```
字符流 -- 词法分析 -> 单词流 -- 语法分析 -> 语法分析树 -- 语义分析和 IR 生成 -> IR -- 目标代码生成和优化 -> 目标代码
```

自顶向下语法分析

LL(k) 分析：确定性的**推导**

- 从左 (L) 到右扫描，每次替换最左 (L) 非终结符
- 对文法施加限制，通过向前查看 k 个单词来确定使用的产生式

LL(1) 文法

对于 CFG $G = (V_N, V_T, P, S)$，定义：

First(a)：文法符号串 a 所有可能推导出符号串的首个终结符。

```
X = Vn | Vt | {ϵ} | {v | A -> u in P and v in u.suffix}
for x in X:
    if x in Vt | {ϵ}:
        First[x] = {x}
    else:
        First[x] = {}
while First 在迭代中变化:
    for A -> β in P:
        First[A] |= First[β]
    for Y1Y2...Yk in {v | A -> u in P and v in u.suffix}:
        if ϵ in Y1, ..., Yi-1 and ϵ not in Yi:
            First[Y1Y2...Yk] = union(First[Y1], ..., First[Yi]) - {ϵ}
        elif ϵ in Y1, ..., Yk:
            First[Y1Y2...Yk] = union(First[Y1], ..., First[Yk])
```

Follow(P)：P 后所有可能跟随的终结符，不包括 ϵ.

```
Follow[S] = {'#'}
while Follow 在迭代中变化:
    for A -> αBβ in P:
        Follow[B] = First[β] - {ϵ}
        if ϵ in First[β]:
            Follow[B] |= Follow[A]
```

PS(A->a)：产生式所有可能推导结果的起始终结符。

```
if ϵ in First[a]:
    PS[A->a] = First[a]
else:
    PS[A->a] = (First[a] - {ϵ}) | Follow[A]
```

G 是 LL(1) 文法，当且仅当 $\forall A \in V_N, \forall A \to \alpha, A \to \beta \in P$

$$PS(A \to \alpha) \cap PS(A \to \beta) = \emptyset$$

LL(1) 分析

递归下降法

```
def ParseS():
    ''' 以 S -> AaS | B 为例 '''
    if lookahead in PS[S->AaS]:
        ParseA()
        MatchToken('a')
        ParseS()
    elif lookahead in PS[S->B]:
        ParseB()
    else: raise Exception("syntax error")
```

表驱动法

预测分析表 + 下推栈

```
stack = ['#', S]
while True:
    top = stack.pop()
    if top in Vn:
        assert table[top, lookahead] is not empty
        top.extend(reversed(table[top, lookahead]))
    elif top in Vt:
        assert match_token(top)
    elif top == '#':
        assert match_token('#')
        break
    else: raise Exception()
```

消除左递归

- 许多文法在消除左递归和提取左公因子后可以变换为 LL(1) 文法

左递归 $P \to P_1\alpha_0, P_1 \to P_2\alpha_0, \cdots, P_n \to P\alpha_0$.

n = 0，$P \to P\alpha$ 称直接左递归。

对于无回路，无 epsilon 产生式的文法：

直接左递归消除：

```
P -> Pa | b 变换为：
P -> bR
R -> aR | epsilon
```

间接左递归消除：

```
将文法非终结符排列为 A1, ..., An
for i in range(1, n+1):
    for j in range(1, i):
        设 Aj 全部产生式 Aj -> a1 | ... | ak
        将 Ai -> Aj r 变换为：
        Ai -> a1 r | ... | ak r
    消除 Ai 的直接左递归
```

提取左公因子：比较直观，不细述。

- 不含左递归和左公因子的文法也不一定是 LL(1) 的。

LL 分析的报错机制：有点复杂，好像没考过。

自底向上语法分析

短语：句型的所有可规约串，也是其语法分析树所有子树的果实。

直接短语：句型的所有一步可规约串，是语法分析树中不包含其它子树的子树果实。

句柄：句型的最右直接短语。

- 对无二义文法来说，其句型的分析树是唯一的，句柄也是唯一的。

LR 分析基础

L 表示从左向右扫描，R 表示归约到是右句型的句柄。

记 i 为栈顶状态，a 为输入串当前符号：

```
while True:
    if ACTION[i, a] == Sj:
        push j
    elif ACTION[i, a] == Rj:
        记第 j 条产生式为 A -> β
        pop |β| 项
        记当前栈顶状态为 k，push GOTO[k, A]
    elif ACTION[i, a] == acc
        return
    else: raise Exception()
```

- ACTION 表指出在栈顶状态和当前输入符号下做什么
- GOTO 表指出依 A -> β 规约后转向什么状态
- 要求分析表中不存在**移进-规约冲突**和**规约-规约冲突**。

LR(0)

增广文法和活前缀

对于文法 G，增加产生式 S' -> S，得到增广文法 G'.

