l: list, cFunc: cont, cSL: ?) =
 let var r1 := printTableLink{SL=SL, cFunc=cFunc, cSL=cSL}
 function doListX(SL: printTableLink, l: list) =
 let var r2 := doListXLink1{SL: printTableLink, l=l}
 in if r2.l=nil then SL.cFunc(SL.cSL)
 else let function doRest(SL: doListXLink1) =
 doListX(SL.SL, SL.l.tail)
 var i := r2.l.head
 var r3 := doListXLink2{SL=r2, i=i,
 doRestFunc=doRest, doRestSL=r2}
 function again(SL: doListXLink2) =
 putInt(SL.SL.SL, SL.i+SL.i,
 SL.doRest.func, SL.doRestSL)
 in putInt(SL.SL.i, again, r3)
 end
 in doListX(r1, l)
 end
```

因为 c 是一个 continuation 类型，是一个函数，所以我们要拆分成 cFunc 和 cSL，把它的 escape variable 都放到 cSL 里去。下一个我们就可以注意 staticlink，每一个函数进来的时候，第一个参数就是存了 staticlink 的 record。

```
link types
type mainLink = { ... }
type printTableLink = {SL: mainLink, cFunc: cont, cSL: ?}
type cont = ? -> answer
type doListXLink1 = {SL: printTableLink, l: list}
type doListXLink2 = {SL: doListXLink1, i: int,
 doRestFunc: cont, doRestSL: doListXLink1}
```

最后我们来讲一下 type，它是根据 rewrite 的结果来定的。cSL 这里为什么是一个问号

呢？因为 staticlink 不能用常规的 typechecking 来考虑了。

## What is a ? Type?

- The type of cont's static link is unknown
  - Marked as a question mark (?)
  - Because cont's static link has multiple functions

```
function printTable(SL: mainLink, l: list, cFunc: cont, CSL: ?) =
 let var r1 := printTableLink{SL=SL,cFunc=cFunc,cSL=CSL} from main()
 function doListX(SL: printTableLink, l: list) =
 let var r2 := doListXLink1{SL: printTableLink, l=l} acutally exit
 in if r2.l=nil then SL.cFunc(CSLSL)
 else let function doRest(SL: doListXLink1) =
 doListX(SL.SL, SL.l.tail)
 var i := r2.l.head
 var r3 := doListXLink2{SL=r2, i=i,
 doRestFunc=doRest, doRestSL=r2}
 function again(SL: doListXLink2) =
 putInt(SL.SL.SL, SL.i+SL.i,
 SL.doRest.func, SL.doRestSL)
 in putInt(SL.SL,i, again,r3)
 end
 in doListX(r1,l)
 end
 end
```

9

因为 exit 里我们知道没有什么内容，它可能对应的就是没什么内容的一个类型。

- '?' suggests converted programs cannot be type-checked in a normal fashion

这样 closure conversion 就讲完了，更多的就是我们不希望用堆上分配，而是函数内空间的内容来处理，这样便于我们优化。

## Efficient Tail Conversion

接下来我们讲尾调用消除。

## Functions in the tail position

---

- Formal definition:  $f(x)$  within the body of  $g(y)$  is in tail position if
  - Calling  $f$  is the last thing that  $g$  does before return
  - Calling to  $f$  is referred as 'tail call'
- For example,  $B_i$  are in tail but  $C_i$  are not
  - `let var x := C1 in B1 end`
  - `if C1 then B1 else B2`
  - $C_1 + C_2$

一个函数  $f(x)$  在  $g(y)$  return 之前做的最后一件事情，那么  $f$  就是  $g$  的 tail call。而  $C_1$  和  $C_2$  都不是，最后一件事情是把它们加起来。

### Why are tail calls important?

---

- Tail calls can be implemented more efficiently than ordinary ones
  - A function has the same return value as its tail call
  - Perhaps we can avoid pushing the return value to stack
- Optimization: implementing tail calls like a jump
  - Using jump instructions instead of calls

tail call 的特性和别的 call 不太一样。也就是尾递归返回的时候会有多次 return value 的返回，我们希望少做几次 return，把尾递归使用 jump 一样执行。inline 就是把函数体合在一起，没有 call 和没有 jump，直接换成新的函数的第一条指令去执行。

而这里保证了函数的完整性，可以减少一些内存操作。那么怎么实现 tail call 呢？它主要有两个任务

1. call 的时候把控制流给 callee
2. 当 callee 执行完之后，还需要把控制流返回给 caller。

## Implementing a tail call

---

- A tail call has two missions
  - Passing the control flow to callee
  - Returning from the caller (as it ends after the call)
- It contains four steps
  - Move actual parameters into argument registers
  - Restore callee-save registers
  - Pop the stack frame of the calling function
  - Jump to the callee
- It contains four steps removed by coalescing
  - ~~- Move actual parameters into argument registers~~
  - ~~- Restore callee-save registers~~
  - ~~- Pop the stack frame of the calling function~~
  - Jump to the callee (**cheap**) not required for simple functions

如果我们用递归的话，我们就可以这样做。对于简单函数，也不需要去做 callee-save register，只需要 jump 过去就可以了。

## Functional PL has many tail calls

---

- Functions are passed as parameters and invoked in the end of function body
  - Every call in printTable is a tail call!

```
function printTable(SL: mainLink, l: list, cFunc: cont, cSL: ?) =
 let var r1 := printTableLink{SL=SL,cFunc=cFunc,cSL=cSL}
 function doListX(SL: printTableLink, l: list) =
 let var r2 := doListXLink1{SL: printTableLink, l=l}
 in if r2.l=nil then [SL.cFunc] SL.cSL
 else let function doRest(SL: doListXLink1) =
 doListX(SL.SL, SL.l.tail)
 var i := r2.l.head
 var r3 := doListXLink2{SL=r2, i=i,
 doRestFunc=doRest, doRestSL=r2}
 function again(SL: doListXLink2) =
 putInt[SL.SL.SL, SL.i+SL.i,
 SL.doRest.func, SL.doRestSL]
 in putInt[SL.SL,i, again,r3]
 end
 in doListX(r1,l)
 end
```

68

所以即使我们做了内联，还是有很多小函数，其中有很多是 tail call。

## printTable after compilation

- All calls are replaced with jumps
  - Now it looks like a loop-based function

|             |                                                                                                     |
|-------------|-----------------------------------------------------------------------------------------------------|
| printTable: | allocate record r1<br>jump to doListX                                                               |
| doListX:    | allocate record r2<br>if l=nil goto doneL<br>i := r2.l.head<br>allocate record r3<br>jump to putInt |
| again:      | add SL.i+SL.i<br>jump to putInt                                                                     |
| doRest:     | jump to doListX                                                                                     |
| doneL :     | jump to SL.cFunc                                                                                    |

我们把它们全部换成 jump。

## Compared with an imperative version

```
let
 type list = {head: int,
 tail:list}

 function printTable(l: list) =
 while l <> nil
 do let var i := l.head
 in.putInt(i);
 putInt(i+i);
 l := l.tail
 end

 var mylist := ...
 in printTable(mylist)
end
```

|             |                                                                                               |
|-------------|-----------------------------------------------------------------------------------------------|
| printTable: | allocate stack frame<br>jump to whileL                                                        |
| whileL:     | if l=nil goto doneL<br>i := l.head<br>call putInt<br>add i+i<br>call putInt<br>jump to whileL |
| doneL:      | return                                                                                        |

这里给出了一个 imperative 的实现。

## Now both lead to Rome!

---

- The logic looks very similar after compilation
  - The most difference comes from heap vs. stack

|             |                                                                                                     |             |                                                   |
|-------------|-----------------------------------------------------------------------------------------------------|-------------|---------------------------------------------------|
| printTable: | allocate record r1<br>jump to doListX                                                               | printTable: | allocate stack frame<br>jump to whileL            |
| doListX:    | allocate record r2<br>if l=nil goto doneL<br>i := r2.l.head<br>allocate record r3<br>jump to putInt | whileL:     | if l=nil goto doneL<br>i := l.head<br>call putInt |
| again:      | add SL.i+SL.i<br>jump to putInt                                                                     |             | add i+i<br>call putInt                            |
| doRest:     | jump to doListX                                                                                     |             | jump to whileL                                    |
| doneL :     | jump to SL.cFunc                                                                                    | doneL:      | return                                            |

翻译到这一步，我们的性能和 imperative 的性能就基本上一致了。无非就是我们的 record 是在堆上分配的，会导致性能稍微差一点。

## The last inefficiency: heap-allocated record

---

- Heap allocations are less efficient
  - The heap structure is usually more complex
  - Not to mention garbage collection
- What can we do for further optimizations?
  - More advanced closure conversion: reducing heap allocation
  - Escape analysis: stack-allocated those records (r1)
  - Fast heap allocation and GC

这边就讲完了我们的 pure-fun tiger。作者花了很多功夫讲纯函数的性能未必不如 imperative。

# Lazy Tiger

Does PureTiger always ensure equational reasoning?

- An important principle is  $\beta$ -substitution:
  - If  $f(x) = B$  with function body  $B$ , then any application  $f(E)$  to an expression  $E$  is equivalent to  $B$  with every occurrence of  $x$  replaced with  $E$ :
  - $f(x) = B \rightarrow f(E) = B[x \mapsto E]$
- An concrete example
  - $B: x^2; f(i+i) \equiv (i+i)^2$

```
let f(x: int):int=
 x*x
 in f(i+i)
 end
```



```
let f(x: int):int=
 x*x
 in (i+i)*2
 end
```

我们之前追求 equational reasoning，我们是否达到了这个目标呢？我们对 IO 也做了 continuation，没有 side-effect。但是我们没有完全成功，这里提供了一个场景。

- But consider another example:
  - $B: \text{if } y > 8 \text{ then } x \text{ else } -y$ ; seems equal
  - But if  $y = 0$ : the left hangs while the right returns 0

```
let
 function loop(z:int):int=
 if z>0 then z
 else loop(z)
 function f(x:int):int=
 if y>8 then x
 else -y
 in
 f(loop(y))
 end
```



