一种特殊的上下文无关文法及其语法分析*

张瑞岭

(中国科学院软件研究所计算机科学开放研究实验室 北京 100080)

摘要 SAQ 系统是一个进行软件规约获取、检验和复用的实验系统,其中以上下文无关文法表示的概念是规约的一部分.SAQ 要求将概念的词法和句法定义结合在一个上下文无关文法中.如果用常规的上下文无关文法描述诸如程序设计语言和自然语言等一些复杂概念的语法,则需要把诸如空格和回车等没有实质意义的分隔符包含到语法中去(这种描述方法称为朴素表示法),使得语法描述很累赘.为此,作者设计了一种特殊的上下文无关文法,它把通常上下文无关文法定义中的非终极符集合和终极符集合进行细化.用这种文法可以相对简洁地描述程序语言和自然语言等复杂概念的完整定义,而且,其相应的语法分析效率较朴素表示法有所提高.同时,给出相应于这种特殊的上下文无关文法的语法分析和语法树生成算法.这些算法分别由在通用上下文无关语法的 Earley 分析算法及其相应的语法树生成算法的基础上改进而得到.

关键词 上下文无关语言,语法分析,语法树.

中图法分类号 TP314

上下文无关(Context-Free,CF)文法[1]是广泛用于描述自然语言和程序设计语言的语法,相应地,CF 语法分析广泛地应用于程序设计语言的编译器和解释器实现以及自然语言的理解和翻译等领域中.

本文提出一种特殊的 CF 文法及其语法分析方法,它是为形式规约获取系统 SAQ[2,3]设计的.该系统在规约库的支持下进行形式规约的获取、复用和检验,用 CF 文法表示的概念是规约的一部分.在基于复用的概念获取(在获取新概念的过程中复用已知概念)和概念检验(检验获取的概念是否符合要求)的过程中,需要一个通用的 CF 语法分析器,用来判定一个字符串是否为一个概念的合法句子(即概念的实例).为此,我们用 Earley 算法[4]实现了一个 CF 语言的通用分析器.[5]但在进行程序设计语言和自然语言等一些复杂概念的语法分析时,我们发现,如果按常规处理方法,即先经过词法分析,再进行句法分析,不能满足我们的要求,因为我们需要的是将词法分析任务包含在内的完整的语法分析;如果用通常的 CF 文法去描述包括词法结构信息在内的完整语法定义,则需要把诸如空格和回车等没有实质意义的分隔符包含到语法中去(我们把这种方法称为朴素表示法),使得语法定义很不直观.在 SAQ 系统中,概念的语法描述用于定义运算,所谓运算,就是 CF 语言上的递归函数[6],所以,要求概念的定义尽可能简洁、自然.为此,我们设计了一种特殊的上下文无关文法,它把通常的上下文无关文法定义中的非终极符集合和终极符集合进行细化.用 SAQ 系统中术语来说(在 SAQ 中概念名与非终极符等价),即把概念分为 Token 和 Non-token 两种类型.对于 Token 型概念,如整数、标识符等简单概念,它们的实例被看成由字符组成的单词,即一个实例是一个 Token;而对于 Non-token 型概念,如程序设计语言和自然语言等复杂概念,它们的实例被看成由多个 Token 组成的 Token 串,Token 之间以空格、回车等分隔符分开.Non-token 型概念的定义中可以出现 Token 型子概念.下面,我们先给出这两种概念的严格定义,接着介绍对应于 Non-token 型概念的语法分析和语法树生成算法及其实现,然后进行算法分析并给出实验结果,最后进行讨论.

1 定义和记号约定

一个 Token 型概念的定义用通常的 CF 文法表示,我们用 Earley 算法实现其语法分析.并在 Earley 算法所产生的分析结果的基础上构造语法树.[7]本文假定已知 Token 型概念的语法分析和语法树生成算法.

---

* 本文研究得到国家自然科学基金、国家 863 高科技项目基金和国家"九五"攻关计划基金的支持.作者张瑞岭,1969 年生,博士生,主要研究领域为机器学习.

