翻译器：翻译器是能够完成从一种语言到另一种语言变换的软件。

编译器：一个编译器就是一个程序，它可以阅读以某一种语言（源程序）编写的程序，并把该程序翻译成另一个等价的恶。用另一种语言（目标语言）编写的程序。编译器的重要任务之一是报告它在翻译过程发现的源程序中的错误。

解释器：解释器是另一种常见的语言处理器。他并不通过翻译的方式生成目标程序。从用户的角度，解释器直接利用用户提供的输入执行程序中指定的操作。

源程序

编译器

目标程序

输入

目标程序

输出

源程序

输入

解释器

输出

语言语处理系统：

预处理的程序：创建一个可执行的目标程序还需要一些其他程序。一个源程序可能被分割成若干个模块。并存放于独立的文件中。把源程序聚合在一起的任务有时会由一个被称为预处理器的程序独立完成。预处理器还负责把那些称为宏的缩写形式转化为源语言的语句。

链接器：大型程序经常被分为多个部分编译，因此，可重定位的机器代码有必要和重定位的目标文件以及库文件连接到一起，形成真正在机器上运行的代码。

预处理器

修改后的源程序

目标汇编程序

可重定位机器代码

编译器

汇编器

连接器/加载器

源程序

可执行的目标程序

库文件

可重定位目标文件

编译器的结构：

一个编译器就是一个程序，它可以阅读以某一种语言（源程序）编写的程序，并把该程序翻译成另一个等价的恶。用另一种语言（目标语言）编写的程序。编译器的重要任务之一是报告它在翻译过程发现的源程序中的错误。

词法分析

可执行程序

（词法单元）

（语法树）

语法分析

语义分析

中间代码生成

代码优化

代码生成

（语义分析结果）

（中间表示）

（中间表示）

符号表

* 编译器的前端与后端
  + 前端：只依赖于源语言
  + 后端：依赖于目标机器，一般独立于源语言，与中间语言有关
  + 前端+后端：组合
    - 取一个编译器前端，重写它的后端以产生同一源语言在另一机器上的编译器
    - 把几种不同的语言编译成同一种中间语言，让不同的前端使用同一后端，从而得到一台机器上的几个编译器
    - 编译的几个阶段常用一趟/遍（pass）扫描来实现，一趟/遍扫描包括读一个输入文件和写一个输出文件

编译器各阶段的主要任务：

* 词法分析

逐个读构成源程序的字符，把它们组成词法单元(token)流。

* 语法分析

把词法记号流依照语言的语法结构按层次分组，以形成语法短语

* 语义分析

检查程序的语义正确性，以保证程序各部分能有意义地结合在一起，为后面代码生成阶段收集类型信息。

类型转换

类型检查

* 中间代码生成

中间代码位于高级编程语言和机器语言（目标程序）之间

* 代码优化
* 改进代码，以产生执行较快的机器代码
* 目标程序生成

生成可重定位的**机器代码**或**汇编码**

* 符号表管理

编译器的一项重要工作是记录源程序中使用的标识符，并收集每个标识符的各种属性。

* + 这些属性提供标识符的存储分配、类型和作用域信息。
  + 如果是过程标识符，还有参数的个数和类型、参数传递方式和返回值类型。

符号表是为每个标识符保存一个记录的数据结构，记录的域是标识符的属性。

* + 该数据结构允许我们迅速地找到一个标识符的记录，在此记录中存储和读取数据。
* 出错管理

每个阶段都可能发现源程序的错误。发现错误后，该阶段必须处理此错误，使得编译可以继续进行，以便进一步发现源程序的其他错误。

* + 词法分析：当前被扫描的字符串不能形成语言的词法记号。
  + 语法分析：记号流违反语言的语法规则。
  + 语义分析：编译器试图找出语法正确但对所含的操作来说是无意义的结构，如相加的两个标识符，其一是数组名，另一个是过程名。

词法分析器：

逐个读构成源程序的字符，把它们组成记号(token)流，并完成和用户接口的一些任务

过滤空白和注释

将编译器生成的错误消息与源程序的位置联系起来

词法分析器

语法分析器

符号表

记号(token)

