语义分析导论

1️⃣ 语义的概念

1. 语法和语义：语法决定代码怎么写，语义决定代码的逻辑和含义(代码是什么/能做什么)
2. 语义的核心：语义性质(能决定代码含义的特性)，合法性(序在逻辑上有效)

2️⃣ 语义分析概览

1. 任务：充当"逻辑警察"，如检查是否声明/类型匹配等，验证程序程序合法性(而非正确性)
    - 静态检查：每个运算符的操作数是否符合其预期的类型
    - 控制流检查：确保程序的跳转和流程控制是合法的
    - 唯一性检查：同一作用域内，标识符的定义是否唯一
    - 名称相关检查：函数调用时，传入的参数数量和类型是否与函数定义匹配
2. 特点：区别于词法分析/语法分析都具有形式化的规则，语义规则多用自然语言描述(非形式化)
3. 输出：语义分析将输出语义信息(填满的符号表)，中间代码(AST/三地址码)

符号表

1️⃣ 内容：表头和登记项

1. 表头：逻辑上一个作用域(通常是一个函数)对应一个符号表，表头记录了作用域元数据

表头组成	解释
`outer`指针	指向其外层(父)作用域的符号表，当前符号表找不到某名字时会沿`outer`指针回溯
`width`	占用空间，即所有局部变量占用的总内存宽度
`argc/arglist`	过程参数个数，过程参数列表
`rtype`	函数(过程)返回结果的类型

2. 登记项：登记了代码中的用户定义的每个实体(变量/常量/函数名...)，并记录其名字+语义属性
    - 构成成分：固定部分(`name`记录名字+`type`记录大类(`INT/ARRAY/FUNC`)，及特有部分(视不用类而定)

`type`	类型特有部分
`INT`	`offset`(变量的内存偏移地址)
`ARRAY`	`etype`(数组元素种类)，`base`(基地址)，`dims`(数组维度)，`dim[i]`(第i维长度)
`FUNC`	`return_type`指示函数返回的类型，`parameters`指示形参列表

    - 生命周期：登记项并不是在创建时就完善的，而是先绑定后填充

阶段	时机	操作
创建与绑定	遇到变量声明语句	将该符号的固定部分(名称+类型)放入符号表，特有部分缺省
更新与查询	后续代码生成等	不断得到并更新特有部分，填充其缺省部分

2️⃣ 操作：实现其功能的支撑

1. 操作的类型

操作	形式	备注
插入	`bind(name, attributes)`	插入一个新符号及其属性
查找	`lookup(name)`	是最频繁的操作，优先查找当前作用域，然后才是父作用域
进入	`newtab()`	代码进入一个新的作用域(函数)时，创建一个新的子函数符号表
退出	`exit_scope()`	代码离开当前作用域时，销毁当前的子符号表，返回父表

2. 操作的机制
    - 全局栈`symtab`：编译器维护的全局的唯一的栈，存放指向符号表的指针
    - 当前符号表：任何时刻认定位于`symtab`栈顶的符号表，就是当前正在处理的作用域
        - 自动挡：所有的`bind()`和`lookup()`操作都默认针对当前符号表
        - 手动挡：即`bind1(table,...)`和`lookup1(table,...)`，用第一个参数指定操作的符号表

3️⃣ 功能：贯穿了语义分析阶段&中间代码生成阶段

1. 语义分析阶段：
    - 插入操作：遇到声明标识符的语句时，检查是否重复定义了该标识符，如果没有则将其插入表中
    - 查找操作：遇到使用标识符的语句时，检查该名字是否在表中，不在则报错(未声明标识符)
2. 代码生成阶段：
    - 分配多大空间：查询符号表中变量的类型，如遇到`int`则分配4字节
    - 分配在哪里：对于表中局部变量，编译器会计算其偏移地址并写回符号表，供后续机器码使用

属性文法概述

1️⃣ 基本组成：CFG+属性+属性方程(语义规则)

