程序设计语言原理

Principle of Programming Languages

裘宗燕

北京大学数学学院

2012.2～2012.6

2. 程序设计语言的定义

- 语言的基本元素
- 词法和语法
- 语法的形式描述
- 朴素的语义描述
- 操作语义
- 前后条件和 Hoare 公理
- 指称语义

语言的字符集

大多数语言采用文本表示形式，其中的程序就是字符的序列（一维形式）

完全可以采用其他形式定义语言，如以图形形式描述程序的语言，特定二维形式的程序的语言。下面讨论的许多情况类似，不专门讨论

为了定义程序的形式，一个语言必须选定一个基本的字符集合

字符集：允许出现在语言的程序里出现的字符的全体

一些采用标准的编码字符集或其子集，一些语言不定义具体字符集

有些语言区分了可用在语言的主要部分的字符，与可作为数据或语言之外的描述的字符（可用于字符字面量、字符串字面量，或出现在注释里）

实例：

- **Java采用Unicode字符集，ASCII之外的字符可用于注释、标识符、字符和字符串字面量（用于标识符可能并不好）**
- **C未规定字符集，要求基本字符集包含大小写字母、数字和29个特殊字符，其他字符可用在注释、字符和字符串常量里，效果由具体实现解释**

词法元素

一个程序，可以看着是给定字符集上的一个字符序列

- 是最低级（最简单）的看法
- 没有考虑语言本身的要求和程序中的层次结构

程序语言中最低级的形式单元是词法元素

- 词法元素就是程序语言里的“单词”

在字符序列观点向上一层，是把一个程序看着一些词法元素的序列

- 由程序的字符序列得到词法元素序列的过程就是词法分析

下面介绍各种词法元素：

标识符：文字形式的词法对象，用于表示

- 语言里的关键字
- 程序对象的名字

词法元素

标识符有规定的构造规则，最常见的规则是字母开头的字母数字序列

- 有些语言允许更宽松的标识符形式，如在 Lisp 里 !go+3 是合法标识符

标识符是程序里的名字，作为对象的标识

- 为各种用户定义的对象命名（如变量、常量、函数、过程、类型等）
- 命名对象可以在程序里的一定区域内通过名字使用

特殊的名字

关键字：语言规定了特殊意义的标识符，如 C 中的 if，while，for

保留字：语言中规定了特殊意义，而且不允许程序员用于其他用途的标识符。C 语言的关键字都是保留字，Fortran 没有保留字

显然，在程序里将关键字另作他用不是聪明的做法

关键字和保留字通常用于表示各种语言结构（这种关键字本身没意义，只是作为结构的标志），也可能被给定了特殊的意义

词法元素

运算符：有预定义意义的特殊字符或特殊字符序列（可能有标识符）

运算符的语义就是语言定义的运算（操作），如 Java 的 new

常规语言用运算符表示各种算术、逻辑运算

例： C定义了很大的一组运算符

空白符：通常包含空格、制表符和换行字符。在不同语言里有不同作用

例如：

- Fortran里的空白没有作用（除了出现在字符串里的情况）
- 大多数语言里的空白只起分隔作用，没有语义
- Snobol 中空白作为串连接运算符，Mathematica 作为乘法

分隔符：用于分隔程序里的不同词法元素的特殊符号或标识符

空格，换行和制表符等，通常作为语法元素的分隔符

词法元素

括号是成对出现的分隔符，用于标记语法单位（程序结构）的开始和结束

- 常见的有分程序括号 **begin … end**，或者 **{ … }**
- 多数括号只用于界定一个范围，并无实际的语义

  这种括号完全可以用其他方式代替

  例如 Python 用缩格来表示一个语法结构的范围
- 有的括号有特殊意义，如表达式里的 **(…)** 通常表示运算的优先结合

注释：程序里不表示任何语义的字符序列，用于提供有关程序的辅助信息，供人阅读（或供特殊处理程序使用）。常把一个注释看作一个空格

注释的形式多种多样：

- **Fortran** 采用独立的注释行，用特殊的行标志
- **C** 和 **Pascal** 采用一对注释括号括起的任意字符序列（历史悠久）
- **Ada** 和 **C++** 采用特殊符号开始直至行尾的注释形式（现在很常见）

