CH2 词法分析

2.1 扫描处理

1.只有一个词义:保留字；无限多个语义:标识符

2.2 正则表达式

[abcd] 表示 (a|b|c|d)
[b-gM-Qkr] 表示 [bcdefgMNOPQkr]
M? 表示 (M|?) ，即 M 出现 0 次或 1 次
M+ 表示 (M · M*) ，即 M 出现至少 1 次
.表示除换行符外的所有单个字符
"a.+*"，引号中的字符串匹配其自身

2.3 有穷自动机

1.DFA:M含字母表Σ、状态集S、转换函数T:S×Σ→S、初始状态S0∈S 以及接受状态A属于S.
2.NFA:M 由字母表Σ、状态集S、转换函数T:S×(Σ∪{ε})→P(S)、初始状态S0及接受状A集合

2.4 正则表达式到 DFA

1.**NFA-DFA转换:**列出所有状态的ε闭包；将初始状态的ε闭包作为新的初始状态；然后计算每个新状态里各个字符的转移的闭包，作为新的状态，包含原接受状态的新状态都是接受状态。

2.5 DFA状态数最小化:

1. 将目前的**终止状态**放入一个集合，将其它状态放入另一个集合
2. 遍历字母表，如果两个状态输入相同的字符转化到不同的状态，则拆分到不同集合，如果相同则保留在同一个集合
3. 遍历结束后，处于同一集合的状态属于等价状态，合并为新状态
4. 回步骤 1，直到没有拆分出任何新集合

2.5 LEX

两条规则以消除二义性：Longest Match，选择第一个匹配的正则表达式

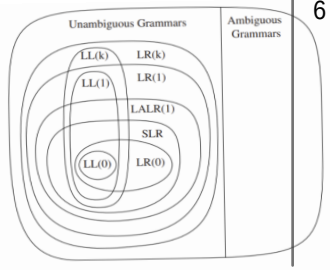
CH3 上下文无关文法分析

3.1 CFG

1.**二义性** Ambiguity:文法会使字符串有多个语法分析树，即该字符串有多种最 右推导
消除二义性例子：
E -> T+E | T , T -> F*T | F , F->id|(E)
2.分析树(concrete sytax tree)；AST (syntax_tree)去除了终结符和非终结符信息,仅保留了语义信息；

LL(1) Parsing Example　E→TX　T→int Y|(E)　X→+E|ε　Y→*T|ε

Stack	Input	Action
E$	int * int $	TX
TX$	int * int $	int Y
int YX$	int * int $	terminal
YX$	* int $	* T
* TX$	* int $	terminal
TX$	int $	int Y
int YX$	int $	terminal
YX$	$	ε
X$	$	ε
$	$	ACCEPT



3.2 递归下降分析 | Recursive Descent Parsing

是自顶向下分析 (Top-Down Parsing) 的一种。Top-Down Parsing，就是从顶 (start symbol) 开始，从上到下、从左到右构造 parse tree
流程：遍历语法树每个结点，若是Terminal则看下一个输入是否是该Terminal，是则接受，否则backtrack；若是Non-terminal，则尝试转化为其第一个产生式，继续检查，如果出问题就回溯回来尝试下一个产生式，如果回溯回来发现产生式都检查完了则出问题了，回溯。左递归Left-Recursive
S推导式最左边是S就是左递归，可转化为右递归：

$$S \to S\alpha_1|\dots|S\alpha_n|\beta_1|\dots|\beta_m$$
$$S \to \beta_1 S'|\dots|\beta_m S',\ S' \to \alpha_1 S'|\dots|\alpha_n S'|\epsilon$$

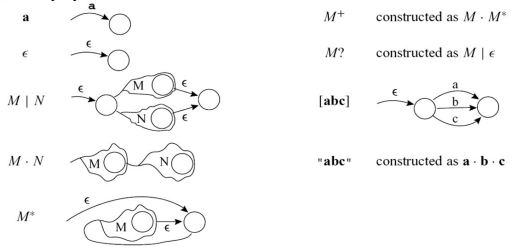
3.3 LL(1) Parsing

1.预测分析 (Predictive Parsing) 也是一种 Top-Down Parsing,查看当前输入位置之后的几个 token 来决定使用哪一个 production，从而避免了 backtracking
第一个 L：left-to-right scan of input
第二个 L：leftmost derivation
k：predict based on k tokens lookahead
2.提取左因子 | Left Factoring

E -> T+E | T
E -> TX , X -> +E | epsilon
3.构造Predictive Table

- For each production $A \to \alpha$ in G do:
  - For each terminal $t \in First(\alpha)$ do
    - $T[A, t] = \alpha$
  - If $\varepsilon \in First(\alpha)$, then for each $t \in Follow(A)$ do
    - $T[A, t] = \alpha$

First Sets 注意包括epsilon
Follow Sets 注意不要包括epsilon
4.LL(1)动作：
①生成(generate),利用文法将栈顶的 N 替换成串,串反向进栈
②匹配(match):将栈顶的记号和下一个输入记号匹配 ③接受(accept):接受字符串④错误error
5.LL(1)文法是无二义性的，即任意规则A →α1 | α2，有First(α1)∩First(α2)为空
6. LL(1)左因子和左递归都要弄



3.4自底向上分析 | Bottom-Up Parsing

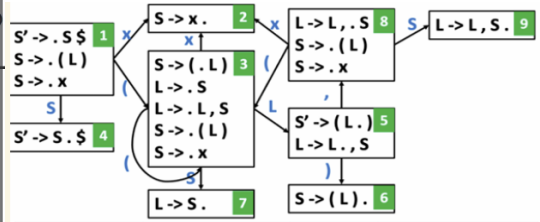
从输入的 token 流一步步简化为 start symbol，是最右分析的一个逆过程，所有操作都会发生在 | 紧邻的左右两边，只有 2 种操作：**Reduce**，规约：将 | 左边的一些 token 用某条 production 反向转换为 Non-terminal；**Shift**，移进：将 | 右移一位，即将下一个 terminal 纳入考虑。

3.5 LR(0) Parsing

1. LR(k) 表示 Left-to-Right Scan, Rightmost Derivation, Predict Based on k Tokens of Lookahead。LR(0)无二义性。
2.LR(0) Item即产生式加个点
3.LR(0) Parsing Table 的构造
先按规矩画出NFA，再转化为DFA并最小化
sk 表示将对应的 terminal 压栈，然后移到到状态 k；**gk** 表示将对应的 non-terminal 压栈，然后移到状态 k；**rk** 表示用第 k 条 production 规约，这里 r1, r2, r3 分别为 S'->S, S->(S)S, S->EPSILON；**accept** 表示接受。

- 对于 DFA 中的每一个 $S_i \xrightarrow{t} S_j$，我们令 `T[i, t] = sj`，其中 S_i 是状态 i，t 是一个 terminal；
- 对于 DFA 中的每一个 $S_i \xrightarrow{X} S_j$，我们令 `T[i, X] = gj`，其中 X 是一个 non-terminal；
- 对于每一个可以规约（即句点在 rhs 最后）的 LR(0) item，对其所在状态 i 以及每一个 terminal t，令 `T[i, t] = rk`，其中 k 是 LR(0) item 对应的 production 的序号。

4.LR(0) Parsing Table 的使用



Stack (states)	Stack (symbols)	Input	Action
1		(x) $	shift 3
1, 3	(	x) $	shift 2
1, 3, 2	(x	) $	reduce 2 S -> x
1, 3	(S	) $	goto 7
1, 3, 7	(S	) $	reduce 3 L -> S
1, 3	(L	) $	goto 5
1, 3, 5	(L	) $	shift 6
1, 3, 5, 6	(L)	$	reduce 1 S -> (L)
1	S	$	goto 4
1, 4	S	$	accept