```
let
 function loop(z:int):int=
 if z>0 then z
 else loop(z)
 function f(x:int):int=
 if y>8 then x
 else -y
 in if y>8 then loop(y)
 else -y
 end
```

左边导致死循环，但是它们函数体不一样，而右边不会导致死循环。这里的问题就是，左边因为我们的函数调用值总是在进去之前就算出来了，右边因为我们展开了，我们是先执行了  $f$ ，而是再执行了  $loop$ 。

# Introducing lazy evaluation

---

- A program compiled with lazy evaluation will not evaluate any expression unless its value is demanded by some other part of the computation
  - i.e., on-demand evaluation
- A language with lazy evaluation is a **lazy language**
  - This part introduces Lazy-Tiger

我们要引入延迟计算：如果没有使用这个值的话，我们就不去计算它。满足这样这个条件的就是 **lazy language**。我们在做 **deep learning** 的时候也会很常用这种方法，比如矩阵的一系列算子丢给编译器分析一下，它就会输出一个执行顺序。

首先我们平时用的 **strict language**，通常使用 **call-by-value** 来传递参数。也就是在计算  $f(g(x))$  时，哪怕  $f$  没有使用这个参数，也会先计算  $g(x)$ 。

## Call-by-name evaluation

---

- Most (strict) languages use **call-by-value** to pass function arguments
  - $f(g(x))$ : first  $g(x)$  is evaluated even if  $f$  does not use it
- **Call-by-name** replaces each variable with a **thunk**
  - E.g., each integer is replaced with a function value of type  $() \rightarrow \text{int}$

`let var a := 5+7 in a+10 end` → `let function a() := 5+7 in a() + 10 end`

而在函数式中，通常是使用 **call-by-name** 的，它就是把变量变成一个 **thunk**。**thunk** 没有什么实际的值，其实就是一个名字摆在这里。如果对 **int** 声明 **thunk** 的话，那么我们得到的就是  $() \rightarrow \text{int}$ 。这样我们就可以把原先的变量定义成一个函数。这样就不是一个直接计算了，而是到后面调用  $a()$  的时候才变成了一个 **lazy** 的东西。

假设我们要在二叉树上做 **call-by-name**，

- Recap: the original imperative program

```

type key = string
type binding = int
type tree = {key: key,
 binding: binding,
 left: tree,
 right: tree}
function look(t: tree, k: key): binding =
 if k < t.key
 then look(t.left, k)
 else if k > t.key
 then look(t.right, k)
 else t.binding

```

## The problem with call-by-name

---

- A thunk may be executed many times
  - In the tree example, a node is expanded every time it is traversed
  - Now the tree has to be reconstructed even in read-only look()!
- The problem can be solved with **call-by-need**
  - Every thunk can be evaluated only once

通过这样一个转换以后确实可以 lazy 化，可以包含一些没有展开的 tree node 来减少计算。但是一个 node 可能会被展开很多次。

## The problem with call-by-name

---

- A thunk may be executed many times
  - In the tree example, a node is expanded every time it is traversed
  - Now the tree has to be reconstructed even in read-only look()!
- The problem can be solved with **call-by-need**
  - Every thunk can be evaluated only once

优化方法就是 call-by-need，在用 call-by-need 的时候，每个 thunk 只需要被 evaluate 一次，这是一个记忆性的方法，evaluate 一次以后就会有一个 slot 存储结果。

**2021/12/24**

我们还是从 call-by-name 开始，我们知道之前的是 call-by-value。调用一个函数，肯定

是先算里面再算 invocation。其实我们的所有变量都可以定义为一个函数，返回值为原来的类型。

## Call-by-name evaluation

- Most (strict) languages use call-by-value to pass function arguments
  - $f(g(x))$ : first  $g(x)$  is evaluated even if  $f$  does not use it
- Call-by-name replaces each variable with a thunk
  - E.g., each integer is replaced with a function value of type  $(\text{--}) \rightarrow \text{int}$

```
let var a := 5+7 in a+10 end → let function a() := 5+7 in a() + 10 end
```

1

其实在正常语言里也可以理解，

比如说

Let ...

In a

End

我们也可以理解为这是一个没有参数的函数。这里就是把这个东西显式化了，我们这里把变量都变成了函数。要显式 invoke 它才能返回的东西。变成函数之后，它就不再对应内存中的一个值了，而是一段代码。所以当我们变成 call-by-name，这些变量就多了一层 tricky 的含义。

之前我们有二叉树的一个 key 和 binding 结构，现在我们改成所有的访问都变成了 call-by-name 式的访问。

- Recap: the original imperative program

```

type key = string
type binding = int
type tree = {key: key,
 binding: binding,
 left: tree,
 right: tree}
function look(t: tree, k: key): binding =
 if k < t.key
 then look(t.left, k)
 else if k > t.key
 then look(t.right, k)
 else t.binding

```

- Recap: the original imperative program

```

type key = string
type binding = int
type tree = {key: ()->key,
 binding: ()->binding,
 left: ()->tree,
 right: ()->tree}
function look(t: ()->tree, k: ()->key): binding =
 if k() < t().key() must evaluate...
 then look(t().left, k)
 else if k() > t().key()
 then look(t().right, k)
 else t().binding

```

|                                                                                                                                                                                                                                                                            |                                                                                                                                                                                                                                                                                                                      |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| type key = string<br>type binding = int<br>type tree = {key: key,<br>binding: binding,<br>left: tree,<br>right: tree}<br>function look(t: tree, k: key): binding =<br>if k < t.key<br>then look(t.left, k)<br>else if k > t.key<br>then look(t.right, k)<br>else t.binding | type key = string<br>type binding = int<br>type tree = {key: ()->key,<br>binding: ()->binding,<br>left: ()->tree,<br>right: ()->tree}<br>function look(t: ()->tree, k: ()->key): binding =<br>if k() < t().key()<br>then look(t().left, k)<br>else if k() > t().key()<br>then look(t().right, k)<br>else t().binding |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

//call-by-value

//call-by-name

所以当我们要做后面的二叉树的查找操作时，我们真的要做比较的时候，比如有些地方是不调用的，就是继续把 `k` 这个函数传进去了。

比如 `t().left` 这个也是一个 `lazy` 的结构，在我们调用它之前是不用展开的。什么时候我们真正需要拿到它的值呢？比如我们上述的小于号的地方，我们必须拿到值才可以比大小。这时候我们就要调用 `k()` 和 `t().key()`。

所以我们有一个 `call-by-need`，也就是只有要被算的时候才会被计算出来。

## Call-by-need

我们把 `thunk` 扩展一下变成 `memo slot`。有 `ep` 我们就可以拿到所有的 `escape parameter` 和 `static link`。因为我们把所有东西都保存再堆上了，并且函数和 `ep` 具有相同的生命周期了。状态就从 `unevaluated` 变成 `evaluated`。每次算完就调用以后就不会再去计算了。

|                                                                                                                                                                                                                 |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>let   type intfun = int -&gt; int   function add (n:int) : intfun =     let function h(m:int): int = n+m       in h     end   var addFive: intFun := add(5)   var twenty := addFive(15)   in ... end</pre> | <pre>let   type intfun = int -&gt; int   function add (n:int) : intfun =     let function h(m:int): int = n+m       in h     end   var addFive: intFun := add(5)   var twenty := intThunk{func=twentyFunc, memo=EP}   in ... end</pre> <p><b>Stack frames    Heap records</b></p> <p>The diagram illustrates the memory layout for the 'Call-by-need' example. It shows the main stack frame containing the variable <code>addFive</code> and its <code>EP</code>. Two <code>SL</code> (Static Link) pointers are shown, each pointing to a heap record. These heap records contain <code>SL</code> and the value <code>5</code>. Arrows indicate that both heap records also point back to the <code>EP</code> in the main stack frame. A separate stack frame at the bottom is labeled as <code>addFive's closure</code>, containing <code>EP</code> and <code>MC</code>. An arrow points from this closure stack frame to one of the heap records, labeled as <code>addFive's machine code</code>.</p> |
| 原先的形式，这个函数我们正常就是 <code>pure-fun tiger</code> 。这个 <code>in</code> 里面可能要用到这个 <code>twenty</code> ，比如 <code>print</code> 或者小于号，这样就是通过观察的方法促使它去计算。                                                                  | 如果我们要换成 <code>lazy-tiger</code> 的话，就要把 <code>twenty</code> 换成 <code>intThunk</code> ，它的 <code>function</code> 就是 <code>twentyFunc</code> ，其实就是 <code>addFive(15)</code> ，而 <code>emeo</code> 是 <code>EP</code> 。其实 <code>lazy-func</code> 的话，这个结构会更复杂，这里简单回顾一下我们会用到的函数。 <code>addFive</code> 的 <code>EP</code> 要包含                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |

这个 5。

改出来的结果就是这样，会比较复杂。

```
type intThunk = {func: ?->int, memo: ?}
type intfunc = {func: (?<-,intThunk)->int, SL: ?} SL/EP
type intfuncThunk = {func: ?->intfunc, memo: ?} //这里的类型是比较自由的，所以这里都标了问号，比如 staticlink 可以放一个结构体，里面包含各种各样的逃逸的东西。 memo 是 EP，也是可以各种类型。
//所以我们在做生成的时候不太关心类型检查的事情
function evaluatedFunc(th: intThunk): int = th.memo
function twentyFunc(mythunk: intThunk): int = //这个对应的是没有 evaluated 的，所以要做一系列计算。传入的是 mythusk， invoke 就是传入自己去做 invoke。
let var EP := mythunk.memo
var add5thunk: intfuncThunk := EP.addFive //通过 memo 去拿 addFive 这个函数。因为这个函数在里面没有体现，所以它会被放到 escape record 里去。
var add5: intfunc := add5thunk.func(add5thunk) //传入的参数就是自己，都是以 t.func(t) 的形式。
var fifteenThunk:=intThunk{func=evaluatedFunc,memo=15}
var result: int := add5.func(add5.SL, fifteenThunk) //这里就是真的做了 5 + 15 的计算。
in mythunk.memo := result;
mythunk.func := evaluatedFunc;
result
end
...
var twenty := intThunk{func=twentyFunc, memo=EP}
```

我们在做这里面的时候， fifteen\_thunk 肯定也是去调用 evaluation 来得到值。这里就不再是一个纯函数的语言了。

## An example: addFive

- Auxiliary declarations are required