词法元素

噪声词（**Noise word**）：无语义的单词，可能有助于程序的阅读

例如，有些语言里 **if** 之后的 **then** 可选，它就是噪声词

字面量（文字量，**literal**）：直接写出的特定类型的数据

常见的是各种数值字面量，字符和/或字符串字面量

字面量的意义就是与之对应的数据值

例如，在 C 程序里，字面量 **020** 表示 **int** 类型的整数 **16**；字面量 **0x28L** 表示 **long** 类型的整数 **40**

注意：字面量不仅有值，还有类型。C 程序里的字面量 **235** 和 **235.** 是不同的字面量，分别为 **int** 和 **double** 类型

一些语言允许直接写出数组/结构的值，如 **Ada** 和 **C99**

这种值称为“聚集值”，描述的是具有结构的复合对象的值

ANSI C 只在变量初始化处有聚集值描述形式，其他地方不能用

词法和词法分析

构成程序的基本词法元素包括标识符、运算符、字面量、注释等

复杂的词（标识符、各种字面量）需要明确定义的构词法，即词法

处理源程序的第一步是词法分析：

- 编译器处理表示源程序的字符序列，根据词法规则做词法分析，将源程序切分为符合词法的段，识别出源程序的有意义单词（token）
- 词法分析得到一个（表达被分析的源程序的）单词流，流中各单词标明了词法类别（是“标识符”，“整数”，“加号”，“左括号”，等等）
- 词法分析中抛弃所有无保留价值的分隔符（如空白符）和噪声词

例：对 int main (void) { return 0; } 做词法分析，得到的单词序列是：

"int" "main" "(" "void" ")" "{" "return" "0" ";" "}" 共10个单词

类别：标识符，左/右圆括号，左/右花括号，整数，分号

词法分析

词法分析通常采用最长可能原则，确定可能成为单词的最长字符串。例

`staticint`是一个标识符，而不是关键字 `static` 和 `int`

`x+++y` （C语言）的分析结果是：`x`，`++`，`+`，`y`；而不是`x`，`+`，`++`，`y`

`x+++++y` 是 `x`，`++`，`++`，`+`，`y`（语法错，`++` 的运算对象非左值）

早期的 Fortran 中存在复杂的词法问题。例如：

```
DO 10 I = 1.5
```

的意思类似于 C 语言的 `DO10I = 1.5`

而 `DO 10 I = 1,5`

是枚举循环头部，类似于C 的 `for (I = 1; I < 5; ++I) ...`

后来的语言都避免了这类问题，保证单词很容易识别（能局部识别）

人们已经开发了许多帮助构造词法分析器的自动工具，如lex/flex等

有格式语言和自由格式语言

从程序的格式看，文本形式的语言分为两类：

- 有格式的语言。语言对程序的格式有特定要求，也就是说，程序中的书写格式有特定的语义价值
- 自由格式语言（无格式语言）。语言对程序的格式无任何要求。把程序简单看着字符序列，所有格式字符（换行符和制表符）都看作空白

早期出现了许多有格式语言

Fortran 要求一行不超过 **72** 个字符（**73-80**列为可缺省的行序号），前 **5** 列为标号区，第 **6** 列用于写续行符。一个语句只能写在一行里，跨行时随后的行必须标明是续行

新近出现的 **Python** 语言也是有格式语言

其中用退格和对齐表示程序的嵌套结构，如一个循环体里的语句序列

人们在程序语言是否应该有规定格式方面有争论，这不是大问题

语法

语法规定位于词法层次之上的程序结构。例如：

- 表达式的合法形式
- 基本语句的形式，组合语句的构成方式
- 更上层的结构，如子程序、模块等的构成形式

语法用于确定一个输入序列是否合法的程序。但什么是“合法”？

程序存在多个不同层次的合法性问题：

1. 局部结构　　例：**C**程序里的 **if** 之后是不是左括号，括号是否配对

2. 上下文关系　　例：变量使用前是否有定义，使用是否符合类型的要求

3. 深层问题　　例：使用变量的值之前，变量是否已经初始化

语言的语法定义通常只描述 **1**（程序形式中的上下文无关部分）

编译程序通常检查条件 **1** 和 **2**，有人称 **2** 为“静态语义”

语法的形式定义

语法定义需要严格地定义程序的复杂结构。在定义 Algol 60 语言时，Backus 提出了一种形式化的语法定义方式，称为 Bachus 范式，BNF

- 一般用类似 BNF 的描述形式严格定义程序的上下文无关结构
- 有人研究过用功能更强的技术描述语法（如 W-文法，用于 Algol 68）

BNF（基本 BNF ）提供了两个元符号（语法符号）：

- ::= 或者 → 表示“定义为”
- | 表示“或者”