1. CFG：只定义了程序除逻辑以外的现状结构是否合法，是程序的骨架
2. 属性文法：对CFG的扩展

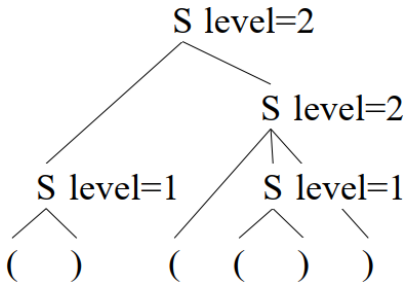
- 对每个文法符号：都附加一个属性，用来承载信息，如可以给`E`附上值`val`的属性
- 对每个产生式：配备零至多个属性方程(语义规则)，定义了属性的计算/传递方式，如E.val=E1.val+T.val`

2 属性与属性方程

1. 属性有关概念：
   - 属性类：比如对于文法符号`S`，其有一个`level`属性，则文法规则中的`S.level`就是一属性类
   - 属性实例：在具体的语法分析树中，某个特定节点(注意一个文法符号可以占用多个结点)拥有的属性
   - 属性值：属性实例的任意取值，如某结点上`S.level=1`
2. 属性方程概念：
   - 产生式：编号为p且$p:X\to X_1,\dots,X_n$，左部编号为$p[0]=X$，右部编号为$p[i]=X_i$
   - 属性：记$p[i].a_j$表示，对产生式p中第i个符号X_i，其第j个属性是a_j
   - 属性方程：X_i的第j个属性s_j，是关于产生式中所有文法符号的所有属性的函数
     - $p[i].a_j=f_{ij}(p[0].a_1,\dots,p[0].a_k,\dots,p[n].a_1,\dots,p[n].a_k)$
3. 综合示例文法：
   - 产生式：属性名为`level`，属性方程中`S/S[0]`表示产生式左部的S，`S[1]/S[2]`依次表示右部的S

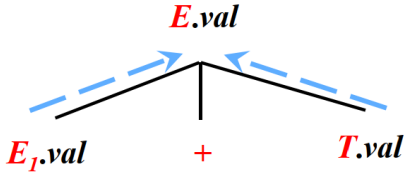
$$
\begin{aligned}
&S \to SS && \{\ S[0].level = \max(S[1].level, S[2].level)\ \}\\
&S \to (S) && \{\ S[0].level = S[1].level + 1\ \}\\
&S \to () && \{\ S.level = 1\ \}
\end{aligned}
$$

   - 语法树：即属性的语法树表示，或者带注释的语法树

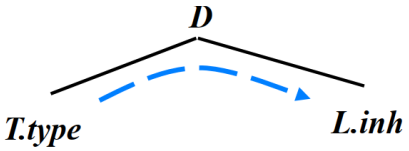


3 属性的类型

1. 综合属性：
   - 含义：在分析树中的非终结符A对应节点，其综合属性只能由其本身或其子节点的属性值定义
   - 示例：对于产生式$E\to E_1+T$，关联语义规则$E.\text{val}=E_1.\text{val}+T.\text{val}$，val就是$E$的综合属性



   - 补充：终结符也有综合属性，但是是由词法分析器直接提供的词法值，与语义规则无关
2. 继承属性：
   - 含义：在分析树中的非终结符A对应节点，其继承属性只能由其本身/父/兄弟节点的属性值定义
   - 示例：对于产生式$D\to TL$，关联语义规则$L.\text{inh}=T.\text{val}$，inh就是$L$的继承属性