取下一个记号

源程序

* 词法记号：由记号名和属性值构成的二元组，属性值不是必须项，记号名是语法分析的输入符号。
* 模式：一个记号的模式描述属于该记号的词法单元的形式。和一个给定模式匹配的字（字符串）的集合成为该模式的语言。
* 词法单元：是源程序中匹配一个记号模式的字符序列，由词法分析器识别为该记号的一个实例。

串和语言的定义

* 字母表：符号的有限集合

∑ = { 0, 1}，ASCII，Unicode

* 串：符号的有穷序列

0110, ε

* 语言：字母表上的一个串集

{ε, 0, 00, 000, …}, {ε}, ∅

* 句子：属于语言的串

正则表达式：

* + 正则表达式是一种描述模式的重要表示方法
  + 正则表达式表示的简单语言叫做正则集

正则定义：

* 对正则表达式命名，使表示简洁

d1 → r1

d2 → r2

. . .

dn → rn

* 各个 di 的名字都不同
* 每个 ri 都是**∑ ∪**{d1, d2, …, di-1 }上的正则表达式

转化图：

* 状态转换图描绘词法分析器被语法分析器调用时，词法分析器为返回下一个记号所做的动作

不确定的有限自动机：

* 一个数学模型，包括
  + 有限的状态集合***S***
  + 输入符号集合**∑**
  + 转换函数***move* : *S*** × **(** ∑∪**{**ε**} )** → ***P(S*)**
  + 状态 ***S0*** 是唯一的开始状态
  + ***F* ⊆ *S***是接受状态集合

确定的有限自动机：

* 一个数学模型，包括
  + 有限的状态集合***S***
  + 输入符号集合**∑**
  + 转换函数***move* : *S*** ×∑→ ***S*** ,且可以是部分函数
  + 状态 ***S0*** 是唯一的开始状态
  + ***F* ⊆ *S***是接受状态集合

NFA和DFA的区别：

从数学模型上看，除了转换函数不同，其他相同

NFA:转换函数***move* : *S* × ( ∑∪{ε} ) → *P(S*)**

DFA：转换函数***move* : *S* × ∑ → *S*** ,且可以是部分函数

* NFA:
  + 一个符号标记离开同一状态有多条边
* DFA:
  + 一个符号标记离开同一状态只有一条边
  + 任何状态下都没有ε转换

语法分析器的作用：

* 语法分析器读取词法分析器提供的记号流，检查它是否能由源语言的文法产生，输出分析树的某种表示。

语法分析器类型：

词法分析器

语法分析器

符号表

记号

取下个

记号

源程序

分析树

前端的其余部分

中间表示

* 通用语法分析（Cocke-Younger-Kasami算法，Earley算法）：效率低，不能用于编译器产品
* 自顶向下语法分析：从语法分析树的根节点开始向叶子节点构造语法分析树
* 自底向上语法分析：从叶子结点开始，逐渐向根节点方向构造语法分析树

上下文无关语法的定义：

* CFG是一个四元组（*VT , VN , S, P*）
  + *VT* : 终结符的有限集合
  + *VN*: 非终结符的有限集合，且*VN ∩VT* = *Ø*
  + *S* : 开始符号，非终结符中的一个
  + *P* : 产生式的有限集合，产生式形式:*A* →α ，其中*A∈VN ，α∈(VN∪VT )\**

推导：

* 把产生式看成重写规则，把符号串中的非终结符用其产生式右部的串来代替
* 产生式A→γ, 且α、b是文法的任意符号串，则αAb ⇒ αγb称为**直接推导**
* 若α1⇒α2⇒*…*⇒αn，则为**零步**或**多步推导**，记为：α1⇒\*αn
  + 若α1=αn，为**零步推导**
  + 若α1≠αn，则为**至少一步推导**，记为：α1⇒+αn
* 上下文无关语言
  + 由上下文无关文法生成的语言是上下文无关语言(CFL)。
  + 等价的文法
  + 如果两个文法产生同样的语言，则两个文法等价。
* 句型
  + 文法 G 的开始符为 S，S ⇒\* α，α可能含有非终结符，则 α 叫做文法 G 的句型。句子是只含终结符的句型。
* 最左推导：每次直接推导均替换句型中最左边的非终结符

E ⇒lm −E ⇒lm −(E) ⇒lm −(**E** + E) ⇒lm −(**id** + E) ⇒lm  −(id + id)