3.6 SLR(1) Parsing

SLR(1) 分析表与LR(0) 分析表一致，除了rk的设置。SLR不可能是二义性。

- 对于每一个可以规约的 LR(0) item（假设是 `A->α.`），对其所在状态 i 以及每一个 terminal t，**如果 t ∈ FOLLOW(A)**，令 `T[i, t] = rk`，其中 k 是 LR(0) item 对应的 production 的序号。

唯一的区别是，LR(0) 对可归约项所在状态的所有 terminal 进行 reduce，而 SLR(1) 只对那些下一个符号在对应 non-terminal 的 Follow Set 的情况进行 reduce。表的使用方法也与LR(0)一样。

3.7 LR(1) Parsing

LR(1) 在 SLR(1) 的基础上进行了进一步优化。LR(1)item会跟上下一个的输入。

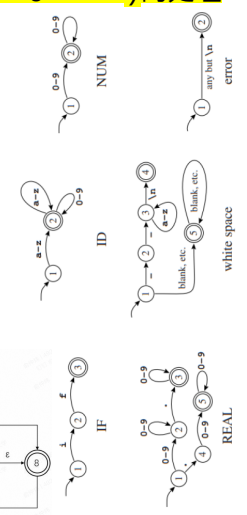
- 对于 DFA 中的每一个 $S_i \xrightarrow{t} S_j$，我们令 `T[i, t] = sj`，其中 S_i 是状态 i，t 是一个 terminal；
- 对于 DFA 中的每一个 $S_i \xrightarrow{X} S_j$，我们令 `T[i, X] = gj`，其中 X 是一个 non-terminal；
- 对于每一个可以规约（即句点在 rhs 最后）的 LR(1) item，对其所在状态 i 以及每一个 lookahead symbol t，令 `T[i, t] = rk`，其中 k 是 LR(1) item 对应的 production 的序号。

3.8 LALR(1) Parsing

LR(1) 的状态有点太多了,LR(1) 的简化版本 LALR(1) (Look-Ahead LR(1)) 被提出了。与LR(1)的区别是，对DFA状态进行合并：若状态里所有LR(1)item的产生式（包括点号位置）一样，只是后面跟的输入不一样，则可以合并，后面跟的输入用/分隔。可能引入rr冲突

悬挂else(dangling else)的处理

$S \to$ if E then S else S
$S \to$ if E then S
$S \to$ other
解决办法:引入新非终结符用于if-else的匹配问题,M是匹配的 else;U 是未匹配的 else
$S \to M$
$S \to U$
$M \to$ if E then M else M
$M \to$ other
$U \to$ if E then S
$U \to$ if E then M else U



First(X) (First Set of X) is computed iteratively:

- **Base case:**
  - if X is a terminal symbol, First(X) = {X}
  - if X is a non-terminal symbol, initialize First(X) = {}
- **Inductive case:** if X is a non-terminal symbol and X -> Y1 Y2 ... Yk, First(X) = First(X) ∪ First(Y1 Y2 ... Yk) = First(X) ∪ F1 ∪ F2 ∪ ... ∪ Fk, where:
  - F1 = First(Y1)
  - F2 = First(Y2) if Y1 $\stackrel{*}{\to}$ ε, otherwise ∅
  - ...
  - Fk = First(Yk) if Y1 Y2 ... Y_{k-1} $\stackrel{*}{\to}$ ε, otherwise ∅

Follow(X) (Follow Set of X) is computed iteratively:

- **Base case:** initially, we assume nothing follows X, i.e., initialize Follow(X) = {}
- **Inductive case:**
  - For any strings α, β, if Y -> αXβ, then Follow(X) = First(β)
  - For any strings α, β, if Y -> αXβ and β $\stackrel{*}{\to}$ ε, Follow(X) = Follow(X) ∪ Follow(Y)

Non-Recursive Predictive Parser

- S→(S) S | ε

M[N, T]	(	)	$
S	S -> (S)S	S -> ε	S -> ε

Steps	Parsing Stack	Input	Action
1	$S	()$	S -> (S)S
2	$S)S(	()$	match
3	$S)S	)$	S -> ε
4	$S)	)$	match
5	$S	$	S -> ε
6	$	$	**accept**

The symbol on top of the stack:

- Terminal: Match
- Non-terminal: Derive

CH5 语义分析

5.1 符号表(symbol table)

我们用哈希表实现符号表，哈希表每个元素是指针，指向一个链表，一条链表称为一个bucket

5.1命令式(imperative)

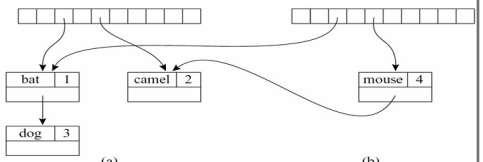
修改同一个表（比如哈希表）来加入新的绑定，就是实验文档里的那种

5.3 函数式(functional)

永不修改旧表，而是生成新表，新表中原本有bucket的位置会直接指向旧表的bucket，而不是复制过来

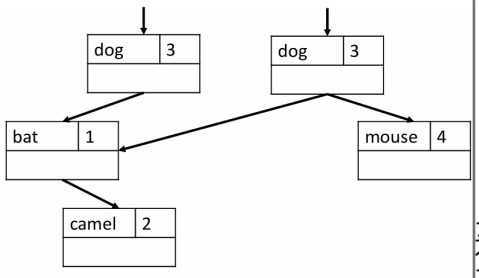
实现方法：二叉搜索树（BST）

特点：不会直接操作原符号表，而是创建新的 BST节点

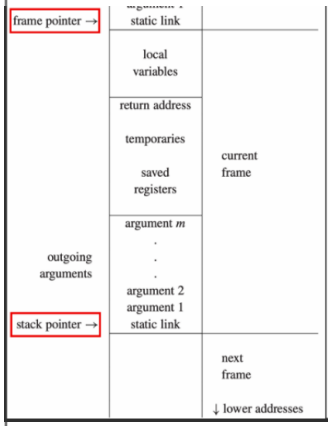


例子：m1={bat->1,camel->2,dog->3},

添加新绑定mouse→4，需要复制root到插入节点父节点路径上的所有结点，剩余部分共用).然后插入新节点，结果如下：



P5 Activation Records



寄存器存：部分函数参数；返回地址；函数返回值；部分本地变量。

栈存：额外函数参数；大体积变量；通过引用传的变量，或者这个变量有被取地址；有些语言允许函数内定义函数，内部函数访问外部函数参数时得存栈上；数组

嵌套函数变量访问问题

Static Links 是一个指向外部函数栈帧的指针。

全局数组 Display，以嵌套深度为 index 存储该深度函数的最近一次调用的栈帧地址。

Lambda Lifting编译时直接看调用的函数会用到哪些外部变量，直接当参数传进去

P6 Intermediate Code

Three-Address Code：x = y op z，四元式 quadruple存储三地址码：(op,x,y,z)

Intermediate Representation Trees

Expressions (T_exp): the computation of some value, possibly with side effects

CONST(*i*)	The integer constant *i*
NAME(*n*)	The symbolic constant *n* (assembly language label)
TEMP(*t*)	Temporary *t*. i.e., virtual register
BINOP(o, e1, e2)	The application of binary operator *o* to operands *e1*, *e2*. The integer arithmetic operators: **PLUS**, **MINUS**, **MUL**, **DIV**; the integer bitwise logical operators: **AND**, **OR**, **XOR**; the integer logical shift operators: **LSHIFT**, **RSHIFT**; the integer arithmetic right-shift: **ARSHIFT**.
MEM(*e*)	The contents of *wordSize* bytes of memory starting at address *e*. When MEM is used as the left child of a MOVE, it means "store", but anywhere else it means "fetch".
CALL(*f*, *l*)	A procedure call: the application of function *f* to argument list *l*. Evaluate from left to right.
ESEQ(*s*, *e*)	The statement *s* is evaluated for side effects, then *e* is evaluated for a result.

