CH1 导入

1.编译器将高级语言变成汇编语言(Assembly Program)，一种用于编写和构建计算机指令的低级语言。它基于给定的机器指令集编写，可以被汇编器转换成为机器码。

2.前端：将代码文本转化为抽象表示，完成 lexing 和 parsing。它需要对语言进行理解，检查语法正确性；中端：将抽象表示变为中间形式，完成 type checking、analysis、optimizaition。编译器会尝试使用各种技术来改善程序效率和可读性，比如代码优化、循环展开、函数内联等；后端：负责将中间形式转换成目标机器代码。需要进行寄存器分配、代码生成、对象文件链接等操作，最终生成可执行文件。

3.一些小概念：模块有 a prefix unique to that module；每个 class 要有初始构造函数；右侧规则是结构体的 union 或者携带一个值

CH2 词法分析

2.1 scanning process 扫描处理

1.某些记号只有一个词义(lexeme)：保留字；某些记号有无限多个语义：标识符都由 ID 表示。

2.2 regular expression 正则表达式

0.不具备记忆、计数比较、递归的能力

1.M*>ab>a|b 顺序

2.相同的语言可以用不同的 RE 表示

3.R+：R(R*)；R?：R|ε；[abc]：a|b|c；[a-z]：(a|b|c|..|y|z)；[^ab]除 a 或 b；[^a-z]除 a-z

4.RE 匹配优先匹配保留字；最长字串优先

2.3 finite automata 有穷自动机

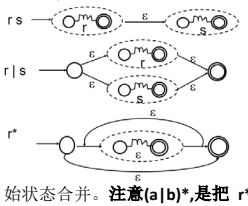
1.DFA：M 由字母表Σ、状态集 S、转换函数 T：S×Σ→S、初始状态 S_0∈S 以及接受状态 A⊂S。

2.错误状态默认不画，但是存在；错误状态下的任何转移均回到自身，永远无法进入接受。

3.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 S_0 以及接受状态 A 的集合。

2.4 RE To DFAs 正则表达式到 DFA

2.Thompson 结构通过ε转移将 NFAglue together

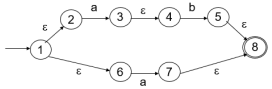


3.被合并的那个接受状态如果没有从它到其他状态的转移时，可以将该接受状态和后面的起始状态合并。**注意(a|b)*,是把 r*中间的换了，记得可以一个 e 直接过去**

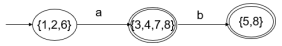
4.子集构造的过程：

首先列出所有状态的ε闭包；然后将初始状态的ε闭包作为新的初始状态；然后计算在每个新状态下在**各个字符上（都去做）**的转移的闭包作为新的状态，转移自然成为新的转移；包含原接受状态的所有新状态都是接受状态

PS：ε闭包首先包含自身。下面步骤缺一个所有状态的ε闭包；S 代表是哪几个状态的闭包得到的 S′



s	S′	S'_a	S'_b
1	1,2,6	3,4,7,8	
3,7	3,4,7,8		5,8
5	5,8		



4.DFA 状态数最小化：**最小状态数的 DFA 唯一**。

步骤：创建两个集合，一个包含所有接受，另一个是剩余；考虑每个字符上的转换，如果所有的接受在 a 上都有到接受的转换，或是都有到非接受的转换，那么这就定义了一个从新接受到新非接受的转移；如果两个接受有转移但是不在相同集合或是一个有转换，一个没转换，那么两个接受在该字符上被区分，从而分割出了新的状态集合；重复如此。

CH3 上下文无关文法与分析

3.2 CFG

1. 左推导：每次总是把最左边的非终结符转换；右推导：每次总是把最右边的非终结符转换左递归：定义 A 的推导式的右边第一个出现的是 A；右递归：定义 A 的推导式邮编最后一个出现的是 A；

3.3 Parse tree and AST 分析数和抽象语法树

1.同一个申存在多个推导即多个分析树

2.分析树(**concrete sytax tree**)是一个作了标记 labeled 的树，内部节点是非终结符，树叶是终结符；对一个内部节点运用推导时，推导结果从左到右依次成为该内部节点的子节点

3.最左推导和前序编号对应，最右推导后序

4.AST(syntax tree)去除了终结符和非终结符信息，仅保留了语义信息；一般用左孩子右兄弟

3.4 Ambiguity 二义性

1.定义：带有两个不同的分析树的串的文法

2.解决方法①设置消歧规则 disambiguating rule，在每个二义性情况下指出哪个是对的。无需对文法进行修改，但是语法结构就不是单纯依赖文法了，还需要规则②修改文法。

4.修改文法时需要同时保证优先级和结合律 precedence and associativity

5.在语法树中，越接近根，越高，优先级越低；左递归导致左结合，右递归导致右结合

6.将相同优先级的运算符分组叫做 precedence cascade 优先级联

7.通过最近嵌套规则 most closely nested rule 解决 else 悬挂问题；另一种方案是为 else 语句使用一个括号关键字(end if {} fi 都可)

8.inessential ambiguity 是无关紧要的二义性，虽然语法树各不相同，但是语义相同，例如算术加法虽然可结合但是结合顺序无关紧要

3.5 EBNF

1.A→a{b}表 b 可重复，花括号在右是左递归

A→a[b]表 b 可选

CH4 自顶向下分析

相比于自顶向下分析，自底向上分析更加高效，能够处理更复杂的语言结构

第一个 L 是从左到右处理，第二个 L 是最左推导，1 代表仅使用 1 个符号预测分析方向

4.1 recursive-descent 递归下降

1.将一个非终结符 A 的文法规则看作将识别 A 的一个过程的定义。递归下降需要使用 EBNF：将可选[]翻译成 if，将重复{}翻译成 while 循环

4.2 LL(1)

2.第一列标号；第二列为分析站内容，底座在左，栈底标注$；第三列显示了输入，从左到向右，$表示输入结束；第四列为动作

3.动作：①生成，利用文法将栈顶的 N 替换成串，串反向进栈②匹配：将栈顶的记号和下一个输入记号匹配③错误

• S→(S) S | ε

Step	Parsing	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

4.Definition of LL(1) Grammar: A grammar is and LL(1) grammar if the associated LL(1) parsing table has at most one production in each table entry.分析表中的每个项目中至多只有一个产生式。LL(1)文法是无二义性的

6.LL(1)面对重复和选择的解决方法：消除左递归 left recursion removal 和提取左因子 left factoring。

7.简单直接左递归：A → Aα|β，αβ是 N，且β不以 A 开头。$A \to \beta A', A' \to \alpha A' | \varepsilon$

8.普遍直接左递归：

$A \to A\alpha_1 | A\alpha_2 | \dots | A\alpha_n | \beta_1 | \beta_2 | \dots | \beta_m$

$A \to \beta_1 A' | \beta_2 A' | \dots | \beta_m A'$

$A \to \alpha_1 A' | \alpha_2 A' | \dots | \alpha_n A' | \varepsilon$

9.一般的左递归，不能带有ε产生式和循环

for i:=1 to m do

for j:=1 to i-1 do

replace each grammar rule choice of the form Ai→Ajβ by the rule

Ai → $\alpha_1\beta|\alpha_2\beta|\dots|\alpha_k\beta$, where Aj → $\alpha_1|\alpha_2|\dots|\alpha_k$ is the current rule for Aj

Remove, if necessary, immediate left recursion involving Ai 其中 m 是 N 的个数

10.提取左因子

$A \to \alpha\beta|\alpha\gamma$。$A \to \alpha A', A' \to \beta|\gamma$

4.3 first and follow sets

1.Fisrt 定义：令 X 为一个 T 或 N 或ε，Fisrt(X)由 T 或ε组成。①若 X 为 T 或ε，Fisrt(X)={X}②若 X 为 N，对于每个产生式 $X \to X_1 X_2 \dots X_n$，First(X)都包含了 $First(X_1)-\{\varepsilon\}$。若对于某个 i<n，所有的 $Fisrt(X_1),\dots,First(X_i)$都含有ε，则 First(X)也包括了 $First(X_{i+1})-\{\varepsilon\}$。若所有 $Fisrt(X_1),\dots,First(X_n)$都含有ε，则 First(X)也包含ε。

2.定理：A non-terminal A is nullable if and only if First(A) contains ε

3.Follow 定义：若 A 是一个 N，那么 Follow(A)由 T 和$组成。①若 A 是$，直接进入 Follow(A)②若存在产生式 B→αAγ，则 First(γ)-{ε}在 Follow(A)中 ③若存在产生式 B→αAγ，且ε在 First(γ)中，则 Follow(A)包括 Follow(B)

PS：③更常见的情况是 B→αA，那么 Follow(A)包括 Follow(B)

4.First 关注点在产生式左边，Follow 在右边

5.LL(1)分析表 M[N,T]的构造算法：为每个非终结符 A 和产生式 A→α重复以下两个步骤：①对于 First(α)中的每个记号 a，都将 A→α添加到项目 M[A,a]中 ②若ε在 First(α)中，则对于 Follow(A)中的每个元素 a(包括$)，都将 A→α添加到项目 M[A,a]中

FOLLOW(S) = {), $ }
FOLLOW(S') = {), $ }
FOLLOW(E) = { +), $ }

FIRST(E) = {num,(}
FIRST(S') = {+, ε}
FIRST(S) = {num,(}.

S→ES′ S′→ε|+S E→num|(S)

	num	+	(	)	$
S	S→ ES′		S→ ES′		
S'		S′→ +S		S′→ ε	S′→ ε
E	E→ num		E→ (S)		

6.LL(1)文法的判别：A grammar in BNF is LL(1) if the following conditions are satisfied.①For every production Ai→$\alpha_1|\alpha_2|\dots|\alpha_n$, $First(\alpha_i) \cap First(\alpha_j)$ is empty for all i and j, $1 \le i, j \le n, i \ne j$ ②For every non-terminal A such that First(A) contains ε, First(A)∩Follow (A) is empty.

