基于扩展的BNF文法的通用语法分析算法

杨明辉 郭肇德

(南京通信工程学院，南京 210016)

A GENERAL SYNTAX ANALYSIS ALGORITHM BASED ON EXTENDED BNF GRAMMAR

Yang Minghui and Guo Zhaode

(*Nanjing Communication Engineering Institute, Nanjing 210016*)

Abstract In this paper, we show how to represent an EBNF (Extended BNF) grammar in form of a syntax graph. We then give an algorithm to translate an EBNF grammar into its syntax graph form. We proceed to present a general parsing algorithm applicable to various procedural high level language and make considerations of error recovery and the insertion of semantic routines.

摘要 本文介绍了如何用句法图来表示扩展的 BNF 文法 (EBNF)，给出一个将 EBNF 文法翻译成句法图算法．然后我们给出一个适用于各种过程性高级语言的通用语法分析算法，同时考虑了语法错误的恢复和语义子程序的嵌入．

§1. 引 言

递归下降语法分析是最普遍地应用于编译实现的方法之一．它有两种具体形式，一种是按给定规则为遇到的每一个语言的语法编制特定的递归子程序，另一种是构造单一的通用分析程序，各个语言的文法规则以初始数据结构 (本文称为句法图) 的形式送到该通用程序里，这种形式称为句法图驱动的递归下降语法分析．二者相比较，前者需要编制很多递归子程序，但分析速度稍快；后者则非常灵活，能够方便地修改文法规则，所以特别适合于作为可扩充语言的研究工具．

用句法图表示文法有一个缺点，就是计算机不能直接读图，而通用分析程序的句法图数据结构必须在语法分析开始之前用某种方法构造好．在这方面，文法的巴科斯范式 (BNF) 表示法是对通用分析程序的理想输入形式．文献 [1] 给出了一个将 BNF 文法变成句法图的翻译程序，并且给出了在其上运行的通用分析算法．

本文 1989 年 12 月 16 日收到，1990 年 9 月 24 日定稿．作者 **杨明辉**，助工，1990 年硕士毕业于南京通信工程学院，目前主要从事程控交换系统，计算机通信网，通信软件等方面的研究工作．**郭肇德**，教授，主要从事人工智能，软件工程，程控交换系统方面的研究工作．

可是，在很多语言的定义中，语法描述经常采用一种扩展的 BNF (EBNF) 形式，最常见的包含重复结构 — 用 $\{\alpha\}$ 表示符号串 α 重复 0 次或多次；任选结构 — 用 $[\alpha]$ 表示符号串 α 是任选的；组合结构 —$(\alpha_1|\alpha_2|\cdots|\alpha_n)$ 表示任选其中一个 α_i. 由于使用这种表示方法通常可以消除左递归，并且比较直观，易于与语言的静态语义产生联系，所以经常被采用. 文献 [1] 的翻译算法不能直接用于 EBNF 文法，而必须先将 EBNF 文法通过增加新的非终结符转变为 BNF 文法. 这样不但不方便，而且会破坏语言定义的直观形式以及语法和静态语义之间的联系，给研究可扩充语言带来不便.

我们在研制 SDL (Specification and Description Language) 语言到 CHILL 语言的翻译程序时发现，用句法图可以很方便地表示 EBNF 方法，便于随时修改所选取的 SDL 语言子集和引入新的语言成分. 针对 EBNF 表示的文法，我们给出其句法图表示形式和相应的翻译算法，然后给出一个通用语法分析算法，并讨论了语法错误的恢复和语义动作的嵌入方法.

§ 2. EBNF 文法的句法图表示

文法的数据结构表示早先由 Irons[2] 提出，之后 Cohen 和 Gotlieb 给出了 BNF 的一种句法图表示，这种句法图后来又为一些作者广泛使用 [1][3][4][5]. 文献 [1] 给出了一种更精致的表示形式，它使用 PASCAL 语言的变体记录来表示句法图的一个结点. 我们采用这种形式来表示 EBNF 文法，通过一个例子来说明.

给定 EBNF 文法 G_1 (S):

S::=a S | b (d | M) [c]

M::={ e S }

相应的句法图表示如图 1 所示 (ϕ 表示空指针值 nil).