- evaluate 即求值
- ESEQ 是连接，先执行 statement 得到副作用（环境 B），再计算 e 得到结果作为表达式的返回

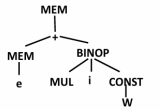
Statements (T_stm): performs side effects and control flow – no return value

MOVE(TEMP *t*, *e*)	Evaluate *e* and move it into temporary *t*.
MOVE(MEM(e_1), e_2)	Evaluate e_1, yielding address *a*. Then evaluate e_2, and store the result into *wordSize* bytes of memory starting at *a*.
EXP(*e*)	Evaluate *e* and discard the result.
JUMP(*e*, *labs*)	Transfer control (jump) to address *e*. The destination *e* may be a literal label, as in **NAME(*lab*)**, or it may be an address calculated by any other kind of expression.
CJUMP(*o*, e_1, e_2, *t*, *f*)	Evaluate e_1, e_2 in that order, yielding values *a*, *b*. Then compare *a*, *b* using the relational operator *o*. If the result is true, jump to *t*; otherwise jump to *f*.
SEQ(s_1, s_2)	The statement s_1 followed by s_2.
LABEL(*n*)	Define the constant value of name *n* to be the current machine code address.

Name(n) uses symbol n, Label(n) defines symbol n

- EXP(e) 计算表达式然后将结果丢掉，用来将表达式转化为语句（只需要表达式的副作用，不管其返回值）
- JUMP 指跳转到 e 的地址继续执行，注意这里 e 不一定是表达式
- CJUMP 为条件跳转，t 和 f 是两个地址
- SEQ 就是把两个语句连起来，功能类似 C 语言的分号

R tree of a[i] is

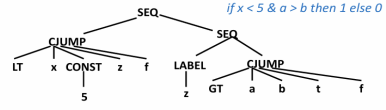


- MEM(e) represents a
- W is the word size

Example: ***if x < 5 then a > b else 0***

- x < 5 translates into Cx(s1)
- a > b will be translated as Cx(s2)

Similar to *if x < 5 & a > b then 1 else 0*



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

liminate ESEQs,将 ESEQ 往上提

- 如果 expression 是一个常量，那么它可以和任何 statement 进行交换
- 如果一个 statement 是空的，那么它可以和任何 expression 进行交换

move CALLs to top level

对于 BINOP(PLUS, CALL(...), CALL(...))，第二个CALL会覆盖第一个CALL的RV reg，需要用临时变量存储第一个返回值：**CALL(f, args)-> ESEQ(MOVE(TEMP t, CALL (f, args)), TEMP t)**

eliminate SEQ

重复使用：SEQ(SEQ(a, b), c) = SEQ(a, seq(b, c))，将左子树上的 SEQ 移动到右子树，变成 SEQ(s1, SEQ(s2, ..., SEQ(sn-1, sn)...))

Basic Blocks & Traces

分块的方法就是遇到label就是块的开始，遇到jump就是块的结束，如果一个块最后没有jump我们就加一个跳到下一条指令的jump

A trace is a sequence of statements that could be consecutively executed during the execution of the program and can include conditional branches

通过 DFS 搜索出所有的 trace，然后处理：

Perform some minor adjustments:

- **For any CJUMP immediately followed by its false label:**
  - Let alone (there will be many of these).
- **For any CJUMP followed by its true label:**
  - Switch the true and false labels and negate the condition.
- **For any CJUMP(cond, a, b, lt, lf) followed by neither label:**
  - invent a new false label lf'
  - rewrite the single CJUMP statement as three statements:

```
CJUMP(cond, a, b, lt, lf')
LABEL lf'
JUMP(NAME lf )
```

一个程序可以有多条不同的、重叠的 trace，我们希望构建出一组 traces，每个 block 只会出现在一条 trace 上，且每个 trace 都不会有循环，而且这一组 traces 刚好覆盖整个程序，且 trace 越少越好

p7-Instruction Selection

- Optimum tiling：全局最优，cost 最小的覆盖
- Optimal tiling: 局部最优，选择任意两个瓦片进行合并，代价没有变得更低

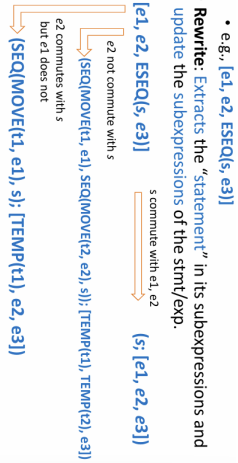
Canonical Trees

规范树定义：

1.不包含 SEQ 和 ESEQ 2.每个 CALL的父节点一定是 EXP 或 MOVE

性质：

1. 至多一个 statement，且一定是根节点；
2. CALL 的孩子一定不是 CALL；
3. CALL 的父节点一定是根节点；
4. 最多一个 CALL



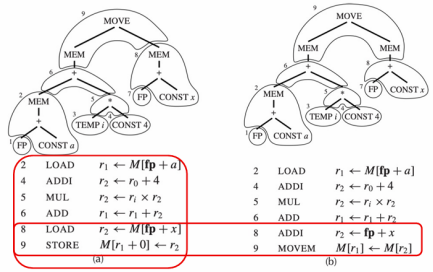
- **Rewrite**: Extracts the "statement" in its subexpressions and update the subexpressions of the stmt/exp.
- e.g., [e1, e2, ESEQ(s, e3)]

The Jouette architecture

Name	Effect	Trees
—	r_i	TEMP
ADD	$r_i \leftarrow r_j + r_k$	+
MUL	$r_i \leftarrow r_j \times r_k$	*
SUB	$r_i \leftarrow r_j - r_k$	-
DIV	$r_i \leftarrow r_j / r_k$	/
ADDI	$r_i \leftarrow r_j + c$	+ CONST
SUBI	$r_i \leftarrow r_j - c$	- CONST
LOAD	$r_i \leftarrow M[r_j + c]$	MEM + CONST
STORE	$M[r_j + c] \leftarrow r_i$	MOVE(MEM + CONST, MEM)
MOVEM	$M[r_j] \leftarrow M[r_i]$	MOVE(MEM, MEM)

- BINOP(PLUS, x, y) => +(x, y)
- r0 = 0
- The actual values of CONST and TEMP nodes will not always be shown.

- The tree of *a[i] := x* tiled in two ways:



- Tiles 1, 3, and 7 do not correspond to any machine instructions, because they are just registers (TEMPs).

Algorithms for Instruction Selection

- **Maximal Munch**：局部最优，保证任意两个瓦片都不能合并

认为越大的瓦片越好，贪心算法，自顶向下遍历 IR tree，用尽可能大的瓦片进行覆盖当前的结点，再递归检查剩下的，遇到多个选择时随便选一个即可

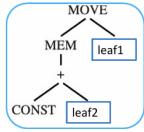
- Dynamic Programming：全局最优

就是自底向上，每到一个结点，计算最小的 cost，一直到根节点

- **The cost of node *x*** (denoted as *f(x)*) is the cost of the best tiling that covers the subtree rooted at *x*.
  - $f(x) = \min_{\forall \text{ tile } t \text{ covering } x} (c_t + \sum_{\forall \text{ leaf } i \text{ of } t} f(i))$
  - c_t refers to the cost of tile *t*.

Given the IR Tree with root node n

- First, the costs of all the children (and grandchildren, etc.) of node n are found recursively.
- Then, each tree pattern (tile kind) is matched against node n
- Each tile has zero or more leaves, where subtrees can be attached.
- For each tile *t* of cost c_t that matches at node *n*, the cost of matching tile *t* is: ($c_t + \sum_{\forall \text{ leaf } i \text{ of } t} f(i)$) *f(i)* has already been computed)
- The tree pattern leading to minimum cost is chosen



SPILL PRIORITY 注意10是循环次数，根据实际情况更改

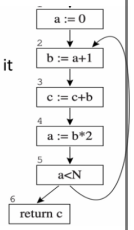
对于节点 a，其溢出优先级计算公式：$Priority = \frac{Out_{use+def} + 10 \times In_{use+def}}{D}$。其中，

- $Out_{use+def}$ 为 a 在循环外的 use 与 def 总数.
- $In_{use+def}$ 为 a 在循环内的 use 与 def 总数.
- D 为 a 的 degree，只包含实线。

优先级值越小，说明优先级越高。

p8-Liveness Analysis

- **def** of a *variable*: the set of graph nodes that define it
- **def** of a *graph node*: the set of variables that it defines
- **use** of a *variable* or *graph node* is similar.