* 最右推导：每次直接推导均替换句型中最右边的非终结符

E ⇒rm −E ⇒rm −(E) ⇒rm −(E + **E**) ⇒rm −(E + **id**) ⇒rm  −(id + id)

语法分析树的性质：

* 根结点是开始符号
* 叶子结点是终结符或ε
* 内部结点是一个非终结符
* 若A→x1x2…xn，则A是一个非终结符，x1x2…xn 是终结符或非终结符。
* 一个文法生成的**语言**是它的某个分析树生成的串的集合。为给定的符号串找到一棵分析树的过程称为串的**语法分析**。

二义性：

* 一个文法，如果存在某个句子不止一棵分析树，或者说这个句子存在不止一种最左（最右）推导，那么称这个文法是二义的。
* 原因：在产生句子的过程中某些直接推导有多于一种选择。

文法的优缺点：

* 优点
  + 文法给出了精确的，易于理解的语法说明
  + 自动产生高效的分析器
  + 可以给语言定义出层次结构
  + 以文法为基础的语言的实现便于语言的修改
* 缺点
  + 文法只能描述编程语言的大部分语法，不能描述语言中上下文有关的语法特征

分离词法分析器理由：

* 既然正则集是上下文无关文法，为什么用正则式定义词法？
  + 词法规则非常简单，不必用上下文无关文法
  + 对于词法记号，正则式描述简洁且易于理解
  + 从正则式构造出的词法分析器效率高
* 从软件工程角度看，词法分析和语法分析的分离有哪些好处？
  + 简化设计
  + 编译器的效率会改进
  + 编译器的可移植性加强
  + 便于编译器前端的模块划分
* 能否把词法分析并入到语法分析中，直接从字符流进行语法分析？
  + 若把词法分析和语法分析合在一起，则必须将语言的注解和空白的规则反映在文法中，文法将大大复杂
  + 注解和空白由自己来处理的分析器，比注解和空格已由词法分析器删除的分析器要复杂得多

形式语言：

* 文法 G=(VN，VT，P，S)
  + 0型文法：α→β，α,β∈(VN∪VT)\*，|α | ≥ 1；
  + 1型文法：α→β,（S→ε除外）满足 |α| ≤ |β| ；
  + 2型文法：A→β，A∈VN，β∈(VN∪VT)\*；
  + 3型文法：A→a 或 A→aB，A,B∈VN，a∈VT。

|  |  |  |
| --- | --- | --- |
| **文法** | **语言** | **自动机** |
| 短语文法（0型） | 短语结构语言 | 图灵机 |
| 上下文有关文法（1型） | 上下文有关语言 | 线性界线自动机 |
| 上下文无关文法（2型） | 上下文无关语言 | 下推自动机 |
| 正规文法（3型） | 正规集 | 有限自动机 |

程序不同层次的错误：

* + **词法错误**：包括标识符、关键字或运算符拼写错误和没有在字符串文本上正确地加上引号。
  + **语法错误**：包括分号放错地方、花括号“{”，“}”多余或缺失。C语言或Java语言中，一个case语句的外围没有相应的switch语句等。
  + **语义错误**：包括运算符和运算分量之间的类型不匹配。
  + **逻辑错误**：因程序员的错误推理而引起的任何错误。

错误恢复策略：

* **恐慌模式的恢复**
  + 一旦发现错误就不断丢弃输入中的符号，一次丢弃你一个符号，直到找到同步词法单元集合中的某个元素为止
* **短语层次的恢复**
  + 发现错误时在余下的输入上进行局部纠正
* **错误产生式**
  + 通过预测可能遇到的错误，在当前的文法中添加特殊的产生式
* **全局纠正**
  + 在处理一个错误的输入串时通过最少的改动将其转化为语法正确的串