本文通讯联系人:张瑞岭,北京 100080,中国科学院软件研究所计算机科学开放研究实验室

本文 1997-05-16 收到原稿,1997-11-14 收到修改稿

一个 Non-token 型概念的定义用一个类 CF 文法 G 表示，它把通常 CF 文法定义的四元组中的非终极符集合分解为两个集合 V_N^n 和 V_N^t，终极符集合分解为两个集合 V_T^t 和 V_T^c，文法 G 记为

$$G=(V_N^n,V_N^t,V_T^t,V_T^c,P,X),$$

其中 V_N^n 为 Non-token 型非终极符集合，V_N^t 为 Token 型非终极符集合，V_T^t 为 Token 终极符集合，V_T^c 为字符终极符集合，P 为产生式集合，X 为开始非终极符. 我们用 $V_N=(V_N^n\cup V_N^t)$ 表示全体非终极符集合，其元素用大写字母 $A,B,C,\ldots$ 表示. $(V_N\cup V_T^t)^*$ 表示由非终极符和 Token 终极符组成的串(包括空串)，其元素用希腊字母 $\alpha,\beta,\gamma,\ldots$ 表示. $(V_N^t\cup V_T^c)^*$ 表示由 Token 型非终极符和字符终极符组成的串. $(V_T^t)^*$ 表示由 Token 终极符组成的串(简称 Token 串). 产生式集合 P 可以表示为

$$P=\{X\to\alpha|X\in V_N^n,\alpha\in(V_N\cup V_T^t)^*\}\cup\{X\to\alpha|X\in V_N^t,\alpha\in(V_N^t\cup V_T^c)^*\}.$$

上式表明，P 中产生式具有如下特性：当产生式左部 X 为 Non-token 型非终极符时，其右部不会出现字符终极符；当产生式左部 X 为 Token 型非终极符时，其右部不会出现 Non-token 型非终极符和 Token 终极符.

我们用 λ 表示空串，$\varnothing$ 表示空集合，概念 Y 的文法定义记为 $G(Y)$，概念 Y 的所有实例组成的集合(即 $G(Y)$ 定义的语言)记为 $L(G(Y))$. 我们用 $\alpha\Rightarrow\beta$ 表示$\exists\ \gamma,\delta,\eta,A$ 满足 $\alpha=\gamma A\delta,\beta=\gamma\eta\delta$ 且 $A\to\eta$ 是一个产生式. $\alpha\overset{*}{\Rightarrow}\beta$ 表示$\exists\ \alpha_0,\alpha_1,\ldots,\alpha_m(m\geqslant 0)$ 满足

$$\alpha=\alpha_0\Rightarrow\alpha_1\Rightarrow\ldots\Rightarrow\alpha_m=\beta.$$

这里，序列 $\alpha_0,\ldots,\alpha_m$ 称为从 α 到 β 的一个推导.

在下面的描述中，非终极符与概念名等价. 概念 Y 关于其实例 s 的语法树等价于 Y 关于 s 的最右推导产生式序列.

2　算法及实现

下面给出 Non-token 型概念的语法分析算法，该算法由 Earley 算法改进而来. 算法 1 判断 Token 串 s 是否为 Non-token 型概念 X(X 的定义为 G)的合法实例，记 $s=t_1t_2\ldots t_n$(其中 $t_i\in V_T^t,i=1,\ldots,n$). 算法(以下称为识别器)的基本思想如下：给定文法 G 和 Token 串 s，识别器从左向右扫描 s，每扫过一个 Token t_i，识别器构造一个状态集 S_i，其中每个状态形如 $(A\to\alpha\cdot\beta,j,T_\alpha)$，这里 $A\to\alpha\beta$ 为 P 中一产生式，其中 α 为已分析部分；$j(\leqslant i)$ 表示该状态由 S_j 中某状态产生而来；T_α 是个集合，用于记忆 α 中各个 Token 型非终极符分别接受 s 中某个 Token $t_m(j<m\leqslant i$，对 α 中某个具体的 Token 型非终极符，对应的 m 值可以在语法分析过程或语法树生成过程中确定)对应的语法树. T_α 中的元素是一个二元组 $(Locate(Y,\alpha),Tree(Y,t_m))$，$Locate(Y,\alpha)$ 表示 Token 型非终极符 Y 在串 α 中的位置，$Tree(Y,t_m)$ 表示概念 Y 关于 Token t_m 的语法树. 算法 1 描述了各状态集 S_i 中状态的产生过程.