- Example
  - def(3) = {c}, def(a) = {1, 4}
  - use(3) = {b, c}, use(a) = {2, 5}

A variable is live on an edge if:
- this variable will be used later
- this variable will not be re-defined before being used

Live-in: A variable is live-in at a node if it is live on any of the in-edges of that node;
Live-out: A variable is live-out at a node if it is live on any of the out-edges of the node

Notions:
- **in[n]**: the live-in set of node n (the variables that is live-in at node n)
- **out[n]**: the live-out set of node n (the variables that is live-out at node n)

$$in[n] = use[n] \cup (out[n] - def[n])$$
$$out[n] = \bigcup_{s \in succ[n]} in[s]$$

The algorithm for finding a solution to these equations by iteration:

```
for each n
  in[n] ←{}; out[n] ←{}
repeat
  for each n
    in'[n] ← in[n]; out'[n] ← out[n]
    in[n] ← use[n] ∪ (out[n] − def[n])
    out[n] ← ∪s ∈succ[n] in[s]
until in'[n] = in[n] and out'[n] = out[n] for all n
```

- use[n] and def[n] can be obtained by analyzing statement n, and thus are treated as known.
- **Rule 1**: If $a \in in[m]$ for $\forall m \in succ[n]$, then $a \in out[n]$
- **Rule 2**: If $a \in use[n]$, then $a \in in[n]$
  - i.e., if statement n uses variable a, a is live on entry of n
- **Rule 3**: If $a \in out[n]$ and $a \notin def[n]$, then $a \in in[n]$

Interference Graphs

- At any nonmove instruction n that defines a variable a, where out[n] = {b1, ..., bj}, add interference edges (a, b1), ..., (a, bj).
- At a move instruction n, which is a := c, where out[n] = {b1, ..., bk}, add interference edges (a, b1), ..., (a, bk) for any bi that is not the same as c.

- 第一条即我定义新变量时不应使用还会活跃的变量在使用的寄存器
- 第二条即针对进行复制操作的 MOVE 指令，我们不给 a 和 c 之间添加冲突边

此外，每条 MOVE 指令的两个变量用虚线连接，即合并边，如果已有干涉边就不用了

p9-Register Allocation

an NP-complete problem

Build & Simplify

画图，每个结点代表一个变量，一条边代表他们有冲突（不能被分配同一个 reg）
记我们有 k 个 reg，如果一个结点的邻居数量少于 k，就删掉这个结点
simplify 先只处理非 MOVE 相关结点
用 stack 按序储存删掉的结点，如此不断重复，直到没结点了，或剩下的都是 significant degree 的结点（邻居数 >= k）

Coalescing

如果两个结点是虚线（合并边）连接的，即这其中一个变量复制于另一个变量，且二者没有冲突边，那么就可以将他俩合并，但是新节点的邻居可能变多，所以我们采取保守合并，保证合并后的图不会变成非 k-colorable graph

1. **Briggs**(两者都不是预着色)
   - **新节点**应当有少于 K 个邻居是 significant degree 的
   - 确保 simplify 后能剩下少于 K 个邻居
2. **George**(有一个预着色)
   - 对于合并前的 A 和 B，需满足其中一个条件：
     - A 所有邻居都与 B 有冲突
     - A 所有邻居都是 in-significant degree
   - 符合第一个条件的边不会被重复增加至新节点，符合第二个条件的边能被优化，于是新节点的边被压制在合并前的数量

Coalescing 后再 simplify，直至去除所有合并边
如果没法去除所有合并边，进入 freeze 阶段，找一个 **low degree** 的 MOVE 相关结点，我们 freeze 这个 MOVE 操作，将虚线变为实线，放弃合并

Spill

只剩下 significant degree 的结点时，我们选择一些结点储存到 mem 而非 reg 里，这些节点与其它结点不再冲突，然后我们会将其删除并 push 进 stack,spill 后继续 Simplify，如此循环，直到生成空图

Select
从 stack pop 结点，一个一个染色，重建冲突图

p11-Garbage Collection

Mark-and-Sweep Collection
通过 DFS 来检索出所有结点并将其标记。
所有未标记的结点都是不可达结点，我们就通过 Sweep 进行回收，即线性遍历 record，Link the unmarked nodes in a linked list (freelist)，并且消除所有标记，以便下一轮回收。
这个算法的问题是使用了递归算法 DFS，栈可能爆，我们就用一个 explicit stack instead of recursion，改成非递归。

Pointer Reversal

基于上一种方法，但将 DFS 的 stack 储存在有向图里。
遇到新 record 时：
1. 标记 record
2. 将 record 里的指针反转，指向 DFS 路径的前继节点
3. 返回的时候恢复指针

Reference Counts

记录每个 record 被多少个指针指向，然后没人指向它就说明是不可达，直接回收
当然维护的开销也变大了，而且没法回收环垃圾

Copying Collection

将 mem 分为两部分：from-space：给用的，to-space：不给系统用的。
当 from-space is exhausted 时，copy all reachable records to to-space，反转两者角色，复制后的to space是紧凑的，没有 fragmentation。
复制时在其旧的 record 里设置一个 forwarding pointer 指向新的 record，借此确认该 record 有被复制，然后使用 Cheney's Algorithm，用 BFS 遍历，其引入了特殊指针 scan，对已复制的 record 进行遍历，检查并更新所有指针。

- **Pros**
  - Objects/records are not moved during GC
  - Be able to handle cyclic references

Pointer Reversal

- **Cons**
  - Normal execution must be suspended
  - Leads to fragmentation in the heap
    - Cache misses, more complex allocation
- **Pros:**
  - Simplicity: no stack or pointer reversal required
  - Run-time proportional to #rechable_objects
  - Leave free space contiguous
    - Automatic compaction eliminates fragmentation

Copying Collection

- **Cons:**
  - Half of memory is wasted
  - Poor locality (at least for the Cheney's algorithm)
  - Precise type information required (Pointer or not)

- **Pros:**
  - Simple to implement
  - Immediate collection
    - i.e., reduce the time between the object becoming garbage and its reclaimation
  - Incremental collection
    - cell management interleaved with program execution; no "stop-and-collect" effect

Reference Count

- **Cons:**
  - Fails to reclaim all unreachable objects
  - Can be slow if a large collection is initiated
  - Noticeably slows down assignments

CC算法代价：每一次回收H/2-R个字，摊还代价是每个分配字为c3*R/(H/2-R)条指令

CH6 活动记录

6.1 栈帧，也叫活动记录

- **定义：**栈中存放函数的局部变量/参数/返回地址/临时变量的这片区域为该函数的活动记录(activation record)或栈帧(stack frame).
- 帧指针的变化：一个函数g调用f时①sp指向g传给f的第一个参数②f分配栈帧(sp-栈帧大小)③进入f时旧的sp变成当前帧指针；fp旧值被保存到栈帧内，新的帧指针变成旧sp④f退出时把fp拷贝给sp，取回原先保存的fp即可
- 逃逸变量(elapsed variable)：①传地址变量 ②被取地址 ③被内层嵌套函数访问的变量.

Ch7 IR

对于cjump和jump,不知道label的具体值,需要使用真值标号回填表(true patch list)和假值标号回填表(false patch list)

简单变量：存放在栈帧的变量v转化为MEM(BINOP(PLUS, TEMP fp, CONST k)), k是栈帧内v的地址偏移.