- 目的是让开始符号不出现在任何产生式的右部。

活前缀：对于 $S' \Rightarrow_{rm}^{*} \alpha Aw, w \in V_T^*, A \Rightarrow \beta \in V_T$, 即 β 是右句型 $\alpha\beta w$ 的一个句柄，则 $\alpha\beta$ 的所有前缀都是文法 G[S] 的活前缀。

LR(0) Item

一个 LR(0) Item 是右端某一位置有原点的产生式。

LR(0) 状态

LR(0) FSM 的任一状态都是 LR(0) Item 集的闭包。

```
def Closure(I):
    J = I
    while True:
        for A -> a.Bb in J:
            for B -> r in P:
                if B -> .r not in J:
                    J.add(B->.r)
        if 本轮循环中 J 不变:
            return J
```

LR(0) 状态转移

$$\mathrm{GO}(I,x)=\mathrm{CLOSURE}(J)$$
$$J=\{A\to aX.b|A\to a.Xb\in I\}$$

由此可逐步构造出整个 LR(0) FSM.

LR(0) 分析表

1. 若项目 $A\to\alpha.a\beta\in I_k$，且对于 $a\in V_T$，$\mathrm{GO}(I_k,a)=I_j$，则 ACTION[k, a] = Sj.
2. 若项目$A\to\alpha.\in I_k$，则对 $\forall a\in V_T$, 记 $A\to\alpha$ 是 G 的第 j 个产生式，则 ACTION[k, a] = Rj.
3. 若 $S'\to S.\in I_k$, ACTION[k, #] = acc.
4. 若对于 $A\in V_N$, $\mathrm{GO}(I_k,A)=I_j$, 则 GOTO[k, a] = j.

SLR(1)

对 LR(0) 分析表构造算法第二条略作修改：

2. 若项目$A\to\alpha.\in I_k$，则对 $\forall a\in V_T\cap a\in\mathrm{Follow}(A)$, 记 $A\to\alpha$ 是 G 的第 j 个产生式，则 ACTION[k, a] = Rj.

LR(1)

LR(1) Item

在 LR(0) Item 的基础上增加一个终结符，表示产生式右端完整匹配后所允许在余留符号串中的下一个终结符。

LR(1) 状态

```
def Closure(I):
    J = I
    while True:
        for A -> α.Bβ, a in J:
            for B -> γ in P:
                for b in First[βa]:
                    if B -> .γ, b not in J:
                        J.add(B->.γ, b)
        if 本轮循环中 J 不变:
            return J
```

初始状态为 CLOSURE([S'->S, #])

LR(1) 状态转移

$$\mathrm{GO}(I,x)=\mathrm{CLOSURE}(J)$$
$$J=\{A\to aX.b,r|A\to a.Xb,r\in I\}$$

LR(1) 分析表

2. 若项目$A \to \alpha .,b \in I_k$，记 $A \to \alpha$ 是 G 的第 j 个产生式，则 ACTION[k, b] = Rj.

其他部分与 LR(0) 基本一致。

LALR(1)

LR(1) item [A -> α.β, a] 中的 A -> α.β 称为该 item 的芯。

如果 LR(1) FSM 的两个状态包含的芯完全相同，那这两个状态就是同芯的。

合并 LR(1) FSM 所有同芯状态，就得到了 LALR(1) FSM.

- 合并不会产生新的移进-规约冲突，只可能产生规约-规约冲突

LALR(1) 分析表的生成过程与 LR(1) 相同。

- 二义文法在 LR 分析中的应用：在某些情况下显式规定规约/移进的优先级，可以用 LR 分析处理二义文法。
- LR 分析中的错误处理：在分析表中规定错误类型。
- LR(1) 发现错误的速度更快。

符号表

符号表体现了作用域信息。所有作用域共用一个全局符号表，每个作用域都有各自的符号表。

对于程序中某点，该点的当前作用域和所有外层作用域构成了**开作用域**，其它作用域成为**闭作用域**。只有在该点的开作用域中声明的名字才是可访问的。

可以用栈记录开作用域。

语法制导的语义计算

属性文法

语义规则

- 复写规则 $X.a := Y.b$
- 基于语义函数的规则 $b := f(c_1, \cdots, c_k)$ 或 $f(c_1, \cdots, c_k)$