语法描述里的其他符号都是终结符或者非终结符：

- 终结符是被定义的语言里的符号，最终可以出现在程序里
- 非终结符是为了写语言的语法而引入的辅助符号

一个非终结符表示语言中的一个语法类

例如，我们可能用 <stmt> 表示语句，用 <expr> 表示表达式

语法定义：BNF

一个语言的 BNF 语法定义由一组产生式组成，产生式的形式是：

左部 ::= 右部

- 左部：总是一个非终结符（上下文无关）
- 右部：用 | 分隔的一个或多个终结符和非终结符的序列

例子：

$$
\begin{array}{lcl}
\langle digit \rangle & ::= & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
\langle digits \rangle & ::= & \langle digit \rangle \mid \langle digit \rangle \langle digits \rangle \\
\langle integer \rangle & ::= & \langle digits \rangle \mid + \langle digits \rangle \mid - \langle digits \rangle \\
 & & \\
\langle cond\_stmt \rangle & ::= & \texttt{if } \langle bool\_expr \rangle \texttt{ then } \langle stmt \rangle \mid \\
 & & \texttt{if } \langle bool\_expr \rangle \texttt{ then } \langle stmt \rangle \texttt{ else } \langle stmt \rangle
\end{array}
$$

这是最原始的 BNF 形式，用 <...> 表示非终结符，《Algol 60修订报告》采用这种形式。后来人们常用它的一些变形形式（见下）

需要解决的小问题：语言里的 | 等怎么表示

语法定义：BNF 的变形

一个完整的小语言的语法（取自PLP）：

$$
\begin{array}{lcl}
program & ::= & stmt\_list\ \texttt{\$\$} \\
stmt\_list & ::= & stmt\ stmt\_list \mid \epsilon \\
stmt & ::= & id\ \texttt{:=}\ expr \mid \texttt{read}\ id \mid \texttt{write}\ expr \\
expr & ::= & term\ term\_tail \\
term\_tail & ::= & add\_op\ term\ term\_tail \mid \epsilon \\
term & ::= & factor\ factor\_tail \\
factor\_tail & ::= & mult\_op\ factor\ factor\_tail \mid \epsilon \\
factor & ::= & \texttt{(}\ expr\ \texttt{)} \mid id \mid literal \\
add\_op & ::= & \texttt{+} \mid \texttt{-} \\
mult\_op & ::= & \texttt{*} \mid \texttt{/}
\end{array}
$$

其中正体（打字机体）表示终结符，斜体表示非终结符，ε 表示空串

语法类 ***program*** 表示完整的程序

语法定义：BNF的变形

语言手册里常见的一种形式（C 语言/C++ 语言都采用）

enum-name:
    identifier
enum-specifier:
    enum ***identifier***$_{opt}$ { ***enumrator-list***$_{opt}$ }
enumerator-list:
    enumrator-definition
    enumratoe-list , ***enumerator-definition***
enumerator-definition:
    enumerator
    enumerator = ***constant-expression***
enumerator:
    identifier

斜体表示非终结符，打字机体是终结符。冒号换行后退格表示“定义为”，并列放置表示“或者”，下标 **opt** 表示可选

语法定义：扩充的BNF

用基本 BNF 描述语法常需引进辅助的非终结符，产生式较多。为描述方便，人们提出了一些扩充形式，称为 EBNF（没有标准），描述能力不变

增加元符号：

- [...]　　表示其中的内容可选（出现 0 次或者一次）
- {...}*　　表示其中内容可以出现 0 次或者多次
- {...}+　　表示其中内容可以出现 1 次或者多次
- 用圆括号表示分组范围

前面例子，现在可重新写为：

$$\langle digit \rangle \quad ::= \quad 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

$$\langle integer \rangle \quad ::= \quad [+ \mid -]\{\langle digit \rangle\}^{+}$$