追踪静态链：MEM(+(CONST Kn, MEM(+(CONST Kn-1, … MEM(+(CONST K1, TEMP fp))…))); k1~kn-1是各个嵌套函数的静态链位移

数组变量下标：a[i]表示为MEM(+(MEM(e), BINOP(MUL, I, CONST W))

7.3 声明

函数被翻译为**入口处理代码(prologue)/函数体(body)**和**出口处理函数(epilogue)**组成的汇编语言代码.

①入口处理函数包含：(1)声明一个函数开始的伪指令(2)函数名字的标号定义(3)调整栈指针的一条指令用于分配新的栈帧(4)将逃逸参数保存至栈帧的指令，以及将非逃逸参数传送的新临时寄存器指令(5)保存在此函数用到的callee-save寄存器 **②入口处理之后是：**(1)函数的函数体 **③出口函数位于函数体之后，包含：**(1)将返回值传送至专用与返回结果的寄存器(2)用于恢复callee-save的寄存器取数指令(3)恢复栈指针，释放栈帧(4)return指令(5)声明函数结束的伪指令

动规的比例常数比Maximal大，因为要遍历两遍

- Suppose:
  - K: the average matching tile contains K non-leaf (labeled) nodes.
  - N: the number of nodes in the input tree
  - K′: the largest number of nodes that ever need to be examined to see which tiles match at a given subtree.
  - T′: the number of different patterns (tiles) that match at each tree node, on average
- Maximal munch: proportional to (K′ + T′) N / K
- Dynamic programming: proportional to (K′ + T′) N
- K, K′ and T′ are constant, the running time of all of these algorithms are linear.

CH13 垃圾收集

13.2 标记-清扫式算法

Mark phase:
for each root v
  DFS(v)
function DFS(x)
if x is a pointer into the heap
  if record x is not marked
    mark record x
    for each field fi of record x
      DFS(x.fi)

Sweep phase:
$p \leftarrow$ first address in heap
while $p <$ last address in heap
  if record p is marked
    unmark p
  else let f_1 be the first field in p
    $p.f_1 \leftarrow$ freelist
    freelist $\leftarrow p$
  $p \leftarrow p$+(size of record p)

Cost of Mark-and-Sweep Collectio

- H: heap size, R: reachable data
- **Time of GC**
  - **Mark**: DFS takes time proportional to R
  - **Sweep**: takes time proportional to H.
  - Total time: $c_1R + c_2H$
- GC replenish the freelist with H-R words
- Amortized cost $(c_1R + c_2H)/(H - R)$
  - If R is closed to H, the cost is very High

Benefit: H words instead of H activation records

```
function DFS(x)
if x is a pointer and record x is not marked
    mark record x
    t ← 1
    stack[t] ← x
    while t > 0
        x ← stack[t]; t ← t − 1   // pop an item from the stack
        for each field fi of record x
            if x.fi is a pointer and record x.fi is not marked
                mark x.fi
                t ← t + 1; stack[t] ← x.fi
```

// push the start of DFS on stack

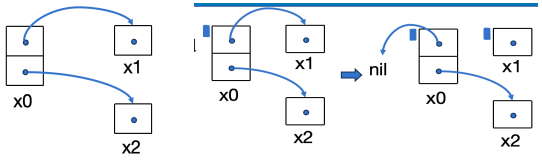
消除ESEQ的例子

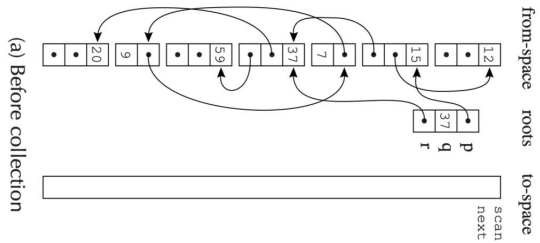
ESEQ(s_1, ESEQ(s_2, e)) ESEQ(SEQ(s_1, s_2), e)

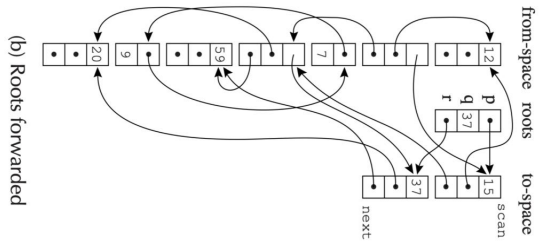
Previous	After
BINOP(op, e_1, ESEQ(s, e_2))	ESEQ(MOVE(TEMP t, e_1), ESEQ(s, BINOP(op, TEMP t, e_2)))
CJUMP(op, e_1, ESEQ(s, e_2), l_1, l_2)	SEQ(MOVE(TEMP t, e_1), s, CJUMP(op, TEMP t, e_2, l_1, l_2))

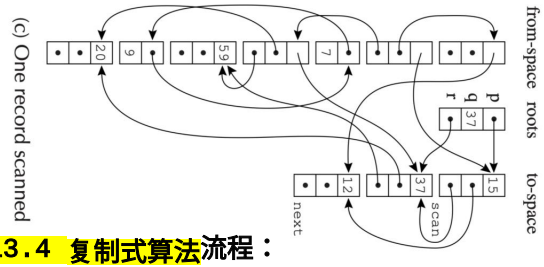
Pointer Reversal

```
function DFS(x)
if x is a pointer and record x is not marked
    t ← nil                                   • done: how many fields in each record
    mark x; done[x] ← 0                         have been processed
    while true                                • t: the top of the stack
        i ← done[x]
        if i < # of fields in record x
            y ← x. fi                         To better understand, suppose y points to record1
            if y is a pointer and record y is not marked
                x. fi ← t; t ← x; x ← y       // let x.fi point to the old top of stack (its DFS
                mark x; done[x] ← 0             parent) ; Push x to the stack; Start to process the
            else                                stored x.fi
                done[x] ← i + 1
        else
            y ← x; x ← t                      // store x to y, pop the top of the stack to x
            if x = nil then return            // back to the root node, terminate
            i ← done[x]
            t ← x. fi; x. fi ← y              // restore the correct link
            done[x] ← i + 1                   // start to process the next field of the DFS parent
```









13.4 复制式算法流程：

①收集初始化：初始化指针next指向to-space的开始.每当from-space发现一个可到达记录,便把它复制到to-space的next所指位置,同时式next增加该记录的大小

②转递(forwarding)：使一个指向from-space的指针p转而指向to-space

(1)p指向的使from-space中已经复制的记录,则p.fi是指明副本在何处的特殊的转递指针

(2)p指向 from-space中一个尚未复制过的记录，则将它复制到next所指的位置;同时将转递指针赋给p.f1

(3)p不是指针或者指向from-space以外的指针,对p不做任何事情.

传递指针算法伪代码

```
function Forward(p)
    if p points to from-space
    then if p. f1 points to to-space
        then return p. f1
        else for each field fi of p
                next. fi ← p. fi
            p. f1 ← next
            next ← next + size of record p
            return p. f1
    else return p
```

Cheney's Algorithm

```
scan ← next ← beginning of to-space
for each root r
    r ← Forward(r)    // increase next, but keep scan unchanged
while scan < next
    for each field fi of record at scan
        scan.fi ← Forward(scan.fi)
    scan ← scan + size of record at scan
```

①位于scan和next之间区域包含的是已复制到to-space但子域还没有传递的记录:子域指向from-space

②位于to-space开始和scan之间的是已复制并且以转递的记录,这一区域的所有指针均指向to-space.

③while循环会使得scan向next移动,复制记录也会导致next移动.

④当所有可达数据都被复制到to-space后scan追上next.

