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

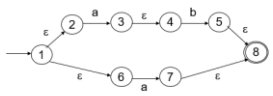
1.*repetition>concatenation 连>alternation 选
2.相同的语言可以用不同的 RE 表示
3.R+：R(R*)；R?：R|ε；[abc]：a|b|c；[a-z]：(a|b|c|..|y|z)；[^ab]除 a 或 b；[^a-z]除 a-z
4.RE 匹配优先匹配保留字；最长字串优先

2.3 finite automata 有穷自动机

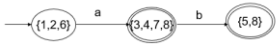
1.DFA：M 由字母表Σ、状态集 S、转换函数 T：S×Σ→S、初始状态 S_0∈S 以及接受状态 A⊂S。
2.错误状态默认不画，但是存在；错误状态下的任何转移均回到自身，永远无法进入接受。
3.NFA：M 由字母表Σ、状态集 S、转换函数 T：S×(Σ∪{ε})→P(S)、初始状态 S_0 以及接受状态 A 的集合。

2.4 RE To DFAs 正则表达式到 DFA

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.同一个串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.最左推导和前序遍历对应，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

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
4.分析表中的每个项目中至多只有一个产生式。

• $S\to(S)\ S\mid\varepsilon$

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

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 → βA′，A′ → αA′|ε
8.普遍直接左递归：
A → $A\alpha_1|A\alpha_2|\dots|A\alpha_n|\beta_1|\beta_2|\dots|\beta_m$
A → $\beta_1A'|\beta_2A'|\dots|\beta_mA'$
A → $\alpha_1A'|\alpha_2A'|\dots|\alpha_nA'|\varepsilon$
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 → α1β|α2β| ... |αkβ , where Aj →
α1|α2| ... |αk 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 → αβ|αγ。A → αA′，A′ → β|γ

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→X_1 X_2 ...X_n，First(X)都包含了 First(X_1)-{ε}。若对于某个 i<n，所有的 Fisrt(X_j),...,First(X_i)都含有ε，则 First(X)也包括了 First(X_{i+1})- {ε}。若所有 Fisrt(X_1),...,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)
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($\alpha_i$)∩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

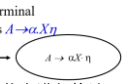
Yacc 基于 LALR(1)；使用显式栈完成分析。

5.1 概览

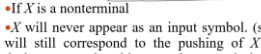
1.动作为①shift，将 T 从输入开头移到栈顶②reduce 使用产生式 A→α将栈顶的α规约成 A③accept 分析栈为开始符号，输入栈为空时的动作④error
2.注意统一额外加一个 S′作为新的开始符号
3.推导中的 N 和 T 的每个中间串都称作右句型 right sentential form，这样的句型都被分析栈和输入分隔开(即使某一边空了也 OK)。在每一种情况下，分析栈的符号序列都被称为右句型的可行前缀 viable prefix(分析栈空时，可行前缀为ε)
3.若语法无二义性，则句柄唯一

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

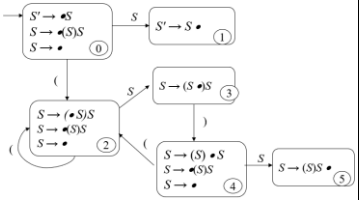
1.LR(0)的项就是在右边带有区分位置的产生式，同时就是 LR(0)的 FA 中的一个状态
2.NFA 构造算法：按照上述规则，从添加的S′→S 作为起始状态进行构造。



•If X is a nonterminal
•X will never appear as an input symbol. (such a transition will still correspond to the pushing of X onto the stack during a parse, but this can only occur during a reduction by a production $X\to\beta$.)