每个非终结符 N 都有一个形如图 1 (b) 的头结点，其中 Symbol 是 N 的外部 (或内部) 表示， ENTRY 指向表示该非终结符产生式的句法图的开始结点. 产生式右边的每一个终结符或非终结符都用一个形如图 1 (c) 的结点表示，如果它表示一个终结符，则 x 为该终结符的外部 (或内部) 表示；否则， x 为指向该非终结符的头结点的指针. 一个特殊的结点是如图 1(d) 所示的 ε 结点，它总是用作重复结构或任选结构中第一个符号的最后一个下选 (alt).

用 PASCAL 语言的数据类型来表示句法图的结点，可以有如下定义：

```
{ HEADER }  TYPE  HPOINTER=↑ HEADER;
                  HEADER=RECORD
                              SYMBOL: CHAR;
                              ENTRY: POINTER
                          END;
{ NODE }  TYPE  POINTER=↑ NODE;
                NODE=RECORD
                          ALT, SUC: POINTER;
                          CASE TERMINAL: BOOLEAN OF
                          TRUE: (TSYM: CHAR);
                          FALSE: (NSYM: HPOINTER)
                END;
```

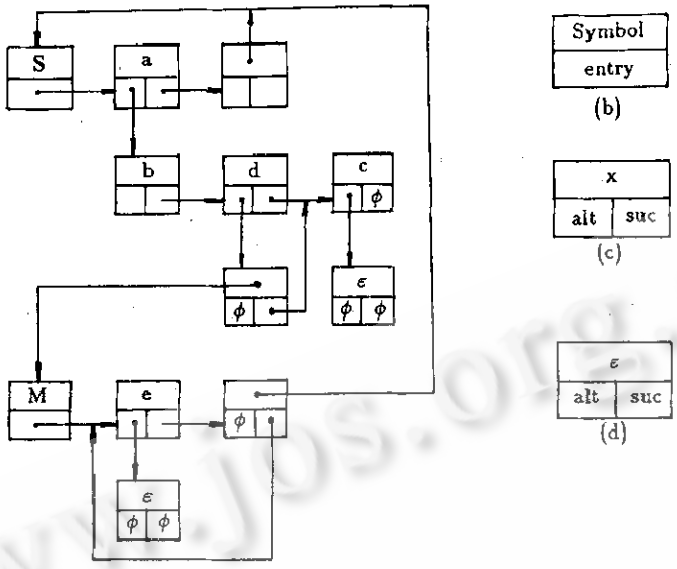


图 1　文法 G_1 (S) 的句法图

句法图的构造规则简单，见图自明．经过推广使用的句法图来表示 EBNF 文法有两个优点：(1) 更加一般化，通用性强．它当然适用于 BNF 文法，并且对于使用 EBNF 定义的语言来说，只须对文法作微小修改 (消除回溯，我们对 SDL 语言的 EBNF 文法的处理经验证实了这一点), 从而避免了增加很多新的非终结符．因此保持了语言定义所固有的直观性以及语法和静态语义之间的联系，这对于语言的研究是十分有利的；(2) 在使用 EBNF 定义的语言中，一般情况下非终结符产生式都不会推导出空串 ε. 例如在 SDL 语言定义中，约 260 个产生式，只有 3 个不常使用的非终结符可以推导出空串 ε. 在使用 EBNF 文法时，我们可以使用重复结构或任选结构来消去这样的非终结符及其产生式，以使通用语法分析程序变得简单．例如在文法 G_1 (S) 中，M 可以推导出空串 ε, 这时我们可以将文法 G_1 (S) 改为 G_2 (S): S::=a S | b (d |{ e S }) [c], 从而消去了 M 及其产生式．而非终结符 M 的语义子程序 (如果有的话) 可以嵌入表达式 {e S } 的 ε 结点中.