4.5 error recovery

1.遇错后的不同层次反应：给出一个错误信息后①尽可能准确定位②尝试进行错误矫正 error repair③分析程序从错误程序中推断 infer 出正确程序

2.some important considerations:①尽快判断出错误的发生②错误发生后，必须挑选一个位置恢复 resume 分析，尽可能找到多的真的错误③避免出现错误级联(一个错牵出数个假错)④避免错误的无限循环

3.panic mode 应急模式，递归下降中的错误矫正。基本机制为每个递归过程提供一个额外的由同步记号组成的参数。遇到错误是，就向前扫描，并且一直丢弃记号知道遇到一个同步记号，从这里恢复分析。Follow 集合是同步记号中的重要一员。First 集合可以避免跳过开始新的主要结构的重要记号，也可以在更早时候检测错误。同步记号随着递归不断传递并增加新值。

4.LL(1)中没有递归，因此额外增加一个栈存同步记号，算法生成每个动作前，都调用 checkinput：或者在分析表中的空格中补全错误处理，共有三种可能①若当前输入为$或是在 Follow(A)中，将 A 从栈中弹出，记作 pop②当输入不是$或不在 First(A)∪Follow(A)中，看到一个为了它可以重新开始分析的记号后，再弹出该记号，记作 scan③特殊情况下压入一个新的 N

CH5 自底向上分析

Yacc 基于 LALR(1)；使用显示栈完成分析。

5.1 概览

1.动作为①shift，将 T 从输入开头移到栈顶②reduce 使用产生式 A→α将栈顶的α规约成 A③accept 分析栈为开始符号，输入栈为空时的动作④error

2.注意统一**额外加一个 S′作为新的开始符号**

3.**活前缀**无二义性时示例：S→AdB →AdBdb。使用规则 B→b|Bdb|ε。活前缀有 A，Ad，AdB，AdBd，AdBdb：活前缀不超过最右句柄的右端，此处句柄是 Bdb (分析栈空时，可行前缀为ε)。有二义性时：E′->E->E+n->n+n。此时 n+n 的可行前缀是 n and E and E+

3.若语法无二义性，则句柄唯一

5.2 FN of LR(0) items and LR(0)

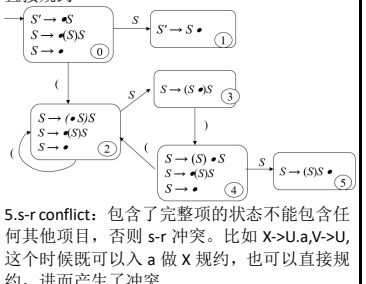
1.LR(0)的项就是在右边带有区分位置的产生式，同时就是 LR(0)的 FA 中的一个状态

3.DFA 构造算法：

每个新状态都是一个产生式的ε闭包。

其中在闭包步骤中通过ε添加到状态中的项目与引起状态的项目，前者叫闭包项 closure item，后者叫做核心项 kernel item。若有一个文法，核心项唯一判断出状态以及转换，那么只需要指出和心想就可以完整地表示出 DFA。

4.LR(0)分析算法的定义：Let s be the current state (at the top of the parsing stack). Then actions are defined as follow:①If state s contains any item of the form A → α · Xβ (X is a T). Then the action is to shift the current input token on to the stack.② If state s contains any complete item (A → α ·), then the action is to reduce by the rule A → α ·如果为 T 直接移进；如果包含完整项，直接规约



5.s-r conflict：包含了完整项的状态不能包含任何其他项目，否则 s-r 冲突。比如 X->U.a,V->U,这个时候既可以入 a 做 X 规约，也可以直接规约，进而产生了冲突

6.r-r conflict：两个完整项共存则出现 r-r 冲突，比如 U->X. V->X. 此时 X 不知道怎么归约

7.LR(0)文法不可能是二义的

8.A grammar is LR(0) if and only if ①Each state is a shift state (a state containing only shift items) or a reduce state (containing a single complete item).

这里的 r1 都是用 S′→S • 规约，应该写成 accept

State	Input			Goto
	(	a	)	A
0	s3	s2		1
1	r1	r1	r1	
2	r2	r2	r2	
3	s3	s2		4
4			s5	
5	r3	r3	r3	

5.3 SLR(1)

1.SLR(1)算法定义：LR(0)移进规则不变；规约时要求输入必须在 follow 中

2.SLR(1)不可能是二义性

3.A grammar is SLR(1) if and only if, for any state s, the following two conditions are satisfied:②For any two complete item A → α ·and B → β · in s, Follow(A) ∩ Follow(B) is empty.两个完整项的 Follow 集不相交；①待移进的终结符不能是完整项的 Follow 元素；比如 X->d.c;M->d.,则若 M 的 follow 集里有 c，则不可以

4.自底向上分析中右递归可能引起栈溢出，需要避免

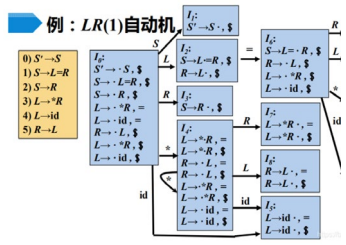
5.SLR(1)中的两种冲突，sr 冲突使用消岐规则：优先移进；rr 冲突基本是设计出问题

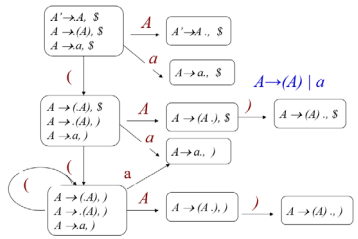
5.4 LR(1) and LALR(1)

1.LR(1) items：[A → α · β, a]前面是 LR(0)项，后面是 lookahead token

2.LR(1)的起始状态[S′→ • S,$]的闭包

3.





4.LR(1) definition: let s be the current state (at the top of the paring stack). Then actions are defined as follows:①If state s contains LR(1) item of the form [A → α · Xβ, a], X is T and X is the next token in the input string.②If state s contains LR(1) item [A → α ·, a], the next token in the input stream is a.③If the next input token is such that neither of the above two cases applies, error.

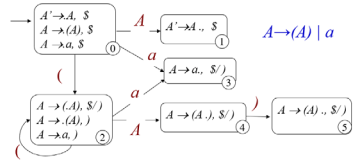
5.LR(1)文法不可能二义性

6.A grammar is LR(1) if and only if, for any state s. The following two conditions are satisfied: ①For any item [A → α · Xβ, a], X is T. There is no item in s of the form [B→γ ·,X]，否则 sr 冲突②There are no two items in s of the form [A → α ·, a] and [B → β ·, a]，否则 rr 冲突——看分析表

7. (1) A→(A)，(2) A→a 的分析表

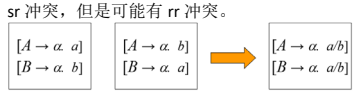
State	Input				Goto
	(	A	)	$	S
0	s2	s3			1
1				accept	
2	s5	s6			4
3				r2	
4			s7		
5	s5	s6			8
6			r2		
7				r1	
8			s9		
9			r1		

8.LALR(1)将先行合并

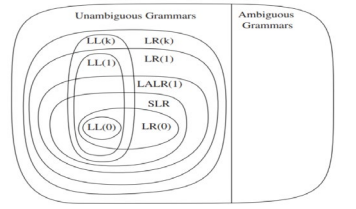


9.A grammar is an LALR(1) grammar if no parsing conflicts arise in the LALR(1) parsing algorithm.

10.如果文法是 LR(1)，那么 LALR(1)中必然没有 sr 冲突，但是可能有 rr 冲突。



12.通过传播先行 propagating lookahead 的处理从 LR(0)项目的 DFA 直接计算出 LALR(1)的 DFA 是可能的。



5.7 Error recovery

1.LR(1)比 LALR(1)或 SLR(1)更早检测出错误；LALR(1)和 SLR(1)都比 LR(0)更早

2. There are three possible alternative actions:①Pop a state from the stack.②Successively pop tokens from the input until a token is seen for which we can restart the parse.③Push a new state onto the stack.

3. When an error occurs is as follows:①Pop states from the parsing stack until a state is found with nonempty Goto entries.②If there is a legal action on the current input token from one of the Goto states, push that state onto the stack and restart the parse.③If there is no legal action on the current input token from one of the Goto states, advance the input.