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：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
6.r-r conflict：两个完整项共存则出现 r-r 冲突
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

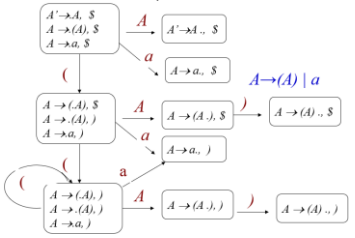
State	Input			Goto
	(	a	)	A
0	s3	s2		1
1	r1	r1	r1	
2	r2	r2	r2	
3	s3	s2		4
4			s5	
5	r3	r3	r3	

5.3 SLR(1)

1.SLR(1)算法定义：①If state s contains any item of form A → α · Xβ (X is T), then the action is to shift the current input token onto the stack, and the new state to be pushed on the stack is the state containing the item A → αX · β②If state s contains the complete item A → α · and the next token in the input string is in Follow(A), then the action is to reduce by the rule A → α ·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:①For any item A → α · Xβ (X is T), there is no complete item B→γ · in s with X in Follow(B).②For any two complete item A → α ·and B → β · in s, Follow(A) ∩ Follow(B) is empty.待移进的终结符不能是完整项的 Follow 元素；两个完整项的 Follow 集不相交
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)

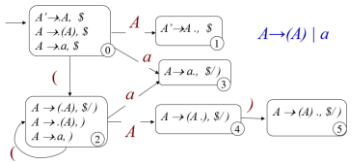
1.LR(1) items：[A → α · β, a]前面是 LR(0)项，后面是 lookahead token
2.LR(1)的起始状态[S′→ • S,$]的闭包
3.LR(1)转移的定义：①非空转移 Given an LR(1) item [A → α · Xγ, a], X is T or N, there is a transition on X to the item [A → αX · γ, a]②空转移 Given an LR(1) item [A → α · Bγ, a], B is a N, there are ε-transitions to itrem [B→ · β, b] for every production B→β and for every token b in First(γa).第一条规则永远不会创建新的先行；实际情况中，往往是γ本身就是ε，此时从格式[A → α · B, a]到[B→ · β,a]可得到ε转移



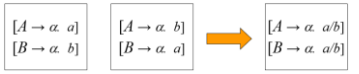
4.LR(1) definition: let s be the current state (at the top of the paring stack). Then actions are defined as follows:①If state s contains LR(1) item of the form [A → α · Xβ, a], X is T and X is the next token in the input string.②If state s contains LR(1) item [A → α ·, a], the next token in the input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



11.文法包含 LR0 ⊂ SLR1 ⊂ LALR1 ⊂ LR1。
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. Three possible alternative actions:① Pop a state.②Successively pop tokens from the input until a token is seen for which we can restart the parse.③Push a new state.
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.

CH6 语义分析

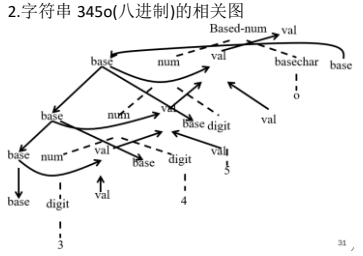
编译器完成的是静态语义分析

6.1 Attributes and attribute grammars

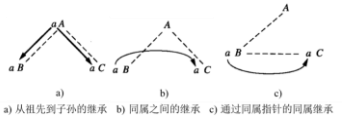
1.属性的计算及将计算值与语言结构联系的过程称作属性联编 binding。执行之前联编的属性是静态的，期间是动态的。
2.属性计算通过属性等式 attribute equation 或语义规则 semantics rule 表示。
$X_i.a_j=f_{ij}(X_0.a_1,\dots,X_0.a_k, X_1.a_l, \dots, X_1.a_k,\dots X_n.a_1, \dots X_n.a_k)$
3.属性文法表，左侧列为 Grammar Rule，右侧为 Semantic Rules
4.属性等式中可以出现的表达式集合称作属性文法的元语言 metalanguage。元语言限于算术式、逻辑式以及 if switch 等语句。也可以使用函数。

6.2 Algorithms for attribute computation

1.给定一个属性文法，每个文法规则选择有一个相关依赖图。文法规则中的每个符号在图中都有用每个属性 $X_i.a_j$ 标记的节点，对于每个属性等式 $X_i.a_j = f_{ij}(\dots, X_m.a_k, \dots)$相关于文法规则从在右边的每个节点 $X_m.a_k$ 到节点 $X_i.a_j$ 有一条边(表示 $X_i.a_j$ 对 $X_m.a_k$ 的依赖)。
2.字符串 345o(八进制)的相关图