- 补充：终结符不具有继承属性，或者说终结符从词法分析器获得的属性被强制归为综合属性

属性求值

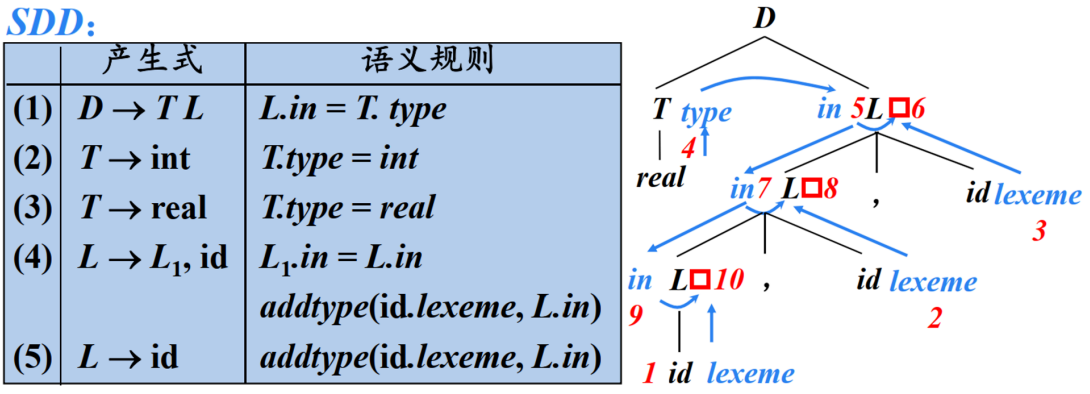
1 属性求值：

1. 含义：为语法树上的每一个节点，计算出它所有属性的值，aka绑定
2. 分类：按求值的时间点差异可分为
    - 静态求值：在编译时完成，编译器在翻译代码时就将值定下来，运行期间不再改变
    - 动态求值：运行完成时完成，如运行时根据用户输入在计算出值，如`cin>>x;`后`x`的值
3. 方法：(下表)

方法	描述	优劣
依赖图拓扑排序	构建依赖图→对图进行拓扑排序(即正确计算顺序)	最根本，但低效
树遍历法	反复深度优先遍历语法树，来计算属性	遍历数过多
语法制导法	在语法分析过程(LL/LR)就进行属性计算	最高效

2 基于依赖图的属性求值

1. 核心：语法树中各属性间可能存在依赖，表现为有向无环图(正确的文法必定无环)
2. 结构：分析树每个结点的每个属性都对应一图节点，若$X.a$依赖于$Y.b$则令有向边$Y.b \to X.a$



- 对于某个结点，可以让综合属性在右，继承属性在左
- 红框表示`addtype`动作，其执行依赖于下面的`id.lexeme`(源于词法分析器)和左边的`L.in`

3. 排序：为所有结点属性标号X_k，如果$X_i \to X_j$则令$i<j$，将所有X_k排序即得到拓扑排序
    - 注意：排序不唯一，例如上述例子中可以是`1,2,3,4,5,6,7,8,9,10`或`4,3,2,1,5,7,6,9,8,10`

3 树遍历法的属性求值

1. 先根遍历与后根遍历法
    - 先根遍历：先处理根节点再往下处理子节点，天然自顶向下，天然契合继承属性的信息流向
    - 后根遍历：先处理子节点再往下处理根节点，天然自底向上，天然契合综合属性的信息流向

2. S-/L-属性文法
    - S-属性文法：只使用综合属性，信息永远纯粹自底向上，用后根遍历处理最高效
    - L-属性文法：可使用综合属性，限制结点的继承属性只依赖其父亲或左边的兄弟，信息从左到右

语法制导的属性求值

1️⃣ 语法制导翻译的概述

1. 含义：语法分析的同时，以属性求值过程为框架，将语法树中符号(尤其是根节点)翻译成中间代码
2. 方法：什么样的属性文法+什么样的语法分析器
    - 自底向上：S-属性文法+LR分析器，在LR分析器执行规约时，同时计算父节点的综合属性
    - 自顶向下：L-属性文法+LL分析器，在LL分析器自顶向下预测一个产生式，同时自左向右继承信息

2️⃣ S-属性文法的计算

1. 核心机制：新增属性栈
    - 新增结构：以标准LR分析器为基础(符号栈+状态栈)，再新增若干属性栈，有多少个属性就对应多少栈
    - 同步逻辑：符号栈每一次压入/弹出时，必须同时在所有属性栈上进行一次相应的压入/弹出
2. 计算过程：以$A\to XYZ$的规约为例
    - 弹出：Z出栈$\to Z.a$出栈$Z.b$出栈(同步逻辑会确保这些属性都恰好在属性栈栈顶)，然后YX依次同操作
    - 计算：利用已弹出的元素，计算语义规则$A.a=f(X.a,Y.a,Z.a)/A.b=f(X.b,Y.b,Z.b)$
    - 入栈：将A压入符号栈，计算出的新属性值$A.a/A.b$压入各自的属性栈
3. 综合示例：括号层次和数量识别文法
    - 文法：(下表)