```
type intThunk = {func: ?->int, memo: ?}
type intfunc = {func: (?, intThunk)->int, SL: ?} SL/EP
type intfuncThunk = {func: ?->intfunc, memo: ?}
function evaluatedFunc(th: intThunk): int = th.memo
function twentyFunc(mythunk: intThunk): int =
 let var EP := mythunk.memo
 var add5thunk: intfuncThunk := EP.addFive
 var add5: intfunc := add5thunk.func(add5thunk)
 var fifteenThunk:=intThunk{func=evaluatedFunc,memo=15}
 var result: int := add5.func(add5.SL, fifteenThunk)
 in mythunk.memo := result;
 mythunk.func := evaluatedFunc;
 result
 end
...
var twenty := intThunk{func=twentyFunc, memo=EP}
```

9

所以这里展现的是允许过程中的一个结果，修改了 memo 和 thunk 的值来达到记忆化的实现。

一方面我们要引入 lazy 的东西，让所有东西变得复杂，value 不再是 value，而下面的实现就要不断修改 thunk 里面的成员，好处就是可以 lazy 地做这些事情，让我们算过一次以后就不用再算了。

### Back to the previous example

- When  $f(\text{loop}(y))$  is called,  $\text{loop}(y)$  is lazily evaluated
  - As no one uses the value of  $\text{loop}(y)$  when it is called
  - $\text{loop}(y)$  remains a binding until its value is used

```
let
...
function f(x:int):int=
 if y>8 then x
 else -y
in f(loop(y))
end
```

15

我们继续回到可能死循环的例子，因为我们使用了 lazy 的方法，现在这个  $\text{loop}(y)$  因为没有人观察它（我们永远会走 else 这条分支），这个  $\text{loopfunc}$  没有人用，就不会计算它。这个地方  $y$  因为是有大于号，所以  $y$  是必须计算的，因为这个会影响控制流。

# Lazy 函数式程序的优化

我们来看看 lazy 对优化的影响。

虽然 lazy 这个东西被我们弄得很复杂，但是在 strict functional program (严格函数式程序) 和 imperative program (命令式程序) 中使用到的大多数优化方法还是可以使用的，比如 Loop-based, CSE。

我们之前得到的结果是 pure-fun 的优化的最终性能是和 imperative 差不多的。

但是 lazy-compiler 可以做更多的优化，主要是靠 equational reasoning。我们可以简单介绍一些这一些优化。（invariant hoisting, dead-code removal, deforestation）

## Invariant hoisting

- Equational reasoning helps by finding all 'fixed' values
  - Consider the following program
  - $h(i)$  always returns the same value for a given  $i$

```
type intfun = int->int
function f(i: int): intfun =
 let function g(j: int) = h(i) * j
 in g
 end
```

## Invariant hoisting

- Equational reasoning helps by finding all 'fixed' values
  - Consider the following program
  - $h(i)$  always returns the same value for a given  $i$
  - Optimization: moving  $h(i)$  outside  $g$

```
type intfun = int->int
function f(i: int): intfun =
 let var hi := h(i)
 function g(j: int) = hi * j
 in g
 end
```

这个和我们之前循环的 hoisting 区别不大。以后  $g$  这个函数就不需要再去算  $h(i)$  了，这里就非常严格地依赖于 equational reasoning。因为我们改变了执行顺序，在 strict language 中可能就 hang 住了。

## Invariant hoisting

---

- Can we do this for imperative language?
  - Possibly no: we may cause side-effects during calling h
    - In C++ we have const keyword to mark no-side-effect
    - Another alternative property is called 'idempotent'

```
type intfun = int->int
function f(i: int): intfun =
 let var hi := h(i)
 function g(j: int) = hi * j
 in g
end
```

另一种方式是证明这个函数是幂等的。

### Dead code removal

---

- Equational reasoning can find determine more dead code
  - The following program:  $g(x)$  can be removed
  - Strict functional:  $g(x)$  may infinite-loop
  - Imperative:  $g(x)$  may have side effects

```
function f(i:int): int =
 let var d := g(x)
 in i+2
 end
```

有了 equational reasoning 以后，这些就可以很轻易地被消除，因为我们没有调用函数。

## Deforestation

---

- We may break programs into modules
  - E.g., producers and consumers (pipes)

```
type intList = {head: int, tail: intList}
type intfun = int->int
type int2fun = (int,int)->int
function sumSq(inc: intfun, mul: int2fun, add: int2fun): int =
let
 function range(i: int, j: int) : intList =
 if i>j then nil else intList{head=i, tail=range(i+1,j)}
 function squares(l: intList) : intList =
 if l=nil then nil
 else intList{head=mul(l.head,l.head), tail=squares(l.tail)}
 function sum(accum: int, l: intList) : int =
 if l=nil then accum else sum(add(accum,l.head), l.tail)
in sum(0,squares(range(1,100)))
end
```

它把一个 program 拆成了很多 module，这个例子要做的事情就是生成一个 1~100 的链表，求平方再加起来。square 就是对每个元素拿出来调用乘法。  
去森林化的效果就是变成了一个比较紧凑的函数。

## Deforestation

---

- What is the problem for the program?
  - Too many intermediate lists!
    - More list allocation during runtime
    - Perhaps more overhead on GC
- A possible optimization: deforestation

```
type intList = {head: int, tail: intList}
type intfun = int->int
type int2fun = (int,int)->int
function sumSq(inc: intfun, mul: int2fun, add: int2fun): int =
let
 function f(accum: int, i: int, j: int) : int =
 if i>j then accum else f(add(accum,mul(i,i)),inc(i))
 in f(0,1,100)
end
```

总的来说，我们就把之前的函数合在一起了，我们可以认为这是某种程度的内联，但是合成的能力比我们之前的内联强很多。

- Previously:
  - inc(1), inc(2), ... inc(100), mul(1,1), mul(2,2), ..., mul(100,100), add(0, 1), add(1, 4), ..., add(328350, 10000)
- After:
  - mul(1,1), add(0,1), inc(1), mul(2,2), add(1,4), inc(2), ..., mul(100,100), add(328350, 10000), inc(100)
- Equal this time, but not always:
  - Function invocations have side effects
  - But a lazy language can always use this optimization

现在是乘一下加一下，它本质没有减少计算，但是它把函数的执行顺序改变了。

## Conclusion for lazy language optimizations

---

- Lazy evaluation fixes the partial failure in the pure functional language
  - Now the equational reasoning always holds regardless of halts
- This enables more optimization opportunities
  - A function always returns the same result regardless of its body
    - Getting rid of annotations (like C++)
    - And analysis for the function body

我们介绍了函数式优化的方法和实现的方法，而面向对象开始就大家更加熟悉一点。原先的面向过程的 tiger 要变成面向对象的 tiger 的差别还是挺大的。

- Aka. OBJECT-Tiger
- Which keywords/syntax rules should be added?
  - Class? (class A extends B)
  - Fields? (private/public)
  - Method? (A::abc())
  - .....

也就是我们需要支持 class, private/public, namespace 等。当然 method 本身可能也分 private 和 public。我们来看一个 object-tiger 的例子，其实在面向对象中一个比较重要的例子如下：

## An OBJECT-Tiger example

---

```

let start := 10

class Vehicle extends Object {
 var position := start
 method move(int x) = (position := position + x)
}

class Car extends Vehicle {
 var passengers := 0
 method await(v: Vehicle) =
 if (v.position < position)
 then v.move(position - v.position)
 else self.move(10)
}

class Truck extends Vehicle {
 method move(int x) =
 if x <= 55 then position := position + x
}

var t := new Truck
var c := new Car
var v : Vehicle := c
in
 c.passengers := 2;
 c.move(60);
 v.move(70);
 c.await(t)
end

```

39

wrapped in let

在 tiger 没有支持分开编译的事情，比如 java 中就需要指定包名。这里讲的是一个交通工具的例子，交通工具有一个 position 和 move 函数。下面我们就定义了子类 Car 和 Truck，而 Car 还有 passenger 和 await。我们接下来就可以通过 new 关键字来做。我们也可以定义向父类的转换。而 Object 是一个 predefined 的类型，是所有面向对象类型的父类，也称为 Top，在别的语言中一个 Bottom 的概念，它是所有类型的子类，也就是 Exception 的情况。

OO-tiger 中没有构造函数，我们认为执行的所有函数就是它的构造器。注意这里的石油东西都是 public 的，所有东西都可以直接拿来使用。

上面稍微介绍了一下很像 java 的语法，其实 `description` 加了是比较少的。类型的定义，我们需要全部改掉，现在的所有的类型都必须以类型的方式来定义。类型必须是 `class description`。

## Syntax rules

---

- Adding descriptions

```
dec → classdec
classdec → class
class-id extends class-id { {classfield} }
classfield → vardec
classfield → method
method → method id(tyfields) = exp
method → method id(tyfields) : type-id = exp
```

- Adding expressions

```
exp → new class-id
exp → lvalue . id()
exp → lvalue . id(exp[, exp])
```

在 `expression` 中，我们现在的左值是一个对象，我们就可以增加 `new` 一个变量。相对于函数式，面向对象这里要修改的会比较多一点。

`self` 不在 AST 里去做，也不会识别一个 `keyword`，只会识别为一个正常的 `token`，它被处理为一个隐式的参数，在运行时自动地绑定到对象上去。如果我们处理的是一个对象，它对应一个 `temporary`，那么 `self` 会自动和它绑定在一起。这更多地是一个 `runtime` 时候的事情。

## What about `self`?

---

- Self is not a keyword in OBJECT-Tiger
  - It is an implicit parameter for each method
  - Automatically bound to the object during runtime

```
Class Car extends Vehicle {
 ...
 method await(/*self: Car,*/ v: Vehicle) {
 if (v.position < position)
 then v.move(position - v.position)
 else self.move(10)
 }
}
var c := new Car
c.move(60); -> move(c, 60);
```

48

认为就是传入的时候标志了一个 `self` 进来。编译的时候会被转化为 `move(c, 60)`；不同语言里的设计会不太一样 `c` 和 `c++` 还有 `java` 就是作为一个 `hidden parameter` 来传的，

因为我们在调用的时候并不会传入 self。

## Self in other OO languages

---

- C++/Java: *this* keyword as a hidden argument
  - But this is a pointer in C++, a reference in Java

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

class Test
{
private:
 int x;
 int y;
public:
 Test (int x = 0, int y = 0) { this->x = x; this->y = y; }
 Test setX(int a) { x = a; return *this; }
 Test setY(int b) { y = b; return *this; }
 void print() { cout << "x = " << x << " y = " << y << endl; }
};

int main()
{
 Test obj1;
 obj1.setX(10).setY(20); // chained calls
 obj1.print();
 return 0;
}
```

