**Chapter 1 Overview**

**phase**(Source code) Scanner (Token) Parser (Syntax Tree) Semantic analyzer (Annotated Tree) Source code optimizer (Intermediate code) Code generator (Target code) Target code optimizer (Target code) **literal表symbol表错误处理参与全程**lex的输出为scanner，IR（语法树、三地址码、pcode）**Related Programs** Interpreter, assembler, linker, loader, preprocessor, editor, debugger, profiler, project manager.**前端**基于源代码语言scanner,parser,semantic

analyzer,intermediate code synthesis

**后端**基于目标语言code generation, optimization analysis **Pass**在生成代码之前处理整个源程序的重复被称为Pass；Pass可能由多个phase组成。单次efficient compilation, less efficient target code. 一般用多个pass (scan+parse, 语义分析+源代码优化, codegen+target-level opt) **Major Data Structure** Symbol Table, Literal Table, Tokens, Syntax Tree, Intermediate Code, Temporary Files. **Error Handling** static(编译时) errors必须被报告（生成有意义的错误消息，并在每个错误之后恢复编译；编译器的每个阶段都需要不同类型的错误处理）；Exception（生成额外的代码来执行运行时测试，确保所有此类错误在执行期间导致适当的事件）

**Chapter 2 词法分析**

**Regular Expressions**

\*优先权最高，连结其次 | alternation最低。

{ }集Φ不包括任何串，而{ε}则包含一个没有任何字符的串。L0 = { ε }

nat = [0-9]+

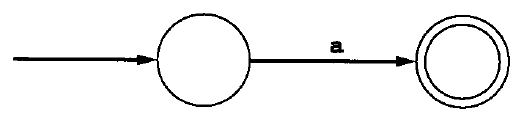
signedNat = (+|-)?nat

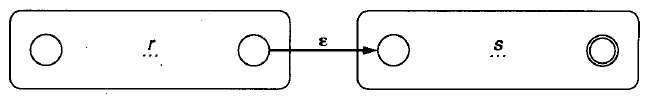
number=signedNat(“.”nat)?(E signedNat) ?

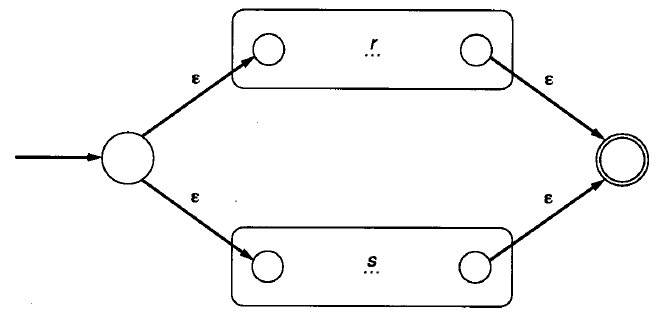
**C语言Comment:**b为/,a为\*,~为非

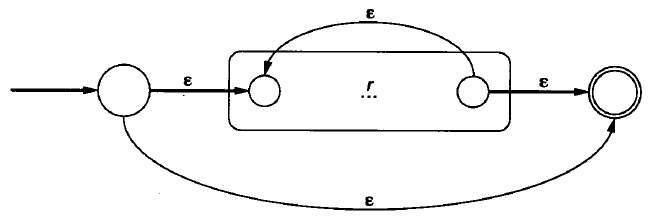
ba(∼**(ab)**)\*ab **×** b\*(a\*~(a|b)b\*)\*a\* **√**

**Thompson’s construction:**









DFA不能处理计数类的language，除非数量有上限（用CFG）

**LEX**输入包含正则表达式的文本文件，以及匹配每个表达式时要采取的操作；输出包含过程yylex的C源代码，它是一个表驱动的DFA实现。文件格式：

定义: %{ C代码%}, 标识符 正则表达式

%%

规则:正则表达式 {动作,C语言}

%%

{用户程序}(在规则段中要用到的一些函数或子程序等)

**yyin**FILE \*类型变量,正在解析的文件**yyout**FILE \*类型变量,输出的位置。**yytext**char \*类型变量，指向刚刚识别到的字符串。**yyleng**整型变量，保存刚刚识别到的字符串的长度。

**Chapter 3 语法分析&CFG**

**Leftmost derivation**: a deri in which the leftmost nonterminal is replaced at each step in the derivation. = preorder =