当然，使用 EBNF 文法会使得一趟扫描的分析速度减慢，这主要是因为建立句法图数据结构的时间增加以及语法分析过程中涉及到的集合运算较多．但是，语法分析只是整个翻译工作的较小的一部分，更多的时间耗费在语义分析及代码生成上，而且在多趟扫描的情况下，第一趟以后的语法分析是在语法正确的基础上进行的，仅起制导作用，分析速度稍快于使用 BNF 的情形．所以，作为可扩充语言的研究工具，特别是在对翻译速度要求不高的预编译 (如 SDL/CHILL 自动转换) 以及对源文件进行多趟扫描的情况下，这种方法是具有吸引力的.

§ 3. 将 EBNF 翻译成句法图

前已述及，在分析程序开始运行之前，必须先将句法图用某种方法构造好．我们参

照文献 [1] 的基本思想设计了一个翻译程序，使之能将 EBNF 文法转变成句法图.

翻译程序的基本思想，是把 EBNF 看作由它自己的语法所刻划的语言，亦即这些特定的语法可以用 EBNF 本身的产生式来描述. 然后对此特定的 EBNF 描述的语言用递归子程序的方法编制识别程序，再对该识别程序加上语义动作就扩展成一个 EBNF 文法的翻译程序，它把输入的 EBNF 文法翻译成相应的句法图.

EBNF 文法的产生式可以用下列产生式描述：

(1) <EBNF 产生式 >::=< 非终结符 >'::='<EBNF 表达式 >.

(2) <EBNF 表达式 >::=<EBNF 项 > {'|' < EBNF 项 >}

(3) <EBNF 项 > ::=< EBNF 因子 > {< EBNF 因子 >}

(4) <EBNF 因子 >::=(< 终结符 > | < 非终结符 >)
　　|'{'< EBNF 表达式 >'}'
　　|'['< EBNF 表达式 >']'
　　|'('< EBNF 表达式 >')'

其中为了区分作为终结符使用的元符号的二义性，特在作为终结符使用时加一对引号. 例如 '{','}' 是终结符 (注意并非 < 终结符 >!) {,} 是元符号.

针对上面的四个产生式，我们编制一组递归子程序，如图 2 所示，其中 VT、VN 分别是 EBNF 文法的终结符集合与非终结符集合，getsym 是扫描程序，它从输入的 EBNF 源文件中读取下一个符号 Sym(在具体实现时，我们另编制一个翻译程序，将 EBNF 文法翻译成一个内部编码文件，以加快建立句法图的速度). error 是一个错误处理过程，它报告错误信息并终止整个翻译程序.

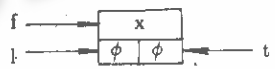
```
procedure ebe; { EBNF 表达式 }
  procedure ebt; {EBNF 项 }
    procedure ebf; { EBNF 因子 }
      begin { ebf }
        if sym in VT+VN then getsym
        else if sym='{' then
              begin getsym; ebe; if sym='}' then getsym else error end
        else if sym='[' then
              begin getsym; ebe; if sym =']' then getsym else error end
        else if sym '(' then
              begin getsym; ebe; if sym=')' then getsym else error end
        else error;
      end { ebf };
    begin { ebt }
      ebf; while sym in VT+VN +['{','[','('] do ebf
    end { ebt };
  begin { ebe }
    ebt; while sym='|' do begin getsym; ebt end
  end { ebe };
procedure ebp;
  begin { ebp }
    getsym;
    if sym in VN then getsym else error;
```

```
    if sym=' ::=' then getsym else error;
    ebe;
    if sym≠' .' then error;
end { ebp };
```

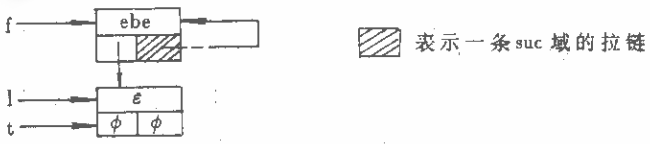
图 2 识别 EBNF 产生式的递归子程序

下面我们在识别程序上添加语义动作，这与句法图有关. 我们描述一些数据结构，每一语言成分都可以用一个图来表示 (图 3), 这个图作为相应识别过程的结果参数被传递，这样就可以把这个识别过程扩展成该成分的翻译程序了. 当然作为结果传递的不可能是图本身，而是指向图中关键结点的指针： f, l 和 t. 指针 f 指向此图的第一个结点，指针 l 指向此图的最后一个下选结点，指针 t 指向此图的第一个后继为 ϕ 的结点 (alt 为下选域, suc 为后继域), 而此图中所有后继为 ϕ 的结点通过它们暂时不使用的 suc 域链接起来. 一旦知道了它们的后继结点 (或者在无后继时为 ϕ), 即可 "反填" 回来，这在编译技术中称为 "拉链反填".