算法 1.

输入：CFG $G=(V_N^n,V_N^t,V_T^t,V_T^c,P,X)$ 和 Token 串 $s=t_1t_2\ldots t_n$.

输出：$S_i(0\leqslant i\leqslant n)$

过程：

```
1   begin
2   S0:={($ → · X,0,∅)};
3   for i:=1 to n do
4       Si:=∅;
5   for i:=0 to n do
6       begin
7           process the states of Si in order, performing one of the following operations on each state: (A→α·Yβ,j,Tα)
8       (1) Predictor: if Y∈VNn then
9               for each production Y→η∈P do
10              add (Y→ ·η,i,∅) into Si;
11      (2)  Scannert: if Y∈VTt and i≠n and Y=ti+1 then
12              add (A→αY·β,j,Tα) into Si+1;
13      (3)  ScannerN: if Y∈VNt and i≠n and ti+1 can be accepted by Y then
14              add (A→αY·β,j,Tα∪{(Locate(Y,αY),Tree(Y,ti+1))}) into Si+1;
15                  if Y∈VNt and λ∈L(G(Y)) then
16              add (A→αY ·β,j,Tα) into Si;
17      (4) Completer: if Yβ=λ then
```

```
18                for each (B→ξ · Aη,m,Tξ)∈Sj(after all states have been added to Sj) do
19                add (B→ξA · η,m,Tξ) into Si;
20          end / * of for * /
21    if there is a state of form (X→α · ,0,Tα)∈Sn then return acceptance,
22      else return rejection.
```

算法1第13行中出现的 t_{i+1} can be accepted by Y 表示 $t_{i+1}\in L(G(Y))$，即 t_{i+1} 被识别为 Token 型概念 Y 的实例，该过程和第14行求 $Tree(Y,t_{i+1})$ 的过程分别用通常的 CF 语法分析算法和语法树生成算法实现. 第15行中 $\lambda\in L(G(Y))$ 表示 Y 包含空字实例，从概念的语法定义判断它是否包含空字实例的简单方法是：考察 $G(Y)$ 中任一产生式，若(1)形如 $Y\to\lambda$；或(2)形如 $Y\to A_1A_2\ldots A_k$，其中 $k\geqslant 1$，$A_i(i=1,\ldots,k)$ 为非终极符，且 $\lambda\in L(G(A_i))$，则 $\lambda\in L(G(Y))$. 若 $G(Y)$ 不存在以上情形的产生式，则 $\lambda\notin L(G(Y))$.

由于算法1是从 Earley 算法改进得到，故它具有与 Earley 算法类似的性质：

① 当开始对 S_i 中的状态进行处理时，$S_0,S_1,\ldots,S_{i-1}$ 中不会有新状态加入；

② 假设 $(A\to\alpha\cdot\beta,j,T_\alpha)\in S_i(\alpha$ 可以为 $\lambda)$，它表示：若 $j<i$，则有 $\alpha\overset{*}{\Rightarrow}t_{j+1}t_{j+2}\ldots t_i$；若 $j=i$，则有 $\alpha\overset{*}{\Rightarrow}\lambda$；

③ 若 G 中无空字产生式(即形如 $B\to\lambda$ 的产生式)，则当前状态集 S_i 中不会出现形如 $(A\to\alpha\cdot,i,T_\alpha)$ 的状态项. 也就是对 S_i 中任一状态 $(A\to\alpha\cdot\beta,j,T_\alpha)$，不会有 $\beta=\lambda$ 且 $j=i$ 的情形. 对该结论，可以简单地证明如下：对于当前状态集 S_i 中任一状态 $(A\to\alpha\cdot\beta,j,T_\alpha)$，有以下可能：

(a) 该状态由当前状态集 S_i 中另一状态经过 Predictor 操作派生而来. 由于 G 中无空字产生式，故此时 $\alpha=\lambda,\beta\neq\lambda$ 且 $j=i$；

(b) 该状态由前一状态集 S_{i-1} 中一状态经过 Scanner 操作(即算法1中11～12或13～14行)派生而来. 此时有 $j<i$.