4.基于一个算法在相关图的构造上进行属性分析是可能的，因为在编译时相关图是基于特定的语法构造的，这个方法称作分析树方法。
5.依赖于对属性等式或语义规则的分析叫做基于规则的方法 rule-based method
6.合成 synthesized 属性：一个属性是合成的，如果在语法树中它所有的相关都从子节点指向父节点。等价的，给定一个文法规则 $A \rightarrow X_1X_2...X_n$，左边仅有一个 a 的相关属性等式有以下形式 $A.a = f(X_1.a_1,...X_1.a_k...X_n.a_1,... X_n.a_k)$
7.一个属性文法中所有的属性都是合成的，那么就称作 S 属性文法 S-attributed grammar。给定分析树或语法树，S 属性文法的属性值可以通过对树进行简单的自底向上或后序遍历计算。
8.继承 inherited 属性：不是合成的属性就是继承的。



a) 从祖先到子孙的继承 b) 同属之间的继承 c) 通过同属指针的同属继承

9.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。与合成属性不同，子孙继承属性计算的顺序十分重要，因为在子孙的属性中继承属性可能有依赖关系。
10.递归遍历程序用前序计算继承属性，用后序计算合成属性，在子节点把继承属性作为参数传递给递归函数调用，并接收合成属性作为那些相同调用的返回值。
11.外部数据结构替代传参返回的典型是符号表
12.我们所使用的分析方法都是从左到右处理的。这等价于要求属性能通过从左向右遍历分析树进行赋值，这对于合成属性不是限制，因为子节点可以用任意顺序赋值。但是对于继承属性，这就意味着相关图中没有向后的依赖(分析树种依赖从右到左)。属性文法确保的这一特性叫 L 属性
13.L 属性定义：属性 $a_1,..., a_k$ 的一个属性文法是 L 属性，如果对每个继承属性 a_j 和每个文法规则 $X_0 \rightarrow X_1X_2...X_n$，$a_j$ 的相关等式都有以下形式 $X_i.a_j=f_{ij}(X_0.a_1,...,X_0.a_k,\ X_1.a_1,...,X_1.a_k,..X_{i-1}.a_1,\ ...X_{i-1}.a_k)$。也就是说，在 X_i 处 a_j 的值只依赖于在文法规则种 X_i 左边出现的符号 $X_0,...,X_{i-1}$ 的属性。
14.S 属性文法必定是 L 属性文法
15.给定一个 L 属性文法，对于 Top-Down 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，LR 分析适合处理合成属性，对于继承属性很难处理。
16.值栈用来储存合成属性，在分析时并行处理

$	3*4+5$	Shift	$	
Sn	*4+5$	Reduce E→n	Sn	E.val = n.val
$E	*4+5$	Shift	$3	
$E*	4+5$	Shift	$3*	
$E*n	+5$	Reduce E→n	$3*n	E.val =n.val
$E*E	+5$	Reduce E→E*E	$3*4	E1.val=E2.val*E3.val
$E	+5$	Shift	$12	
$E+	5$	Shift	$12+	
$E+n	$	Reduce E→n	$12+n	E.val = n.val
$E+E	$	Reduce E→E+E	$12+5	E1.val=E2.val +E3.val
$E	$		$17	

17.在 LR 分析中，处理继承属性的最好技术是使用外部数据结构
18.定理：给定一个属性文法，通过适当地修改文法，而无须改变文法的语言，所有的继承属性可以改变成合成属性。但是并不推荐，因为会造成文法规则晦涩难懂