CH5 抽象语法

5.1 递归下降分析器

对 T 而言，看预测分析表，存在着 id、num、（时的 T->FT'，所以此时碰到这三符号时是送一个 F 给 T'。T'则预测分析表中有遇到+：T'->和遇到*：T'->*FT'。所以也执行返回

```
int T_follow[] = { PLUS, RPAREN, EOF, -1 };
int T(void) {switch (tok) {
case ID: case NUM: case LPAREN: return
Tprime(F());
default: printf("expected ID, NUM, or left-
paren");
skipto(T_follow);
return 0;
}}
int Tprime(int a) {switch (tok) {
case TIMES: eat(TIMES); return Tprime(a*F());
case PLUS: case RPAREN: case EOF: return a;
default: · · ·
}}
```

5.2 命令式风格解释器

此时和 yacc 的解释器一样。

5.3 对比区别

（1）递归下降解析器采用自顶向下的方式进行语法分析，从输入符号开始不断递归调用子过程直到推导出完整的句子。而命令式风格的解释器则更加类似于编译器，采用自底向上的方式分析语法树，一次执行一个操作指令并更新程序状态。（2）递归的使用条件和递归函数；命令用循环分支跳转；（3）递归存储使用栈和堆；命令使用动态分配和管理的数据；（4）命令比递归简单

5.4 Yacc 特性

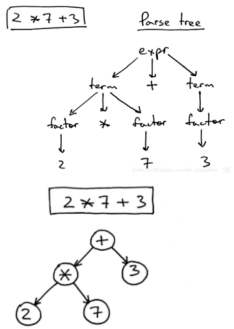
（1）A Yacc-generated parser implement semantic values by keeping a stack of the them parallel to the state stack.（2）When the parser performs a reduction,it must execute a C-language semantic action.（3）When the parser pops the top k elements from the symbol stack and pushes a nonterminal symbol , it also pops k from the semantic value stack and pushes the value obtained by executing the C semanticaction

5.5 Tree

理论看 3.3。补充：每种语法有多种的 parse tree 但是只有相同的 AST 树；AST 树不在乎 generate 只在乎 program constructs；它更加紧密；不用内部节点来表示语法规则；使用操作符/运算符作为根节点和内部节点，而使用操作数作为它们的子节点

A parse tree has exactly one leaf for each token of the input and one internal node for each grammar rule reduced during the parse

这里放图



5.6 Pos

Pos 代表的是源代码位置（行号或者行内位置），存储在 AST 节点中。如果不存储，由于词法分析已经到达末尾，无法显示代码在哪里报错，插入时如下：

```
expr : expr '+' term { $$ = Add($1, $3, pos); }
                | term { $$ = $1; }
```

CH6 语义分析

6.1 符号表

σ 1{int a,b,c}，经过 int j 得到了 σ 2{int a,b,c}+{int j}，经过 string a 又得到了 σ 3{int a,b,c}+{int j}+{string a}。为了实现参数指代形式的变化，有两种方法：函数式，有多个 σ 共存。命令式：全局只有一个符号表，但是符号表动态变换。 即每次都会修改 σ ，当不需要时再重新恢复成之前的 σ ，上述则有 σ 0, σ1, σ2, σ3, σ1, σ0。要求全局中有 an “undo stack” with enough information to remove the destructive updates

6.2 多符号表

受编译语言影响，不同的语言编译时符号表不同，这是受语言特性所决定的。例如下列语言中，ML 语言不允许在 N 中出现 D.d，但是 Java 语言式允许的，也就是说在 Java 中，E、N、D 使用同一个负号表，而 ML 语言则是 E、N、D 各自拥有一个符号表

```
structure M = struct
   structure E = struct
      val a = 5;
   end
   structure N = struct
      val b = 10
      val a = E.a + b
   end
   structure D = struct
      val d = E.a + N.a
   end
end
```

6.3 命令式符号表

使用 Hash 表完成操作，具体思路如下

假设有 a->σ1，现有 a->σ2， 则把 a->σ2 插入到 a 对应的 hash 表的表头位置，待到 a->σ2 使用完毕，再 pop 出对应位置的 Hash 表即可实现操作。这里考到了相应的代码：

```
void insert(string key,void * binding,struct bucket
* table){
        int index;bucketnum++;
        if(bucketnum/size>2)
        {size*=2;table=rehash(table);}
        index=hash(key)%size;
        table[index]=Bucket(key,binding,table[in
dex]);}
_bucket * rehash(_bucket * table){
        _bucket * newTable=(_bucket
*)malloc(sizeof(_bucket *)*size);
for(i=0;i<bucketnum;i++)
insert(table[i]->key,table[i]->binding,newTable);
        free(table);return newTable;}
struct bucket *table[SIZE];
unsigned int hash(char *s0)
{unsigned int h=0; char *s;
for(s=s0; *s; s++)
h = h*65599 + *s;return h;}
struct bucket *Bucket(string key, void *binding,
struct bucket *next) {
struct bucket *b = checked_malloc(sizeof(*b));
b->key=key; b->binding=binding;
b->next=next;return b;}
void insert(string key, void *binding) {
int index = hash(key) % SIZE;
table[index] = Bucket(key, binding, table[index]);}
void *lookup(string key) {
int index = hash(key) % SIZE;
struct bucket *b;
for(b=table[index]; b; b=b->next)
if (0==strcmp(b->key,key)) return b->binding;
return NULL;}
void pop(string key) {
int index = hash(key) % SIZE;
table[index] = table[index]->next;}
```

6.3 函数式符号表

函数式符号表可以使用 binary search trees 二叉搜索树快速进行实现，其思路是：在 d 层添加一个节点，只要新创建 d 个节点，不需要复制整个树

6.4 编译器的绑定

属性 (attributes) 包括编程语言组件的任意特性。例如，标识符的属性就包括其种属（变量、数组、函数名等等）、数据类型、存储位置、长度、值、作用域等。属性的确定时间是有多种可能的。属性值的计算以及将计算出来的值与相关的语言结构进行联系的过程称为属性的 binding，发生的时间称为 binding time。在程序执行之前就进行 binding 的属性称为 static attributes，函数、变量等；在程序执行过程中才进行 binding 的属性称为 dynamic attributes，虚函数、多态等。

6.5 等价类型

结构等价：两个类型当且仅当他们有相同的结构，即语法树时才相同。一般要求数组大小相同、结构体顺序相同，但是也可以不同。

名等价：当且仅当两个类型有相同的类型名时才等价。例如，定义 typedef t1 = int, typedef t2 = int，但是`t1`,`t2`,`int`仍然两两不等价

说明等价：弱化版的名等价，即别名之间可以等价。在上面的例子中，这三个类型等价。

4.Pascal 使用说明等价，C 对结构和联合使用说明等价，对指针和数组使用结构等价。

CH7 活动记录

7.0 栈

1.有一些函数内的局部变量，其需要的生命期可能超过了函数的生命期。pascal 和 tiger 允许函数嵌套，不允许函数作为返回值-可使用栈。C 允许将函数作为返回值，不允许函数嵌套-可使用栈。ML、Scheme 等语言允许函数作为返回值也允许函数嵌套-不可使用栈。

2. 栈通常只在入口处增长，出口处收缩。栈中用来存放函数的局部变量、参数、返回地址、其他临时变量的区域称之为 activation record 或 stack frame。存储的内容有：incoming argument;return address;local variables;outgoing argument；stack links

7.1 储存器组织

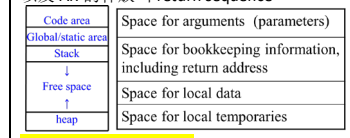
1.寄存器区和 RAM，RAM 分为代码和数据区

2.存储器组织和 AR(activation record)

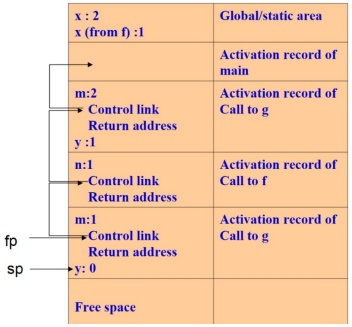
3.AR 位置：fortran77 静态区 C Pascal 栈 LISP 堆

4.pc,sp,fp(当前 AR),ap(保存参数值的 AR 区)

5.AR 存储器分配、计算和保存自变量以及其他必要的寄存器操作叫 calling sequence：放置可由调用程序访问的返回值、寄存器的重新调整以及 AR 的释放叫 return sequence



7.2 Stack-based C Pascal



1.栈随着调用链生长缩小，一个过程可以在栈上有多个不同的活动记录，代表不同的调用。允许递归调用，本地变量重新分配空间。

2.对于所有过程都是全局的语言，栈环境两个要求：①fp 允许访问本地变量②**sp 指向调用栈的最后，管理栈的生长缩小**

3.**控制链 control/dynamic 指向先前的 AR，fp 指向当前 AR，sp 在 fp 下一个**。参数在上方，局部变量在下方（从高地址往下）

实参 n...

实参 1（incoming arguments）

Frame point->stack link(previous frame)

Local variables

Return address

Temporaries

Saved registers

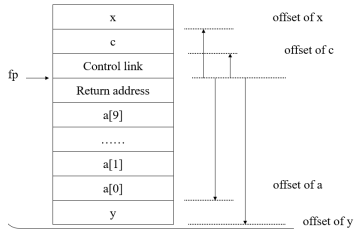
Argument m(outgoing argument)

Argument 1

Stack pointer->Stack link current frame

.....next frame

4.对名称访问时，由 fp 和偏移量获取，大部分偏移量是可以静态计算获得的。向上+向下-



5.调用序列：计算自变量并将其放在过程的新 AR 的正确位置，倒序压栈；f 作为控制链压入 AR 中；修改 fp 为新的 AR 开始(复制 sp 也可)；返回地址存入 AR；执行跳转；

6.返回时，fp 复制到 sp；控制链装载到 fp；跳转到返回地址，即调用该函数的位置；修改 sp 弹出自变量

7.变长数据：指代参数数量可变或者参数大小可变①函数参数量可变(倒序压栈，有一个一般是+4 偏移的量说明总参数数)②数组参数或者局部数组，额外跟踪大小

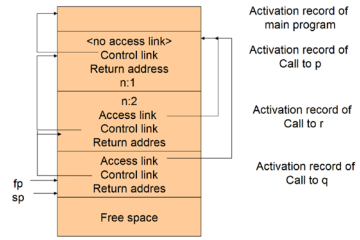
8.对待函数一样对待块不够效率。简单方法是在嵌套的块中处理声明，进入块时分配它们，离开块时销毁

7.3 局部过程基于栈的定义规则

1. 允许局部过程后，叫做 block structure。有三种方法：静态链；display 数组；lambda lifting.