产生式	l(层次)的语义规则	c(数量)的语义规则
$S'\to X$	左边直接接收右边的	左边直接接收右边的
$X\to XX$	左边去右边最大者	左边为右边二者相加
$X\to(X)$	左边为右边加一	左边为右边加一
$X\to()$	左边直接为基础值1	左边直接为基础值1

    - 运行：(下表)

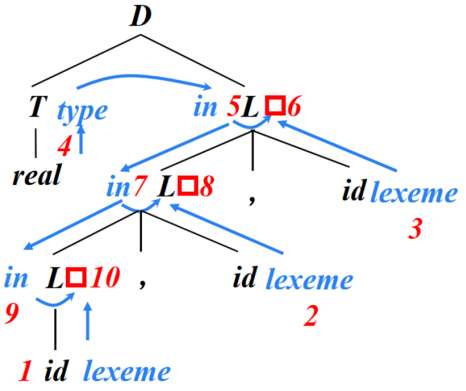
符号栈	level栈	m栈	w#	动作
#	#	#	()(())#	移进
#(	#-	#-	)(())#	移进
#()	#--	#--	(())#	r4
#X	#1	#1	(())#	shift
#X(	#1-	#1-	())#	shift
#X((	#1--	#1--	))#	shift
#X(()	#1---	#1---	)#	r4
#X(X	#1-1	#1-1	)#	shift
#X(X)	#1-1-	#1-1-	#	r3
#XX	#12	#12	#	r2
#X	#2	#3	#	r1
#S	#2	#3	#	acc

- 第一次规约$X\to()$：`(`和`)`并无属性所以出栈了寂寞，但X的`X.l=1`和`X.c=1`皆入栈
- 第二次规约$X\to()$：`(`和`)`并无属性所以出栈了寂寞，但X的`X.l=1`和`X.c=1`皆入栈
- 第三次规约$X\to(X)$：X的俩属性`X.l=1`和`X.c=1`皆出栈，出栈后各自加一重新塞回栈
- 第三次规约$X\to XX$：由于右部有俩X所以属性全部出栈，`l`去最大值/`c`二者相加后塞回栈
- 第四次归约$S'\to X$：就是一个增广文法，走个形式罢了属性值不变

4. 问题&解决：以上方案自然无法处理含继承属性的文法，但可以重写文法和语义规则以曲线救国

SDD：

	产生式	语义规则
(1)	$D \to T\ L$	$L.in = T.type$
(2)	$T \to$ int	$T.type = int$
(3)	$T \to$ real	$T.type = real$
(4)	$L \to L_1$, id	$L_1.in = L.in$
		$addtype$(id.$lexeme$, $L.in$)
(5)	$L \to$ id	$addtype$(id.$lexeme$, $L.in$)



- 该写前：`D -> TV {V.in=T.type}`，直接传递传递类型信息
- 改写后：`D -> TV {for(e in V.in) {bind(e,T.type)}}`，汇总信息并执行最终动作
  - `D`从`T`处收集综合属性`T.type`
  - `D`从`V`处收集包含所有标识符`[e1,e2,...]`名字的列表`V.in`
  - `D`处遍历`V.in`的所有标识符`[e1,e2,...]`，调用`bind`函数将每个标识符名字/类型在符号表登记

中间语言(IR)

1 中间语言概述

1. 是什么：编译器前端和后端的接口，将不同高级语言翻译成中间语言，再将中间语言翻译成汇编

2. 有哪些：线性的(逆波兰表示/三地址码/四元式)，基于图的(抽象语法树/无环有向图)

2️⃣ 三地址指令与代码

1. 核心思想：每个指令只涉及三个地址(变量名/常量/临时变量)，`result = operand1 op operand2`
2. 分类讲解：赋值指令
   - 分类：二元运算`x = y op z`，一元运算`x = uop z`，拷贝`x = z`
   - 示例：将`x = 2*a+(b-3)`翻译成中间代码，构造`t1-t3`的临时变量

```
t1 = b - 3
t2 = 2 * a
t3 = t2 + t1
x  = t3
```