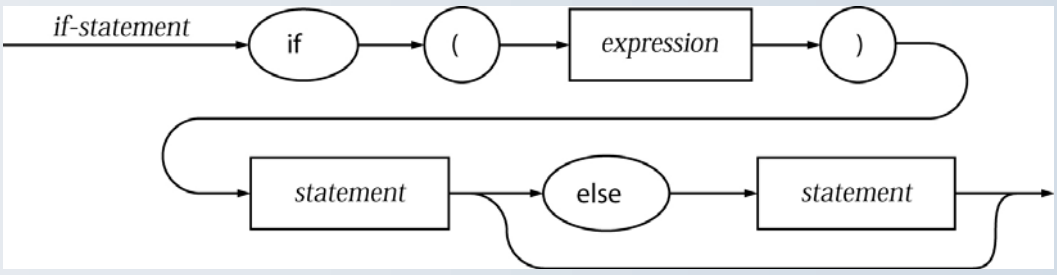
$$\langle cond\_stmt \rangle \quad ::= \quad \texttt{if} \ \langle bool\_expr \rangle \ \texttt{then} \ \langle stmt \rangle \ [\texttt{else} \ \langle stmt \rangle]$$

语法定义：语法图

另一种常见语法描述方式是语法图

- 圆（椭圆）表示终结符，矩形表示非终结符
- 连线形成的通路表示语言结构的合法形式
- 一个语法图是一个非终结符表示的语法类的形式定义
- 一个语言的语法描述是一组语法图

例：描述 C 语言 if 语句的形式的语法图



语法定义

计算机工作者应该熟练掌握形式化的语法描述技术，如 BNF 或扩展的 BNF。学习、工作和研究中都经常需要

- 查阅语言手册，阅读专业书籍
- 工作和学习中的交流。开发软件，写报告和手册，写论文

需要学会：

- 阅读理解用 BNF 写出的语法形式定义
- 用严格的方式思考和描述被处理的信息形式
  - 文本形式的信息可以直接用 BNF 等描述
  - 其他形式的信息可以参考 BNF 形式，设计合适的描述形式