Access link 指向自己外层的最近一次调用。

Access link 要比 fp 先进栈在调用时。下面的示例：p 内定义一个 q 和一个 r，并且 r 内调用 q



2. access link 的维护规则：假设 x 调用 y

（1）若 y 的嵌套深度大于 x，那么 y 直接在 x 中定义嵌套深度为 1，此时 y 的 access link 指向 x 的 activation record

（2）若 y 的嵌套深度等于 x，那么要么是自己调用自己（也就是递归），要么就是定义在全局或者同一个 procedure 内部。比如上面的 p 内一个 q 一个 r。此时 y 的 acess link 直接使用 x 的 access link 即可。

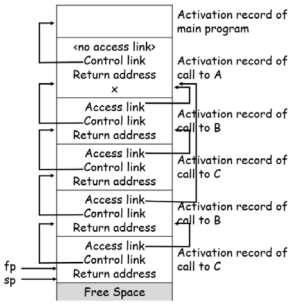
（3）若 y 的嵌套深度小于 x，要么 y 定义在全局，要么 y 定义在 r 内部，而 x 在 r 的某一层。那么此时，y 的 access link 应该为 r。那么看 x，经过 nx-ny+1 次 access link 的寻找，找到了 r 的 activation record，然后将这个地址赋给 y 的 access link。(或者是找 nx-ny 次，把 s1 的 access link-也就是指向 r，赋给 y）

```
Procedure r
   Procedure y
   Procedure s1
      Procedure s2
         Procedure x
```

但是，从实际操作上，其实很简单，就算牢记指向自己外层的最近一次调用

分析这张图：第二次调用 B 要去找自己的 A，而第二次调用 C 呢，则是找 B，但是是去找最近的那个 B

```
program env;
procedure a;
var x: integer;
   procedure b;
      procedure c;
      begin
         x := 2;
         b;
      end;
   begin (* b *)
      c;
   end;
begin (* a *)
   b;
end;
begin (* main *)
      a;
end;
```



7.4 课本其余概念

1. 当 f 调用 g 时，如果 f 用了寄存器 r，g 也用想 r，那么要对 r 进行保护（比如 g 先存这个 r 的值，等 g 结束再还原这个 r），通过这个 r 靠谁来保护，分为 caller-save register（f 保护）和 callee-save register（g 负责保护）

2. 参数传递：现代机器中，传递参数时前 4-6 用寄存器传，剩余的用内存传。但是会出现函数调用函数的情况，比如 f(a,b,c)调用 g(z)，此时 z 要用的寄存器被占了，有四个方案

- 一些函数不会调用其他函数，称之为 leaf procedures，不需要 write their incoming arguments to memory
- 一些编译器可以自动优化，使得编译器顺序变化，a-r1,z-r7
- g(z)时 a 不会再被用到，此时这个寄存器没作用，直接使用
- 一些体系结构有寄存器 windows，不会有 memory traffic
- 把放在寄存器里的值放到内存里去

还有一些语言，例如 C 语言里需要参数的连续性

3. 返回地址：f 在 a 处调用了 g，返回的时候返回到 a+1，现代机器使用一个寄存器来保存这个返回地址。非叶把返回地址放到栈帧中（除非有特殊的过程间寄存器），叶子不需要保存，有寄存器。该步骤是 CALLED procedure 来完成的

4. 变量存放位置：

放在寄存器里的：function parameters;return address;function result

放在内存（stack frame）里的：

（1）被用作传地址参数的变量，也就是加了&的变量（2）该变量被嵌套在当前过程内的过程访问（但并不是绝对的，有时也可放在寄存器

中）（3）变量值太大了不能放到单个寄存器中（但是有的编译器选择用多个寄存器，而不是放到内存）（4）该变量是一个数组，为了引用其地址（5）存在太多局部变量和临时变量，放不下时，需要放一部分进内存里（6）需要用来进行参数传递

Ch8 翻译成中间代码

8.1 IR

编译器的前端进行词法分析、语法分析、语义分析，并且产生中间表示；而编译器的后端对中间表示进行优化，并且将中间表示翻译成机器语言。

8.2 三地址码

一种表示如下所示：

High-level language	Three-address code

read x ; { input an integer }
if 0 < x then { don't compute if x <= 0 }
fact:=1;
repeat
fact:=fact*x;
x:=x-1
until x=0;
write fact { output factorial of x }
end
	read x
	t1=x>0
	if_false t1 goto L1
	fact=1
	label L2
	t2 = fact * x
	fact = t2
	t3 = x - 1
	x = t3
	t4= x= =0
	if_false t4 goto L2
	write fact
	label L1
	halt

而其对应的四元式：

```
                    (asn,t2,fact,_)
                    (sub,x,1,t3)
(rd,x,_,_)          (asn,t3,x,_)
(gt,x,0,t1)         (eq,x,0,t4)
(if_f,t1,L1,_)      (if_f,t4,L2,_)
(asn,1,fact,)       (wri,fact,_,_)
(lab,L2,_,_)        (lab,L1,_,_)
(mul,fact,x,t2)     (halt,_,_,_)
```

对应的三地址码：

```
(0)   (rd,x,_)
(1)   (gt,x,0)
(2)   (if_f,(1),(11))
(3)   (asn,1,fact)
(4)   (mul,fact,x)      (8)   (eq,x,0)
(5)   (asn,(4),fact)    (9)   (if_f,(8),(4))
(6)   (sub,x,1)         (10)  (wri,fact,_)
(7)   (asn,(6),x)       (11)  (halt,_,_)
```

8.3 树结构

1. 基础变量：

整型常数 CONST(i),符号常数 NAME(n),临时变量 TEMP(t)

2. 基础计算(Tree 语言没有一元运算符，要用二元运算去进行)：

BINOP(o,e1,e2)：MEM(e)当其位于 move 左操作时是存储其余时候是读取；CALL(f,l)用 l 的参数列表调用 f；ESEQ(s,e)以 s 结果去看 e；MOVE(TEMP t,e) MOVE(MEM(e1),e2) 前=后；EXP(e)；JUMP(e,labs)跳转到 e；CJUMP(o,e1,e2,t,f)；SEQ(s1,s2)；LABEL(n)

3. 字符串，变量与数组：

（1）追随静态链：一个值层层嵌套得出，此时为了得到一个值要经过多次偏移与基地址的加和：MEM(+CONST KN,MEM(+CONST KN-1,...MEM(+CONST k1,TEMP FP)))

(2) 左值和右值：简言之，左值是出现在等号左边的，可以变化的内容，右值是只能出现在等号右边的，例如 a+2,f(x)。而左值也能出现在右边. 左值是地址，右值是值

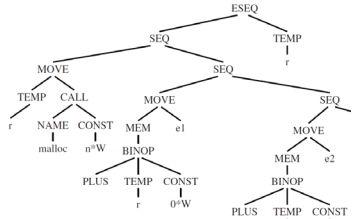
(3)数组值：有些语言如 C，数组是指针常数，a=b 是地址的复制；而有些语言如 Pascal，a=b 则是整体值上的复制。

A[i]=MEM(+(MEM(e),BINOP(MUL,I,CONST W)))

安全性警告：需要检查**数组越界**的安全性，优化编译器在进行静态分析时，也可以将这些边界检查从代码中优化掉，以提高程序的执行效率。但在这个过程中，需要保证程序的运行时安全性，并采取一定的措施避免出现缓冲区溢出等安全隐患

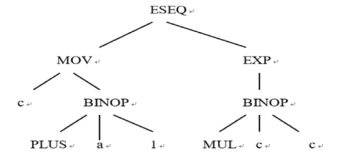
(4) 字符串：所有字符串操作都由运行系统提供的函数来完成，这些函数为字符串操作的结果分配堆空间，并返回指针

(5)记录与数组的创建：申请空间，然后逐个赋值，最后返回地址 r



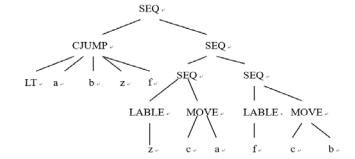
4.基础语句

1. 基础语句：c=a+1; c*c

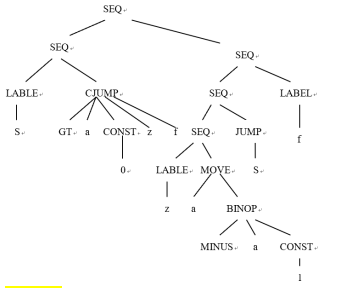


2. 条件语句 if a<b then c=a else c=b

首先由一个 CJUMP 确认两类跳转，再有 seq 做连接，其中的 n LABEL 是 if 的结束



3. 循环语句 while a>0: a=a-1



5. 声明

1. 当在函数体内部声明或定义变量时，编译器会为每个变量分配一定的存储空间；当声明函数时，编译器会为该函数分配一段新的代码区域，用于存储函数体中的代码。

2. 将函数转换为汇编代码需要分为三个部分：前奏（prologue）、主体（body）和结尾（epilogue）。前奏包含了 pseudo-instructions，用于标记函数的开始、定义函数名以及调整栈指针。此外，还需要进行参数和寄存器的保存和初始化操作；主体部分就是函数实际执行的代码了；结尾部分是函数执行结束后需要进行的操作(寄存器指针等跳转)。前奏结尾为了正确和稳定。函数是功能

CH9 基本块与轨迹

9.1 基础概念

树形结构与编译器有冲突：CJUMP（真实为假是直接跳转到后一句，不可以任意跳）、ESEQ（使得子树的不同顺序产生不同值）、call；但是它们存在的意义是，有利于编译器 Translate 阶段。并且，有方法可以进行等价性转换。

在转换时分三步（1）去除 ESEQ 形成规范树（2）分组成基本块（3）形成轨迹，使得 CJUMP 后跟着它的 fasle

9.2 规范树

除了根节点，没有 ESQ 或 ESEQ，并且 call 的父节点是 exp 或者 move(TEMP t,...)，而树的调整规则如下：

补充作业里常见的：

MEM(ESEQ(A,B))=ESEQ(A,MEM(B))

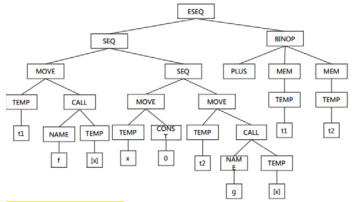
MOVE(ESEQ(A,B),C)=ESEQ(A,MOVE(B,C))