6.3 Symbol table

1.符号表主要操作：增删查
2.符号表是一个典型的字典数据结构。典型实现包括线性表、搜索树以及 hash 表
3.线性表：简单直接的实现，插入操作常数时间，查找和删除是 n 复杂度。对于不在乎编译速度的编译器，十分合适。
4.搜索树：没什么用，删除太过复杂
5.hash 表：三种操作基本都是常数时间，最常用，是最优选择
6.hash 函数把索引键 search key(标识符名字符串)转换成索引范围内的一个证书的 hash 值，对应在该索引的桶 bucket 中。
7.关键问题：如何处理冲突 collision resolution。①开放寻址 open addressing：在冲突时，将该项插入到后续相连的桶 successive buckets 中；性能下降很快且删除操作复杂②分离链表 separate chaining：每个桶都实现为一个线性表，把新的项插入到桶对应的表中(头插入)
8.问题：桶的个数，这个个数在编译器构建期间固定不变；一般从几百到上千；大小实际应用中应为一个素数
9.hash 函数三步：①将字符串中的每个字符转换成一个非负整数②将这些整数按照某种规则组合成一个整数③把整数调整到 0~size-1 的范围内
10.实现一：忽略大部分字符，只把开头的几个字符或者第一个、中间的和最后一个字符的值相加。对于编译器来说并不合适。
11.实现二：直接把所有字符值加起来，但是也不行，temp 和 ptem 一样了
12.实现三：当加上下一个字符的值时，重复地使用一个常量作为乘法因子 α。$h_{i+1} = \alpha h_i + c_i$ $h_0 = 0$ 最后的 hash 值就是 $h = h_n$ mod *size* 等价于下列公式 $h = (\alpha^{n-1}c_1+\alpha^{n-2}c_2 +.....+ \alpha\ c_{n-1}+ c_n)$ *mod size*。惩罚因子的合理选择是 2 的幂，这样可以通过移位来完成乘法
13.四种基本说明 declaration：常量、类型、变量以及过程/函数。①constant：把变量的值绑定到名字上②把新构建的类型绑定到名字上，或是创建已有类型的别名 aliases③把名字绑定到数据类型上，变量声明同时隐含地约束了其他属性，最主要的是说明的作用域 scope of declaration④procedure/function：包含显式和隐式说明
14.保存说明的策略：①使用一张符号表保存所有不同类型的说明的名字，这要求语言禁止在不同类型的说明中使用相同的名字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，并且韩赵语言的语义规则链接到一起，C Pascal Ada 都是这样
15.作用域规则两条：①使用前说明 declaration before use②块结构的最近嵌套规则 most closely nested rule for block structure。In pascal, blocks are the main program; procedure/function declarations. In C blocks are the compilation units, procedure/function declarations; the compound statements ({})
16.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。
17.顺序说明 sequential：在处理时，每个说明加进符号表；并列说明 collateral：替代所有要同时处理的说明，在说明部分的最后立即加入到符号表。然后说明中任意表达式的名字将引用前面的说明，不再处理新的说明；递归说明 recursive：说明可以引用自身或者互相引用
18.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

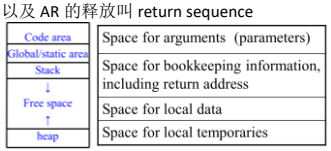
6.4 Data type and type checking

1.仅有的可用于缺省类型名的，结构等价：两个类型当且仅当他们有相同的结构时才相同。等价于语法树相同时才等价。可能会允许数组大小不同、结构体成员顺序不同等；
2.基于类型名的名等价，当且仅当他们是相同的简单类型或有相同的类型名，强等价
3.说明等价，名等价的弱化版，像 t2=t1，t2 时作为类型别名解释的，不是新类型
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 运行时环境

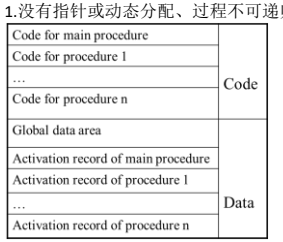
7.1 储存器组织

1.寄存器区和 RAM，RAM 分为代码和数据区
2.存储器组织和 AR
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 Fully static FORTRAN77