CH18 循环优化

Loop Invariant Hoisting

如果某个表达式的值在循环中不会改变，则称它是 loop-invariant 的。

如果操作数 v1 和 v2 都满足以下条件的任意一条，则可以判定赋值语句 x:=v1 OP v2 是 invariant 的：

操作数是常数

- 赋值语句用到的所有定义都在循环外
- 赋值语句仅用到一处定义，且该定义是 loop-invariant 的

即赋值语句中的操作数及其使用的变量必须是 loop-invariant 的，而且不能在循环中因为控制流跳转等原因导致变量的 def 不唯一。

```
L0: t := 0
    a := x
L1: i := i + 1 // not invariant, i在循环中被修改
    b := 7     // invariant, 操作数是常数
    t := a + b // invariant, a在循环外定义, b是invariant的
    *i := t    // not invariant, i在循环中被修改
    if i < N goto L1 else L2
L2: x := t
```

Dominators

必经节点，从入口节点到某节点所有路径都需要经过的节点，自己就是自己的必经节点

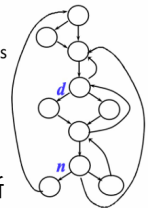
定理：必经节点是有序的，一个节点如果有两个必经节点，那这两个必经节点肯定是有一个是另一个的必经节点

idom(n) 即离 n 最近的非 n 必经节点，除 s0 外的节 点都有且只有一个

- **Algorithm for finding dominators:**
  - let D[n] be the set of nodes that dominate *n*, Then

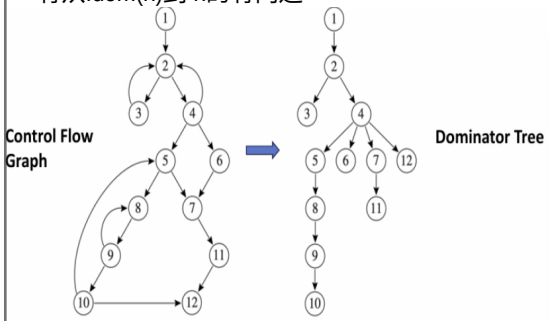
$$D[s_0] = \{s_0\} \qquad D[n] = \{n\} \cup \left(\bigcap_{p \in \text{pred}[n]} D[p] \right) \qquad \text{for } n \neq s_0$$

  - Initialize D[n] (for $n != s_0$) to hold all nodes in the graph.
  - Iteratively update D[n] based on predecessors until you reach a fixed point



Dominator Tree

有从 idom(n) 到 n 的有向边即可画出所有控制流图的节点，以及所有从idom(n)到 n的有向边



safely hoisting 的判据：当且仅当满足如下所有条件时可以提升形如 t:=a OP b的 loop invariant赋值语句 d：

- d 必须 dominate 所有 t 在其 live-out 集合中的 loop exits
- 在循环中对 t 赋值的语句是唯一的
- t 不属于 loop preheader 的 live-out 集合。换句话说，t在进入循环前不是活跃的。

Natural Loops

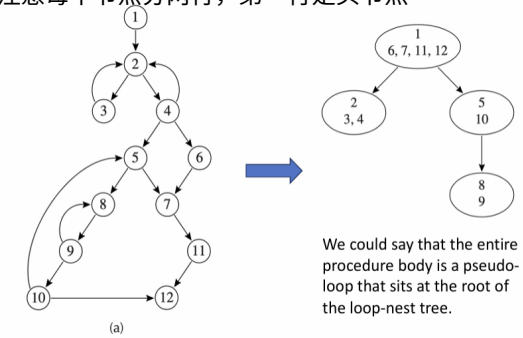
back edge指某节点到自己必经节点的 edge

对于一个 back edge，设其为 n-> h，则其 **Natural Loops**为包含这个 back edge 以及 h-> n 的这么个 循环，这个循环的所有点满足：h 是大家的必经节点 ，大家都有不经过 h 就能到 n 的路径

Loop-Nest Tree

循环嵌套图需要先画出 Dominator Tree，然后找出所有 back edges，以及它们对应的 natural loops。对每个 header,合并所有以h为 header 的 natural loops,得到loop[h]。构造循环 header 之间的树:如果h2在loop[h1]中，那么h1位于 h2上方。

注意每个节点分两行，第一行是头节点



Quiz Two

Consider the following grammar

S' → S $ S → AS S → ε S → (T)

T → T,S T → S A → a A → b

a. Calculate Nullable, FIRST and FOLLOW for nonterminals in the grammar
b. Construct the LL(1) parsing table for the grammar
c. Explain whether the grammar is LL(1) or not

a.

	Nullable	First	Follow
S'	F	(a b $	
S	T	(a b	$),
T	T	(, a b	,)
A	F	a b	$ a b ( ),

b.

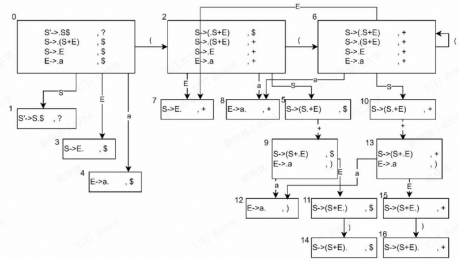
	a	b	(	)	,	$
S'	S'->S$	S'->S$	S'->S$			S'->S$
S	S->AS	S->AS	S->(T)	S->	S->	S->
T	T->S T->T,S	T->S T->T,S	T->S T->T,S	T->S	T->S T->T,S	
A	A->a	A->b				

c. No, because the LL(1) parsing table contains duplicate entries, indicating parsing conflicts.

Quiz Three

1 S'-> S$
2 S -> (S+E)
3 S -> E
4 E -> a

a. Construct the LR(1) states for the grammar above.
b. Give the LR(1) parsing table of the grammar above.



	a	(	+	)	a	$	S	E
0	s4	s2					g1	g3
1						a		
2	s8	s6					g5	g7
3						r3		
4						r4		
5			s9					
6	s8	s6					g10	g7
7			r3					
8			r4					
9	s12							g11
10			s13					
11				s14				
12				r4				
13	s12							g15
14						r2		
15				s16				
16			r2					

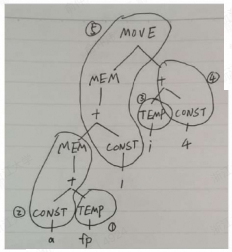
Quiz 4

For the following expression:

MOVE(MEM(+(MEM(+(CONST a, TEMP fp)), CONST 1)), +(TEMP i, CONST 4))

(1) Draw the IR Tree of this expression.
(2) Tile the IR Tree using Maximal Munch and the the *Jouette* architecture.
(3) Show the sequence of *Jouette* instructions of the tiling.

(1)(2)



(3)

```
2  LOAD   r1 <- M[a+fp]
4  ADDI   r2 <- i+4
5  STORE  M[r1+1] <- r2
```

2.1

a. Strings over the alphabet {a, b, c} where the first *a* precedes the first *b*.
c*a(a|c)*b(a|b|c)*

b. Strings over the alphabet {a, b, c} with an even number of *a*'s.
((b|c)*a(b|c)*a(b|c)*)*|(b|c)*

c. Binary numbers that are multiples of four.
(1|0)*00

d. Binary numbers that are greater than 101001.
10101(0|1)
| 1011(0|1)(0|1)
| 11(0|1)(0|1)(0|1)(0|1)
|(0|1)*1 (0|1)*(0|1)(0|1)(0|1)(0|1)(0|1)(0|1)(0|1)

e. Strings over the alphabet {a, b, c} that don't contain the contiguous sub-string
baa.
(a|c)*(b|bc(a|c)*|ba|bac(a|c)*)*

f. The language of nonnegative integer constants in C, where numbers beginning with 0 are octal constants and other numbers are decimal constants.
(00|0[1-7][0-7]*)|(0|[1-9][0-9]*)

g. Binary numbers *n* such that there exists an integer solution of $a^n + b^n = c^n$.
1(1|0)*

费马大定理告诉我们大于2没有**正整数解**，但题目是整数解，对于任意正整数 n，总能存在一组解 a = b = c = 0。

6.3 For each of the variables *a*, *b*, *c*, *d*, *e* in this C program, say whether the variable should be kept in memory or a register, and why.

```
int f(int a, int b)
{ int c[3], d, e;
  d=a+1;
  e=g(c, &b);
  return e+c[1]+b;
}
```

variable	in memory	in register	reason
a		1	pass function parameters in registers
b	1		passed by reference, accessed by a procedure
c	1		an array, accessed by a procedure
d		1	intermediate results of expressions, no used after the function g called
e		1	the function result

```
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
```

a. Show the sequence of machine instructions required to fetch the variable `output` into a register at line 14 of Program 6.3, using static links.
b. Show the machine instructions required if a display were used instead.