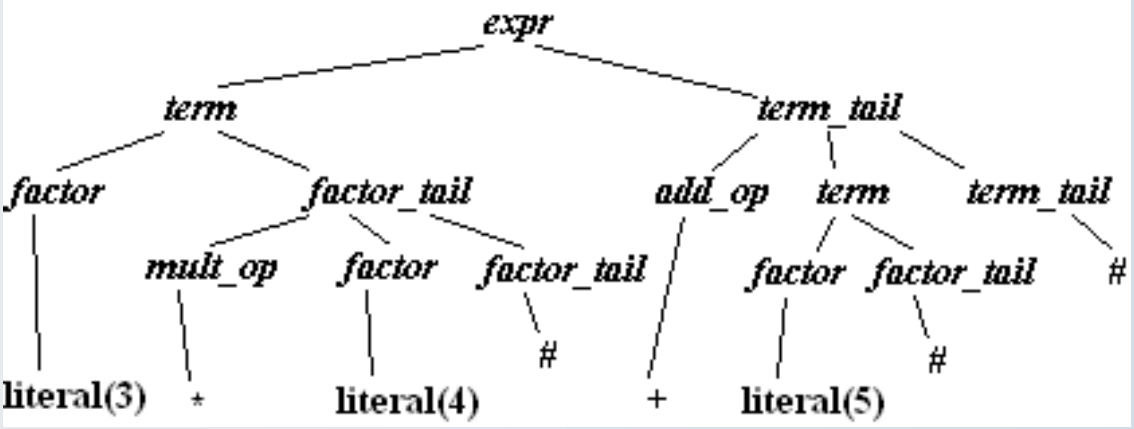
语法：语法分析

语法分析判断输入是否为合语法的程序，并识别出源程序的结构

- 输入：来自词法分析器的单词序列和各单词的词法类别信息
- 输出：某种清晰地表达了源程序的内部构造的表达形式

一种常用的清晰直观的语法结构表达形式是语法分析树

例，3 * 4 + 5 的语法分析树：



语法分析

语法分析是语言翻译过程的一个重要阶段。

语法分析技术是“编译原理”或“编译技术”课程的重要内容

在《程序设计语言——实践之路》和《程序设计语言——原理与实践》中都用一定篇幅讨论了基本的语法分析技术。

本课程不准备详细介绍语法分析的技术

词法分析和语法分析过程并不仅用在编译系统里

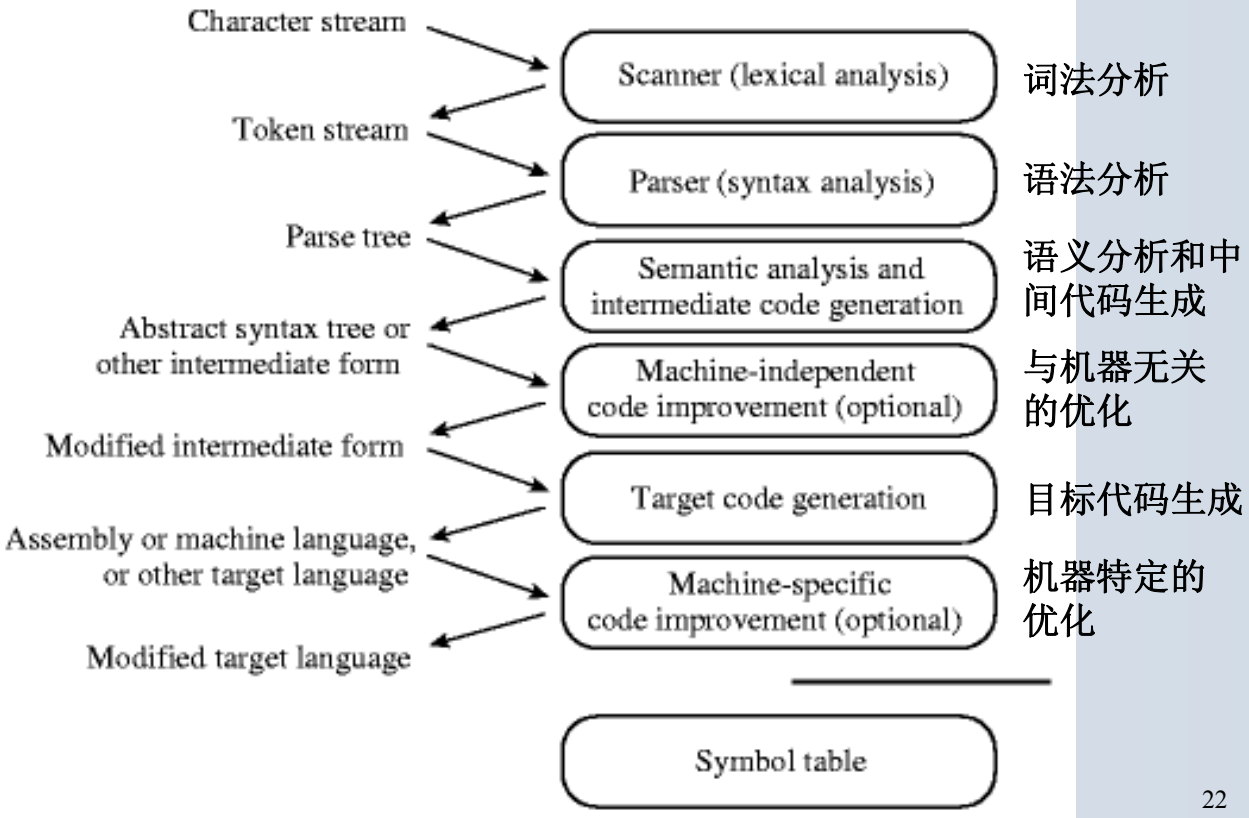
许多计算机软件的用户输入都需要做词法分析和简单的语法分析

今天的大部分语法分析器都是用工具生成的。例如**yacc/bison**

这些工具接受某种类似 **BNF** 的语法描述，生成对应的语法分析器

人们为各种主要语言（**C/C++/Java**等）开发了支持做词法和语法分析的工具或库，有些脚本语言本身就带有词法/语法分析库，可直接使用

常规的语言编译过程

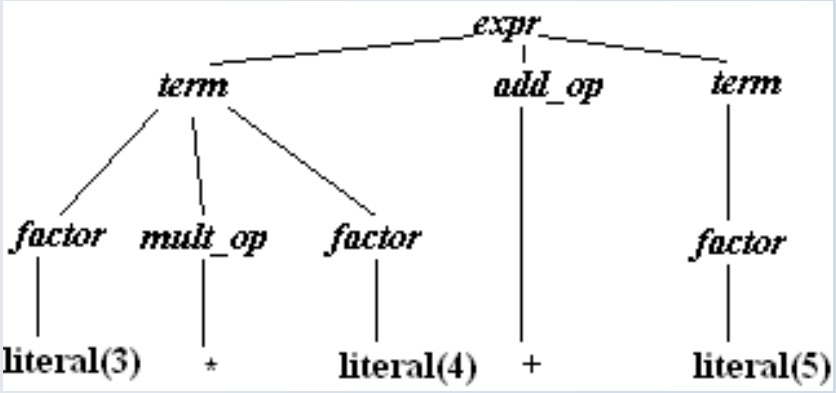


语义分析

语义分析的工作是弄清合法程序的语义，支持后续的代码生成。人们在研究开发高级的静态语义分析技术，以挖掘出程序的深层语义性质，以便检查程序中的深层问题，或者支持更好地生成高效代码