而这里出现了 commute，这个意思是要判断能否交换顺序，比如 s 是 MOVE(MEM(x))，e1 是 BINOP(PLUS,MEM(x),z)，交换会导致语句出问题。采取保守的方式，空语句和常量语句可以进行交换，其余的不可以。

9.3 Call 的转换

CLL 函数将会把结果返回到同一个返回值寄存器 TEMP（RV）中。为了防止覆写寄存器，将每一个 CALL 的结果能存在 TEMP t 里，然后通过 ESEQ 返回这个 t：CALL(fun,args) -> ESEQ(MOVE(TEMP t,CALL(fun,args)),TEMP t)

这里有一道家作里十分复杂的题目：转换的目标是：BINOP(PLUS, CALL(NAME f , [TEMPx]),CALL(NAMEg, [ESEQ(MOVE(TEMPx , CONST0), TEMPx)]))。这里把所有的 ESEQ 判断的东西都移动到了左边。



9.4 线性语句表

SEQ(SEQ(a,b),c) = SEQ(a,SEQ(b,c))

9.5 基本块

必须从头进从尾出。（1）头是 LABEL（2）最后是 JUMP 或者 CJUMP（3）没有其他的 LABEL、JUMP、CJUMP。形成相对简单。下面看一个基本块转成指令：

1、**Label(Begin)**
   Move (Temp m, Const 0)
   Move (Temp v, Const 0)
2、**Label(TestConditionOfV)**
   CJump(≥, v , n, Exit, BodyOfV)
3、**Label(BodyOfV)**
   Move (Temp r , Temp v)
   Move (Temp s, Const 0)
4、**Label(TestConditionOfR)**
   CJump(<, r, n, BodyOfR,IncreaseV)
5、**Label(IncreaseV)**
   Move (Temp v, Binop(+,Temp v, Const 1))
   Jump(TestConditionOfV)
6、**Label(BodyOfR)**

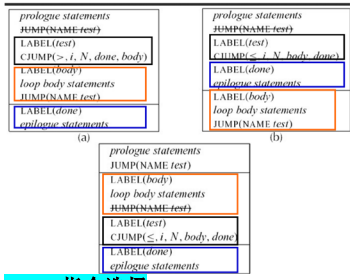
9.6 轨迹生成与完善

1. 生成时，一直持续的做，直到做不下去了，就从基本块里选一个没做过的继续做。

2. 优化：（1）把许多无条件跳转紧接着其目标标签的基本块排在一起，并删除这些无用的跳转语句（2）把 CJUMP 后面紧跟其 false 情况的跳转

3. 完善：有些语句 CJUMP(cond,a,b,lt,lf)这样的句子，若这一整句后面跟的就是 lf 的，那么不变；若后面紧跟的是 lt 的，那么交换 lr 与 lf；如果碰到了后面跟的既不是 lt 也不是 lf，则作下面的改变：CJUMP(cond,a,b,lt',lf')
LABEL lf' JUMP(NAME lf)

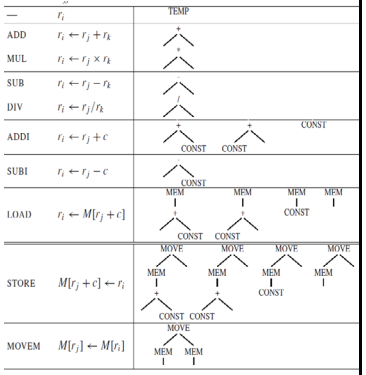
4.最优轨迹：频繁的循环体有单独的轨迹，其中的迭代跳转要最少，这里显示了同样的一些语句，不同的轨迹，其中 C 的跳转最少。



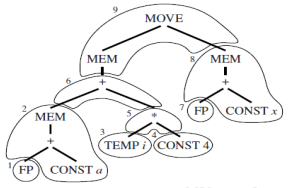
CH10 指令选择

10.1 树型

树型要与机器指令达成对应的转换，目标就是 use no overlapping tiles to cover the IR tree



覆盖转换为指令对应的时候，根据瓦片左边的指令去转换。注意**大部分要使用的是寄存器；MEM 和取 R 要分开；r0 是 0 别占用**



2 LOAD $r_1 \leftarrow M[\mathbf{fp} + a]$
4 ADDI $r_2 \leftarrow r_0 + 4$
5 MUL $r_2 \leftarrow r_i \times r_2$
6 ADD $r_1 \leftarrow r_1 + r_2$
8 LOAD $r_2 \leftarrow M[\mathbf{fp} + x]$
9 STORE $M[r_1 + 0] \leftarrow r_2$

10.2 指令覆盖

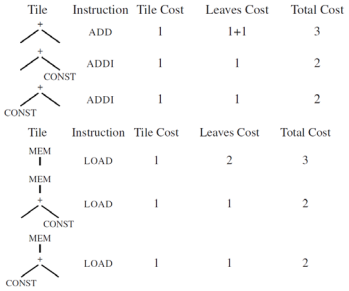
1. 覆盖有两个概念，最优（optimum）覆盖和最佳（optimal）覆盖。
   - optimum 是总和代价最小。
   - optimal 是没有两个瓦片可以合并

2. optimum 一定也是 optimal，而对 optimal 的覆盖是更为简单的。对于 CISC，两者差别明显，对于 RISC，两者差别不大

3. Maximal Munch：the alogrithm for optimal tiling，从根节点开始找最大覆盖，找到后，对剩下的子树，也从其根开始继续找最大的覆盖，直到再也没法找到。本质上是**贪心算法**，不一定全局最优。

4. 动态规划：从底层节点开始匹配，找最优的方法不断向上
- 递归求出节点 n 的所有儿子和孙子的代价
- 将每一种树型与节点 n 进行匹配

这里放两张图 get 意思：首先是对于 addi 的计算，其次是对 MEM 的计算

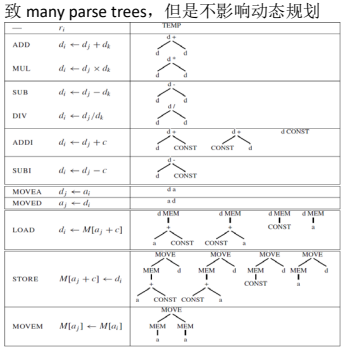


10.3 树文法

1. 基于树形语法的算法，这种算法是**动态规划算法的一种扩展形式**。而这种算法特别适用于**指令集比较复杂、寄存器和寻址模式比较多**的计算机。它的好处是，可以在编译过程中自动选择最佳的寄存器和寻址方式，从而减少程序的指令数和执行时间。

$$d \rightarrow \mathrm{MEM}(+(a, \mathrm{CONST}))$$
$$d \rightarrow \mathrm{MEM}(+(\mathrm{CONST}, a))$$
$$d \rightarrow \mathrm{MEM}(\mathrm{CONST})$$
$$d \rightarrow \mathrm{MEM}(a)$$
$$d \rightarrow a$$
$$a \rightarrow d$$

2. jouette 结构：用 a 代表 registers for addressing，用 d 代表 registers for data。为了让 dp 算法能知道节点使用 a 类寄存器和 d 类寄存器时的最小代价，使用上下文无关文法来描述瓦片，虽然这有**很大的歧义性**，会导致 many parse trees，但是不影响动态规划



3. 但是动态规划要设计却很繁琐，因此开发了代码生成器的生成器。同 yacc 和 lex 一样 the output of a code-generator generator is usually a program in C or Java that operates a table-driven matching engine with the action fragments (written in C or Java) inserted at the appropriate points

10.4 Fast Matching

之前的方法，都要完整的查看很多节点（包括叶节点），才能看清匹配关系，这样会碰到缓慢的问题，这里提出一个快速匹配的方法，直接对节点 n 进行匹配，匹配到了之后就继续匹配，找到第一个 ok 的就结束：比如 match(n){swtich (label(n)) case MEM:... case BINOP... case CONST...}

10.5 CISC

1. 典型的 CISC：32 registers,
•only one class of integer/pointer registers,
•arithmetic operations only between registers,
•three-address instructions of form r1-<r2⊕r3,
•load and store instructions with only the M[reg+const] addressing mode,
•every instruction exactly 32 bits long,
•one result or effect per instruction.

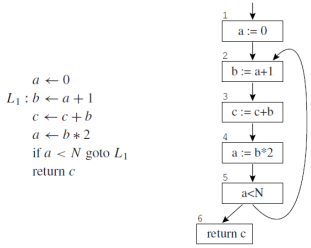
2.typical features
few registers (16, or 8, or 6);
•registers divided into different classes, with some operations available only on certain registers;
•arithmetic operations can access registers or memory through "addressing modes";
•"two-address" instructions of the form r1 ← r1⊕r2;
•several different addressing modes;
•variable-length instructions, formed from variable-length opcode plus variablelength addressing modes;
•instructions with side effects such as "autoincrement" addressing modes.