49

Source: <https://www.geeksforgeeks.org/this-pointer-in-c/>

比如 chained call，在做比较复杂的赋值的时候就会做这种复杂的调用。\*this 就能够直接返回这个对象。

在 python 里，这个 self 对象就不再是隐式的了。

## Self in other OO languages

---

- C++/Java: *this* keyword as a hidden argument
  - But this is a pointer in C++, a reference in Java
- Python: The first parameter is forced to be *self*
  - But the name is not necessarily *self*

OK!

```
def __init__(self) -> def __init__(anyname)
```

我们一定是要定义函数的时候第一个参数设置为 self 对象。

前面是语法介绍，我们来讲一下 oo 中比较重要的继承。

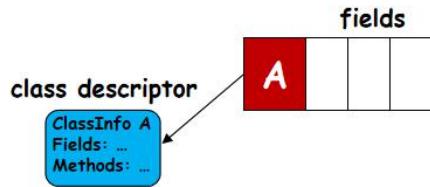
## 单一继承

单一继承就是每个对象有一个父类。Java 和 c#，还有 Swift 等，这些语言都是单一继承的。我们继承的效果，一方面是 field，一方面是 method。

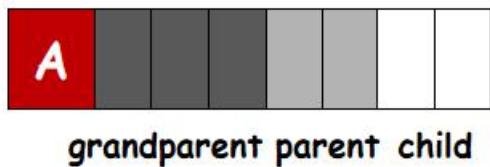
### Field inheritance

---

- Fields are inherited from the parent class
  - How to co-locate them with newly-defined fields?
- Simple for single-inheritance: prefixing
  - Recap: object layout in memory



直接从父类获取字段，子类还会有新的字段，那么怎么排序呢？我们只需要往后放就行了。在 GC 中，我们需要判断 record 里哪些存放的指针。Descriptor 中包含了类型信息、字段有哪些、方法有哪些。这是我们面向对象的结构。继承来的 field 会被放到最前面，按照辈分顺序来放置 field。



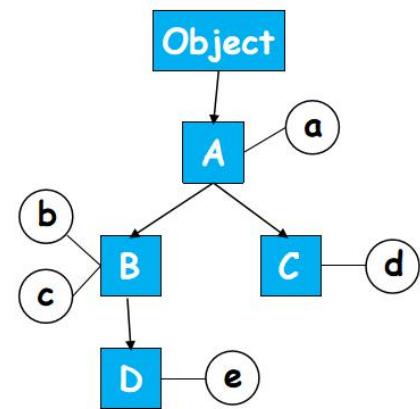
## An example in OBJECT-Tiger

---

code

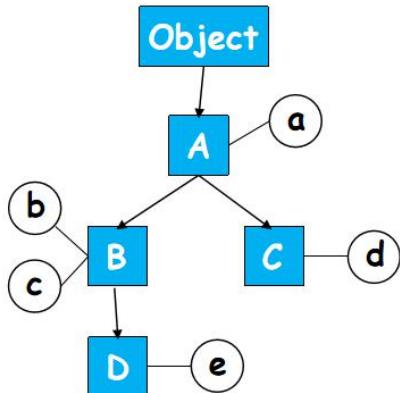
```
class A extends Object {
 var a := 0
}
class B extends A {
 var b := 0
 var c := 0
}
class C extends A {
 var d := 0
}
class D extends B {
 var e := 0
}
```

type hierarchy

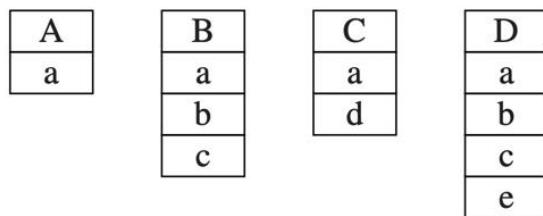


55

## type hierarchy



## object layout



Fields are ordered by following the path!

我们只需要沿着这个继承顺序来走，就可以得到 field 的顺序了。

方法也类似，但是方法有一个不一样的地方是方法可以 `override`。如果名字一样，参数传入一样，那就直接 `override`。

```

let start := 10

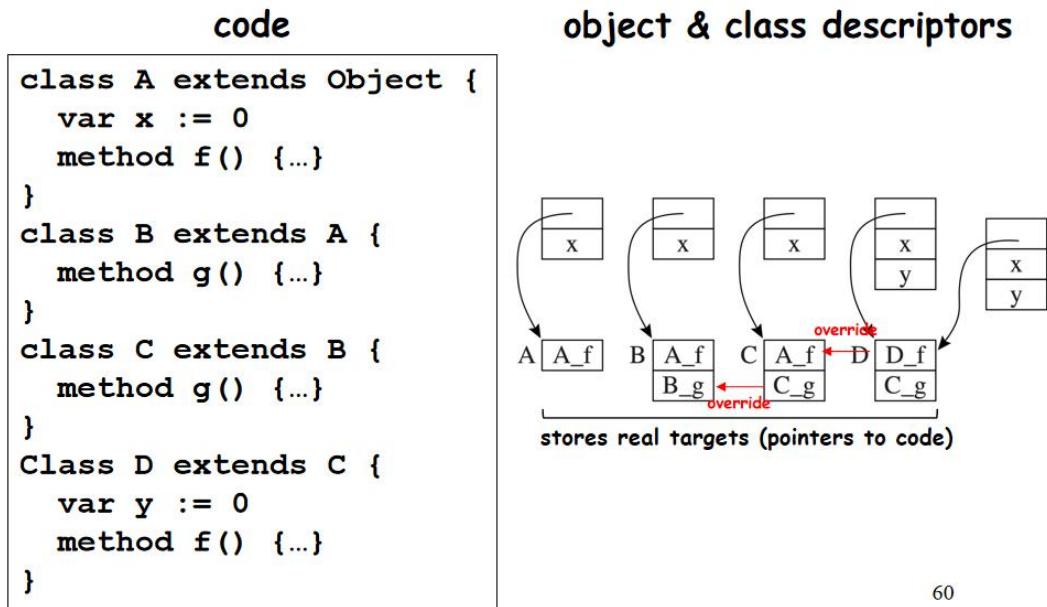
class Vehicle extends Object {
 var position := start
 method move(int x) = (position := position + x)
}
class Car extends Vehicle {
 var passengers := 0
 method await(v: Vehicle) =
 if (v.position < position)
 then v.move(position - v.position)
 else self.move(10)
}
class Truck extends Vehicle {
 method move(int x) =
 if x <= 55 then position := position + x
}

var t := new Truck
var c := new Car
var v : Vehicle := c
in
 c.passengers := 2;
 c.move(60);
 v.move(70);
 c.await(t)
end

```

在 c++ 中如果 field 中重复定义了，就会被定义为两个字段，而方法这里就需要被重载。  
在 c++ 中 descriptor 是通过虚表来做的。

## An example for class descriptors



60

我们就以继承关系来看。B 继承了 A，而它自己声明了一个 B\_g，而 C 继承了 B，重载了新的 g，所以用 C\_g 来覆盖 B\_g。这里基本思路就是有新的我们就继续往下扩展，如果有重载我们就在原地重载。来做相同位置的函数替换。这样就可以使我们的虚表的扩展比较简单。

其实大家的位置是一样的，所以动态的 method lookup 不会很复杂。比如我们要 call C.f()，我们从 C 的 object 拿到这个 C 对应的 descriptor，然后我们要拿到 A\_f 的 pointer，那么就从 offset=0 这个地方去拿。

这个 table lookup 也没有很费时间，总共也就 2 次额外的访问内存操作。

- Is table lookup very costly?
  - Not really! Overridden methods are at the same offset
  - The memory access pattern is fixed:  
 $M[M[\text{obj}+0]+16]$  for f() with any types
  - The overall overhead: 2 more memory accesses

这是单一继承，比较容易。

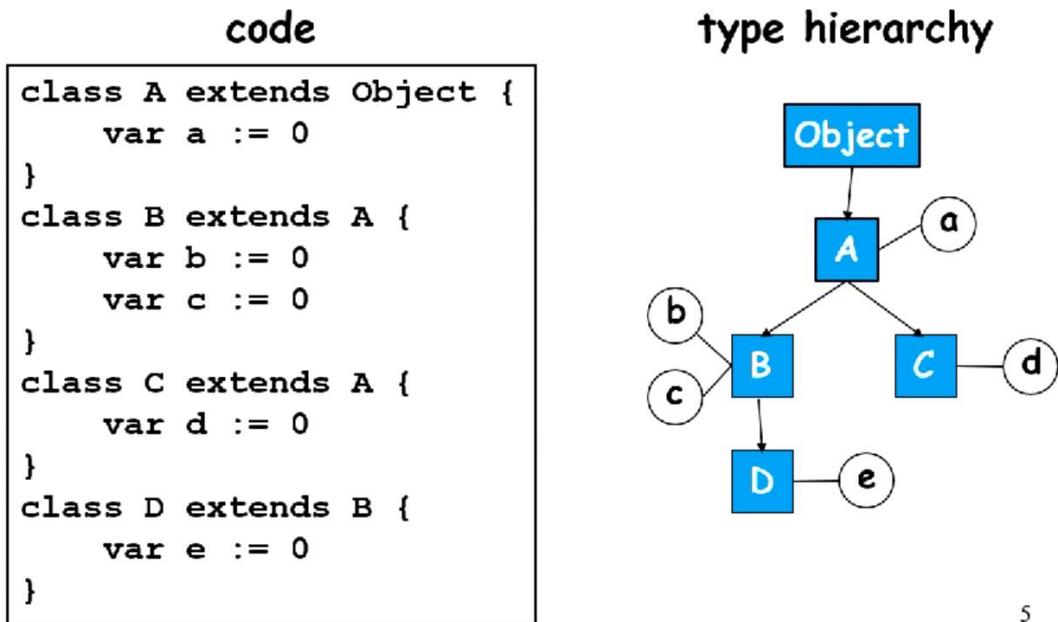
## 多重继承

C++, Perl, Python 等支持多重继承。复杂就复杂在 A 继承于 B 和 C 的话，刚才我们是线性的顺序，那到底把 B 和 C 的 field 哪个放在前面呢，这就需要讨论了。