**Rightmost** = reverse post order =

**parse tree**:叶子结点的中序遍历为input。内部节点是N，叶子为T，对每个内部节点推导时，结果从左到右依次成为该节点的子节点。（编号指推导顺序）最左推导的内部结点的编号为前序编号，最右推导的内部结点的编号为反向后序编号。**对于每一种最左推导，有且仅有一种最右推导与其形成同样的语法分析树；其区别只是树的分支加入的顺序。**

**Ambiguity二义性** grammar的一种推导会生成多种parse tree，非二义性的grammar的多种推导只有一种parse tree,措施Change grammar or disambiguating rule.

**Abstract Syntax Tree:**去除了终结符和非终结符信息，直接为输入的数字等，仅保留语义信息,一般用左孩子右兄弟

**Precedence**:

离根越远，优先级越高。

exp → exp addop exp | term

addop → + | -

term → term mulop term| factor

mulop → \*

factor → ( exp ) | number

**Associativity**:

左递归rule左结合，右递归rule右结合

exp → exp addop exp | term

replacing the rule by

exp → exp addop term |term (左)

exp → term addop exp |term(右)

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(enif)

inessential ambiguity:无关紧要的二义性,虽然parse树各不相同，但是AST相同,例如算术加法虽然可结合但是结合顺序无关紧要。二义性影响parse tree，可能影响AST。

EBNF:A->a{b}表b可重复,花括号在右是左递归,A->a[b]表b可选

Chomsky Hierarchy: 0 unrestricted;

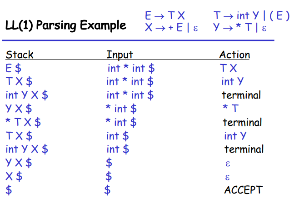
1 context sensitive; 2 context free;

3 regular

**Chapter 4 Top-down parsing**

**Recursive-Descent Parsing**: right hand side of the grammar for A specifies the structure of the code for this procedure, requires the use of **EBNF**. **递归下降**分析需要先将语法消除左递归，不断尝试并回退，找到与input匹配的parse tree。

**LL(1) Parsing**: 消除左递归、合并左因子、计算FIRST、FOLLOW集合，得到table。三种动作**generate & match & accept**



A grammar is an LL(1) grammar if the associated LL(1) parsing table has at most one production in each table entry.

An LL(1) grammar cannot be ambiguous.

**Left Factoring**:

=> ，

**Left Recursion Removal:**

直接左递归：

间接左递归：不能有产生式和循环

for i:=1 to m do 其中m是N的个数

for j:=1 to i-1 do

replace each grammar rule choice of the form Ai→Aj by the rule Ai→, where Aj→ is the current rule for Aj

Remove, if necessary, immediate left recursion involving Ai

Left Recursion Removal不会改变语言，但会改变语法和解析树，这一点给 Parser 带来很大困难。

**先把每个非终结符的集合包含关系写好**

**First找左边有的，Follow找右边有的**

**FIRST set**: 1. If X is a terminal or ε, then First(X) = {X};

2. If X is a non-terminal, then for each production choice X→X1X2 … Xn, First(X) contains First(X1)-{ε}.

3. For each i=2,…,n, if for all k=1,..,i-1, First(Xk)contains ε, then First(X)

constains First(Xi)-{ε}.

4. If all the set First(X1)..First(Xn) contain ε, the First(X) contains ε.

symbol a is nullable，First(a) contains ε

**FOLLOW set**: 没有ε

1. if A is the start symbol, the $ is in the Follow(A).

2. if there is a production B→ α Aγ, then First(γ)-{ε} is in Follow(A).

3. if there is a production B→α Aγ, such that ε in First(γ), then Follow(A) contains Follow(B).

**Grammar is LL(1) if:**

1. For every production A→α1|α2|…|αn, First(αi) ∩ First(αj) is empty for all i and j, 1≦i,j≦n, i≠j.

2. For every non-terminal A such that First(A) contains ε, First(A) ∩ Follow(A) is empty.

**Construct of the LL(1) parsing table**:

For each non-terminal A and

production choice A→ α.