自顶向下语法分析：

* 自顶向下语法分析：
  + 对于任何输入串，从文法开始符号（根结点）出发，自上而下，从左到右地为输入串建立语法分析树。

自顶向下分析可以被看做是寻找输入串的最左推导的过程

自顶向下语法分析无法处理左递归情况

同样也不能处理复杂的回溯技术，回溯导致语义工作推倒重来、难以报告出错的确切位置

非递归下降的预测分析：

* 非递归下降的预测分析器
  + 根据当前的栈顶符号X和输入符号*a*决定分析器的动作，有四种可能：
    - 如果*A*=*a=*$，分析器宣布分析完成成功而停机。
    - 如果*X*=*a*≠$，分析器弹出栈顶符号*X*，并推进输入指针，使之指向下一个符号。
    - 如果*X*是非终结符，程序访问分析表*M*；若*M*[*X*,*a*]是*X*的产生式，例如*M*[*X*,*a*]={*X*→*UVM*}，那么分析器用UVM代替栈顶的*X*，并让U在栈顶。作为输出，在此假定分析器打印出所用的产生式，当然也可以执行其他代码。如果*M*[*X*，*a*]指示错误，则分析器调用错误恢复例程。

**a**

**+**

**b**

**$**

**输入**

**预测分析程序**

**分析表M**

**输出**

**X**

**Y**

**Z**

**$**

**栈**

First：

* FIRST(α)被定义为可从α推导得到的串的首符号的集合，其中α是任意的文法符号串。

**FIRST(*α* ) = {*a* | *α* ⇒\* *a*…, *a* ∈ *VT*}**

特别是，*α* ⇒\* ε时，规定ε ∈ FIRST(*α* )

Follow：

* FOLLOW(*A*)被定义为可能在某些举行中紧跟在 *A* 右边的终结符号的集合。

**FOLLOW(*A*) = {*a* | *S* ⇒\* …*Aa*…，*a*∈*VT*}**

如果 *A* 是某个句型的最右符号，那么$属于FOLLOW(*A*)

LL1（1）：

* + 任何两个产生式*A* →*α | β* 都满足下列条件：
    - FIRST(*α* ) Ո FIRST(*β* ) = ∅
    - 若*β* ⇒\* ε ，那么FIRST(*α*) Ո FOLLOW(*A*) = ∅
  + LL(1)文法有一些明显的性质
    - 没有公共左因子
    - 不是二义的
    - 不含左递归

自顶向上语法分析：

* 一个自底向上语法分析过程对应于为一个输入串构造语法分析树的过程，它从叶子节点开始，逐步向上到达根节点。即把输入串**归约**成文法的开始符号。

归约：

* 若一个子串和某个产生式的右部匹配，则用该产生式的左部符号代替这个子串。

句柄：

* 句型的句柄是和某产生式右部匹配的子串，并且，把它归约成该产生式左部的非终结符代表了最右推导过程的逆过程的一步。

移进归约

* 移进−归约语法分析是自底向上语法分析的一种形式。
* 使用栈来保存文法符号，并用一个输入缓冲区来存放将要进行语法分析的其余符号。
* 移进−归约语法分析的四种动作：
  + 移进：把下一个输入符号移进栈
  + 归约：分析器知道句柄的右端已在栈顶，然后确定句柄的左端在栈中的位置，再决定用什么样的非终结符代替句柄
  + 接受：分析器宣告分析成功
  + 报错：分析器发现语法错误，调用错误恢复例程

移进归约冲突：

* 要想很好地使用移进−归约方式，尚需解决一些问题:
  + 如何决策选择移进还是归约
  + 进行归约时，确定右句型中将要归约的子串
  + 进行归约时，如何确定选择哪一个产生式

LR语法分析器模型：

输入

LR分析程序

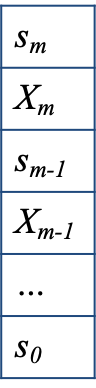
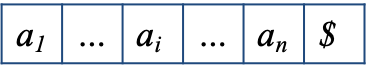
输出

栈

*action*

*goto*

语法分析表



LR语法分析的特点：

* 栈中的文法符号总是形成一个可行前缀
* 分析表的转移函数本质上是识别可行前缀的DFA
* 栈顶的状态符号包含了确定句柄所需要的一切信息
* 是已知的最一般的无回溯的移进−归约方法
* 能分析的文法类是预测分析法能分析的文法类的真超集
* 能及时发现语法错误
* 手工构造分析表的工作量太大