3. 用以下方式解决问题：
（1）寄存器较少：不受限制地生成 TEMP 结点，并假设寄存器分配器能够很好完成工作
（2）寄存器分类：将操作符和结果分开 mov eax,t2_ eax<-t2; mul t3_ eax<-eax*t3; edx<-garbage_ mov t1,eax_ t1<-eax
（3）两地址指令：增加额外指令
Mov t1,t2---t1<-t2; add t1,t3---t1<-t1+t3
(4) 令选择阶段将每个 TEMP 节点转换为一个“寄存器”引用。解决方案：在操作之前将所有操作数加载到寄存器中，并在操作后将它们存回内存。
(5) 多种寻址方式的优势：•占用更少的寄存器，•指令编码更短。通过一些工作，可以使树匹配指令选择工具选择 CISC 寻址模式，但使用简单的类 RISC 指令编写的程序同样可以达到快速的执行效果。
(6) 可变长度指令•对于编译器来说并不是真正的问题；•一旦指令被选择，汇编器只需简单（虽然繁琐）地发出编码即可。
(7)有副作用的指令：(a)忽略地址自增指令希望它们会自动消失(b)采用树形匹配的代码生成器的上下文中，用一种特殊的方式来匹配特殊方言(c)使用不同的指令算法，不基于树型

CH11 活跃分析

11.1 基础概念

1. 活性分析是编译器中的一个重要步骤，用于确定哪些变量是活跃的，以及哪些临时变量在同一时刻同时被使用。这个过程可以帮助编译器实现代码优化、寄存器分配等任务，从而提高程序的执行效率和空间效率。
2. control-flow graph **控制流图**：如果语句之后可以紧接着执行，就有一条边
3. Liveness：我们**从后往前分析**活跃性，如果 x 在 p'语句块被使用到（取用它的值），那么便有一条从 p->p'的活跃。

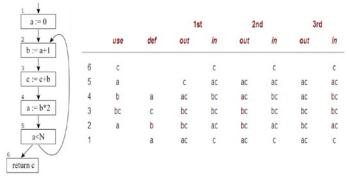


4. example：如图所示，b 的活跃范围是{2->3，3->4}，a 的活跃范围是{1->2，4->5->2}活跃（这里的 a<N 是对 a 的使用，a=b*2 不算对 a 的使用）。这两个范围内，a/b 可以放在一个寄存器里，另外的 c 可以放在另一个寄存器里。这就是，靠活跃分析，来做寄存器分配。

11.2 基本算法

1. live-in：A variable is live-in at a node if it is live on any 任何 of the in-edges of that node
2. live-out：A variable is live-in at a node if it is live on any 任何 of the out-edges of that node
3. - If a variable is in use[n], then it is live-in at node n. That is, if a statement uses a variable, the variable is live on entry to that statement.
- If a variable is live-in at a node n, then it is live-out at all nodes m in pred[n].
- If a variable is live-out at node n, and not in def [n], then the variable is also live-in at n. That is, if someone needs the value of a at the end of statement n and n does not provide that value, then a's value is needed even on entry to n.
4. 推导方程：在实际操作中，从后往前迭代，先计算 out 集合，再计算 in 集合。但是注意控制流图如果有 goto 的无条件跳转语句，就不要写出来，直接跳转。Succ-后继：

for each n
  $in[n] \leftarrow \{\}$; $out[n] \leftarrow \{\}$
repeat
  for each n
    $in'[n] \leftarrow in[n]$; $out'[n] \leftarrow out[n]$
    $in[n] \leftarrow use[n] \cup (out[n] - def[n])$
    $out[n] \leftarrow \bigcup_{s \in succ[n]} in[s]$
until $in'[n] = in[n]$ and $out'[n] = out[n]$ for all n



5. Basic block：如果一个点只有一个前驱和一个后继，那么这样的 node 可以与其余的进行合并
6. One variable at a time：上面我们是基于整体性的做数据流方程的，但是，对于大型程序或者复杂的代码，同时计算所有变量的数据流信息可能会非常耗时。因此，当只需要某个变量的数据流信息时，可以单独使用深搜方法计算该变量的信息。
7. 算法分析：数据流方程的集合有两个比较好的表示方法，位数组（array of bits）和链表(sorted lists of variables)。位数组是用一位来表示一个元素的状态，并运算需要 N/K 次操作。链表则是其中的成员是集合的元素，合并的大小和链表的长度有关。假设有 N 变量，计算机每个字 K 位，则集合是稀疏的（元素小于 N/K），则使用有序表表示，如果是密集的，则使用数组更好。数据流算法最坏是 O(N^4)，但是实际操作往往在 O(N)-O(N^2)之间

11.3 最小不动点

1. 编译器绝不可能判断一个给定的标号是否可达，所以必须使用保守近似值。因此，数据流方程的任何解都是一个近似的保守解（conservative approximation），即我们会误认为一个变量也是活跃的，哪怕它并不活跃。这使得编译器所使用的寄存器比实际需要的多，但是能保证生成的代码一定是正确的。
2.会有一个最小的解，称为最小不动点（Least Fixed Points）
3. - 动态活跃： Avariable a is dynamically live at node n if some execution of the programgoes from n to a use of a without going through any definition of a.- 静态活跃：Avariable a is statically live at node n if there is some path of control-flow edgesfrom n to some use of a that does not go through a definition of a.
- 动态活跃=>静态活跃，优化编译器根据静态活跃，因为一般计算不出动态活跃信息

11.4 冲突图

1. 阻止 a 和 b 放入同一个寄存器，使用矩阵或者无向图，实际上是看 in 集，in 集里同时出现的就是冲突的
2. 要避免人为制造冲突。在每个非 move 指令中，对于被定义的变量 a 和此时处于活跃状态的变量 b1,...,bj，在它们之间添加干涉边(a, b1),...,(a, bj)，表示它们之间存在干涉关系。同样地，在每个 move 指令 a ← c 中，如果变量 c 处于活跃状态且不等于 bi，则在变量 a 和 bi 之间添加干涉边(a, b1),...,(a, bj)，表示它们之间也存在干涉关系。

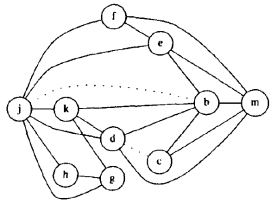
CH12 寄存器分配

12.1 定义

1. 寄存器分配是编译器中非常重要的一个阶段，用于将程序中的众多临时变量和寄存器进行映射，以便程序能够在少量的寄存器中高效地运行。在实现上，可以将 MOVE 指令的源操作数和目的操作数分配到同一个寄存器中，以便 MOVE 指令被删除。同时，为了实现寄存器分配，可以采用干涉图染色算法（Interference Graph Coloring Algorithm）

12.2 图着色算法

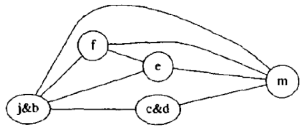
1. 分为四个步骤：构造（根据 In 图）、简化（把邻居**少于 K** 的的点去除）、溢出（乐观估计是邻居可能同色，放入栈中）、选择（出栈分配颜色）、重新开始（产生了实际溢出，把变量放出内存，改写了代码，重新进行流程，一般不超过 1-2 次）
2. 比如本图是 mcbfejdkhg3



12.3 合并节点

1. 合并的原因是两个变量有 move 关系，也就是在某一刻会出现 a=b，并且 a 与 b 之间没有干涉边（即不存在，不能放入同一个寄存器的冲突），那么新节点 ab 可以合并，**其所有边就是 a 和 b 之前的边**

2. 合并要采取保守的策略：**- Briggs: a、b 合并后的节点 ab，它的高度数(>=K 条边)的邻节点数量<K。- George: a 的每一个邻居 t，要么 t 与 b 已有冲突，或者 t 是低度数节点。这里尽量使用让整体简单的**
3. 能合并的两个点，必须只有一条虚线连接，如果既虚线也实线就不能合并，必须要做简化。并且，合并与简化的顺序是会影响的，书中规定**先简化**，并且引入冻结的概念
4. 如果简化和合并都不能再进行，我们寻找一个度数较低的传送有关的结点。冻结这个结点所关联的那些传送指令:即放弃对这些传送指令进行合并的希望。这将导致该结点(或许还有与这些被冻结的传送指令有关的其他结点) 被看成是传送无关的，从而使得有更多的结点可简化。然后，再重新开始简化和合并阶段。



(b)

5. 如果在发现需要溢出之前已经进行了节点合并，那么这些合并操作将会被保留下来；但是，如果在发现需要溢出之后进行了节点合并，那么这些合并操作将会被丢弃

12.4 预着色结点

1. 图中会有一些节点，代表的就是真实寄存器，比如 rax，六个参数寄存器，callee-save 寄存器等，这些在 codegen 时会被显式使用到，所以不可避免地会出现在图中。它们不能被简化和溢出，可以视作度为无限大。它们具有特定颜色。同时， 这些节点的活跃范围必须很小。因为通常可以将机器寄存器用作普通的临时变量。只有活跃范围小才能保证能够使用
2. 着色算法调用简化合并溢出直到只剩下预着色节点，然后选择才开始向冲突图中加点和着色。前端必须保持预着色节点的活跃范围很小，可以通过生成保存和恢复预着色点值的 MOVE 来实现。比如这里把 callee-save 的 r7 放入 t231，如果对寄存器要求高 t231 溢出，不然的话合并 t231 和 r7 并移除 move

```
Enter:    def(r7)
      t231←r7
        ⋮
      r7 ← t231
Exit: use(r7)
```

3. 对于一个本地变量或编译器临时变量，如果它不跨越任何过程调用，通常应该分配到一个 caller-save register，这样就可以避免寄存器的保存和恢复操作。但是，如果一个变量需要跨越**多个过程**调用，那么就应该将它分配到一个 callee-save register 中，这样只需要在进入和退出被调用过程时保存和恢复该寄存器一次即可。
4. 如果一个变量 x 需要跨越一个过程调用，并且它与一些已经预先分配的 caller-save 寄存器发生冲突，或者与 all the new temporaries created for callee-save registers 冲突(t231)，那么就可能需要将该变量溢出到内存中。进行溢出时，将会选择一个度数较高但使用较少的节点 t231 来溢出，而不是直接溢出变量 x
5. 实操：这里的代码，先做控制流程图，然后求出 in 和 out 表。最后用 in 里的做冲突图。这里的 r3 是 callee-save 的，所以放临时变量 c 里面。由于所有的点都不能简化 or 冻结，所以计算溢出优先级溢出 c，接着合并 ae，br2，ae 和 r1(颜色为 r1r2r3)；最终发现 c 实际溢出，没有颜色能够分配给它了。

```
Enter: c←r3
    a ← r1
    b ← r2
    d ← 0
    e ← a
loop: d ← d + b
    e ← e − 1
    if (e > 0) goto loop
    r1 ← d
    r3 ← c
    return   (r1, r3 live out)
```

```
int f(int a, int b) {
    int d = 0;
    int e = a;
    do { d = d+b;
         e = e-1;
    } while (e>0);
    return d;
}
```