3. 分类讲解：控制流指令
   - 条件分支的三地址码：`IF x rop z THEN l`，满足条件则跳转到标号为`l`的指令去执行
   - 无条件转移的三地址码：`GOTO l`，强行跳转到标号为`l`的指令处
4. 分类讲解：访存指令
   - 读取内存的三地址码：`d = M[r]`，从地址`r`处读取内存中的值，并存入变量`d`
   - 写入内存的三地址码：`M[d] = r`，将变量`r`的值，存入到地址为`d`的内存单元中
5. 分类讲解：过程调用指令
   - 参数传递的三地址码：`PAR tm`，将变量`tm`作为参数传递给即将调用的过程
     - 函数有几个参数就执行多少次`PAR`
   - 过程调用的三地址码：`v = CALL f,m`，调用名为`f`的过程，并告知它有`m`个参数，返回值存入变量`v`中
   - 过程返回的三地址码：`RETURN t`，过程执行完毕，返回一个值`t`
6. 综合示例：只有一个符号表的

```
// 源代码                 // 中间代码
input x;                  // INPUT x
                          // t1 = 0

if 0 < x then             // IF t1 < x THEN l1 ELSE l2
  fact = 1                // LABEL l1
                          // fact = 1
                          // GOTO l3
else
  fact = 2;               // LABEL l2
                          // fact = 2

repeat                    // LABEL l3
  fact = fact * x;        // fact = fact * x
  x = x - 1               // x = x - 1
until x == 0;             // IF x == 0 THEN l4 ELSE l3

print fact                // LABEL l4
```

```
                        // PRINT fact
```

7. 综合示例：有两个符号表

```
// 源代码 (Source Code)            // 中间代码 (Intermediate Code)
// ------------------------------------------------------------------
                                   // @code=[
x = 123 + fact(5,1);
                                   //   t4 = 123
                                   //   t5 = 5
                                   //   t6 = 1

                                   //   -- 准备函数调用 fact(5,1) --
                                   //   PAR t6      // 将第二个参数 t6(值为1) 压入
                                   //   PAR t5      // 将第一个参数 t5(值为5) 压入

                                   //   -- 执行调用 --
                                   //   t7 = CALL fact, 2
                                   //   调用fact函数，告知有2个参数，返回值存入t7

                                   //   -- 计算最终结果 --
                                   //   x = t4 + t7   // 执行 123 + (fact的返回
值)

print x                            //   PRINT x     // 打印x
                                   //   ]

// 源代码 (Source Code)            // 中间代码 (Intermediate Code)
// ------------------------------------------------------------------
int fact(int n; int a;){           // fact@code=[

  if (n==1)                        //   IF n==1 THEN l1 ELSE l2

    return a                       //   LABEL l1:
                                   //   RETURN a              // 返回 a
                                   //   GOTO l3               // 无条件跳出函数

  else                             //   LABEL l2:
    return fact(n-1, n*a);
                                   //   -- 准备递归调用 fact(n-1, n*a) --
                                   //   t1 = n - 1            // 计算第一个参数
                                   //   t2 = n * a            // 计算第二个参数
                                   //   PAR t2                // 传递第二个参数
                                   //   PAR t1                // 传递第一个参数

                                   //   -- 执行递归调用 --
                                   //   t3 = CALL fact, 2

                                   //   -- 返回递归调用的结果 --
                                   //   RETURN t3
};                                 //   LABEL l3:
                                   //   ] (函数结束)

```

3️⃣ 三地址码转四元式

1. 四元式：将三地址码的三个地址+三地址码的运行结果，放在一起构成四个字段的元组
    - `k(op, arg1, arg2, result)`中`k`是编号，其余对应了`result = arg1 op arg2`
2. 综合示例

行号	三地址码	四元式 `(op, arg1, arg2, result)`
100	`INPUT x`	`(INPUT, x, _, _)`
	`t1 = 0`	
101	`IF t1 < x THEN l1`	`(J<, t1, x, 103)`
102	`GOTO l2` (else部分)	`(J, _, _, 105)`
103	`LABEL l1:`	
	`fact = 1`	`(=, 1, _, fact)`
104	`GOTO l3`	`(J, _, _, 106)`
105	`LABEL l2:`	
	`fact = 2`	`(=, 2, _, fact)`
106	`LABEL l3:`	
	`fact = fact * x`	`(*, fact, x, fact)`
107	`x = x - 1`	`(-, x, 1, x)`
108	`IF x == 0 THEN l4`	`(J=, x, 0, 110)`
109	`GOTO l3` (else部分)	`(J, _, _, 106)`
110	`LABEL l4:`	
	`PRINT fact`	`(PRINT, fact, _, _)`