LR文法 *vs*  LL文法：

* *LR(K)*文法:**向前看k个输入符号**能够知道一个产生式的右部所能推导出的所有符号串，进而识别出这个产生式右部的出现。

|  |  |  |
| --- | --- | --- |
|  | LR(1)方 法 | LL(1)方 法 |
| 建立分析树的方式 | 自 下 而 上 | 自 上 而 下 |
| 归约还是推导 | 规 范 归 约 | 最 左 推 导 |
| 决定使用产生式的时机 | 看见产生式右部推出的整个终结符串后，才确定用哪个产生式进行归约 | 看见产生式右部推出的第一个终结符后，便要确定用哪个产生式推导 |

|  |  |  |
| --- | --- | --- |
| 对文法的显式限制 | 对文法没有限制 | 无左递归、无公共左因子 |
| 分析表比较 | 状态×文法符号  分析表大 | 非终结符×终结符  分析表小 |
| 分析栈比较 | 状态栈，通常状态比文法符号包含更多信息 | 文法符号栈 |

|  |  |  |
| --- | --- | --- |
| 确定句柄 | 根据栈顶状态和下一个符号便可以确定句柄和归约所用产生式 | 无句柄概念 |
| 语法错误 | 决不会将出错点后的符号移入分析栈 | 和LR一样，决不会读过出错点而不报错 |

LR(0)项目:

* 在右部的某个地方加点的产生式
* 加点的目的是用来表示分析过程中的状态

LR(0)自动机

* 根据文法LR(0)项目构造识别可行前缀的DFA

LR(0)项 =》 LR(0)自动机

* 文法的所有LR(0)项构成一组规范LR(0)项集，这些规范项集对应LR(0)自动机的状态。

从文法构造的识别可行前缀的DFA的一些特点

* 概念：有效项目

如果*S* ′⇒\**rm αAw* ⇒*rm αβ*1*β*2*w*，那么就说项目*A*→*β*1∙*β*2对可行前缀*αβ*1是有效的

* 一个项目可能对好几个可行前缀都是有效的
* 一个可行前缀可能有多个有效项目
* 一个可行前缀*γ* 的有效项目集就是从这个DFA的初态出发，沿着标记为*γ*的路径到达的那个项目集（状态）

LR(1)项目：

* LR(1)项目：

重新定义项目，让它带上搜索符，成为如下形式：

[*A*→*α* ∙*β*, *a*]

* LR(1)项目[*A*→*α ∙β*, *a*]对可行前缀γ有效
  + 如果存在着推导 *S* ⇒\**rm δAw* ⇒*rm δαβw*，其中：
    - *γ* = *δα*；
    - *a*是*w*的第一个符号，或者*w*是ε且*a*是$

LALR：

* 研究LALR的原因
  + 规范LR分析表的状态数偏多
* LALR特点
  + LALR和SLR的分析表有同样多的状态，比规范LR分析表要小得多
  + LALR的能力介于SLR和规范LR之间
  + LALR的能力在很多情况下已经够用
* LALR分析表构造方法
  + 通过合并规范LR(1)项目集来得到

语法制导翻译：

* 主题：使用上下文无关文法来引导对语言的翻译
* 用途：
* 类型检查和中间代码生成
* 完成特殊任务的语言,如排版

属性文法：

* 通过把属性附加到代表语法结构的文法符号上，将语义信息和程序设计语言的语法结构联系起来，属性的值是用与文法产生式相联系的语义规则来计算的
* 语法制导定义和语法制导翻译：
  + - 语法制导定义SDD：文法产生式和语义规则分开
      * 说明关于语言翻译的高层次规格，隐藏了许多具体实现细节，使用户不必显式地说明发生的顺序
      * 易读、适合作为对翻译的规约（描述）
    - 语法制导翻译方案SDT：文法产生式和语义规则交错
      * 把语义规则用{}括起来，插入到规则右部的合适位置上，指明了语义规则的计算顺序，以便说明某些实现细节
      * 高效、适合用于翻译的实现

翻译过程：输入字符串=〉语法分析树=〉依赖图=〉语义规则计算顺序

* + - 对输入符号(记号/终结符)串进行语法分析，构建语法分析树
    - 根据需要遍历语法分析树，得到描述节点属性间依赖关系的依赖图
      * 当实现一遍编译程序时，可在分析期间计算语义规则而不明显地构造语法分析树和依赖图
    - 由依赖图得到语义规则的计算顺序
      * 计算语义规则：生成代码、在符号表中保存信息、发出错误信息或完成其他活动
    - 按上述计算顺序在语义分析树节点处进行语义规则计算，得到翻译结果