也就是不可能有 $\beta=\lambda$ 且 $j=i$ 的情形，从而证明了上述结论. 反之，若 G 中包含空字产生式，则当前状态集 S_i 中有可能包含形如 $(A\to\alpha\cdot,i,T_\alpha)$ 的状态项，该结论可类似地得到证明.

根据以上性质，算法1的实现方法如下：

(1) 将 P 中产生式从1开始编号，产生式 $\$\to X$ 编号为0；

(2) 对应于 G 中每个非终极符 A 有一个产生式编号链表，这些产生式的共同特点是左部为 A，该链表用于进行 Predictor 操作. 同时对应于每个非终极符 A 设一标志位，以保证就一个状态集的各个状态的处理过程中，对于一个非终极符的 Predictor 操作最多进行一次(由形如 $(A\to\alpha\cdot Y\beta,j,T_\alpha)$(这里 $Y\in V_N$)的状态引起的 Predictor 操作称为对 Y 的 Predictor 操作)；

(3) 每个状态集合中的状态按加入的先后顺序置于一个链表中，并按链表中的顺序对各状态依次处理；

(4) 根据性质③，当 P 中包含空字产生式时，算法1中第17～19行的 Completer 运算不能直接进行，即为了满足18行括号中的内容而需做特殊处理. 举例来说，由于 P 中包含空字产生式，所以状态集 S_i 中可能出现一状态形如 $(A\to\alpha\cdot,i,T_\alpha)$($\alpha$ 可以为 λ)，对该状态进行 Completer 操作，即对 S_i 中'·'后为 A 的状态，将'·'移至 A 后得到的新状态加入 S_i，但当前时刻可能有某一'·'后为 A 的状态尚未加入到 S_i 中. 对这种情况的处理是，定义一个非终极符集合 CV_N(在开始处理状态集 S_i 前置 $CV_N=\varnothing$)，一旦处理的当前状态形如 $(A\to\alpha\cdot,i,T_\alpha)$，则将 A 加入 CV_N；在向 S_i 中加入新状态时进行检查，设加入状态为 $(B\to\gamma\cdot Y\beta,m,T_\gamma)$，若 $Y\in V_N$，且 $Y\in CV_N$，则将 $(B\to\gamma Y\cdot\beta,m,T_\gamma)$ 一并加入 S_i；

(5) 某 Token 可能在 Token 串 s 中多次出现，这样，同一个 Token 型概念对同一个 Token 的分析过程可能会重复多次. 为避免这种情况，在语法分析过程中维护一张表，表中元素为三元组(Token 型概念名，Token，分析结果). 这样，在算法1的第13行中，在 Token 型概念 Y 对 Token t_{i-1} 分析之前，先查表，若表中已有项 $(Y,t_{i+1}$，分析结果)，则无需再分析.

对 Token 型概念，合法实例的语法树生成算法在文献[7]中已有所描述，在此基础上，我们给出对于 Non-token 型概念的相应算法. 如果 s 为 G 的句子，算法2由算法1产生的分析表出发，求生成 s 的最右推导产生式序列 π. 首先置 π 为空，在 S_n(n 为 s 的长度)中找到形如 $(X\to\alpha\cdot,0,T_\alpha)$($X$ 为开始非终极符)的表项，调用过程 RightParse$((X\to\alpha\cdot,0,T_\alpha),n)$，最终 π 中包含了生成 s 的最右推导产生式序列(即语法树).

算法2.

输入：CFG $G=(V_N,V_N',V_T',V_T,P,X)$，Token 串 $s=t_1\ldots t_n$ 和算法1产生的关于 s 的状态集 $S_0,S_1,\ldots,S_n$.

输出：关于 s 的最右推导产生式序列 π.