2021/12/28

我们今天继续讲面向对象的语言。先复习一下上节课讲的单一继承。

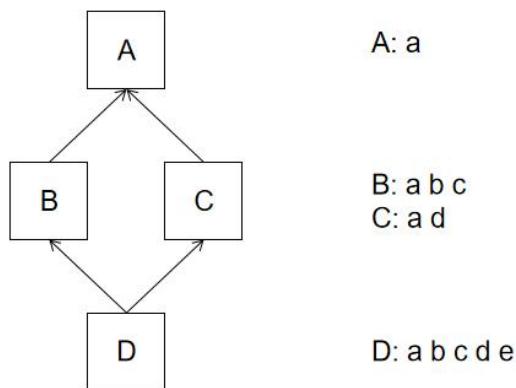
## An example in OBJECT-Tiger



5

单一继承都是线性往下扩增的，一定是可以排出一个顺序的。我们通过一个固定的 offset 就可以拿到这个函数和 field。

而这节课讲到的多重继承就会复杂一点，因为这种线性序就不成立。没有多重继承的语言也有对应的 feature 来实现类似的功能。多重继承会导致 layout 不能在进行简单的排列。



我们可以强行让 D 的 field 按照某个顺序排列，但是我们会发现：

```
C c = new D()
c.d
```

当我们想把 D 强制转化为 C 去访问的时候，C.d 和 D.d 的 offset 是不相同的。而我们在静态的时候又判断不了这个类型，就对我们编译产生了麻烦。

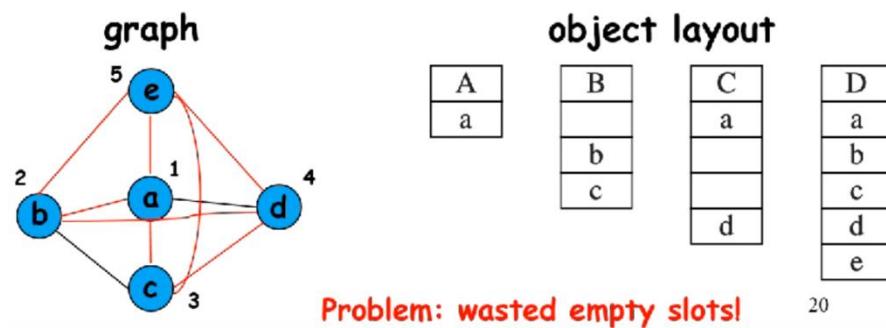
## Global Graph Coloring

为了解决这个问题，就提出了图染色问题。如果不同 field 在同一个类型里出现了，那么它们就不能染成同一个颜色。也就是不同的 field 不能放在同一个位置。

所以这个图染色问题的顶点是不同的类型名，而边是两个 field 共存在同一个类中了。

我们举个例子：

```
class A extends Object { var a := 0 }
class B extends Object { var b := 0; var c := 0 }
class C extends A { var d := 0 }
class D extends A, B, C { var e := 0 }
```

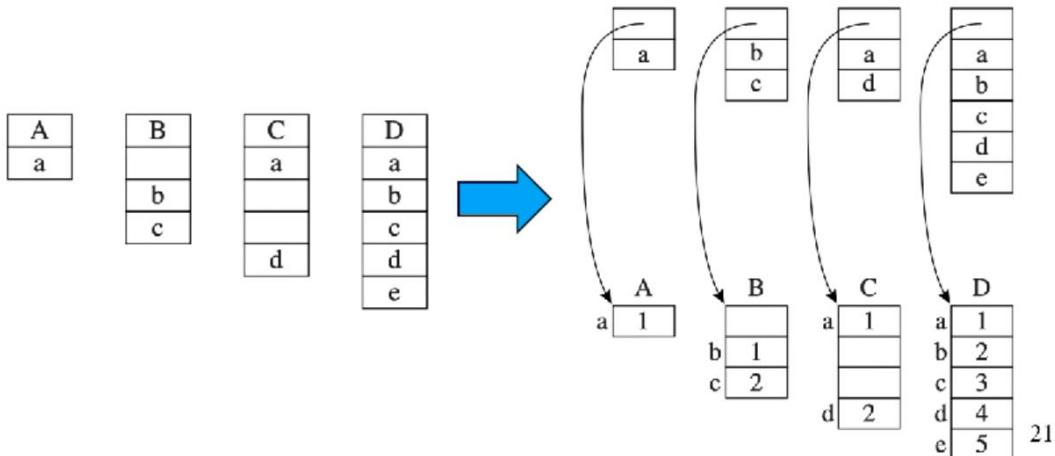


最终我们发现构成了一个完全图，我们就可以直接对它做染色并编号。

我们得到的这个 layout 符合我们对于单一继承的假设。比如说 C、D 类型的 d 一定在第四个位置。但是后果就是我们会有一些空的 offset 在这个地方，会造成一些浪费，可能造成比较大的内存压力。对于更加复杂的类型继承，可能会更麻烦。

## Advanced solution: coloring on classes

- Real colors (offsets) are presented in class descriptors
- Fields are compacted in objects



我们通过类型的固定位置拿到 `d`, 然后再去对象中拿紧凑表达的序号。为什么要这样做呢? 因为类型的数量通常是认为小于对象的数量的。所以我们在类型里面浪费一些内存影响没有这么大。

## Advanced solution: coloring on classes

- Why preferring class descriptors for coloring?
  - The number of types  $\ll$  the number of objects
- Problem: the offset for each field is not fixed
  - E.g., the offset for `b` is 0 in `B`, 1 in `D`
  - Dynamic lookup is required for field access

这里的问题就是说, 本来的对象可以直接 `lookup`, 但是现在我们每次都要做一次间接的访问到 `class descriptor` 才可以访问到 `field`。

步骤:

- 从 `field` 的 `offset = 0` 处拿到类型的 `descriptor`。
- 从类型 `descriptor` 中, 找到我们想访问的 `field` 的 `offset`。

- 再拿着这个 offset 去访问对象。

对于对象中的方法，我们也是放到图中去做染色，不过方法会放到类型里面，所以也不需要在对象里去存方法的信息。

## What about methods?

---

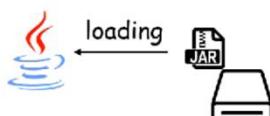
- Still using the global coloring strategy
  - Method names can be mixed with field names in the same graph
    - Field entries → offset
    - Method entries → code address for method
- The cost for dynamic lookup is similar
  - Single inheritance also requires a lookup

我们拿到方法信息以后不用再去 data 中去会查了，我们可以直接拿到代码的位置直接去执行了。

## New problems with dynamic linking

---

- Global graph coloring assumes classes are **statically linked** together
  - A special-purpose linker can help achieve that
- However: many OO systems can load classes during runtime
  - E.g., Java's dynamic class loader



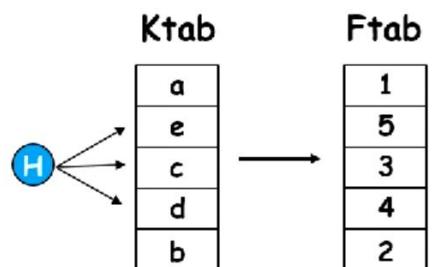
25

之前考虑的都是静态方法和字段的情况。我们有一个链接器，把这些都连接起来。比如 C++ 的话就用 g++ 编译起来然后用一个 linker 自动链接起来。我们的要求是，类型信息在编译的时候都知道，因为如果我们不知道类型消息，我们是不能做这个图的染色的。所以都是编译器和链接器去完成的，没有什么动态的东西，运行的时候就直接用这样的偏移量就可以了。

通常现在的 OO 语言都会有运行时，支持动态加载的。以 java 为例，比如我们从网上下载一个 jar 包，运行的时候可以加载一些 jar 包进来。因为我们可能有一个超大的 jar 包，这样我们就可以不需要一开始就全部把类型加载进来。坏处就是我们不能用之前的静态方法了，因为很多类型信息是此时我们不知道的。

## Solution: Hashing

- Building a hash table for each class descriptor
  - Ftab: containing field offsets and method code address
  - Ktab: containing field/method names
- The table is agnostic to multiple inheritance
  - Fields do not need to have fixed offsets

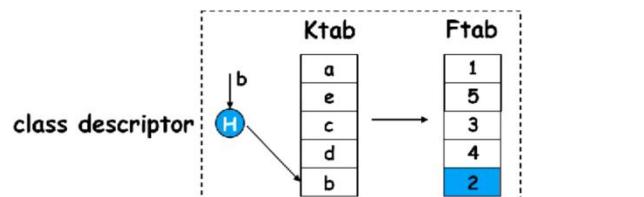


26

这样我们就用了一个哈希的表达方法，不再遵循之前的规律。我们给每个类型放一个哈希表，分为 key-table（放 field 的名字）和 field-table（放 field 的 offset）。这种情况下，不管单一继承和多重继承都是适用的，来了以后我们无非就是哈希查表即可。

## Solution: Hashing

- Steps to fetch a field (say field b)
  - Fetch the class descriptor at offset 0
  - Fetch the field name from offset Ktab +  $\text{hash}_b$
  - Compare field name with the input name (EQUAL!)
  - Fetch the field offset from Ftab +  $\text{hash}_b$  (2)
  - Fetch the field from object + 2



28

我们进来以后无脑查表即可。

## Overhead analysis

---

- Four more instructions (seems tolerable)
- However, it is much more costly than coloring
  - Hashing: usually requires several binops
  - String equality check:  $O(N)$
  - Hash entries can collide (leading to more accesses)
- Perhaps a reason why Java does not have MI

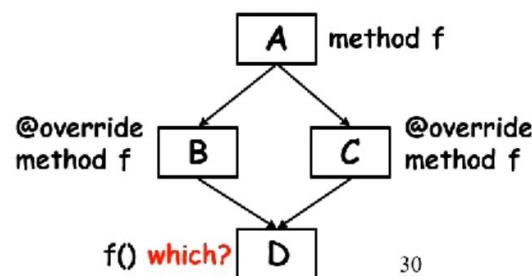
这四条指令的粒度是非常粗的，里面可能有一些非常耗时的指令，比如哈希计算。所以这肯定不是几个 cycle 能结束的事情。判断 string 是否相等又是一个  $O(N)$  判断，更不用说 hash 的 entry 可能有冲突，又要做冲突处理。所以哈希这个数据结构可能比较简单，几十行就能实现，但是会有很多问题。

1. bucket 分布不均匀，有的链可能很长，这就不能控制访问时间。
2. 闭哈希不够的情况下，我们可能就要 resize。哈希表使用的 slot 的数量并没有到 100% 但是不能继续插入了。

## Another disadvantages of MI

---

- Can cause ambiguity with multiple super classes
- Also known as a “diamond problem”
  - Both B and C overrides f
  - Now which f is used by D?