对于产生式 $A \to a$ 的语义规则 $b := f(c_1, \cdots, c_k)$,

- 若 b 是 A 的属性，则 b 是 A 的**综合属性**，自下而上传递
- 若 b 是产生式右部某文法符号 X 的属性，则 b 是 X 的**继承属性**，自上而下传递。

语义计算

- 树遍历方法：构造依赖图 + 拓扑排序
- 语法分析遍的同时，计算属性。
    - S-属性文法：只包含综合属性。

      可在 LR 分析中增加语义栈。
    - L-属性文法：包含综合属性和继承属性，产生式右端文法符号继承属性的计算只能依赖于该符号左边文法符号的属性，只能依赖产生式左端符号的继承属性。

      可用 DFS 后序遍历生成。

翻译模式

与属性文法相比，可以显式地表达动作和属性计算的次序。

限制：S-属性文法或 L-属性文法

自上而下

LL(1) 分析 + 递归下降法，对于非终结符 A，其解析函数 ParseA 以 A 的继承属性为参数，返回 A 的综合属性。

```
def ParseS(Sf):
    '''
    S -> {B.f:=S.f} B {S1.f:=S.f+1} S1 {S.v:=S1.v+B.v}
    S -> epsilon {S.v:=0}
    '''
    if lookahead in ['0', '1']:
        Bf = Sf
        Bv = ParseB(Bf)
        S1f = Sf + 1
        S1v = ParseS(S1f)
        Sv = S1v + Bv
    elif lookahead in ['#']:
        Sv = 0
    else: raise Exception()
    return Sv
```

消除翻译模式文法的**直接左递归**：

```
A -> A1 Y {A.a:=g(A1.a, Y.y)}
A -> X {A.a:=f(X.x)}
变换为：
A -> X {R.i:=f(X.x)} R {A.a:=R.a}
R -> Y {R1.i:=g(R.i, Y.y)} R1 {R.a:=R1.a}
R -> epsilon {R.a:=R.i}
```

自下而上

LR 分析 + 语义栈

- 从翻译模式中**去掉嵌在产生式中间的语义规则集**，引入新的非终结符 N 和产生式 N -> ε.

  若语义规则集中无任何属性：

  ```
  R -> + T {print('+')} R1
  变换为：
  R -> + T M R1
  M -> epsilon {print('+')}
  ```

  若语义规则集中有关联属性：

  ```
  R -> a A {C.i:=f(A.s)} C
  变换为：
  R -> a A {M.i:=A.s} M {C.i:=M.s} C
  M -> epsilon {M.s:=f(M.i)}
  ```

这样使所有除复写规则外的语义规则都出现在产生式的末端。

- 在自下而上规约中，**分析栈中继承属性的访问**总是通过栈中已有文法符号的综合属性间接进行的。

  记 top 为规约前栈顶位置。

```
S -> a A {C.i := A.s} C | b A B {C.i := A.s} C
C -> c {C.s := g(C.i)}
```

  在 C -> c 的规约中，需要的综合属性 A.s 可能在 top-1，也可能在 top-2，为了解决这个问题，引入新非终结符：

```
S -> a A {C.i := A.s} C
S -> b A B {M.s:=A.s} M {C.i:=M.i} C
M -> epsilon {M.i:=M.s}
C -> c {C.s := g(C.i)}
```

  这样 C.i 一定能通过 top-1 得到。

- **继承属性的模拟求值**

  通过增加新非终结符，将继承属性经一定运算后的值存入栈中。

```
S -> a A {Ci:=f(A.s)} C
变换为：
S -> a A {M.i:=A.s} M {C.i:=M.s} C
M -> epsilon {M.S=f(M.i)}
```

- 有时，改变基础文法可以用**综合属性代替继承属性。**

语义检查和 IR 生成

lecture7 详细叙述了一个语法制导的静态类型检查的例子。

– **以下是定义某个简单语言的上下文无关文法（将用于本讲的设计示例）** $G[P]$ **：**

$P \to D\ ;\ S$

$D \to V\ ;\ F$

$V \to V\ ;\ T\ L \mid \varepsilon$

$T \to$ boolean $\mid$ integer $\mid$ real $\mid$ array [num] of T $\mid$ ^T

$L \to L\ ,\ \underline{\text{id}} \mid \underline{\text{id}}$

$S \to \underline{\text{id}} := E \mid$ if E then $S \mid$ if E then S else $S \mid$ while E then S

$\quad \mid S\ ;\ S \mid$ break $\mid$ call $\underline{\text{id}}$ (A)