(1)

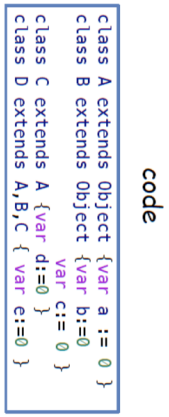
1. fetch indent's static link, i.e., the frame pointer to show
2. fetch show's static link, i.e., the frame pointer to prettyprint
3. fetch ***output*** from the frame of ***prettyprint***

(2)

1. fetch D[1], i.e., the frame pointer of ***prettyprint***
2. fetch ***output*** from the frame of ***prettyprint***

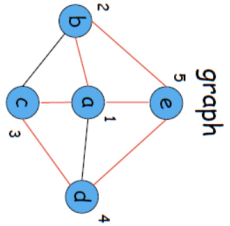
Multiple Inheritance of Data Fields

Field layout转化为 graph-coloring problem，edge 表示两个 fields 出现在同一个类中，field 最终的 offset 表示 color。如果不同 field 在同一个类中出现，则不能共享同一个 offset。
下图有误，所有边均应相连



code

coloring (with offset)



graph

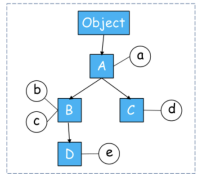
CH14 面向对象

在Tiger中实现类需要解决如下问题：
Field layout：类的 fields 如何在内存中分布。
Method dispatch：如何选择调用的具体 method 实现，或者说调用 method 的时候跳转到哪个地址。
Membership test：如何检查某 instance 是否属于某个类。

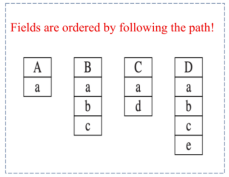
拓展的 Object-Tiger 中，存在一个预定义的类Object，不含 fields 或者 methods。使用 class B extends A {...}定义类B、扩展类A，A的 method 可以被B重载，重载需要保证参数和返回值类型一致；但是 field 不能被重载。

Single Inheritance的Field layout

使用 **prefixing**，如果B继承自A，将A的 fields 放在B的 record 的开头



type hierarchy



object layout

Method Code Generation

每个 method 编译成一段位于特定地址的 code，编译方式和普通函数几乎无异。语义分析阶段，每个变量都包含一个指向 class decriptor 的指针，descriptor 里面包含：一个指向父类的指针;包含这个类所有的 method instances 的列表;每个 method instance 对应于一个机器码 label。

Method dispatch

对于 static method 的调用 x.f()，编译器将会：
1.在对象 x对应的类 C中搜索 method f。
2.如果 C中有 f，则将 x.f()编译为到 label C_f 的 call，否则继续在父类中寻找。
3.直到在某个祖先 A中找到为止，则编译为 label A_f 的 call。
这个过程在编译时间即可完成，无需等到运行时。

对于 dynamic method 的调用 x.f()，编译器将会：1.在 x偏移为 0 处找到 class descriptor d。2.由于方法 f的偏移量是确定的（记为 F），从 d 中偏移为 F处获取 method-instance pointer p。3.call p

safely hoisting

违反规则1:

```
// Before hoisting
for (int i = 0; i < n; i++) {
    if (cond) t = a + b;
    // do something
}
use(t); // t used after loop

// After hoisting
t = a + b;
for (int i = 0; i < n; i++) {
    // if (cond) {}
    // do something
}
use(t); // 这里引用的 t 的值不再受 cond 影响
```

违反规则2:

```
// Before hoisting
for (i = 0; i < n; i++) {
    t = a + b;
    // do something
    t = c + d;
    use(t);
}

// After hoisting
t = c + d;
for (i = 0; i < n; i++) {
    t = a + b;
    // do something
    use(t); // 错误
}
```

将 t:=a+b 移动到 L1 之前会导致 *i:=t 使用的 t 值错误。

```
L0: t := 0
L1: i := i + 1
    *i := t // Uses t
    t := a + b // Invariant
    if i < N goto L1 else L2
L2: x := t
```

实际原因是违反了下文规则3。
最开始提到的 loop 与这里的 natural loop 是同一个东西。
根据回边 $x \to h$ 构建 natural loop 的算法：

1. Loop ← $\{h\}$
2. WorkList ← $\{x\}$
3. 当 WorkList 非空时：
   - 取出节点 p
   - 如果 $p \notin$ Loop:
     - 将 p 加入 Loop
     - 将 p 的所有前驱节点加入 WorkList（但不包括回边 $x \to h$）

对于 dynamic method 的处理略复杂：

- 每个 class descriptor 维护一个 dispatch vector（例如 C++ 中的虚表 vtable = virtual table），以及与每个 dynamic method name 对应的 method instance。
- 类似于 field layout，当B继承于A时，B的 method table 中先是指向A的 method 的 entry，再是由B定义的新的 method。
- 如果B中某个 method 重载了A中的，则对应 entry 指向B的实现。

Translate each of these expressions into IR trees, but using the `Ex`, `Nx`, and `Cx` constructors as appropriate. In each case, just draw pictures of the trees; an `Ex` tree will be a Tree `exp`, an `Nx` tree will be a Tree `stm`, and a `Cx` tree will be a `stm` with holes labeled *true* and *false* into which labels can later be placed.

a. `a+5`

b. `output := concat(output,s)`, as it appears on line 8 of Program 6.3. The `concat` function is part of the standard library (see page 525), and for purposes of computing its static link, assume it is at the same level of nesting as the `prettyprint` function.

c. `b[i+1]:=0`

d. `(c:=a+1; c*c)`

e. `while a>0 do a := a-1`

f. `a<b` moves a 1 or 0 into some newly defined temporary, and whose right-hand side is the temporary.

g. `if a then b else c`, where `a` is an integer variable (true if $\neq 0$).

h. `a := x+y`

i. `if a<b then a else b`

j. `if a<b then c:=a else c:=b`

a.

```
+(TEMP a, CONST 5)
```

b.

在Tree Language中没有找到合适的操作来表示函数的参数。
这里可能需要考虑static link的情况。

```
MOVE(MEM(
      +(CONST ko, MEM(
        +(CONST ks1, TEMP FP)))),
    CALL(NAME concat,
    EXPList(
      MEM(
        +(CONST ko, MEM(
          +(CONST ks1, TEMP FP)))),
      TEMP s))))
```

c.

假设 b 是由 MEM(TEMP b) 表示的数组变量：

```
MOVE(MEM(+(MEM(TEMP b), BINOP(MUL, +(TEMP i, CONST 1), CONST W))), CONST 0)
```

d.

```
ESEQ(MOVE(TEMP c, +(TEMP a, CONST 1)), BINOP(MUL, TEMP c, TEMP c))
```

e.

如果按照教材翻译版109页的和111页提供的代码，连续的语句应该表示为SEQ(SEQ(SEQ(x1, x2), x3), x4)这样的形式，ESEQ同理。
但是按照书中118页这个图，SEQ也是可以置于右边的。
这里采用代码中描述的表示方法。

```
SEQ(LABEL test,
  SEQ(CJUMP(GT, TEMP a, CONST 0, t, done),
    SEQ(LABEL t,
      SEQ(MOVE(TEMP a, BINOP(MINUS, TEMP a, CONST 1),
        SEQ(JUMP(test), LABEL done)))))
```

f.