过程：RightParse$((Y\to\beta\cdot,i,T_\beta),j)$

```
1   begin /* RightParse */
2   π_i=π∪{Y→β}, assume that β=R_1R_2...R_m, set k_i=m and h_i=j.
3   while k>0 do begin
4       if (R_k∈V_T') then begin
5         k_i=k-1;h_i=h-1;
6         end
7       else if (R_k∈V_N') then begin
8         if (Locate(R_k,β),Tree(R_k,t_h))∈T_β then begin
9             π_i=π∪Tree(R_k,t_h);
10            k_i=k-1;h_i=h-1;
11            end
12        else begin
13            π_i=π∪Tree(R_k,λ);
14            k_i=k-1;
15            end
16        end
17      else if (R_k∈V_N') then find an item (R_k→γ·,r,T_γ)∈S_h for some r such that (Y→R_1R_2...R_{k-1}·R_k...R_m,i,
            T_{R_1...R_{k-1}})∈S_r then begin
18            recursively call RightParse ((R_k→γ·,r,T_γ),h);
19            k_i=k-1;h_i=r;
20            end
21      end / * of while */
22  end / * of RightParse */
```

例 1：给定 CFG $G(E)=(V_N^0,V_N',V_T',V_T^0,P,E)$，这里

$V_N^0=\{E\}$

$V_N'=\{\mathbf{I},\mathbf{D}\}$

$V_T'=\{1,2,3,4,5,6,7,8,9,0\}$

$V_T^0=\{+,\times,\mathbf{I}$ 的所有实例$\}$

$P=\{E\to\mathbf{I}$,

$E\to E+E$,

$E\to E\times E$,

$E\to(E)$,

$\mathbf{I}\to\mathbf{D}$,

$\mathbf{I}\to\mathbf{ID}$,

$\mathbf{D}\to 0|1|2|3|4|5|6|7|8|9\}$

G 定义的语言为简单的算术表达式. 算法 1 关于 Token 串“12◇×◇(◇12◇+◇34◇)”(◇表示 Token 之间的分隔符)的分析表如表 1 所示. 算法 2 生成相应的产生式序列为$\{E\to E\times E,E\to(E),E\to E+E,E\to\mathbf{I},\mathbf{I}\to\mathbf{ID},\mathbf{D}\to 4,\mathbf{I}\to\mathbf{D},\mathbf{D}\to 3,E\to\mathbf{I},\mathbf{I}\to\mathbf{ID},\mathbf{D}\to 4,\mathbf{I}\to\mathbf{D},\mathbf{D}\to 1,E\to\mathbf{I},\mathbf{I}\to\mathbf{ID},\mathbf{D}\to 2,\mathbf{I}\to\mathbf{D},\mathbf{D}\to 1\}$.

表 1 是 E 关于 Token 串“12◇×◇(◇12◇+◇34◇)”的分析表. 其中 P,S_T,S_N,C 分别为 Predictor, Scanner_T, Scanner_N, Completer 的缩写. 状态$(A\to\alpha,\beta,j,T_e)$中域 T_e 被略去.

表 1

S_0	1	$\$\to\cdot E$	0	initial setting
	2	$E\to\cdot\mathbf{I}$	0	$P(1)$
	3	$E\to\cdot E+E$	0	$P(1)$
	4	$E\to\cdot E\times E$	0	$P(1)$
	5	$E\to\cdot(E)$	0	$P(1)$
S_1	6	$E\to\mathbf{I}\cdot$	0	$S_N(2)$
	7	$E\to E\cdot+E$	0	$C(6,3)$
	8	$E\to E\cdot\times E$	0	$C(6,4)$
S_2	9	$E\to E\times\cdot E$	0	$S_T(8)$
	10	$E\to\cdot\mathbf{I}$	2	$P(9)$
	11	$E\to\cdot E+E$	2	$P(9)$