- 输入：语法树
- 输出：抽象语法树和某些中间数据结构，其中记录分析得到的信息

抽象语法树：从语法分析树中删除所有辅助结点，只保留程序的语法结构信息



3 * 4 + 5 的抽象语法树

语义分析

下面主要简单介绍基本语义分析，最后介绍几个高级的语义分析研究问题

基本语义分析中创建的主要数据结构是符号表（symbol table）

- 分析过程中需要创建和维护这个表的内容
- 需要使用表里的信息

源程序里的每个标识符在符号表里有一个项，以标识符名为关键码，项中记录与该标识符有关的各种信息。可能包括：

- 变量的类型信息
- 数组的维数和大小，记录（结构）的描述
- 类型的相关信息
- 函数的原型，代码信息

这里还有作用域的维护问题（作用域问题在下一章讨论）

语义分析

语义分析器工作中

- 遇到一个标识符的定义/声明，就在符号表里增加一项
- 遇到一个标识符的使用时，到符号表里查找与之相关的信息
- 根据得到的信息做上下文关系检查（定义与否、使用方式、类型等）

语义分析中最基本的问题是类型检查，包括：

- 检查表达式的类型合法性，主要是
  - 运算符或子程序是否应用于类型合适的对象
  - 对象的类型不合适时能否做适当的类型转换
- 赋值和其他类似结构的合法性

类型和类型检查问题，后面还要仔细讨论

语义分析

如果一个静态分析算法可以准确判断程序某方面的动态行为，就说这个算法是精确的。例如，

- 有些语言有精确的类型检查算法，如 **Ada** 和 **ML** 等
- 但多数语言没有，不可能静态解决所有类型问题

即使不存在精确的类型检查算法，还是可以做静态分析检查。一般方法是：

- 完成尽可能多的静态分析和检查
- 对于哪些无法通过静态给出确定回答的问题，生成相应的动态检查代码，要求在程序执行中检查

显然，能静态检查的问题只需在运行前静态检查一次，不能静态检查的问题需要在运行中反复做，影响程序的运行效率

例：**Java** 的类型检查主要是静态的，但一些地方，如向下强制类型转换，动态装载的类等，就需要动态检查

语义分析

语义分析中还可能做一些其他工作：

- 挖掘程序里的隐含信息。例如 Fortran 语言允许隐式变量类型定义（以字母 I～N 开头的变量为整型，其余为实型）
- 一些语言中变量不用定义，遇到新变量名时在符号表里加入一项
- 做其他需要做的事情，例如为全局变量分配空间

符号表的处理：

符号表是编译程序处理中的核心数据结构，常规语言在编译完成并生成目标代码后，就把符号表丢掉

在程序开发的调试阶段，编译程序生成代码后会保留符号表的信息，供程序的 Debugger 和符号运行系统使用

一些语言在程序运行中一直保留并使用符号表（如 Lisp）

Java 字节码目标程序里包含着符号表信息，以便 JVM 装载程序时完成各种检查（Java 虚拟机规范定义了字节码程序的结构和装载过程）

语义分析

嵌入目标程序里的动态检查可能需要在运行中反复进行，因此可能消耗大量运行时间。减少不必要的动态检查是一类重要优化

- 弄清楚程序的可能动态行为，可以在生成运行代码时减少很多不必要的动作，采取更合适的执行方式，这些都是重要的优化

语义分析的一些例子：

- 别名分析：分析程序中不同的描述是否可能表示同一个对象。如果一个对象没有别名，就可能安全地保存在寄存器里，不需要不时与存储器内容同步。这方面的一个重要问题是指针分析
- 逃逸分析：一个对象的所有引用是否都禁闭在某个上下文里，因此可以在栈里分配（不需要在堆里分配），不必锁定就可以访问
- 子类型分析：某变量的值保证属于某子类型，相关操作不必动态确定

语义分析是程序设计语言研究的一个重点，是目前很活跃的研究领域。多核系统的出现又提出了许多新的分析问题，为其注入了新的活力