1. For each token a in First(α), add A→α to the entry M[A,a].

2. If ε is in First(α), for each element a of Follow(A) (a token or $), add A→α to M[A,a].

**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。为每个递归过程提供一个额外的set of synchronizing tokens。遇到错误时,就向前扫描,并且一直丢弃token直到遇到一个同步token,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,并更早检测错误。同步记号随着递归不断传递并增加新值。

**checkinput**在每个procedure中调用两次，1st检查下一个token在不在First中，2nd检查下一个token在不在follow中

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，扫描直到可以重新开始分析的记号后，再弹出该记号③特殊情况下压入一个新的N

**Chapter 5 Bottom-Up Parsing**

Yacc基于LALR(1)；使用显式栈完成分析。自底向上分析是最右分析的一个逆过程，右递归可能引起栈溢出。reduce代表w全是T，是要处理的部分

**shift & reduce & accept**

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，输入为空时的动作④error

先统一加上一个S‘->S。分析栈的倒序即为推导过程，**推导**中的每个**中间串**都称作右句型right sentential form生成右句型前分析栈中对应**移进**的符号序列（包括自己）都被称为右句型的可行前缀 (分析栈空时，可行前缀为ε)



E, E + ,and E+n are all **viable prefixes**

of the right sentential form E+n.

句柄:某reduce的右边部分|输入,reduce

n|+n & E->n; E+n| & E->E+n

若语法无二义性，则句柄唯一

确定解析中的下一个句柄是Shift-Reduce解析器的主要任务。

**LR(0):**

LR(0) item表示凑production’s rhs进度

对于production A->e，则有A->**·**

①X→.αβ接受α后就会变成X→α.β ②如果存在X→γYw,Y→αβ，那么X→γ.Yw 就可以转换为Y→.αβ。用核心项写出DFA，若·后为T则在该状态中加上T的production，被称为闭包项，只需要核心项就可以构造出所在的DFA

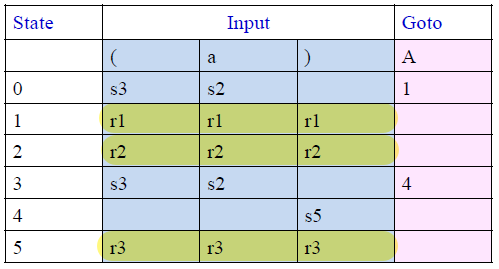
**构造table：**1、对于DFA中的每一个 （状态），令T[i,t]= sj, t 是T；

2、对于DFA中的每一个 （状态），令T[i,X]=gj,其中 X 是N；

3、对于每一个complete（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。

sk表示将对应的terminal压栈，然后移到状态 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.

A grammar is LR(0) if and only if: each state is a shift state( a state containing only “shift” items) or a reduce state containing only a single complete item.



**使用table** 维护分析栈,$0开始，input以$结尾。每次shift将t和状态号push入栈中。若为rx，用x production规约，rhs的长度为m，则分析栈pop m个symbol和state，给分析栈push lhs，用lhs（N）和栈中最顶端的state查得gx，push x；最后两栈中为0和开始符号即accept

**SLR(1):** 在state层面进行look ahead

LR(0)和SLR(1)的DFA一样

**构造table**只有第三条不一样，规约时检查t是否在follow(A)中，在的才加rk

A grammar is SLR(1) if and only if, for any state s: 1. For any item A → α·Xβ in s with X a terminal, there is no complete item B → γ. in s with X in Follow(B). 2. For any two complete items A → α·and B →β· in s, Follow(A) ∩ Follow(B) is empty.

违反第一条代表shift-reduce conflicts, 可以通过自然的消除规则“优先选取shift而不是reduce”，但违反第二条会有reduce-reduce conflicts.