	12	$E\rightarrow \cdot E\times E$	2	$P(9)$
	13	$E\rightarrow \cdot(E)$	2	$P(9)$
S_3	14	$E\rightarrow (\cdot E)$	2	$S_T(13)$
	15	$E\rightarrow \cdot \mathbf{I}$	3	$P(14)$
	16	$E\rightarrow \cdot E+E$	3	$P(14)$
	17	$E\rightarrow \cdot E\times E$	3	$P(14)$
	18	$E\rightarrow \cdot(E)$	3	$P(14)$
S_4	19	$E\rightarrow \mathbf{I}\cdot$	3	$S_N(15)$
	20	$E\rightarrow (E\cdot)$	2	$C(19,14)$
	21	$E\rightarrow E\cdot +E$	3	$C(19,16)$
	22	$E\rightarrow E\cdot \times E$	3	$C(19,17)$
S_5	23	$E\rightarrow E+\cdot E$	3	$S_T(21)$
	24	$E\rightarrow \cdot \mathbf{I}$	5	$P(23)$
	25	$E\rightarrow \cdot E+E$	5	$P(23)$
	26	$E\rightarrow \cdot E\times E$	5	$P(23)$
	27	$E\rightarrow \cdot(E)$	5	$P(23)$
S_6	28	$E\rightarrow \mathbf{I}\cdot$	5	$S_N(24)$
	29	$E\rightarrow E+E\cdot$	3	$C(28,23)$
	30	$E\rightarrow E\cdot +E$	5	$C(28,25)$
	31	$E\rightarrow E\cdot \times E$	5	$C(28,26)$
	32	$E\rightarrow (E\cdot)$	2	$C(29,14)$
	33	$E\rightarrow E\cdot +E$	3	$C(29,16)$
	34	$E\rightarrow E\cdot \times E$	3	$C(29,17)$
S_7	35	$E\rightarrow (E)\cdot$	2	$S_T(32)$
	36	$E\rightarrow E\times E\cdot$	0	$C(35,9)$
	37	$E\rightarrow E\cdot +E$	2	$C(35,11)$
	38	$E\rightarrow E\cdot \times E$	2	$C(35,12)$
	39	$E\rightarrow E\cdot +E$	0	$C(36,3)$
	40	$E\rightarrow E\cdot \times E$	0	$C(36,4)$

3 算法分析

下面我们对算法1的时间复杂度做简单分析,其目的是对本文所述文法对应的语法分析与朴素表示法对应的语法分析的效率做些比较.由于语法分析的实际运行效率与具体的文法和分析串密切相关,因此这里只能从最坏时间复杂度角度对两种方法做简单的定性比较.

我们知道,对通常CF文法G',Earley算法的最坏时间复杂度为$O(n^3)$,即存在一个常数C,Earley算法的时间上限为Cn^3,这里C是与文法G'的规模(由终极符和非终极符数、产生式数以及产生式右部的长度等因素决定)有关的常数,n为待分析串的长度.算法1在Earley算法的基础上得到,因此可以很容易地得到算法1的时间复杂度.需要说明的是,算法1的第13行包含Token型概念对Token的分析过程,在算法1对Token串s的分析过程中,同一个Token可能被多个不同的Token型概念分析,假设每个Token平均被C_t个不同的Token型概念分析,输入文法为G,待分析Token串$s=t_1t_2\ldots t_n$,于是算法1的时间上限为

$$C_tC_{G_t}(|t_1|^3+|t_2|^3+\ldots+|t_n|^3)+C_{G_n}n^3. \tag{1}$$

其中$|t_i|$表示Token t_i的长度($i=1,\ldots,n$),即字符数;C_{G_t}是与Token型概念的文法定义的规模相关的常数,这里假定各Token型概念的规模相同;C_{G_n}是与G_n规模相关的常数,G_n是G中除去左部为Token型非终极符的产生式所得到的文法.式(1)中前一项表示分析所有Token的时间上限,后一项表示在Token层次上分析s的时间上限.