L属性翻译方案和S属性翻译方案：

* L属性翻译方案（L代表从左到右）：包含了所有不必显式构造语法分析树即可完成的翻译方案，即可以在语法分析过程中完成的翻译方案
* S属性翻译方案（S代表综合）：可以很容易和自底向上语法分析（如LR语法分析）过程联系起来的L属性翻译方案

SDD语法制导定义：

* SDD：是上下文无关文法和属性及规则结合
* 属性：和文法符号相关联
  + 每个文法符号都有一个相关的属性集，属性可以代表任何对象：字符串、数字、类型、内存单元或其他对象。
  + 与这些属性相关的信息，即属性值可以在语法分析过程中计算和传递。属性加工的过程即语义的处理过程。
* 规则：和产生式相关联

综合属性和继承属性：

* 每个文法产生式A → α有一组形式为 b = f(c1, c2, …, ck ) 的语义规则，其中b和c1, c2, …, ck 是该产生式文法符号的属性，f 是语义规则的函数
* 综合属性：如果b是A的属性，c1 , c2 , …, ck 是产生式右部文法符号的属性或A的其它属性
* 继承属性：如果b是右部某文法符号X的属性

语法制导的定义可以使用的两种属性。一刻语法分析树结点上的综合属性根据该结点的子节点的属性计算得到。一个结点上的继承属性根据它的父结点和/或兄弟结点的属性计算得到。

SDT语法制导翻译：

* 将**程序片段**附加到一个文法的各个产生式上的表示
* 被嵌入到产生式体中的程序片段成为**语义动作**（semantic action）。语义动作用花括号括起来。

后缀表示：

* + 如果E是一个变量或常量，则E的后缀是本身
  + 如果E是一个形如E1 op E2的表达式，op是二目运算符，那么E的后缀表示是：E1' E2' op，这里E1'和E2'分别是E1和E2的后缀表示
  + 如果E是一个形如（E1）的表达式，则E的后缀表示就是E1的后缀表示

注释分析树:结点的属性值都标注出来的分析树

语义规则的计算方法

* 分析树方法：
  + 刚才介绍的方法，动态确定计算次序，效率低
  + 概念上的一般方法
* 基于规则的方法：
  + （编译器实现者）静态确定（编译器设计者提供的）语义规则的计算次序
  + 适用于手工构造的方法
* 忽略规则的方法：
  + （编译器实现者）事先确定属性的计算策略（如边分析边计算）,（编译器设计者提供的）语义规则必须符合所选分析方法的限制
  + 适用于自动生成的方法

属性依赖图：

* 给定一棵语法分析树和一个SDD，我们在各个语法分析树结点所关联的属性实例之间画上边，以指明位于边的头部的属性值要根据位于边的尾部的属性值计算得到

给定一棵语法分析树和一个SDD，我们在各个语法分析树结点所关联的属性实例之间画上边，以指明位于边的头部的属性值要根据位于边的尾部的属性值计算得到。

S属性的定义：

* 如果一个SDD的每个属性都是综合属性，它就是*S*属性的。

语法树

* 语法树是分析树的浓缩表示：算符和关键字是作为内部结点
* 语法制导翻译可以基于分析树，也可以基于语法树

*L*属性定义

* 如果每个产生式*A*→*X*1…*X*j-1*X*j…*Xn*的每条语义规则计算的属性是*A*的综合属性；或者是*Xj* 的继承属性，但它仅依赖：

该产生式中*Xj*左边符号*X*1, *X*2, …, *Xj*-1的属性；

*A*的继承属性

*S*属性定义属于*L*属性定义

静态检查：

* 静态检查
  + 类型检查：保证运算符被应用到兼容的运算分量
  + 语法分析之后进行的所有语法检查，例如C语言中的一条break指令位于一个for/while/switch语句之内

中间表示

* + 抽象语法树
    - 高层：源语言层次化结构
  + 三地址码
    - 可高可低（通过选择不同的运算符）
  + 高级语言，如C语言
    - C语言灵活通用、可编译成高效机器代码