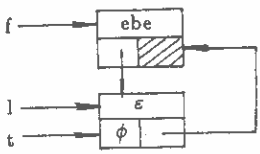
(1) < 终结符号 > 或 < 非终结符 >



(2) '{'<EBNF 表达式 >'}'

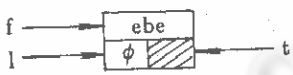


(3) '['<EBNF 表达式 >']'



(4) '('<EBNF 表达式 >')'

(5) < EBNF 项 >



(6) < EBNF 表达式 >

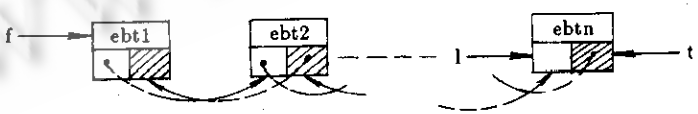


图 3 各个语言成分对应的数据结构

例如，对于表达式 aS | b (d |{ eS }) [c], 过程 ebe (f, l, t) 执行后建立图 4 所示的数据结构.

一旦识别出产生式 S::=a S | b (d |{ eS })[c]. , 即可将指针 t 的拉链全部置为 ϕ, 也就是把 t 所指 ε 结点和 c 结点的 suc 域置为 ϕ.

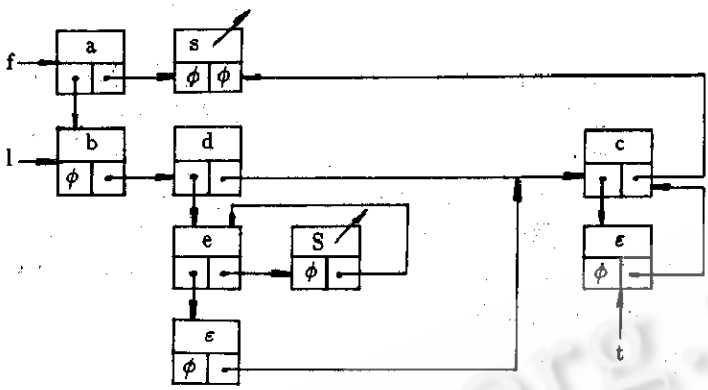


图 4　对表达式 aS | b (d |{ e S }) [c] 建立的句法图

限于篇幅，这里只给出添加了语义动作后的过程 ebf 的一部分程序，见图 5. 这部分程序处理重复结构，其中包含对过程 ebe 的间接递归调用. empty 是一个常数，它表示空串 ε.

```
procedure ebf (var f, l, t: pointer);
  var p, q, r: pointer;
  begin { ebf }
      ⋮
    if sym ='{' then
      begin
        getsym; ebe (f, p, q); (* 对 EBNF 表达式构造句法图 *)
        while q ≠ nil do
          begin
            r: =q ↑. suc; q ↑. suc:=f; q:=r
          end; (* 产生循环结构，将 q 所拉的链全部指向首结点 *)
        new (l);
        with l ↑ do begin
                      terminal: =true; tsym:=empty; alt: =nil;
                      suc: =nil;
                    end;
        p ↑. alt:=l; (* 产生 ε 结点 *)
        if sym='}' then getsym else error
      end
  else
      ⋮
end { ebf };
```

图 5　EBNF 重复结构的翻译

§ 4. 一个递归分析算法

假定 EBNF 文法满足无回溯条件，图 6 给出了通用分析算法流图. 它是一个布尔递归函数，称为 INS (G). 它搜索一个目标非终结符， G 是一个形式参数，它指向该目标非终结符的头结点，初始进入时它被置为文法的开始符号. 根据此非终结符产生出一个符号串来匹配一个给定的输入子串. 如果终结符串是该文法的一个句子，则 INS (G) 返回 TRUE, 否则就返回 FALSE. 图中 sym 是包含当前输入终结符的全局变量，初始时它被置为待检串的第一个符号，假定进入此函数时 sym 总是属于 G 的头符号集. FOL (p) 表示当前句法图中 p 结点的后继符号集，但可以包含 ε.