Node	Uses+Defs outside loop	Uses+Defs within loop	Degree	Spill priority
a	(2 +10×	0)/	4 =	0.50
b	(1 +10×	1)/	4 =	2.75
c	(2 +10×	0)/	6 =	0.33
d	(2 +10×	2)/	4 =	5.50
e	(1 +10×	3)/	3 =	10.33

因此，需要放入内存之中，如图：

```
Enter: c1←r3
    M[c_loc] ← c1
    a ← r1
    b ← r2
    d ← 0
    e ← a
loop: d ← d + b
    e ← e − 1
    if (e > 0) goto loop
    r1 ← d
    c2 ← M[c_loc]
    r3 ← c2
    return
```

Node	Color
a	r1
b	r2
c	r3
d	r3
e	r1

此时可以把所有的变量换成寄存器，再把如 r3<-r3 这样的删除，就得到了最简的指令了

```
enter:  r3 ← r3
        M[c_loc] ← r3
        r1 ← r1
        r2 ← r2
        r3 ← 0
        r1 ← r1
loop:   r3 ← r3 + r2
        r1 ← r1 − 1
        if r1 > 0 goto loop
        r1 ← r3
        r3 ← M[c_loc]
        r3 ← r3
        return
```

12.5 图着色的实现

1. 需要查询 node X 的所有邻居（使用 adjacency list），以及 X 和 Y 是否相邻（二维矩阵）。使用时往往要冗余的同时使用
2. 普通节点 a 与机器寄存器 r 用 Geroge 合并测试，只需要 a 的邻接表不需要 r 的。两个非预着色的普通点用 Briggs 合并测试。
3. Additional data for nodes:–a count of the moves it is involved in (for move-related node)–a count of the number of neighbors (for all nodes)
•Low-degree non-move-related nodes (simplifyWorklist);•Move instructions that might be coalesceable (worklistMoves);•Low-degree move-related nodes (freezeWorklist);•high-degree nodes (spillWorklist).
4. 为了掌握图节点和传送边的情况，节点工作表集合栈如下，每点总在一个 set 或 list 中，并且表和集合互相不相交

precolored: machine registers, preassigned a color.
initial: temporary registers, not precolored and not yet processed.
simplifyWorklist: list of low-degree non-move-related nodes.
freezeWorklist: low-degree move-related nodes.
spillWorklist: high-degree nodes.
spilledNodes: nodes marked for spilling during this round; initially empty.
coalescedNodes: registers that have been coalesced; when $u \leftarrow v$ is coalesced, v is added to this set and u put back on some work list (or vice versa).
coloredNodes: nodes successfully colored.
selectStack: stack containing temporaries removed from the graph.

5. 每一条 move 只在下面的一个集合中：
- **coalescedMoves: moves that have been coalesced.**
- **constrainedMoves: moves whose source and target interfere.**
- **frozenMoves: moves that will no longer be considered for coalescing.**
- **worklistMoves: moves enabled for possible coalescing.**
- **activeMoves: moves not yet ready for coalescing.**

6. 其他数据结构

adjSet: the set of interference edges (u, v) in the graph; if (u, v) ∈ adjSet, then (v, u) ∈ adjSet.
adjList: adjacency list representation of the graph; for each non-precolored temporary u, adjList[u] is the set of nodes that interfere with u.
degree: an array containing the current degree of each node.
moveList: a mapping from a node to the list of moves it is associated with.
alias: when a move (u, v) has been coalesced, and v put in coalescedNodes, then alias (v) = u.
color: the color chosen by the algorithm for a node; for precolored nodes this is initialized to the given color.

CH13 垃圾收集

13.1 定义

指针链无法到达的记录称之为垃圾，需要进行垃圾收集，使得那部分空间可以重新使用，这不是由编译器而是由运行时的系统完成的。我们需要采用一种保守的方法，保证活跃记录都是可到达的，并尽可能减少那些可到达但非活跃的记录数量。

13.2 算法

这里是一个环+一个树
1. 算法 1：Mark-and-sweep garbage collection 对所有的根节点，做一遍深搜，搜到的节点就标记一下。然后对堆内所有节点做判断，把所有标记过的节点去除标记，留待下次垃圾回收时用；对所有未标记节点，链接到一起成为一个 freelist。

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)

2. 垃圾收集的代价：大小为 H 的堆中有 R 个字可达数据，一次垃圾收集的代价是 c1R+c2H，c1c2 为常数，例如 c1 是 10 条指令，c2 是 3 条指令，得到的好处是 H-R 的空间，而分摊代价为：c1R+c2H/H-R
R 接近 H，代价极大；否则的话，H 远大于 R，代价接近 c2；如果 R/H>ratio，就要申请更多空间
3. 优化：手工栈

function DFS(x)
  if x is a pointer and record x is not marked
    $t \leftarrow 1$
    stack[t] $\leftarrow x$
    while $t > 0$
      $x \leftarrow$ stack[t]; $t \leftarrow t - 1$
      for each field f_i of record x
        if $x.f_i$ is a pointer and record $x.f_i$ is not marked
          mark $x.f_i$
          $t \leftarrow t + 1$; stack[t] $\leftarrow x.f_i$

4. 优化：指针反转：done[x]指代现在在访问第几个，为 0 时是自己+1;为 1,2 时是自己的孩子；比如现在有 15->37->59，当操作 37 孩子 59 的时候，x.fi 记录 15，t 记录 37，x=y=59：回溯的时候 y=59 , x=37, i=done[37]=2, t=15,x.fi=59,done[37]++去看下一个

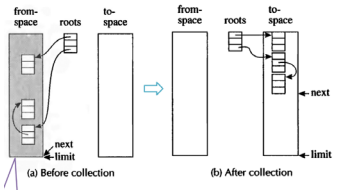
function DFS(x)
  if x is a pointer and record x is not marked
    $t \leftarrow$ nil
    mark x; done[x] $\leftarrow 0$
    while true
      $i \leftarrow$ done[x]
      if $i <$ # of fields in record x
        $y \leftarrow x.f_i$
        if y is a pointer and record y is not marked
          $x.f_i \leftarrow t$; $t \leftarrow x$; $x \leftarrow y$
          mark x; done[x] $\leftarrow 0$
        else
          done[x] $\leftarrow i + 1$
      else
        $y \leftarrow x$; $x \leftarrow t$
        if $x =$ nil **then return**
        $i \leftarrow$ done[x]
        $t \leftarrow x.f_i$; $x.f_i \leftarrow y$
        done[x] $\leftarrow i + 1$

5. Reference Counts：每个记录都包含一个引用计数，用于记录有多少指针指向它。每当一个指针（例如指针 x.fi）被赋值给记录中的一个字段，就会对指针(x.fi)所指向的新记录的引用计数将加一，而原来指向的记录的引用计数将减一。通过这种方式，程序可以跟踪每个记录有多少指针在引用它，以判断何时可以安全地释放该记录所占用的内存资源。当某个记录的引用计数降至零时，说明没有任何指针再指向该记录，可以将该记录放入空闲链表（Free List）中以供后续的内存分配使用。同时，也需要将该记录所指向的所有记录的引用计数减一，以确保内存资源能够被正确地回收和重复利用。

但是存在的问题是：（1）无法回收环构成的垃圾（可以使用数据结构解环、计数与标记清扫相结合的方法缓解）（2）需要更大的代价-因为计数的代码量更大

13.3 碎片收集

解决碎片的问题，把分散的=>紧凑的



1. 当需要回收垃圾时，垃圾回收器会首先将指针 `next` 初始化为指向到空间（To-Space）的开始位置。然后对于从空间（From-Space）中每个可达（Reachable）的记录，都会将其复制到到空间的位置 `next`，并将 `next` 的值增加该记录所占用的空间大小。这样，所有从空间中可达的记录都将得到复制，并存储到到空间中。
2. 转发（Forwarding）。由于在复制过程中，每个从空间中的记录都被复制到了到空间中，因此原先指向从空间中记录的指针也需要修改，使其指向对应的到空间中的记录。具体来说，转发操作的实现可以分为：
(1)如果指针 p 已经指向了到空间中的记录，则直接返回其指向的地址。
(2)如果指针 p 指向了 from-space 中的记录，并且记录已经被复制到了 to-space 中，则将指针 p 的字段 f1 修改为指向到空间中对应的记录，并返回 p.f1。
(3)如果指针 p 指向了从空间中的记录，但是记录还没有被复制到到空间中，则需要先将该记录复制到到空间的位置 next，然后将指针 p 的字段 f1 修改为指向到空间中的记录。最后更新 next 的值，使其指向下一个可用的位置。（就是要指的，一定要先在 to space 里面）

```
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
```

3. BFS: 对于每个程序变量（Root） `r`，都需要通过转发操作将其指向的对象修改为对应的到空间中的记录。这样可以确保从空间中所有可达的记录都会被复制到到空间中，并得到有效对象的转发。

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

CH14 Tiger 语言分析与 lex、yacc

14.1 Tiger 的抽象语法树

1. 首先是其数据结构，内部有 pos。然后是例子为 a := 5; a+1。这里的数字指代的是 pos，采用的是字符计数

```
typedef struct A_var_ *A_var;
Struct A_var_
    { enum { A_simpleVar, A_fieldVar, A_subscriptVar } kind;
        A_pos pos;
        union { S_symbol simple;
                struct { A_var var;
                         S_symbol sym ; } field;
                struct {A_var var;
                        A_exp exp; } subscript;
              } u;
    };
```

```
A_SeqExp(2,
A_ExpList(A_AssignExp(4,A_SimpleVar(2,S_Symbol("a")),A_IntExp(7,5)),
A_ExpList(A_OpExp(11,A_plusOp,A_VarExp(A_SimpleVar(10,S_Symbol("a")))),A_IntExp(12,1))),NULL)))
```