$E \to$ true $\mid$ false $\mid \underline{\text{literal}} \mid \underline{\text{int}} \mid \underline{\text{real}} \mid \underline{\text{id}} \mid E\ \underline{\text{op}}\ E \mid E\ \underline{\text{rop}}\ E \mid E[E] \mid E^{\wedge}$

$F \to F\ ;\ \underline{\text{id}}\ (\ V\ )\ S \mid \varepsilon$

$A \to A\ ,\ E \mid \varepsilon$

语法制导的 IR 生成

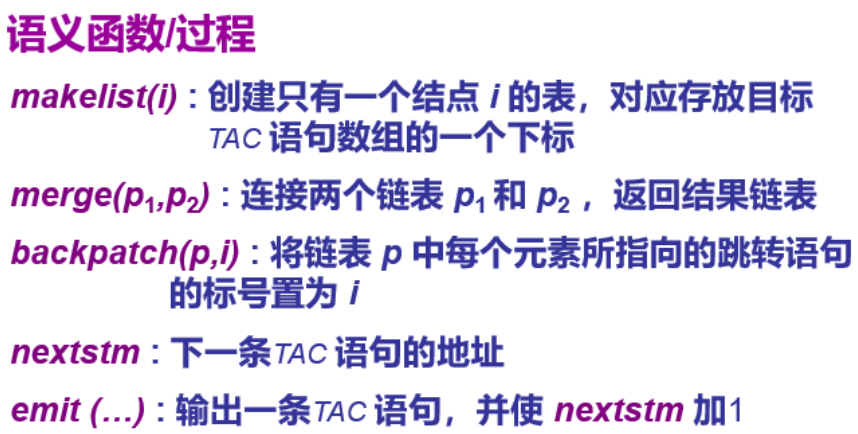
- L-翻译模式：

  布尔表达式的翻译，E.true 和 E.false 分别代表 E 为真和假时对应于程序中的位置。体现短路特性。

- S-翻译模式，拉链与代码回填：

  拉链：一系列跳转语句地址的链表。

  - E.truelist, E.falselist, S.nextlist



这块儿蛮繁琐，需要仔细看ppt.

运行时存储组织

过程活动记录，即栈帧，包含局部变量，实参，临时变量等数据和控制信息。

对于不在当前作用域变量的访问，解决方式有 Display 表和静态链。

Display 表

记录各嵌套层当前过程的活动记录在运行栈上的起始位置.

维护方式：

1. 把整个 Display 表存入活动记录
2. 如果一个处于第 n 层的过程被调用，则只需要在该过程的活动记录中保存 D[n] 先前的值；如果 D[n] 先前没有定义，那么用 “_” 代替。

   这样，只需要维护一个全局的当前 Display 表，大大节省空间。

静态链

在活动记录中保存静态链，指向直接外过程；保存动态链，指向调用本过程的过程。

作用域规则

- 动态作用域规则：外层作用域是运行时的调用者。
- 静态作用域规则：外层作用域是词法意义上的。

```
r = 0.25

def show(): print(r)

def small():
    r = 0.125
    show()

show()
small()
```

Python 是静态作用域规则的，这段代码会输出 `0.25 0.25`; 如果采用动态作用域规则，将输出 `0.25 0.125`.

左值和右值

- 表达式的左值代表存储该表达式值的地址
- 表达式的右值代表该表达式的值

目标代码生成和优化

基本概念

- 基本块
- 流图
- 自然循环

  从流图的首节点触发，到达 n 的任何道路都经过 m，则 m 支配 n，记为 $m\ DOM\ n$.

  节点 n 的所有支配节点组成其支配节点集 $D(n)$.

  如果存在边 n -> d, 且 d DOM n，则 n -> d 是回边，其对应**自然循环**由节点 d，n 和有不经过 d 的通道到达 n 的所有节点组成。

数据流分析

到达-定值流分析

$$OUT[B] = GEN[B] \cup (IN[B] - KILL[B])$$
$$IN[B] = \cup_{b \in P[B]} OUT[b]$$

- GEN [B] 为 B 中定值并可到达 B 出口处的定值点集合;
- KILL [B] 为 B 之外的能够到达 B 入口处、且其定值变量在 B 中重新定值的那些定值点集合;
- OUT [B] 为到达 B 出口处各变量的所有可到达的定值点的集合;
- IN [B] 为到 B 入口处各变量的所有可到达的定值点的集合;