参考教材第111页的程序7-1

```
ESEQ(MOVE(TEMP c, 1),
  ESEQ(CJUMP(LT, TEMP a, TEMP b, t, f),
    ESEQ(LABEL f,
      ESEQ(MOVE(TEMP c, CONST 0),
        ESEQ(LABEL t, TEMP c)))))
```

j.

if e1 then e2 else e3会将e2和e3视为表达式，调用unEx函数
如果直接调用，结果如下：

```
ESEQ(CJUMP(LT, TEMP a, TEMP b, t, f),
  ESEQ(LABEL f,
    ESEQ(MOVE(TEMP r, ESEQ(MOVE(TEMP c, TEMP b), CONST 0)),
      ESEQ(JUMP(done),
        ESEQ(LABEL t,
          ESEQ(MOVE(TEMP r, ESEQ(MOVE(TEMP c, TEMP a), CONST 0)),
            ESEQ(JUMP(done),
              ESEQ(LABEL done, TEMP r)))))))
```

如果进行特殊处理，结果如下

```
ESEQ(CJUMP(LT, TEMP a, TEMP b, t, f),
  ESEQ(LABEL f,
    ESEQ(MOVE(TEMP c, TEMP b),
      ESEQ(JUMP(done),
        ESEQ(LABEL t,
          ESEQ(MOVE(TEMP c, TEMP a),
            ESEQ(JUMP(done),
              ESEQ(LABEL done, CONST 0)))))))
```

f :
$c \leftarrow r_3$
$p \leftarrow r_1$
if $p = 0$ goto L_1
$r_1 \leftarrow M[p]$
call f *(uses r_1, defines r_1, r_2)*
$s \leftarrow r_1$
$r_1 \leftarrow M[p+4]$
call f *(uses r_1, defines r_1, r_2)*
$t \leftarrow r_1$
$u \leftarrow s + t$
goto L_2
L_1 : $u \leftarrow 1$
L_2 : $r_1 \leftarrow u$
$r_3 \leftarrow c$
return *(uses r_1, r_3)*

g.

书中**条件表达式**一节有相应的描述

```
ESEQ(CJUMP(NEQ, TEMP a, CONST 0, t, f),
  ESEQ(LABLE f,
    ESEQ(MOVE(TEMP r, TEMP c),
      ESEQ(JUMP(done))
        ESEQ(LABEL t,
          ESEQ(MOVE(TEMP r, TEMP b),
            ESEQ(JUMP(done),
              ESEQ(LABEL done,
                TEMP r)))))))
```

h.

```
MOVE(TEMP a, +(TEMP x, TEMP y))
```

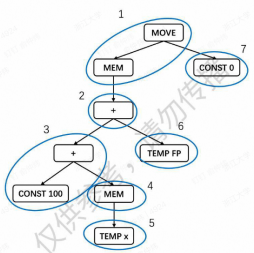
i.

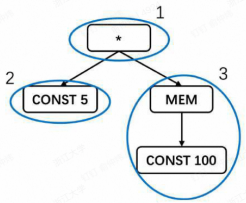
```
ESEQ(CJUMP(LT, TEMP a, TEMP b, t, f),
  ESEQ(LABEL f,
    ESEQ(MOVE(TEMP r, TEMP b),
      ESEQ(JUMP(done),
        ESEQ(LABEL t,
          ESEQ(MOVE(TEMP r, TEMP a),
            ESEQ(JUMP(done),
              ESEQ(LABEL done, TEMP r)))))))
```

9.1 For each of the following expressions, draw the tree and generate *Jouette*-machine instructions using Maximal Munch. Circle the tiles (as in Figure 9.2), but number them *in the order that they are munched*, and show the sequence of *Jouette* instructions that results.

a. MOVE(MEM(+(+(CONST$_{1000}$, MEM(TEMP$_x$)), TEMP$_{fp}$)), CONST$_0$)

b. BINOP(MUL, CONST$_5$, MEM(CONST$_{100}$))

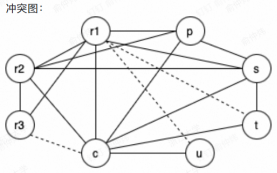





```
4 load r1 <- M[rx + 0]
3 ADDI r1 <- r1 + 100
2 ADD  r1 <- r1 + FP
7 ADDI r2 <- r0 + 0
1 STORE M[r1+0] <- r2
```

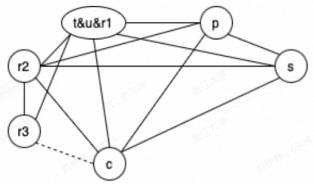
```
2 ADDI r1 <- r0 + 5
3 LOAD r2 <- M[r0 + 100]
1 MUL r1 <- r1 * r2
```

冲突图：



1. **Coalesce**

判断 t 和 r1 合并：因为 r1 为预着色节点，使用 George criterion 判断 t 的邻居，均符合 George 的条件。
u 同理。
因此，合并 t&u&r1。



2. **Freeze**

此时没有可以简化的节点了，也没有可以合并的节点。可以冻结c和r3之间的MOVE

3. **Spill c**

仍然没有可以简化的节点，也没有可以合并的节点，也不存在可以冻结的的节点。因此只能进行spill。

计算spill 分数，程序中没有循环，因此不需要考虑循环内的def和use

	defs	uses	degree	score
p	1	3	4	1
s	1	1	4	0.5
c	1	1	4	0.5

这里选择 c 进行 spill（s 和 c 都可以）

此时，degree[p] = 3, degree[s]=3

4. **Spill s**

此时，degree[p] = 2

5. **Simplify p**

此时图中只剩下了预着色节点

6. **Select**

Select Stack
p
s
c

pop p，color r3

pop s, actual spill

pop c, actual spill

7. **Rewrite**

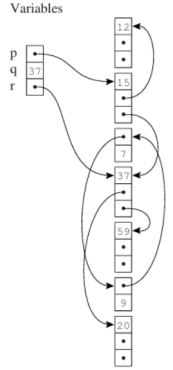
因为存在actual spill，因此对程序进行重写

```
        M[sloc] <- s1
        r1 <- M[p+4]
        call f
        t <- r1
        s2 <- M[sloc]
        u <- s2 + t
        goto L2
L1:     u <- 1
L2:     r1 <- u
        c2 <- M[cloc]
        r3 <- c2
        return
```

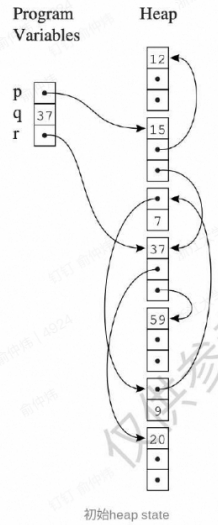
```
        c1 <- r3
        M[cloc] <- c1
        p <- r1
        if p=0 goto L1
        r1 <- M[p]
        call f
        s1 <- r1
```

13.2 Run Algorithm 13.6 (pointer reversal) on the heap of Figure 13.1. Show the state of the heap, the `done` flags, and variables `t`, `x`, and `y` at the time the node containing 59 is first marked.

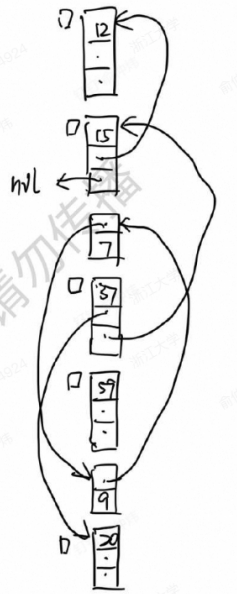
```
let
  type list = {link: list,
               key: int}
  type tree = {key: int,
               left: tree,
               right: tree}
  function maketree() = ···
  function showtree(t: tree) = ···
 in
  let var x := list{link=nil,key=7}
      var y := list{link=x,key=9}
   in x.link := y
  end;
  let var p := maketree()
      var r := p.right
      var q := r.key
   in garbage-collect here
      showtree(r)
  end
end
```



13.2



初始heap state



此时的heap state

done[15] = 2
done[12] = 3
done[37] = 2

done[20] = 3
done[59] = 0 or uninitialized

t → 37 所在 record
x → 59 所在 record
y → 59 所在 record