**LR(1):**在item层面进行look ahead

LR(1) items:[A→α·β,a]前面是 LR(0) item，后面是lookahead token，在FOLLOW(A)中

LR(1)的起始状态为[S’->·S,$]的闭包

LR(1) Item在如下两种情况下转化：

①X->.αβ,t接收α后就会变成X->α.β,t

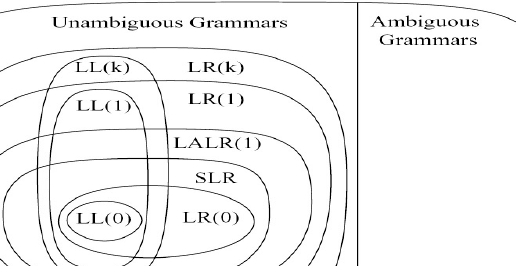
②Given an LR(1) item [A→α·Bγ,a], B is a N, there are ε-transitions to item [B→·β,b] for every production B→β and for every token b in First(γa)

**构造table**只有第三条不一样，规约(complete)时对所在状态i以及每一个lookahead symbol t，令T[i,t]=rk

A grammar is LR(1) if and only if, for any state s: 1. For any item [A→α·Xβ,a] in s with X a terminal, there is no item in s of the form [B→γ·,X] (shift-reduce conflict); 2. Complete item’s lookahead重合(reduce-reduce conflict).

**LALR(1):** 对每个状态，将其包含的所有LR(1) items的第一个分量的集合称为core，将core相同的状态合并，可能出现A->c.,d/e这种多个lookahead symbol的情况

* If a grammar is LR(1), then the LALR(1) parsing table cannot have any shift-reduce conflicts, there may be reduce-reduce conflicts.
* If a grammar is SLR(1), then it certainly is LALR(1)
* LALR(1) parsers often do as well as general LR(1) parsers in removing typical conflicts that occur in SLR(l) parsing.
* If the grammar is already LALR( 1 ), 用 LALR( 1 ) 而不是普通LR的结果是在declare错误前可能会进行一些虚假的reduction
* 通过propagating lookahead的处理可以从LR(0)的DFA直接计算出LALR(1)的DFA(DFA的state相同)



**Error recovery**

LR(1)比LALR(1)或SLR(1)更早检测出错误，LR(0)最晚。自底向上解析器中的一个很好的错误恢复方法：从解析堆栈或输入中删除符号，或同时删除符号。

三种替代方案：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.当出现如下错误时：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.

**Chapter 6 语义分析**

编译器完成的是静态语义分析，构造符号表，在表达式中进行类型推断和检查。

属性的计算及将计算值与语言结构联系的过程称作属性联编binding。执行之前联编的属性是静态的，期间是动态的。

Attribute grammar是属性等式的集合

Xi.aj=fij(X0.a1,…,X0.ak,X1.a1,…,X1.ak,…Xn.a1, …Xn.ak)属性计算通过属性等式Attribute equation或语义规则semantics rule表示，用相关依赖图表示每个文法规则的依赖关系，箭头所指的属性后于箭头出发的属性被计算。属性等式中可以出现的表达式集合称为meta-language，限于算术式、逻辑式和if switch、函数等

字符串345o（八进制）的相关图

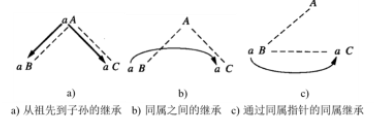


对于非循环acyclic的图DAGs，找到拓扑排序（维护入度为0的节点的集合，每拿出一个放到list中后删除从它开始的边，将删后的入度0的节点加入集合中），得到计算属性的顺序。

合成Synthesized属性：它所有的相关都从子节点指向父节点，每个文法规则中关于这个属性的语义规则都是右定左

如果一个grammar中的所有属性都是合成属性，那称为S属性文法，给定分析树或语法树，其属性值可以通过对树简单自底向上或后序遍历计算

继承Inherited属性：非合成属性

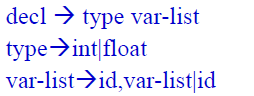
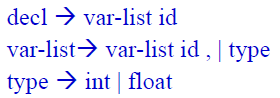