**高层中间表示形式**

**低层中间表示形式**

**源程序**

**目标**

**代码**

**…**

DAG：

* 作用：识别表达式中的公共子表达式
* 比较：DAG vs 抽象语法树

DAG中，代表公共子表达式的节点具有多个父节点

在抽象语法树中，公共子表达式由重复的子树表示，只有一个父节点

类型应用：

* 类型检查：利用一组逻辑规则来推理一个程序在运行时刻的行为，即保证运算分量的类型和运算符的预期类型相匹配
* 翻译时的应用：根据一个名字的类型，编译器可以确定这个名字在运行时刻需要多大的存储空间、插入显式的类型转换、选择正确版本的算术运算符、计算一个数值引用所指示的地址等等方面。

类型表达式：

* + 基本类型是类型表达式
    - 布尔型、字符型、整型、实数型、void（没有值）
    - 子界类型、枚举类型（PASCAL）
  + 类型名是类型表达式（可以为类型表达式命名）
  + 作用于类型表达式的类型构造算子是类型表达式，包括：
    - 数组：类型构造算子array作用于一个数字和一个类型表达式
    - 记录：类型构造算子record作用于字段名和相应的类型表达式
    - 指针：类型构造算子&作用于一个类型表达式
    - 函数：使用类型构造算子-〉构造得到函数类型的类型表达式
    - 笛卡儿积：如果s和t是类型表达式，那么sxt是类型表达式
    - 类型表达式可以包含其值为类型表达式的变量

类型检查：静态vs动态

* + 静态类型检查：由编译器完成
    - 健全的类型系统能静态地确定程序在运行时不发生错误，因此不需要动态检查。
    - 如果某一语言能保证它所接受的程序不会在运行时发生类型错误，则称该语言是强类型语言
  + 动态类型检查：目标程序运行时完成
    - 有些检查只能动态完成

类型等价：

* + 当用图来表示类型表达式时，两个类型之间结构等价，当且仅当以下某个条件为真：
    - 是相同的基本类型
    - 是将相同的类型构造算子应用于结构等价的类型而构造得到
    - 一个类型是另一个类型表达式的名字
  + 如果类型名仅代表自身，则上述前两个条件定义了类型表达式的**名等价**关系。

三地址代码实现形式：

* + 静态单赋值形式：有利于某些代码优化（了解一下即可）
    - 所有赋值针对不同名字变量(一个变量在不同地方使用不同名字)；Ψ函数将不同名字的相同变量合并起来
  + 表示方法比较：间址的使用
    - 使用四元式表示，定义或使用临时变量的三地址语句可以通过符号表直接访问临时变量的地址
    - 使用四元式表示，符号表在值的计算及使用之间提供了一次额外间址，有利于优化。
      * 使用三元式表示，移动一条定义临时值的语句需要我们改变在arg1及arg2数组中所有对该语句的引用，难以优化。
      * 使用间接三元式表示，可以通过statement列表的重新排序来移动语句。

1. 简述编译程序的工作过程。（10）

编译程序的工作过程，是指从输入源程序开始到输出目标程序为止的整个过程，是非常复杂的，就其过程而言，一般可以划分为五个工作阶段：①词法分析，对构成源程序的字符串进行扫描和分解，识别出一个个的单词；②语法分析，根据语言的语法规则，把单词符号串分解成各类语法单位；③语义分析与中间代码产生，即对各类语法单位，分析其汉一并进行初步翻译；④代码优化，以期产生更高效的代码；⑤目标代码生成，把中间代码变换成特定机器上的低级语言指令形式。