下面看看朴素表示法对应的语法分析的情况.设G'用朴素表示法描述与G相同的概念.对于s,朴素表示法对应的语法分析的时间上限为(此时s被视为字符流,而非Token流)

$$C_{G'}(|t_1|+|t_2|+\ldots+|t_n|+\text{Token之间的分隔符数})^3. \tag{2}$$

其中$C_{G'}$是与G'的规模相关的常数.为了对式(1)、(2)的大小进行直观的比较,我们假定

$$C_{G_t}\approx C_{G_n}\approx C_{G'},$$

同时,s中各Token的长度相同,即$|t_1|=|t_2|=\ldots=|t_n|=m$,并忽略Token之间的分隔符,于是式(1)、(2)的比较问题简化为比较$C_tnm^3+n^3$与$(nm)^3$的大小.对m进行归纳,可以证明:当$m,n\geqslant 2$且$C_t<\frac{m^3-1}{m^3}n^2$时,有

$$C_1nm^3+n^3<(nm)^3.$$

显然，$C_1<\frac{m^3-1}{m^3}n^2$ 的条件很容易满足，因为对一具体的文法 G，有

$$C_1\leqslant G\text{ 中 Token 型非终极符数，}$$

而一般文法中 Token 型非终极符较少，以 C 程序语言为例，其语法定义中 Token 型非终极符（概念）有整数、数字、字母和标识符等．因此，我们可以认为，本文所述方法对应的语法分析的时间上限不会高于朴素表示法对应的语法分析的时间上限．

以例 1 中文法 $G(E)$ 为例，若用通常的 CF 文法 $G'(E)$ 描述相同的概念，则

$G'(E)=(\{E,I,D,S\},\{+,\times,0,1,2,3,4,5,6,7,8,9,□\},P,E)$，其中□表示空格符，且

$P=\{E\to I,$

$E\to ES+SE,$

$E\to ES\times SE,$

$E\to(SES),$

$I\to D|ID,$

$D\to 0|1|2|3|4|5|6|7|8|9,$

$S\to □|SS\}$

$G'(E)$中非终极符 S 表示一个或多个空格，显然 $G'(E)$比 $G(E)$显得繁琐．我们用 Earley 算法实现的 $G'(E)$对串“12□×□(□12□+□34□)”的分析过程中共生成 297 个状态，而用算法 1 实现的 $G(E)$对串“12◇×◇(◇12◇+◇34◇)”的分析过程中共生成 176 个状态（包括算法 1 的第 13 行中 Token 型概念对 Token 的分析过程中生成的状态数）．

4　实验结果

上面对算法 1 的效率分析以及与朴素表示法对应的语法分析效率的比较，似乎不足以说明问题，因为，首先在实际应用中，对多数文法而言，算法的效率远小于其时间上限，甚至可能是线性的；其次，很多常数（如 C_1,C_{Gl},C_{Gn},C_G 等）的大小关系仅仅是对大多数情形的估计；另外，本文方法虽然可能减少了分析过程中生成的状态数（事实上并不总是如此），但有一些额外操作（如实现方法 5 中表的维护）也占用一定时间．因此，真正说明问题的是实验结果．

作者在 SUN SPARC 1100E 上实现了算法 1 和算法 2．陈自明在提出问题和试用本文算法方面做了许多工作．作为 SAQ 系统的应用实例之一，他设计了 BASIC 到 C 的语言转换器[6]，即在 SAQ 系统中定义 BASIC 到 C 的转换函数，该函数的定义域和值域分别为 BASIC 和 C 语言（当然该函数的定义中包含若干子函数的定义）．函数运行过程中频繁调用语法分析和语法树构造算法．最初，他用朴素表示法描述 BASIC 和 C 的语法定义，并在此基础上定义 BASIC 到 C 的转换函数．他随即发现，BASIC 和 C 的语法定义很繁琐，大量的分隔符干扰了函数定义过程．于是，作者提出本文的方法并予以实现．实验结果表明，新方法下，BASIC 和 C 语言的语法定义以及转换函数的定义相对简洁、自然，而且函数运行速度有所提高．陈自明运行的函数实例包括将若干个小 BASIC 程序转换成相应的 C 程序，这些小程序包括：角谷猜想试验、梵塔问题、八皇后问题、shell 排序等，函数运行速度提高幅度一般在 5～30%之间．通过实验，我们还发现，若函数参数（即 BASIC 源程序）中重复出现的 Token 越多，则新方法下函数运行速度提高幅度越大．