主程序可以是：

```
sym:= 待检串的第一个符号;
G:= 文法开始符号之头结点的指针值;
if sym in FIRST (G) then
        if INS (G) and sym= 输入串结束符号
        then 匹配成功
        else 匹配失败
else 匹配失败;
```

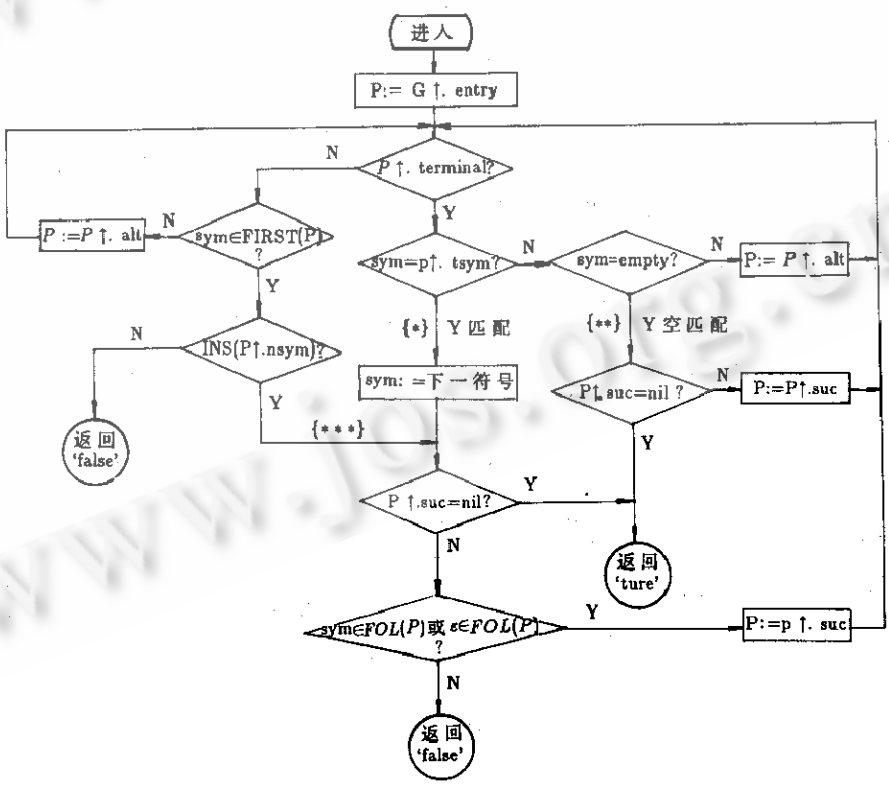


图 6　布尔递归函数 INS (G) 的流图

§ 5. 语法错误恢复和语义动作的嵌入

5.1 语法错误的恢复

上面的算法仅能完成句子识别的任务，只要碰上错误的输入，分析程序的任务即告完成，即返回 FALSE, 程序也就终止. 对于真正的编译程序来说，这显然是不够的. 编译程序应当给出适当的错误信息，并继续分析以发现更多的错误.

我们使用了递归子程序技术的错误恢复方法，在分析过程中给终存在两个集合：当前结点在句法图中的后继符号集 FOL 和当前非终结符子图的停止符号集 STOP, 每当对一个非终结符进行递归下降分析要进入其子图时，就把当前此非终结符的 FOL 集加到 STOP 集上. 当发现非法符号，即 sym $\notin$FOL, 则跳过后面的输入符号串，直到下一个属于 FOL 集或者 STOP 集的符号为止. 这一过程可以描述为：

```
procedure error;
  begin
    print error message;
    while not (sym in FOL+STOP) do sym: = 下一符号;
  end;
```

STOP 集初始时为 { 输入串结束符号 }, FOL 随着分析过程的每一步在不断改变. 经过 error 处理后，如果 sym $\in$ FOL, 则可以在当前子图中继续分析下去，否则就返回 FALSE, 退到上一个子图进行这一过程.