多重继承可能会有二义性，如上图所示。B 和 C 把 f 方法重载掉，那么 D 直接调用 f 的话，应该调用谁的实现呢？多重继承本身并不提供解决方案。

## But what does Java have? Interface

---

- An interface is an abstract class with **no concrete components**
  - Usually only contains a collection of methods
    - "Specify what a class can do but not how"
  - Fields must be public, static, and final
    - So it can only become a constant
- Multiple inheritance can also be implemented with interfaces (implements A,B)
  - But with more restrictions

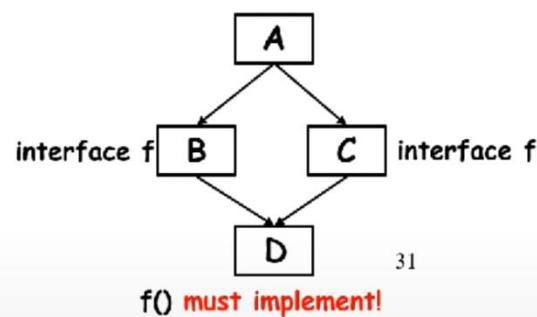
interface 就是一个抽象类，认为里面不应该有具体的 component，通常情况下我们只定义一些方法类。

Comparable 就是一个很常见的 interface，它里面可能就定义了 `bool cmp(Comparable b);`

## Another disadvantages of MI

---

- Can cause ambiguity with multiple super classes
- Also known as a "diamond problem"
- Interface avoids this problem
  - B and C (interfaces) cannot override f



如果 interface 中出现了多重继承的话，B 和 C 中只有两个对方法的定义，所以不会有冲突的问题。interface 就解决了 diamond problem。

多重继承给语言更多的灵活性，但是也在正常的字段访问和方法访问中加了更多的复杂性和额外开销。

## Membership Testing

类型检查，也包括了子类检查。这比我们之前的一对一的 type check 要复杂一些。因为有 subtype 的话，我们要做更多的一些比较。

### Recap: which type cast is safe?

---

- Casting to a super type is always safe (upcast)
  - Fields/methods in the super class can be accessed by the sub-class
- Casting to a sub-type is not (downcast)
  - Child class may define new methods/fields not present in the super class
- How to allow upcast while avoid incorrect downcast?

我们先来讲一下类型转换的事情。在 C 里面很常见的事情就是和 void\* 之间的相互类型转换。首先我们要讲转换成一个 supertype 是永远安全的。但是子类就不一定安全了，因为父类的定义可能是不全的，一些新方法和新字段父类可能没有。所以我们希望 upcast 是可以做的，并且避免不正确的 downcast。

## Type testing and casting

---

- A normal type testing and casting would be:

```
if (a.isClass(A)) {
 A b = (A)a;
 b.somemethod();
}
```

- OO languages have supported this feature

|                                                                                                                                                        | Modula-3      | Java           |
|--------------------------------------------------------------------------------------------------------------------------------------------------------|---------------|----------------|
| Test whether object x belongs class C, or to any subclass of C.                                                                                        | ISTYPE (x, C) | x instanceof C |
| Given a variable x of class C, where x actually points to an object of class D that extends C, yield an expression whose compile-time type is class D. | NARROW (x, D) | (D)x           |

很多语言会加上 `typetesting` 来帮助程序员判断对象的类型。很多 OO 语言支持这个 feature。

一个简单的方法就是我们拿到对象的 class descriptor。如果不相等，我们在拿对象的超类去判断。

### How to implement instanceof?

---

- A simple way is to perform the following loop:
  - Recursively compare types with the input type (C)

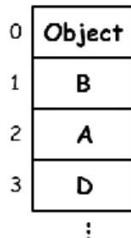
```
t1 <- x.descriptor
L1: if t1 = C goto true
 t1 <- t1.super
 if t1 = nil goto false
 goto L1
```

- The recursive comparison takes time
  - Can we have a faster approach?

如果是多重继承，那就是一个带递归/广度优先搜索的向上遍历问题。这里的问题就是递归式的比较久。

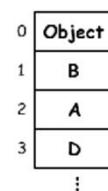
## Solution: displays of parent classes

- Each class descriptor stores such a display
  - E.g., class D extends A extends B extends Object
  - The display should look like below:



一个简化的方法是通过 `display` 来实现的。

- To implement `x instanceof D`:
  - Fetch the class descriptor at offset 0
  - Fetch the 3rd class-pointer slot
  - Compare with D
- Why is it true?



利用的思路就是说 D 的 `display` 一定是长这样的。如果是 D 的子类，一定是在 D 下面的。通过类型系统的这个排列，我们通过正向的去找就可以加速了。

### An alternative solution: typecase

- `instanceof` testing is not that 'object-oriented'
  - Programmers are expected to use dynamic methods
  - The dynamic lookup can directly reach the correct method
- Modula-3 has `typecase` - more beautiful and efficient

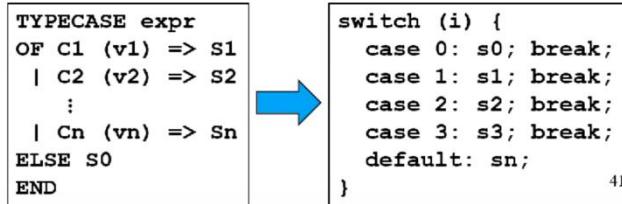
```
TYPECASE expr
OF C1 (v1) => S1
| C2 (v2) => S2
:
| Cn (vn) => Sn
ELSE S0
END
```

39

这个其实就把我们熟悉的语法扩展，扩展到可以接收任意的 expression，然后根据类型判断来分别处理。

## Implementation of typecase

- Is it possible to implement Typecase with jump tables? (**NO**)
  - Modules are separately compiled, with conflicting integers
  - Even integers can be assigned, we may miss clauses with super classes



module 可能是隔离编译的，这样就不容易去做。并且继承的时候是分成两个关系的，不同类 id 不同，但是父类应该是可以和子类匹配上的。所以 typecase 并不容易用 jump table 实现。其实 typecase 就是一个语法糖，就是把一些 if-else 编制在一起。

## Features in modern OO languages: case class

- Scala uses **case class** for pattern matching
  - Objects in case classes are not initialized with new
- Pattern matching can directly use both classes and fields
  - More powerful than traditional switch operations

```
case class Student (ID: String)
val Mingyu = Student("5150379000")
```

42

```
person match {
 case Student(id) => id
}
```

43

scala 就集成了 case class。判断一下是不是 `Student` 类型，如果是的话就把 `id` 拿出来。然后 scala 里定义 class 的方式其实和数据库中的表的方式是差不多的，所以就有人把 scala 语言应用到数据库里。

## What about C++?

---

- C++ has different kinds of type cast instructions
  - **Static cast**: casting without runtime check
    - You usually know the original type of the object
    - Unsafe

```
void meth(void *data) {
 A *a = static_cast<A*>(data);
}
```

c++有一些要兼容 C 的考量，和一些 feature 的考量。比如 static\_cast 就是类型直接转换，编译器假设我们是知道我们在做什么的。也就是我们认为 void\* 是所有类的超类。

- **Dynamic cast**: casting with runtime checks
  - Similar to membership testing in Java

```
A* a = new A();
B* b = dynamic_cast<B*>(a); //OK
```

dynamic\_cast 就会帮助我们判断类型，通过检查以后我们非常确定就是这个类型。

- **Reinterpret cast**: forced conversion
  - Compilers keep silence on your (possibly evil) conversion
- **Const cast**: add/remove the **const** modifier
  - Used when the compiler complains on related issues
- **Regular cast (C-style cast)**
  - Using casting methods above until one succeeds

还支持一些强制转换的函数。

## Private fields and methods

---

- The `private` keyword can be used for information hiding
  - Private fields/methods cannot be accessed outside the object
- Privacy is enforced by type-checking
  - Encountering `c.x/c.f()` → check if `x/f` is private

真实的数据我们可以通过 `private` 隐藏起来，对外暴露一些函数。

## Private fields and methods

---

- Different languages have different protection rules for private fields/methods
  - Accessible only to the class that declares them
  - Accessible to the declaring class and any subclasses (`protected` in C++)
  - Accessible only within the same module as the declaring class (package, namespace)
  - Read-only outside the declaring class, but writable by methods of the class

不同语言对于 `private` 的含义是有分歧的。比如认为只有自己这个类能访问，子类能不能访问引入了 `protected`。还有可能是同一个 `module` 都可以访问到这个变量。

`java` 可以通过一个紧密的方式把数据放在一起，这样就会比很多间接引用的访问比较快。

## Classless languages

---

- Some OO languages do not use **class** at all
- Each object has whatever methods/fields it wants
  - Type checking is usually dynamic (done at runtime)
- A typical classless language is JavaScript

```
var myObj = {
 method0 () {
 return 0;
 }
}
← define methods in an object
```

51

每个对象想定义什么语言就定义什么语言，所以它就是动态类型的。

## Object creation in classless languages

---

- Many objects are created by *cloning*
  - Copying from an existing object and then modifying/extending it
  - Copied objects share the same descriptor with the original (template) object → "pseudo-classes"
- JavaScript can create objects with **constructor functions**

每个对象隐含了类型信息，我们把它叫做“伪类”。如果我们想复用一些函数，我们可以对对象 copy 完以后继续添加新的方法。

## A JavaScript example for object creation

- First, defining an object constructor
  - Similar to the constructor in a class

```
function Person(name, id) {
 this.name = name;
 this.id = id;
}
```