14.2 Tiger 的语法分析部分

1. 符号表
1.1 将将每个字符串转换为符号（Symbol）模块，因为（1）比较符号的相等性很快（只需进行指针或整数比较）。（2）提取整数哈希键很快（以便我们可以创建一个将符号映射到其他值的哈希表）。（3）比较两个符号的大小关系（按照某种任意的排序方式）很快（以便我们可以创建二叉搜索树（4）**有两个 mapping，分别是 type 和 value environment**
1.2. 使用破坏更新的环境
1.3. 需要有一个 auxiliary stack，showing in what order the symbols were “pushed” into the symbol table. As each symbol is popped, the head binding in its bucket is removed.
1.4. 需要有一个 a global variable top，showing the most recent Symbol bound in the table：pushing: copy top into the prevtop field of the Binder;"stack" is threaded through the binders.
2. 编译器的绑定
Tiger 有两个独立的 name space，一个是类型的名字空间，一个是函数和变量的名字空间：Tiger 的基本类型是 int 和 string。每一种类型要么是基本类型，或者就是构造成的记录、数组等。
3. 表达式的类型检查
包括：变量、下标和域的类型检查
4. 声明的类型检查
变量声明、类型声明、函数声明、递归声明的检查。

14.3 Tiger 的栈帧

1. 若使用 MIPS 架构，那么 mipsframe.c 中将#include”frame.h”,一般而言可以假定编译器中与机器无关的部分是以 include+frame =F_newframe()来实现的；编译器使用 frame 不用知道目标机特征
2. 栈帧如下，函数 g，True 代表逃逸要放在存储器 F_newFrame (g, U_BoolList (TRUE, U_BoolList (FALSE,U_BoolList (FALSE, NULL))))
3. Inframe(X)表示栈帧中偏移,InReg(84)代表放在寄存器里。F_formals 指明了运行时存放参数的位置，是从 calee 角度看的，caller 和 callee 角度是不一样的。因此 newFrame 函数必须计算：(1)参数内怎么看参数，是在寄存器还是内存(2)为了实现 view shift，需要哪些指令
3. 栈帧的描述,存储了•instructions required to implement the "view shift,"•the number of locals allocated •the label at which the function's machine code is to begin
4. 局部变量：allocLocal 根据 boolean 返回
5. 非逃逸的局部变量可以分配到寄存器，逃逸的必须在栈帧，FindScape 遍历 AST 寻找，要在 semantic analysis 前完成。当大于深度 d 使用 a，x 的 escape 设为 true
6. 临时变量和标号：temporary 代表暂存寄存器的值，是局部变量的抽象名；label 代表准确地址还需要确定的位置，是静态存储器地址的抽象名
7. Frame 不应知道静态链信息，由 Translate 负责
8. 追踪层次信息：每次使用 Tr_newLevel,Semant must pass the enclosing level value

14.5 LEX

```
#include <stdio.h>
int char_count = 0, word_count = 0, line_count = 0;
int flag = 0;
%}
WORD [A-Za-z0-9]+
CHANGE [" "\n\t]
%%
\n {//换行符
    ++char_count;
    ++line_count;
    if(flag == 1)//如果之前是一直在读单词的
        ++word_count;
    flag = 0;//重新开始记数
}
{CHANGE} {
    if(flag == 1)
        ++word_count;   // 之前是一个单词，碰到 CHANGE 可以重新计数
    flag = 0;       // 重新开始记录
    ++char_count;
}
{WORD} {
    if(flag == 0)
        flag = 1; //只有前面是一个空格或者换行符，才能重新开始记数
    char_count += yyleng;
}
. {
    flag = 2; // 如果在单词处理的时候，末尾发现不是字符，不能记单词数
    ++char_count;
}
%%
int main() {
    yylex();
    if(flag == 1)
        ++word_count;   // 1: read in a word
    printf(" char: %d\n word: %d\n line: %d\n", char_count, word_count, line_count);
    return 0;
}
int yywrap()
{
        return 1;
}
```

14.6 YACC

计算器程序

```
%{
#include "y.tab.h"
#include <stdio.h>
%}

NUM [1-9]+[0-9]*|0
%%
{NUM}
sscanf(yytext,"%d",&yylval.inum); return NUM;//yylval.inum = atoi(yytext);
"+"                     return ADD;
"-"                     return SUB;
"*"                     return MUL;
"/"                     return DIV;
"^"                     return POW;
"("                     return LPAREN;
")"                     return RPAREN;
\n                      return CR;
[ \t]+                  /* ignore whitespace */;
.                       //yyerror("Unknown Character");
%%

%{
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
int yylex(void);
void yyerror(char *);
%}

%union{
   int inum;
   double dnum;
}
%token ADD SUB MUL DIV POW LPAREN RPAREN CR
%token <inum>NUM
%type <inum> expression term factor

%%
        line_list: line
                | line_list line
                ;

        line : expression CR    {
                printf(">>%d\n",$1);
            }

            expression: term
                | expression ADD term {$$=$1+$3;}
                | expression SUB term     {$$=$1-$3;}
                ;

            term: factor
                | term MUL factor {$$=$1*$3;}
                | term DIV factor                   {
if ($3!=0) $$=$1/$3;

else {yyerror("Division by zero."); exit(1);}
                                                    }
                ;
            factor: NUM {$$ = $1;}
                |     LPAREN expression RPAREN {$$ = $2;}
                |     SUB factor {$$ = -$2;}
                |     factor POW factor {$$ = pow($1, $3);}
                ;
%%
void yyerror(char *str){
fprintf(stderr,"error:%s\n",str);
}

int yywrap(){

return 1;
}
int main()
{

yyparse();
}
```

CH15 历年卷概念

1. There is only one parse tree for a string of an unambiguous grammar. True(已消除二义性)
2. LL(1)没有 reduce,有 match generate accept
3. Left recursion is commonly used to make operations left associative
4. LR(1) parser can detect errors earlier than LR(0) True(外面的早一些发现错误)
5. Yacc use **LALR(1)**
6. The symbol table will not carry the data type
7. 如果画 NFA 没有要求，可以简化画，但是注意像(01)*这样的其实是两个状态间的来回
8. **A grammar is ambiguous if it has 2 different derivations or 2 different parse trees for a sentence (False)**
9. If a grammar is LR(1) but not LALR(1),There are not shift-reduce conflicts in its parsing table of LALR(1) **(True,只有 reduce-reduce 的问题)**
10. LR(1) item[A->a.Br,a],FOLLOW(B)={ra}
11. A LR(1) parser cannot parse any left-recursive CFG without ambiguity **False**
12. There is only one parse tree for the string of an ambiguous grammar **True**
13. Finding the next handle is the man task of LR parser **True**
14. The parse tree will not completely reflect the derivation steps for a string **False**
15.four parameters passing methods:
(1)值传递 | Pass by Value
就是 C 语言中的参数传递方式。
(2)引用传递 | Pass by Reference
传递变量的引用，即传递其在存储空间中的位置。在函数中所做的一切更改都会作用于这个变量本身。
(3)值结果传递 | Pass by Value-Result
将实参从左到右逐个复制到形参中，在函数运行结束后再逐个将其复制回原来的位置。
(4)名字传递 | Pass by Name
名字传递的思想是：直到函数真正使用了某个参数时才对其赋值，因此也称为 延迟赋值(delayed evaluation)。等价的做法是将函数在调用的原位进行展开。

```
int i = 0;
void p(int x, int y)
{ x += 1;i += 1;y += 1;}
main()
{
    int a[2]={1,1}; p(a[i], a[i]);
    printf("%d %d\n",a[0],a[1]);
}
```

(1)的结果是 1 1，不变
(2)的结果是 3 1，相当于两次 a[0]+1
(3)的结果是 2 1，相当于先取 p(a0,a0)两个 1 到里面，把 x=2 放入 a0,再把 y=2 放入 a0
(4)的结果是 2 2，相当于是 a[i] += 1; i += 1; a[i] += 1;
16. The output of the scanner is token
17. Lex is a tool that is a lexical analyzer generator
18. Static variables is not commonly found in the stack frame
19. **caller operation push the return address in the stack frame**
20. 终结符$不应该出现在 LR 里
21. In the production B->aAr,e 不在 follow(a)里
22. Scopes of the variables are intercrossed sometimes **Flase**
23. Yacc can not use ambiguous grammars **False**
24. The best choice of data structure of the symbol table is HASH table
25. LR(0) is the least powerful
26. We do error recovery with (1)add error production(2)modify the parsing tables(3)modify the parsing engine **but not** eliminate the conflict
27. **sp pointer** is commonly found in a stack frame(activation record)
28. The motivation to divide the complier is to **provide portability of costomer**
29.Heap management is related to mark and sweep;memory compaction;display but not have **stop-and-copy**
30.历年题里，我认为 compute the argument should be done by the caller
31.bookkeeping-control link
32.delayed used in pass by name
33. semantic analysis output an annotated tree
34. use access link to retrieve nonlocal data declared within another procedure(stack-based environment with local procedure)
35.yacc keep a value stack parallel to parsing stack
36. a traverse order of the dependency graph must be undirected and cylic
37. Intermediate code can be very high level or it can closely resemble target code. **True**
38. The general organization of runtime storage will contain the code area, the global/static area, a stack, a free space as well as a heap. **True**
39. When a procedure is called, an activation record P is generated on stack. There may be activation records on stack corresponding to sibling children nodes of P in the activation tree-True
40. 3-address codes have 3 fields **True**
41. The activation record kept in the stack is always directly pointed by frame pointer (fp) **F**