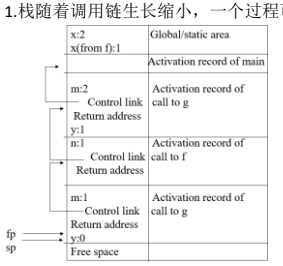
1.没有指针或动态分配、过程不可递归调用；所有数据静态，执行期间保持大小固定。不需要保存除返回地址外的额外环境信息。通过地址可以直接访问任何变量。



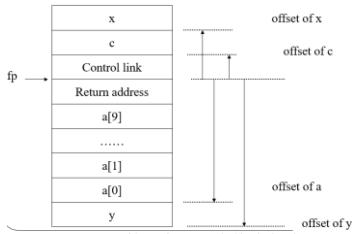
2.调用过程时，计算每个自变量，将其保存到被调用过程的活动中适当参数位置，然后保存返回地址，然后跳转。
3.参数值是隐式储存引用。需要一个额外的引用来访问参数。数组参数无需复制

7.3 Stack-based C Pascal

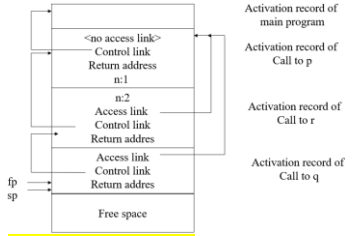
1.栈随着调用链生长缩小，一个过程可以在栈上有多个不同的活动记录，代表不同的调用。允许递归调用，本地变量重新分配空间。



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②sp 指向调用栈的最后
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
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；修改 sp 弹出自变量
7.变长数据：①函数参数可变(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
9.允许局部过程后，前面失效，因为没有提供对非局部且非全局的数据的引用。引入 static/access link。指向当前 AR 的上一层静态作用域。Access link 要比 fp 先进栈在调用时。可以使用 display 数据结构代替 access link。



7.4 Dynamic memory LISP

1.栈环境在这种情况下会导致悬挂引用。
2.活动退出后 AR 还在存储器，此后某个时刻重新分配。特征：对象、方法、继承、动绑
3.基于 malloc 和 free 的堆管理是 manual
4.mark and sweep，在 malloc 失败前都不会调用，调用时寻找可欲引用的所有存储器释放没被引用的。通过两遍扫描完成。需要额外的存储(用于 mark)，速度很慢
5.stop and copy/two space：将存储器分为两部分，每次只从一部分分配，mark 轮时将路过的所有都复制到另一半存储器中。自动压缩。
5.generational GC：将存在时间足够长的对象复制到永久空间，之后不再 GC，减少了搜索量。

7.5 Parameter passing

CH8 代码生成

8.1 IR

1.四元式(op,rs1,rs2,rd)，如果用 op 代替 rd 那么可以缩短到三元式，然后每行三元式前面加行号，代表三元式存结果的数组/链表的下标，之后引用临时变量直接(行号)就好了
2.P-code 包括一个代码存储器、未指定的存放命名的数据存储器、临时数据栈还有一些保持栈和寄存器
3.从中间代码生成目标代码的标准技术：宏拓展 macro expansion 和静态模拟 static simulation
4.Falsejump: if_false t1 goto L1 无条件跳转：goto
if (E) S1 else S2 => <code to evaluate E to t1>; if_false t1 goto L1;<code for S1>;goto L2;label L1; <code for S2>;label L2;
while (E) S => label L1;<code to evaluate E to t1>; if_false t1 goto L2;<code for S>;goto L1;label L2