继承属性的计算可以通过对分析树或语法树的前序遍历或前序/中序遍历的组合来进行，在子孙的属性中继承属性可能有依赖关系。

如果继承属性不依赖于任何合成属性，那么就可能在分析树或语法树的一遍遍历中计算所有的属性。

L属性文法：每个继承属性和每个产生式，该继承属性的值只依赖于规则rhs中该属性左边出现的符号的属性。S属性文法都是L属性文法**。**给定一个 L 属性文法，对于Top-Down（递归遍历程序）可以通过把继承属性转换成参数、把合成属性转换成返回值；对于Bottom-up，用**value栈（YACC维护与parse栈并行，每个symbol有一个value）**处理合成属性，用符号表或非局部变量等外部数据结构保存继承属性值并增加e产生式

继承属性改为合成属性改成子节点向父节点传递(没有递归向下传递的继承也可),便于自底向上分析

把type放在最深的叶节点最开始分析

符号表：字典数据结构，插查删，存储数据类型、作用域、存储器中的最终定位。线性表、搜索树（删除太复杂）、**hash表**（都常数）open addressing冲突后填下一个（性能下降）separate chaining对每个桶一个链表，包括函数名（最外层），新项插在最前面。Bucket大小为素数，多作用域可以维护符号表栈(近的作用域靠近bucket)

**declaration**①constant②type Pascal的类型声明以及C中的struct和union说明、typedef③Variable变量声明同时约束其他属性,最主要的是声明作用域Fortran和C④procedure/function包含显式和隐式

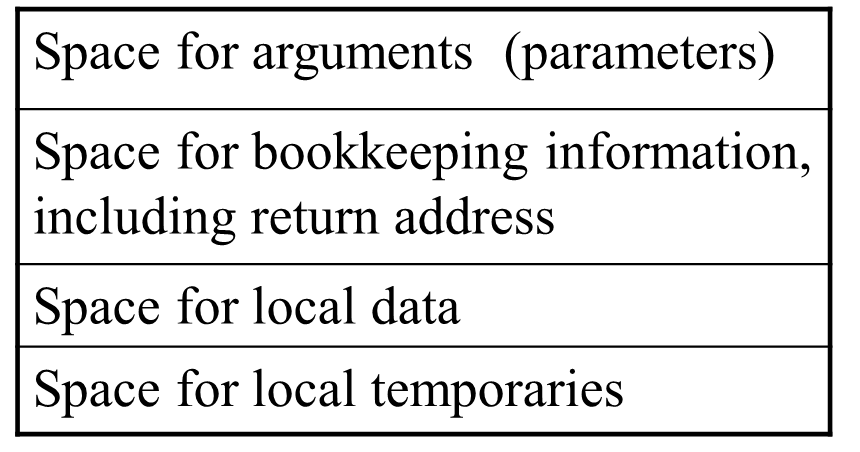
**策略**①使用一张符号表保存所有不同类型声明的名字,禁止在不同类型的生命中使用相同的名字 ②对每种说明使用不同的符号表 ③程序的不同区域(函数或过程)都有独立的符号表，并且根据语言的语义规则链接到一起，C Pascal Ada都是这样。**作用域规则**①使用前声明(C, Pascal) ②最近嵌套规则

同层次中不能使用相同的名字，这要求每次插入符号表前都进行一次查找。

顺序声明：在处理符号表时，将每个声明添加到符号表中；并列声明collateral：声明不直接添加，在新表中累计，处理完所有声明后添加到现有表；递归声明：说明可以引用自身或者互相引用。

顺序调用可以通过修改代码顺序完成；相互调用就需要在开始加一个函数原型的说明。**类型等价**结构等价,可用于缺省类型名，语法树相同；名等价,类型名相同；说明等价,显式声明别名。Pascal使用声明等价,C的结构和联合使用声明等价,指针和数组使用结构等价。

**Chapter 7 运行时环境**

**存储器组织**寄存器区和RAM，RAM分为代码和数据区,AR中存有：pc,sp,fp(当前AR),无sp,ap指向AR

**calling sequence**为活动记录分配内存,参数的计算和存储, 存储和设置影响调用的寄存器.**returning sequence**将返回值放置在caller可以访问的位置,在汇编语言中使用ax,恢复现场,释放活动记录的内存。**caller**计算参数并将其放置在callee可找到的位置。**都可以**保存返回地址、簿记信息和寄存器现场。