1．词法分析

词法分析的主要任务是从左向右扫描每行源程序的符号，按照词法规则

从构成源程序的字符串中识别出一个个具有独立意义的最小语法单位，

并转换成统一的内部表示(token)，送给语法分析程序。

2．LL(1)文法

若文法的任何两个产生式*A* → *α | β*都满足下面两个条件：

（1）FIRST(*α* ) ∩ FIRST(*β* ) = φ；

（2）若*β* ⇒\* ε ，那么FIRST(*α* ) ∩ FOLLOW( *A* ) = φ。

我们把满足这两个条件的文法叫做**LL(1)文法**，其中的第一个L代表从左

向右扫描输入，第二个L表示产生最左推导，1代表在决定分析器的每步

动作时向前看一个输入符号。除了没有公共左因子外，LL(1)文法还有一

些明显的性质，它不是二义的，也不含左递归。

3．语法树

句子的树结构表示法称为语法树(语法分析树或语法推导树)。

给定文法G=(VN，VT，P，S)，对于G的任何句型都能构造与之关联的

语法树。这棵树具有下列特征：

(1)根节点的标记是开始符号S。

(2)每个节点的标记都是V中的一个符号。

(3)若一棵子树的根节点为A，且其所有直接子孙的标记从左向右的排列

次序为A1A2…AR，那么A→A1A2…AR一定是P中的一条产生式。

(4)若一标记为A的节点至少有一个除它以外的子孙，则AVN。

(5)若树的所有叶节点上的标记从左到右排列为字符串w，则w是文法G

的句型；若w中仅含终结符号，则w为文法G所产生的句子。

4．LR(0)分析器

所谓LR(0)分析，是指从左至右扫描和自底向上的语法分析，且在分析的

每一步，只须根据分析栈当前已移进和归约出的全部文法符号，并至多再

向前查看0个输入符号，就能确定相对于某一产生式左部符号的句柄是否

已在分析栈的顶部形成，从而也就可以确定当前所应采取的分析动作 (是

移进还是按某一产生式进行归约等)。

5．语言和文法

文法就是语言结构的定义和描述，是有穷非空的产生式集合。

文法G定义为四元组的形式：

G=(VN，VT，P，S)

其中：VN 是非空有穷集合，称为非终结符号集合；VT 是非空有穷集合，

称为终结符号集合；P是产生式的集合(非空)；S是开始符号(或识别符号)。

这里，VN∩VT=∅，SVN。V=VN∪VT，称为文法G的字母表，它是出现

文法产生式中的一切符号的集合。

文法G所描述的语言用L(G)表示，它由文法G所产生的全部句子组成，即

L(G)={x| S⇒\*x，其中S为文法开始符号，且 }

简单的说，文法描述的语言是该文法一切句子的集合。

四、简答题(共4小题，每小题5分，共20分)

1．编译程序和高级语言有什么区别?

用汇编语言或高级语言编写的程序，必须先送入计算机，经过转换成用机器

语言表示的目标程序（这个过程即编译），才能由计算机执行。执行转换过程

的程序叫编译程序。汇编程序是指没有编译过的汇编语言源文件。编译程序转

换过的叫目标程序，也就是机器语言。

编译程序的工作情况有三种：汇编型、解释型和编译型。汇编型编译程序用来

将汇编语言编写的程序，按照一一对应的关系，转换成用机器语言表示的程序。

解释型编译程序将高级语言程序的一个语句，先解释成为一组机器语言的指令，

然后立即执行，执行完了，取下一组语句解释和执行，如此继续到完成一个程序

止。用解释型编译程序，执行速度很慢，但可以进行人和计算机的"对话"，随时

可以修改高级语言的程序。BASIC语言就是解释型高级语言。编译型编译程序将

级语言编写的程序，一次就会部翻译成机器语言表示的程序，而且过程进行很快，

在过程中，不能进行人机对话修改。FORTRAN语言就是编译型高级语言。

2．编译程序的工作分为那几个阶段?

词法分析、语法分析和语义分析是对源程序进行的分析(称为编译程序的前端)，

而中间代码生成、代码优化和代码生成三个阶段合称为对源程序进行综合(称为

编译程序的后端)，它们从源程序的中间表示建立起和源程序等价的目标程序。

3．简述自下而上的分析方法。

所谓自下而上分析法就是从输入串开始，逐步进行“归约”，直至归约到文法的

开始符号；或者说从语法树的末端开始，步步向上“归约”，直到根节点。

4．简述代码优化的目的和意义。

代码优化是尽量生成“好”的代码的编译阶段。也就是要对程序代码进行

一种等价变换，在保证变换前后代码执行结果相同的前提下，尽量使目

标程序运行时所需要的时间短，同时所占用的存储空间少。

1．什么是S-属性文法？什么是L-属性文法？它们之间有什么关系？

S-属性文法是只含有综合属性的属性文法。 （2分）

L-属性文法要求对于每个产生式A🡪X1X2…Xn，其每个语义规则中的每个属性或者是综合属性，或者是Xj的一个继承属性，且该属性仅依赖于：

1. 产生式Xj的左边符号X1,X2…Xj-1的属性；
2. A的继承属性。 （2分）

S-属性文法是L-属性文法的特例。 （2分）