5.2 语义动作的嵌入

在进行递归下降语法分析的过程中，我们可以执行翻译程序的 "语义" 部分，这只要给每个结点增加一个域 sem 就行了. 这个域可以是一个语义子程序的编号，每当在输入串中识别出某个结点时，就调用相应的语义子程序. 语义子程序的调用是在图 6 中 {*}、{**} 和 {***} 处. 我们约定，如果 p ↑. sem=0 就表示没有语义动作和此结点相联系.

每个结点的语义子程序编号可以嵌进输入的 EBNF 文法中，只要增加一个元符号 @，@ n 表示语义子程序编号为 n, 这样在翻译程序生成句法图时自动将 n 填入相应结点域 sem. 每个终结符或非终结符后面都可以跟形如@ n 的编号，重复和任选之后也可以跟子程序号，这时是嵌入其 ε 结点的 sem 域，组合结构不能跟有语义子程序号，因为没有 ε 结点，这时要嵌入每一个成分中去.

§ 6. 结束语

许多语言的语法定义采用了 EBNF 形式，因为它更加直观和便于理解. 本文讨论了基于 EBNF 的句法图驱动的递归下降语法分析，它适用于各种用 EBNF 定义的过程性高级语言. 我们在编写 SDL/CHILL 翻译程序时使用了这一方法. 输入的 SDL 文法共有 147 个非终结符及其产生式，通用语法分析程序约 100 行 PASCAL 代码，目前已在 VAX 机上实现. 结果表明，在使用中不必大改 SDL 文法，只须对个别产生式作提公因子的处理，便于扩充所选择的 SDL 语言子集和加入新的语义动作. 给使用者提供了很大的灵活性，使得它特别适合于可扩充语言的研究.

参考文献

[1] N. Writh, 算法 + 数据结构 = 程序，科学出版社 (1984).
[2] R.T.Irons, "A Syntax Directed Compiler for ALGOL60", Comm. ACM, 4, 1961.
[3] D.J.Cohen and C.C.Gotlieb, "A List Structure Form of Grammars for Syntactic Analysis", Computing Surveys, 2, 1, 1970.
[4] V.W.SETZER, "Non-Recursive Top-down Syntax Analysis", Software Practice and Experience, 9, 1979.
[5] 施振川，"一种实用的语法识别程序"，计算机学报，1983 年第 4 期.

第四届全国机器学习研讨会—CMLW'93 征文通知

中国人工智能学会机器学习专业委员会 (中国机器学习学会) 与中国计算机学会人工智能模式识别专业委员会定于 1993 年八月中旬在四川省松潘召开第四届全国机器学习研讨会 (CMLW'93), 由中国科学院自动化研究所与成都科技大学承办. 欢迎国内各界专家学者研究人员踊跃投稿. 兹将有关事项通知如下：

一、征文范围：

机器学习的理论与方法	知识获取与应用学习系统
1. 机器学习的一般理论	1. 自动知识获取方法与工具
2. 学习的认识机制与模拟	2. 分类与识别
3. 示例学习与归纳法	3. 问题求解与规划
4. 类比与基于事例法	4. 设计与诊断
5. 讲授学习	5. 自然语言理解
6. 观察与发现学习	6. 语音与图象处理
7. 基于解释的学习	7. 机器人运动控制
8. 神经网络的连接学习	8. 其它应用学习系统
9. 遗传算法	

二、征文要求：1. 论文要反映新的研究思想与成果，未在其它会议或刊物上发表过. 2. 正文字数一般为 6000 字，最多不超过 8000 字. 来稿一式两份. 3. 请附 200 字以内的中英文摘要、关键词和作者的通讯地址. 4. 投稿地址为：100080 北京中关村中国科学院自动化研究所 杨忠祥 研究员收. 5. 会议将出版论文集. 打印清样格式与出版费在录取时另行通知. 6. 论文截止日期：1992 年 12 月 15 日. 录取通知日期：1993 年 1 月 15 日. 清样付印日期：1993 年 3 月 30 日.

中国机器学习学会秘书处
1992 年 5 月 15 日