**FORTRAN77**，完全静态。所有数据静态,每个过程一个AR,只保存返回地址,可以通过地址访问任何变量。调用过程时，每个自变量保存到被调用过程的AR中适当的参数位置,保存返回地址,然后跳转。参数值是隐式储存引用,用额外的引用来访问参数,不需要复制数组参数。

**C**用栈存AR，无局部过程。用fp访问本地变量，sp指向调用栈的最后（栈顶）。控制control/dynamic链指向先前AR的control链，fp指向当前AR。名称访问时用fp+偏移量，向上+向下-



调用时，倒序push参数，push fp作为控制链，fp赋值为sp，push ra，跳转

返回时，sp=fp，pop控制链fp，跳转到ra，改变sp弹出参数 （ar中有fp）

变长参数：倒序压栈，+4偏移的值说明总参数数；额外push一个数组长度

**Pascal（有）**，栈。访问链指向**定义该过程的直接父过程**，用于寻找父过程中的变量，为每个声明计算嵌套层，调用时计算要链几次。可以用display来访问变量



**C++**，对象、方法、继承、动绑。将虚函数表存在静态存储中

**LISP**用堆，使用栈会导致悬挂引用，返回局部参数地址的函数。

关键是回收不可达的对象 align将存储器元素排在合理的字节边界上

malloc和free，手动管理堆的block

memory compaction内存规整

mark和sweep，在malloc失败后才会激活，遍历两次block list，需要额外存储空间，存在较大延迟。

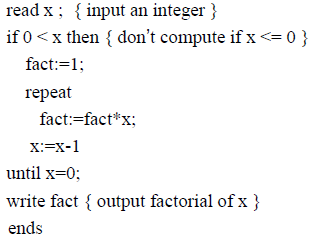
stop and copy/two space，存储器分两部分，每次只从其中一部分分配空间，在mark时将路过的block复制到另一半区域中，自动压缩，仍有延迟。

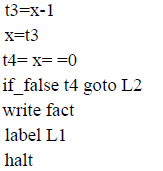
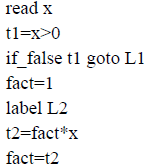
generational garbage collection，将存在时间足够长的对象复制到永久空间，之后不再回收它，减少搜索量。

**参数传递 先写出调用p(a[0],a[0])**

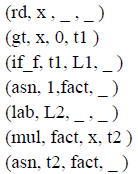
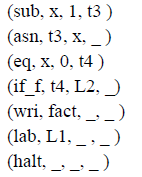
Value, C Pascal Ada默认传递方式，变量的值复制到形参的变量中Reference Fortran77，传递变量在存储空间中的位置，函数内更改作用到本身Value-result将实参从左到右逐个复制到形参中，函数运行结束后再逐个复制回去Name 函数真正使用某参数时才对其赋值，延迟赋值（将函数在调用处展开）

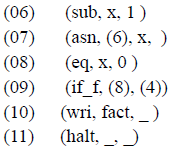
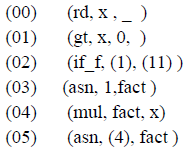
**Chapter 8 Code Generation**





**Quadruple implementation & triples:**



(go,L1,\_,\_) goto无条件跳转

P-code依赖栈，需要更多指令，不用分配新变量名。生成代码依赖macro expansion和static simulation

栈顶值，地址；sto后地址赋为值，俩都pop；stn后栈顶为值。lda push地址，lod值，ldc常数。二元运算符pop两个push结果。adi+ sbi- mpi\* grt> equ== neq!=。fjp label pop布尔值假跳转，ujp跳转，lab L定义标签rdi pop栈顶的地址并读值wri pop栈顶的地址并输出值stp为halt

**代码生成** e→id=e | a; a→a+fac | fac;

fac→(e)|num|id;

|  |
| --- |
| e1.pcode=”ida”||id.strval++e2.pcode++”stn |
| a1.pcode=a2.pcode++fac.pcode++”adi” |
| fac.pcode=”ldc”||num.strval / ”lod”||id.strval |