8.2 P-code

1.地址引用和数组引用
ind i(indirect load) a → *(a+i)
ixa s(indexed addr) a,i(i 在栈顶) → a+s*i
lda x;ldc 10;ixa 1;ldc 2;sto(把常量 2 存入 x 加 10 字节处的地址)
数组引用时常用 ixa elem_size(a)，栈里原本是数组基地址和偏移量
a[t2]=t1 →lda a;lod t2;ixa elem_size(a);lod t1;sto;
2.if 语句和 while 语句
fjp L1 --弹出栈顶的 Boolean 判断，false 则跳
ujp L1 –无条件跳转到 L1
条件语句中的短路也要用跳转做
(x != 0) & & (y = = x) →lod x;ldc 0;neq;fjp L1;lod y;lod x;equ;ujp L2;lab L1;lod FALSE;lab L2;
3.过程和函数(ret 用栈顶数作函数返回值)
声明 int f(int x,int y){return x+y+1;}→
ent f;lod x;lod y;adi;ldc 1;adi;ret;
调用 f(2+3,4);→ mst;ldc 2;ldc 3;adi;ldc 4;cup f;
mst 相当于 begin-args,cup 相当于 call，还有个 csp 是调用内建函数的，我们不考虑
4.其他 p-code
rdi –读入整型存放到栈顶地址 栈顶弹出
wri –将栈顶的值写入输出
grt –比较栈顶两个值的大小并将结果存在栈顶
equ –比较栈顶两个值是否相等结果入栈
stn –同 sto 但是原先栈顶的值仍然留在栈顶
adi,sbi,mpi... --arithmetic expression

8.3 three addr code

1.地址引用和数组引用
地址引用直接采用&，数组引用用[]
2.if 语句和 while 语句
Flase jump:(if_f,t,L,-)无条件跳转(goto,L,-,-)
3.过程和函数
声明 int f (int x, int y){ return x + y + 1; } → entry f;t1 = x + y;t2 = t1 + 1;return t2;
调用 f(2+3,4);→begin_args;t1=2+3;arg t1;arg 4;call f;
4.常用四元式
(gt,x,y,t) -- t=x>y
(asn,src,dst,-) – dst=src
(mul,x,y,dst) – dst=x*y(mul,add,sub,div)

8.9 code optimizations

1.寄存器分配：①增加直接在内存执行的操作的数量和速度②减少在内存直接执行的操作数，但是增加可用寄存器。
2.不必要操作：common subexpression elimination 公共子表达式消除；不再储存不使用的变量或是临时变量；unreachable/dead code elimination
3.高代价操作：小幂次可以改用连乘 reduction in strength；对小整数的乘法替换为移位和加法(4*x 变成 x<<2)；constant folding & constant propagation：去除频繁调用的小过程，可以把过程体内嵌到程序中(找合适地参数代替形参)，也可以消除尾递归 tail recursion removal
4.预测程序行为
5.按照时间对优化分类：在目标代码上的优化叫 peephole op...；一般的优化都是在目标代码前做，这样不依赖于目标机器特性，叫 source-level op...，对应的还有 target-level op...
6.优化之间的顺序很重要，应该在死码消除前完成常量传播
7.按照优化应用的程序范围对优化分类：local, global, interprocedural。局部优化是应用于代码的线性 straight-line 部分，也就是代码中没有跳入或跳出语句。一个最大的线性代码序列称为基本块。拓展超出基本块，但是限制在单个过程中的优化成为全程 global 优化。拓展出过程边界叫过程间优化(很困难)
8.flow graph 用来表示中间代码的过程图。流图的节点是基本块，边来自条件或非条件跳转(作为其他基本块的开始)
9.基本块识别过程：①第一条指令开始一个②每个转移目的标签开始一个③每条跟随在转移之后的指令开始一个
10.计算所有变量的可达定义 reaching definition 集是一个数据流分析问题，变量在每个基本块的开始处。一个定义是一条中间代码指令。如果在块开始处变量保持定义时的值，这个定义成为 reach 基本块。用来常量传播
11.基本块的 DAG。块中使用的来自别处的值表示为叶子节点。其上的操作和其他值表示为内部节点。赋值通过把目标变量或临时变量的名字附加到表示赋值的节点上来表示。
12.DAG 通过两个字典构造，①包含变量名和常数的表，带有可返回当前赋值变量的 DAG 节点的查找操作(可以直接用符号表)②DAG 节点表，带有给出操作和子节点的查找功能。
13.使用 DAG 可以自动得到局部公共子表达式消除，也给消除冗余存储(赋值)带来可能
14.最后还有个 register descriptor 和 addr descriptor。前者给出当前值在寄存器中的变量名；后者给出变量和内存地址的联系。