其中 GEN 和 KILL 集合可以直接得出。OUT 和 IN 集合需要迭代计算：

```
for i in range(1, n+1):
    IN[Bi] = {}
    OUT[Bi] = GEN[Bi]
while IN 在迭代中变化:
    for i in range(1, n+1):
        newin = union(OUT[p] for p in Bi.precursor)
        if newin != IN[Bi]:
            IN[Bi] = newin
            OUT[Bi] = GEN[Bi] | (IN[Bi] - KILL[Bi])
```

IN 的变化导致了 OUT 的变化。

活跃变量分析

- 对程序中的某变量 A 和某点 p 而言，如果存在一条从 p 开始的通路，其中引用了 A 在点 p 的值，则称 A 在点 p 是活跃的.

$$LiveIn[B] = LiveUse[B] \cup (LiveOut[B] - Def[B])$$
$$LiveOut[B] = \cup_{b \in S[B]} LiveIn[b]$$

- LiveUse(B) 为 B 中被定值之前要引用变量的集合
- Def(B) 为在B中定值的且定值前未曾在B中引用过的变量集合
- LiveIn(B) 为 B 入口处为活跃的变量的集合
- LiveOut(B) 为 B 的出口处的活跃变量的集合

其中 LiveUse 和 Def 集合可以直接得出。LiveOut 和 LiveIn 集合需要迭代计算：

```
for i in range(1, n+1):
    LiveIn[Bi] = LiveUse[Bi]
    LiveOut[Bi] = {}
while LiveOut 在迭代中变化:
    for i in range(1, n+1):
        newout = union(LiveIn[p] for p in Bi.successor)
        if newout != LiveOut[Bi]:
            LiveOut[Bi] = newout
            LiveIn[Bi] = LiveUse[Bi] | (LiveOut[Bi] - Def[Bi])
```

到达定值流是前向流，活跃变量流是后向流。

UD 链和 DU 链

- 假设在程序中某点 u 引用了变量 A 的值，则把能到达 u 的 A 的所有定值点，称为 A 在点 u 的 UD 链。
  - 如果 u 所在的基本块 B 中，u 之前有 A 的定值点 d，且 d 能到达 u，则 A 在 u 的 UD 链是 {d}
  - 如果 u 所在的基本块 B 中，u 之前没有 A 的定值点，则 A 在 u 的 UD 链是 IN[B]
- 假设在程序中某点 u 定义了变量 A 的值，且另有某点 s 引用了该值，则把所有此类引用点 s 的全体称为 A 在定值点 u 的 DU 链。

基本块内变量的待用信息和活跃信息

待用信息

对于块内第 i 条语句的某个变量 v，若其下次被引用于语句 j 处，则其待用信息为 j；若其之后未被引用，或 v 被重新定值，则其待用信息为 0.

活跃信息

基本块中某个变量 A 在语句 i 中被定值，在 i 之后的语句 j 中要引用 A 值，且从 i 到 j 之间没有其他对 A 的定值点，则在语句 i 到 j 之间 A 是活跃的。

算法：

```
for idx, (A:=B op C) in reversed(enumerate(TAC)):
    将 A 的活跃(待用)信息附加于 idx 上
    设 A 为非活跃(非待用)
    将 B,C 的活跃(待用)信息附加于 idx 上
    设 B,C 的待用信息为 idx，活跃信息为活跃
```

基本块的 DAG 表示

叶节点用变量名或常数标记，内部节点用运算符标记。所有节点都可有一个附加的变量名表。

讨论三类 TAC 的 DAG 构造 x := y op z, x := op y, x := y.

要点：

- 合并已知量：操作数都为常数时，删除新构造的节点
- 删除多余运算：操作数不都为常数时，检查计算结果是否已存在
- 删除无用赋值：赋值语句中，把 x 从 node(x) 的附加变量表中删除，转移到 node(y) 的附加变量表中。

高效使用寄存器

Ershov 数：表达式求值时所需寄存器数目的最小值

```
# 对于表达式树上的节点 node，其 Ershov 数算法如下：
if isleaf(node):
    node.v = 1
elif node.child_cnt == 1:
    node.v = node.child.v
else: # 有两个孩子
    if node.lc.v == node.rc.v:
        node.v = node.lc.v + 1
    else:
        node.v = max(node.lc.v, node.rc.v)
```

图着色物理寄存器分配算法

构造寄存器相干图，每个伪寄存器为一个结点，如果程序中存在某点，一个结点在该点被定义，而另一个节点在紧靠该定值之后的点是活跃的，则在这两个结点间连一条边。

对相干图 k 着色，使相干的伪寄存器不会被分配到同一个物理寄存器上。