5　讨　论

本文所述算法有效地使用在 SAQ 系统[7]，进行程序语言等复杂概念的语法分析，一方面，词法分析和句法分析被结合到一个完整的语法分析过程中；另一方面，词法分析和句法分析在整个语法分析过程中相对分开进行．实验结果表明，用本文所述语法进行复杂概念的语法描述，相对简洁、自然，而且其对应的语法分析，与用通常的 CF 文法进行语法描述所对应的语法分析相比，在效率上也有所提高．

在例 1 中，求生成相应实例的产生式序列时，可以将 Token 型概念 I 接受 Token 12 和 34 的过程分别简化为一个产生式，如 I→12 和 I→34．实现这一点只需将算法 2 的第 9 和 13 行分别替换成 $\pi_i:=\pi\cup\{R_k\to t_k\}$ 和 $\pi_i:=\pi\cup\{R_k\to\lambda\}$．同时，为了提高语法树生成的效率，可以在算法 1 的状态中再增加一个域，以记忆状态之间的派生关系，从而可以在语法树生成阶段免去有关状态项的查找．

致谢 本文的工作是在董韫美院士的指导下完成的，在此表示感谢. 同时感谢审稿人对本文提出的诸多修改建议.

参考文献

1 Solomaa A. Formal Languages. London: Academic Press, 1973

2 Dong Yun-mei. Collection of SAQ Report no. 1～7. Technical Report ISCAS-LCS-95-09. Laboratory of Computer Science, Institute of Software, The Chinese Academy of Sciences, Aug. 1995

3 Dong Yun-mei *et al*. Collection of SAQ Report no. 8～16. Technical Report ISCAS-LCS-96-1. Laboratory of Computer Science, Institute of Software, The Chinese Academy of Sciences, Mar. 1996

4 Earley J. An efficient context-free parsing algorithm. Communications of the ACM, 1970, 13(2): 94～102

5 张瑞岭. 以上下文无关语言表示的概念的一个交互式学习过程的实现[硕士论文]. 中国科学院软件研究所, 1996
(Zhang Rui-ling. The implementation of an interactive learning procedure for acquisition of concepts represented as context-free languages[M. S. Thesis]. Institute of Software, The Chinese Academy of Sciences, 1996)

6 董韫美. 上下文无关语言上的递归函数. 技术报告 ISCAS-LCS-95-09. 中国科学院软件研究所计算机科学开放实验室, 1995
(Dong Yun-mei. Recursive functions defined on contex-free languages. SAQ Report no. 4. Technical Report ISCAS-LCS-95-09. Laboratory of Computer Science, Institute of Software, The Chinese Academy of Sciences, Aug. 1995)

7 Aho A V, Ullman J D. The Theory of Parsing, Translation and Compiling Vol. 1: Parsing. Prentice-Hall, Inc. 1972

8 陈自明. SAQ 系统用于程序语言的转换器[硕士论文]. 中国科学院软件研究所, 1997
(Chen Zi-min. A convert of programming languages——An application instance of SAQ[M. S. Thesis]. Institute of Software, The Chinese Academy of Sciences, 1997)

A Special Kind of Context Free Grammars and Their Parsing

ZHANG Rui-ling

(*Laboratory of Computer Science Institute of Software The Chinese Academy of Sciences Beijing* 100080)

Abstract SAQ is an experimental system to perform acquisition, verification and reusing of formal specification, in which the lexical and syntactic definitions of one concept should be integrated into one context-free grammar. If employed conventional context-free grammars to describe the overall definitions of complicated concepts such as natural languages and programming languages, separators such as spaces and carrier returns should be included and the definitions should be very messy. To solve this problem, a special kind of context-free grammars is presented. The grammars are obtained by dividing the set of non-terminals and the set of terminals of conventional context-free grammars into two respectively. As a result, the grammatical definitions of complicated concepts are relatively neat; at the same time, lexical analysis and syntax analysis can be integrated into one parsing process. In addition, the authors present the corresponding parsing and derivation tree construction algorithms, which are obtained on the basis of the general parsing method of Earley and its corresponding algorithm of construction of rightmost derivation respectively.

Key words Context-free grammar, parsing, derivation tree.