- Then creating objects with the `new` keyword
  - Not that different with class-based, right?

```
var me = new Person("Mingyu", 5150379000);
```

## TypeScript: JavaScript + Types

- Dynamic check disables opportunities to find bugs during compilation (static)
  - A bug might be exposed when being deployed...
- TypeScript enforces static checks for JS



要进一步发展要解决软件工程问题，老是动态做的话很容易出 bug。所以现在有 typescript，让类型往静态方向调整。

## 2021/12/31

上节课我们提到了 TypeScript。可以看到它的官网的卖点就是 Scale 和 Safety。它给的例子里面，第一个也是检查出来错误的一个情况。所以这个语言本身就是给大家一个更加安全的 js 的状态，这也反映出来的了动态类型的一些问题。加了这些类型以后，我们就可以把很多错误暴露在编译器，对于整个开发流程有好处。

compiler 怎么编译 classless 呢？这个过程和我们之前讲的动态链接和多重继承的哈希表比较像。因为我们的类型信息都是动态生成的，我们可以动态扩展一个类的成员和方法。编译器很难判断有什么类型，最好就是弄一个哈希表各管各的，尽可能少地考虑继承的问题。当然一些全局的优化也是可以做的。

我们知道 js 是 runtime 的语言，和 Java 有点相似的地方，比如说它们都有一个 runtime，js 这里用的是 v8 引擎，也有垃圾回收和动态编译，所以说全局的分析和优化也可以用在 js

的编译优化上。

## De-virtualizing

这边我们简单讲一下面向对象的优化，一个比较重要的优化就是 De-virtualizing

### De-virtualizing dynamic method calls

---

- A very important optimization for OO programs
- Why is it important?
  - Method lookup is costly
    - Even SI is costly as it hinders sw/hw optimizations
    - MI/classless is even more complicated

这是之前我们讲单一继承、多重继承、动态链接的情况都会有虚函数的，这是面向对象带来的。有了多态之后，虚函数就很普遍了。有了超类以后，我们重载虚函数去实现。所以去虚函数就是优化中很重要的一点，我们找一个 **virtual function** 是相对费时间的，就算是单一继承，我们也需要找虚表找到对应的方法，需要 2 次 **memory** 的访存。坏处就是我们不能提前预测要往哪里跳。

常规的 **jump** 是直接把地址写在后面的，如 **jump 0x123456**。这是非常好的，因为我们在跳转之前就可以把这个地址 **prefetch** 到我们的 **cache** 中里。但是对于我们现在这个情况，我们是 **jump** 到一个 **memory** 对应的地方，比如 **%rax**，我们就不能做预测。我们知道在我们的 **y86** 上没有这种情况，因为我们在 **jump** 的时候清楚地知道我们的 **label** 在哪里，但是对于 **indirect jump** 的话，我们知道流水线是先判断再访存。现在访存都不知道在哪，我们怎么 **jump** 呢？

所以这件事情对于流水线不太友好，对于多重继承和 **classless** 就更复杂了。我们讲了几种方法。一种方法是图染色，我们先往下去找，得到 **offset**，拿到 **offset** 之后再去拿。哈希表更复杂一些，因为哈希的查找开销会更高，所以我们还是希望没有虚函数找虚表的这个过程。

### How to de-virtualize?

---

- A global program analysis is required
  - The type hierarchy for each call site should be analyzed
- Rule 1: A method can be de-virtualized with no overridden alternatives
  - All call sites should invoke the same method

怎么做 **de-virtualize** 呢？我们假设方法调用都是知道的，我们可以做一个程序的全局分析，把代码扫一遍。在做全局分析的过程中，我们会知道每一个调用虚函数的地方，都会分析它类型的 **hierarchy**。

举个例子来说，如下的 C++ 代码：

```
A a();
a.method();
```

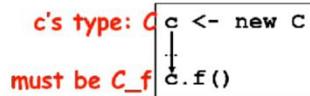
我们肯定知道 `a` 是一个 `A` 类型，那么这里的 `method` 一定是 `A` 类中的 `method`。如果 `A` 类没有自己定义，那么就是父类 `B` 里的 `method`。这种是最简单的情况。

第一个 rule：全局只有一个这个方法的实现，那么我们可以直接 invoke 来做 de-virtualize。

第二个 rule：就是 new C 的话，调用的 `c.method` 一定是 `C` 类的方法。我们就可以把 `jump *rax` 替换为一个 jump label。

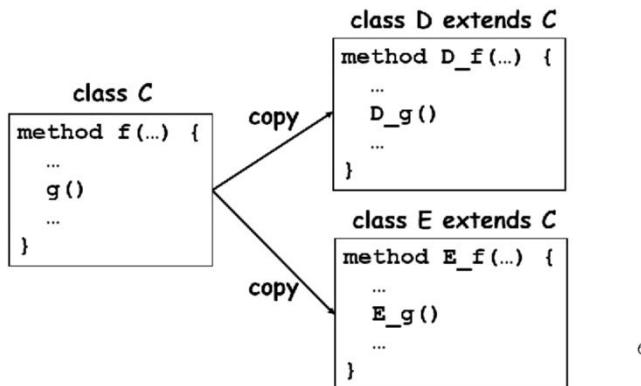
## How to de-virtualize?

- Rule 2: A method can be de-virtualized if the caller's type can be determined
  - Combined with other analysis (type propagation)
  - E.g., although a function `f` can be overridden, but we can determine that the caller `c` must be type `C`



## How to de-virtualize?

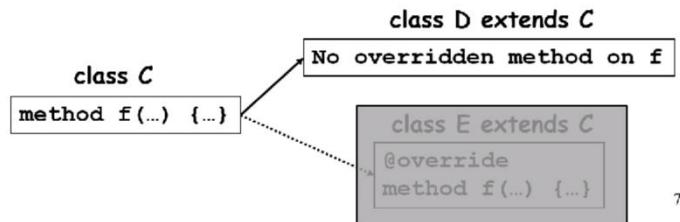
- Rule 3: A method can be de-virtualized if the virtual method is copied for sub-classes



第三种就是有 trade-off 的方法，也就是继承的时候把的方法都 copy 一份。这样调用的时候，每个类都可以走自己的位置。

## How to de-virtualize? (more aggressive)

- A method can be de-virtualized if the virtual method has only one implementation in all **loaded classes**
  - Only loaded classes can be used
  - De-optimize if the invariant is violated

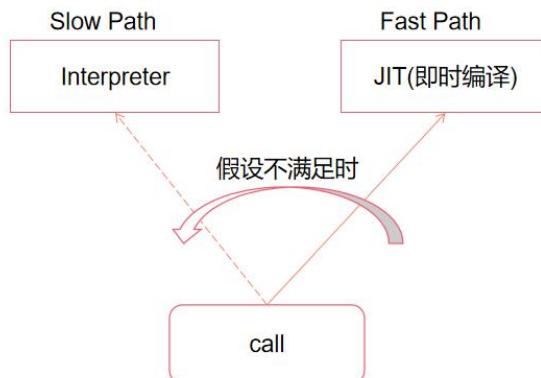


最后一个是假设的优化，可能可以用在更加动态的语言中。因为 **dynamic** 的环境中，我们可以动态加载一些类型进来。我们的优化是基于已经加载的类来做的，我们可以做一些投机的优化，后面如果出现 **invariant** 不一样了，后面再做去优化的过程。也就是我们现在创建了原先的 **slow path** 和基于某个假设做的 **fast path** 的优化，当我们加载新的类进来导致我们优化所假设的条件不成立了以后，我们再从 **fast path** 降级到 **slow path** 做。这种在动态语言里比较常见，比如 **java**、**python**、**js** 的解释器就可以认为是保守的 **slow path**。

比如我们有一个类 **C**，有一个方法 **f**。**D** 没有对 **C** override，而 **E** 对 **f** 做了 override，但是我们从来没有定义过和 **E** 相关的变量。此时我们整个 **f** 只有一个 **C** 的定义，在这样的一个类型系统下，我们可以对 **f** 做 de-virtualize，因为满足 rule 1，可以认为是没有 overridden 的情况，这样就可以直接调用 **f**，不做 look-up。

但是我们一旦加载了 **E**，会对我们整个结构产生影响，因为 **E** 会外挂一个自己对 **f** 的定义，此时我们对 **f** 的 rule 1 假设就不符合了，我们就要回到原先的 **loop-up** 的情况。

在 **Java** 中，它同时存在两套代码，分别通过解释器和即时编译执行。而我们的 **runtime code** 中会有一个 **switch** 机制，一旦假设不符合的情况下，我们就切换回 **slow path** 的 **interpreter** 的情况。



坏处就是代码有两份，有时候 **JVM** 的内存占用可能是因为我们编译的代码太多了。这就是动态语言的好处，是基于动态的 **profile** 的优化，更加 **aggressive** 的一个方法。

## More C++

前面就讲完了书上的面向对象部分，最后我们来讲一下 C++ 的部分。

### Memory Management in C++

---

- C++ does not have a garbage collector
  - Users need to manually manage their objects
    - E.g., new/delete
  - Hard for complicated programs
- Several common bug patterns come from memory
  - Use-after-free, double-free, memory leak...

对于 C++98 来说，是和 C 差不多的。内存都是程序员自己来管。简单的比较好做，因为我们有构造器和析构器。比如我们在 A() 构造器中有 new 的话，那么就在 ~A() 析构器中就 delete 即可。

复杂一点的程序是很难管理的。这也是为什么 JAVA 受欢迎的原因，因为 JAVA 有 GC，很少有 memory leak 的情况。

因为 C++ 的特性，所以经常出现（甚至在内核代码中）都会有 use-after-free, double-free, memory leak 这种 bug。能不能给 C++ 加 GC 呢？我们看了 lab7, tiger 纯静态编译也可以加一个 GC 进去，所以这也完全不可能。

### Can We Add GC to C++?

---

- It is doable if we consider our lab7
  - Data layout (modifying the object metadata)
  - Root detection (add pointer maps before call)
  - Barriers (add during compilation)
  - Triggering (replace the original allocator)
- But hard to imagine...
  - C++ is built for efficiency (GC has its overhead)
  - How to remain compatible with other feature?