**e:**ID{sprint(codestr,”%s%s”,”lda”,$1); emitCode(“stn”);} ‘=’ e{eC(codestr); } | a; **a:**a ‘+’fac {eC(“adi”);} | fac; **fac:** ‘(‘ e ‘)’ | NUM {sprintf(codestr,”%s%s”,“ldc”,$1); eC(codestr);} | ID{sprintf(…,“lod”, $1); eC(…);}

**数组引用a[i+1] = a[j\*2]+3**

t1=j\*2,t2=t1\*elem\_size(a),t3=&a+t2,t4=\*t3,t5=t4+3,t6=i+1,t7=t6\*e\_s(a),t8=&a+t7,\*t8=t5

lda a,lod i,ldc 1,adi,ixa e\_s(a),lda a,lod j,ldc 2,mpi,ixa e\_s(a)ind 0,idc3,adi,sto

结构引用**x.j=x.i**

t1=&x+field\_offset(x,j),t2=&x+f\_o(x,i),

\*t1=\*t2

lda x,ldc f\_o(x,j),ixa1,lda x,ind f\_o(x,i),sto

**p->lc=p;p=p->rc;**

t1=p+f\_o(\*p,lc),\*t1=p,t2=p+f\_o(\*p,rc),p=\*t2

lod p,ldc f\_o(\*p,lc),ixa 1,lod p,sto,lda p,lod p,ind f\_o(\*p,rc),sto

条件分支**if(E) S1 else S2**

E赋值给t1,if\_false t1 goto L1,S1 code,goto L2,label L1,S2 code,label L2

Epush到栈顶,fjp L1,S1code,ujp L2,lab L1,code S2,lab L2

循环分支**while(E) S**

label L1,E赋值给t1,if\_false t1 goto L2,Scode,goto L1,label L2

lab L1,Epush到栈顶,fjp L2,Scode,ujp L1,lab L2

**if(1) while(1) if(0) break else other**

ldc true,fjp L1,lab L2,ldc true,fjp L3,ldc FALSE,fjp L4,ujp L3,ujp L5,lab L4,other,lab L5,ujp L2,lab L3,lab L1

逻辑短路**(x!=0)&&(y==x)**

lod x,ldc 0,neq,fjp L1,lod y,lod x,equ,ujp L2,lab L1,ldc FALSE,lab L2

函数定义（声明）有entry和return point

**int f(int x,int y){return x+y+1;}**

entry f,t1=x+y,x2=t1+1,return t2

ent f,lod x,lod y,adi,ldc 1,adi,ret

函数调用有指令指示参数计算的开始

**f(2+3,4)**

begin\_args,t1=2+3,arg t1,arg 4,call f

mst,ldc 2,ldc 3,adi,ldc 4,cup f

**fn f(x)=2+x fn g(x,y)=f(x)+y g(3,4)**

ent f,ldc 2,lod x,adi,ret,ent g,mst,lod x,cup f,lod y,adi,ret,mst,ldc 3,ldc 4,cup g

**代码优化**1寄存器分配:增加直接在内存执行的操作的数量和速度；减少在内存直接执行的操作数,但是增加可用寄存器数;2不必要操作:common subexpression elimination，避免存储不再使用的变量的值；（必须先完成常量传播）消除不可达代码、跳转优化（跳到下一语句和跳到跳转语句的都优化掉）3、高代价操作，reduction in strength如乘变移位、幂变连乘,constant folding(常数相加)、constant propagation(变量恒定值），过程内嵌、尾递归消除4、predicting program behavior

**按优化的时机分类Peephole optimization**在目标代码上进行优化，通常只能在局部优化。**Target-level optimization**在生成目标代码前，结合目标机器的特性进行优化。**Source-level optimization**在生成目标代码前，以机器无关的方式进行优化。**按优化的范围分类local**限于线性部分 (straight-line segment of code) 的优化，即没有跳进跳出的语句的语句块。一个最大的线性部分称为基本块 (basic block)global，超出基本块但在单个过程内。interprocedural，过程间。优化使用的数据结构：以basic block为结点，构建flow graph。对每个basic block构建DAG来跟踪代码中的计算和赋值（拷贝操作不创建新结点，只是给结点增加一个标号），在DAG的多种拓扑排序中选最好的指令序列。