4️⃣ 逆波兰表示法

1. 普通的中缀表示法：如`a + b`
2. 逆波兰表示法：将操作数挪到后面去，即`a b +`

机器代码生成器

1️⃣ 中间代码转机器码概述

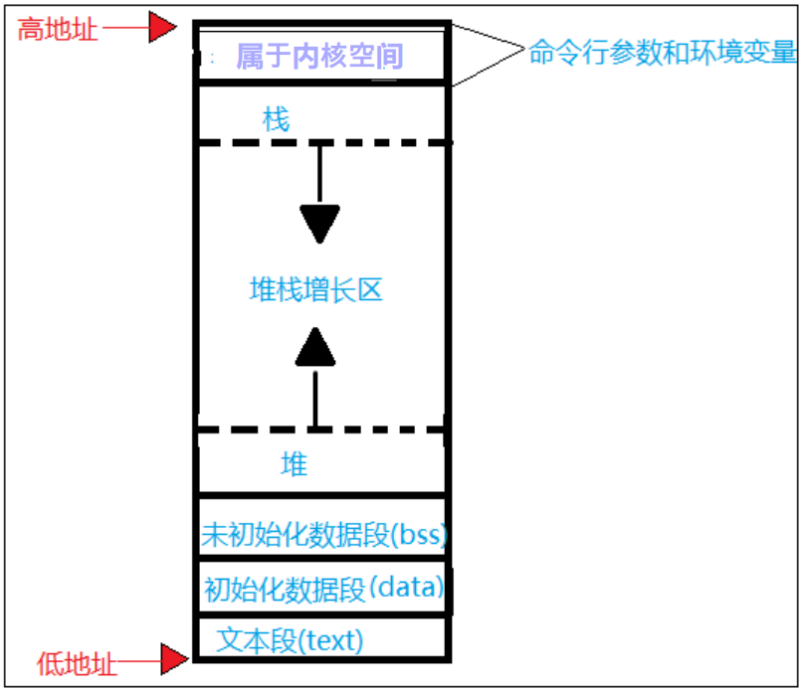
1. 模式库：预定义好的代码模板，指示了什么样的中间代码会匹配并翻译成什么样的机器码
2. 过程：不断消耗输入串(中间代码)的前缀，与模式库对比并翻译，从而构建输出串(机器码)
3. 策略：贪心法(每次都尽可能匹配到剩余串中最长)，线性规划(力图做到匹配的全局最优)

运行时环境

1️⃣ 可执行程序

1. 可执行文件：编译连接好后的一个文件，安静地躺在磁盘里
2. 内存映像：运行可执行文件时，内存中动态的分布，变量值不断变化且有实际分配的堆栈
3. 运行时环境：如何构建内存映像的结构

2️⃣ 内存映像：程序执行时，在内存中构建的独立的地址空间



1. 代码区(文本段)：存放程序编译好的二进制机器码，只读，存储方向自底向上
2. 静态区：存储全局变量/静态变量，可读写，，存储方向自底向上
    - 数据段：存放已显式初始化的全局变量和静态变量
    - BSS段：存放未被初始化/初始化为0的全局变量和静态变量，不用显式地记录变量的值
3. 堆：用于内存动态分配，如`malloc`和`new`时所需空间就从这里割一块，并且是自底向上扩展
4. 栈：用于函数调用(局部环境)，每调用一个函数就会在栈顶创建一个栈帧(存放参数/局部变量/返回地址)
    - 栈增长方向为地址减小方向

3️⃣ 声明宿主：名字在哪里被声明的

1. 对于局部变量：声明宿主就是其所在的函数
2. 对于全局变量：声明宿主就是整个程序(全局作用域)

4️⃣ 栈帧方案

1. 变量`x`：如果`x`在函数`func`中定义，运行时`x`就会被放在`func`的栈帧里，即使`x`是临时变量也一样
2. 函数`foo`：函数本身的代码是静态的放在文本段，但对`foo`调用时系统会创建栈帧，并塞入其局部变量参数等