对于 C++ 来说，因为它本身有虚表这种东西，我们可以改虚表的格式。包括说 root-detection 这种事情，在 call 的时候可以加一个 bitmap 来做。barrier 是说在 GC 中需要有一些不变量来执行并行/增量垃圾回收。triggering 可以让 allocator 来主动触发垃圾回收。C++ 之父说：也不是不可以，但是怎么和以前的 feature 兼容是比较麻烦的。另一个角度，c++ 和 c 更像，更加注重 efficiency。GC 是有性能损耗的。

## A Middle Ground, Perhaps?

---

- Programmers can manually manage objects as they want
  - E.g., malloc/free, new/delete
- But they can also rely on runtime libraries
  - For (perhaps) better reliability
- The middle ground: RAII

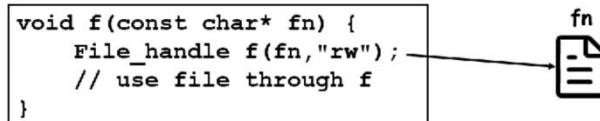
## RAII (Resource Acquisition is Initialization)

c++也不是完全没有提内存管理的方法，比如 RAII，它是一种泛化的思想，并不局限于内存管理。它现在是一种 STL 的方式来辅助使用，并不强制我们使用。

### Resource Acquisition is Initialization (RAII)

---

- Not a specific implementation, but a methodology
  - The life cycle of resource = its corresponding object
  - A file object/handle -> an opened file



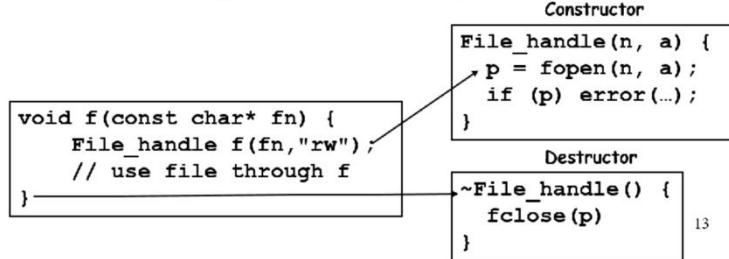
RAII 不是一个具体的实现，更像是一个方法论和思想，如果我们使用它的话，会让我们编程的鲁棒性更好。没有人强制 c++ 中不能使用 malloc free 等。

RAII 其实就是 resource acquisition is initialization 的简称。在面向对象中，万物皆对象，比如文件在 C++ 中也可以认为是一个对象，平时我们用的网络、硬盘、内存都可以抽象成对象，有这么一个概念之后，我们在初始化对象的时候，我们做的就是 resource acquisition，我们把资源和对象绑定在一起了，它们的生命周期应当是一样的。当我们 object 释放的时候，我们也同时释放资源。

上面的代码用了 c-style，所以 RAII 并不局限于面向对象和 C++。打开以后，我们的 `file_handle` 就对应了文件的使用。

## Resource Acquisition is Initialization (RAII)

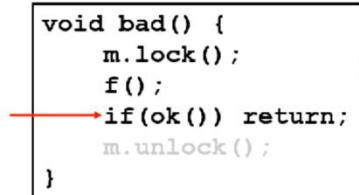
- Not a specific implementation, but a methodology
  - The life cycle of resource = its corresponding object
  - A file object/handle → an opened file



当我们函数退出的时候，`f`自然释放，`c++ runtime`会帮我们调用 `destructor`。这样就实现了一个自然的资源管理。这样就比我们写 `new` 和 `delete` 要简单一些。我们把资源管理很自然地对应到了程序对象的生命周期。

## Any Resources Can be Managed by RAII

- Memory/files are just examples
- Another example: locks
  - Bugs are raised when we forget to unlock
    - What if we return when ok() is true?



上面就是一个把锁认为是资源的例子。下学期讲 OS 的时候，我们还可以把锁认为是访问变量的一个权限 `token`。我们拿到锁才能有访问某个变量的能力。

通常锁就是 `lock` 和 `unlock`，但是问题就是说拿锁放锁容易写错。如上例所示忘记放锁的情况。

## Any Resources Can be Managed by RAI

- Memory/files are just examples
- Another example: locks
  - RAI: adding a lock guard object
    - Mutex acquisition is its initialization
    - Mutex is released when lk leaves its scope

```
void good() {
 std::lock_guard<std::mutex> lk(m);
 f();
 if(ok()) return;
}
```

解决方法就是加了对锁的引用，lk 把拿锁和放锁封装起来了。lk 初始化的函数就是拿锁了，而离开 scope 的时候，析构器就会自然放锁。这样就把 pair 放到了变量里去，把 pair function 的问题隐藏掉了。

## Looking into Source Code

- The C++ standard library (STL) from Microsoft
  - Manipulating the mutex in constructor/destructor

```
template<class _Mutex>
class lock_guard {
public:
 explicit lock_guard(_Mutex& _Mtx) : _MyMutex(_Mtx) {
 _MyMutex.lock(); // no exception is thrown in it
 }
 ~lock_guard() noexcept {
 _MyMutex.unlock();
 }
private:
 _Mutex& _MyMutex;
}
```

16

这里可以清楚地看到 \_MyMutex 的构造的时候就是加锁，而析构的时候就是放锁。任意的资源都可以这样管，我们在 C++98 的时候也可以这样去写。

## What About Dynamic-allocated Memory?

- We can apply RAI to pointers
  - So we get "smart pointers"
    - Initialization: acquiring an object (as resources)
    - Deconstructing: automatically remove the object
- In C++ STL: unique pointers
  - The name suggests it is "unique" to an object

讲了这些以后，我们回到 smart pointer 上。因为动态分配的内存是不知道什么时候恢复的，我们也用 RAI 这个思路就得到了智能指针这件事情。当我们 acquire 的时候，就分配一块内存，而析构的时候就把对象回收。

unique pointer 就暗示了 pointer 的独有性，它会被一个对象独占。

### An Example for Unique Pointers

```
Class Person {
 string name;
 int id;
 Person(name, id){...}
}

int main() {
 std::unique_ptr<Person> p(new Person ("wmy", 5150379000));
}
```

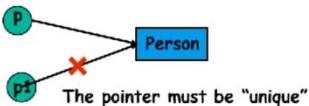
creating a unique pointer wrapping a created object



### An Example for Unique Pointers

```
Class Person {
 string name;
 int id;
 Person(name, id){...}
}

int main() {
 std::unique_ptr<Person> p(new Person ("wmy", 5150379000));
 std::unique_ptr<Person> p1 = p; // Compile error!
}
```



19

### What About Dynamic-allocated Memory?

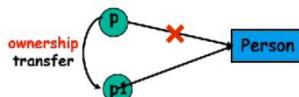
- We can apply RAI<sup>I</sup> to pointers
  - So we get "smart pointers"
    - Initialization: acquiring an object (as resources)
    - Deconstructing: automatically remove the object
- In C++ STL: unique pointers
  - The name suggests it is "unique" to an object

CPU 的 owner 就是当前在运行的程序，而在 unique\_ptr 的 owner 就是拥有 pointer 的那个人。move 就是把 ownership 转移给别人。

## An Example for Unique Pointers

```
Class Person {
 string name;
 int id;
 Person(name, id){...}
}

int main() {
 std::unique_ptr<Person> p(new Person ("wmy", 5150379000));
 std::unique_ptr<Person> p1 = std::move(p); // Using move
}
```



20

unique\_ptr 就三个东西：创建、move 和 deconstruct。

## Show Me the Code

|                                                                                                                                                                                                                                                                                                                                                                                                                                                      |                                   |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------|
| <b>element type</b>                                                                                                                                                                                                                                                                                                                                                                                                                                  | <b>deleter type (real delete)</b> |
| <pre>template &lt;class _Ty, class _Dx&gt;<br/>class unique_ptr {<br/>    using pointer = _Ty*;<br/>private:<br/>    _Compressed_pair&lt;_Dx, pointer&gt; _Mypair;<br/>public:<br/>    unique_ptr(pointer ptr, const _Dx&amp; _Dt):<br/>        _Mypair(..., _Dt, _Ptr){}<br/>    ~unique_ptr() noexcept {<br/>        if (_Mypair._Myval2) {<br/>            _Mypair._Get_first()(_Mypair._Myval2);<br/>        }<br/>    }<br/>    ...<br/>}</pre> |                                   |

23

STL delete 可以被继承，去做一个真正的 delete 工作。

## Basic Idea for std::shared\_ptr

- Similar interfaces compared with unique\_ptr
  - But they can be copied
    - "std::shared\_ptr<Person> p1 = p" is OK
- Using reference counting to memorize the referents
  - A shared\_ptr constructor: +1
  - A shared\_ptr destructor: -1
  - Reaching 0: remove the objects
  - (Making it more similar to GC?)

C++讲完了，我们稍微看一下 Rust。C++是有 library 来实现的，没有一个强制性去要求 share\_pointer。而 rust 就做了很多的 enforcement，要求一定要按照这个语言要求去实现，

因为它的编译器检查非常严格，希望把很多问题暴露在编译器。方法是类似的。

## A Little Peek on Rust References

---

- More systematic definitions on ownership
  - Well-integrated into the language
  - But the idea is similar
  - Strict compiler checks reduce bugs in runtime

```
let s1 = String::from("hello"); // hello
let s2 = s1; // implicit move
println!("{}", s1); // ERROR
```

我们来看一下这个例子，`let` 就是一个赋值。`String::from` 可以认为是一个字符串常量 `hello`。在 C++ 中就会检查 copy constructor，而 rust 这里就是一个 move。move 后 `s1` 就变成 null 了，后面就不能 print。

好处就是一旦我们习惯了 rust 的思维，通过编译之后基本上很少出错。

## More Features/Thoughts Can Explore!

---

- RTTI (Run-Time Type Information)
  - `dynamic_cast` is one of the use case
- Lambda functions
- Co-routines
- Future and asynchronous programming

其实还有很多 feature 可以去探索。这些是 C++ 中已经支持的一些 feature。RTTI 可能就是 `typeid` 在运行时可以得到类型。C++ 中有相应的 lambda 和 closure 的支持。future 允许我们定义一个未来返回值的类型，也就是支持异步编程。C++ 不断在发展出现一些新 feature。

后续是复习内容，略。