Задача разбора (синтаксический анализ)

Даны КС-грамматика G и цепочка x.

$x \in L(G)$?

Если да, то построить дерево вывода для x

(или левый вывод для x, или правый вывод для x).

Существуют различные методы синтаксического анализа для КС-грамматик;

для некоторых подклассов есть эффективные методы, затрачивающие линейное время на Cn на анализ цепочки длины n.

Каждый метод синтаксического анализа предполагает свой способ построения по грамматике программы-анализатора, которая будет осуществлять разбор цепочек.

В основе анализатора может быть автомат с магазинной памятью. Мы рассмотрим другой способ – метод рекурсивного спуска (система рекурсивных процедур).

Анализатор некорректен, если:

-- не распознает хотя бы одну цепочку, принадлежащую языку;

-- распознает хотя бы одну цепочку, языку не принадлежащую;

-- зацикливается на какой-либо цепочке.

Метод анализа *приме́ним* к данной грамматике, если анализатор, построенный в соответствии с этим методом, корректен.

Метод рекурсивного спуска (РС-метод)

Пример: *пусть дана грамматика $G = (\{a,b,c,d\}, \{S,A,B\}, P, S)$, где*

$$P:\quad S \to ABd$$
$$A \to a \mid cA$$
$$B \to bA$$

и надо определить, принадлежит ли цепочка cabad языку L(G).

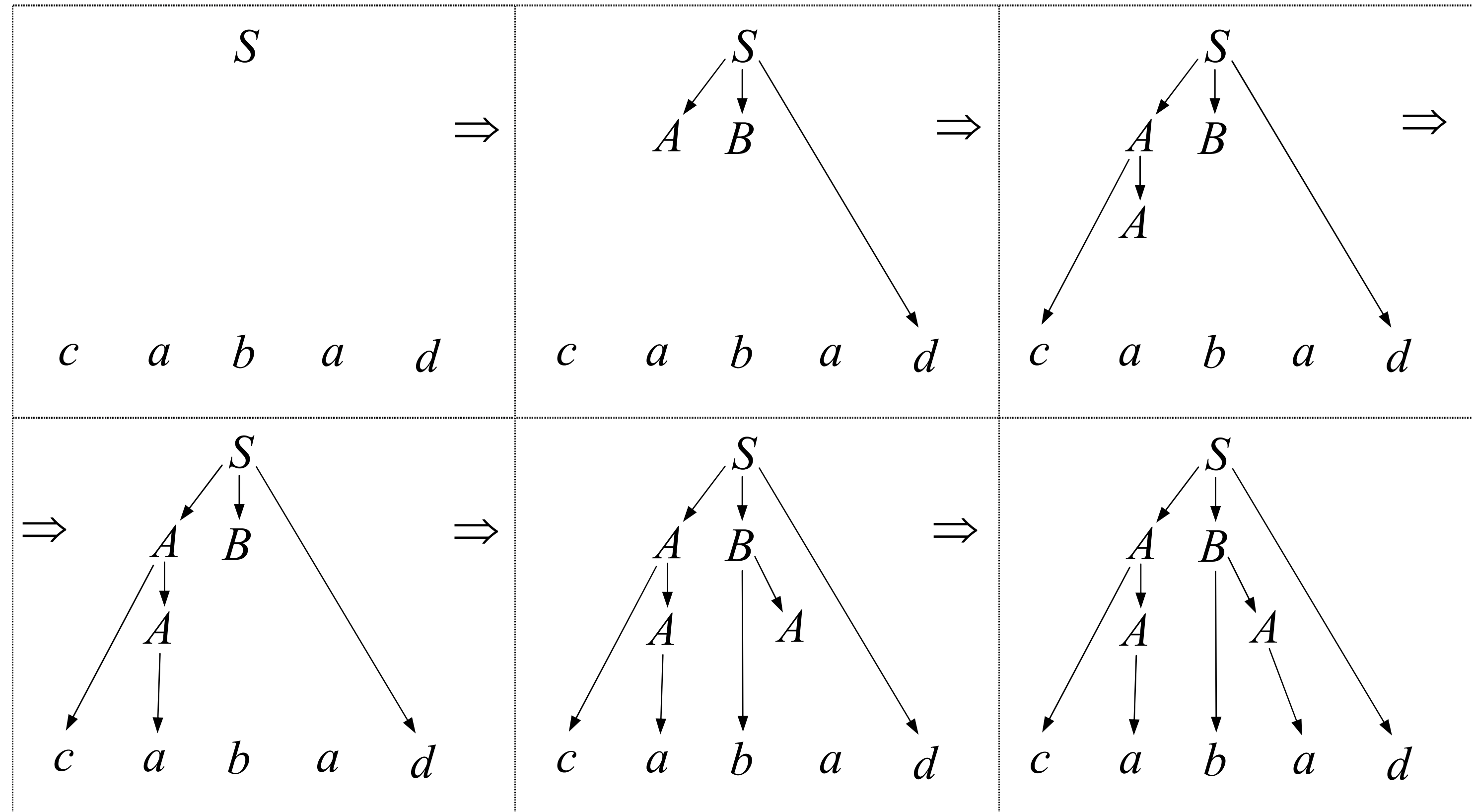
Построим левый вывод этой цепочки:

$S \to ABd \to cABd \to caBd \to cabAd \to cabad$

Следовательно, цепочка принадлежит языку *L(G)*.

$$S \to ABd \to cABd \to caBd \to cabAd \to cabad$$

Построение левого вывода эквивалентно построению дерева вывода методом «сверху вниз» (нисходящим методом) :



***Метод рекурсивного спуска (РС-метод)**:*

Для **каждого нетерминала** грамматики создается своя процедура с **именем** этого нетерминала; ее задача — начиная с указанного места исходной цепочки найти подцепочку, которая выводится из этого нетерминала.

Если подцепочку удалось найти, то работа процедуры считается нормально завершенной и осуществляется возврат в точку вызова, иначе — разбор прекращается и сообщается об ошибке, цепочка не принадлежит языку.

Тело каждой такой процедуры пишется непосредственно по правилам вывода соответствующего нетерминала: терминалы из правой части распознаются самой процедурой, а нетерминалы соответствуют вызовам процедур, носящих их имена.

Программа – анализатор для G_1

G_1:

$S \to ABd$

$A \to a \mid cA$

$B \to bA$

```
#include <iostream.h>

int  c;
void A ();
void B ();
void gc ()
{
   cin >> c;  // считать символ из входного потока
}

void S ()
{
   cout << "S-->ABd, ";      // применяемое правило вывода
   A();
   B();
   if ( c != 'd' )
      throw c;
}
```

```
void A ()
{
   if ( c =='a' )
   {
      cout << "A-->a, ";
      gc ();
   }
   else if ( c =='c' )
   {
      cout << "A-->cA, ";
      gc ();
      A ();
   }
   else
      throw c;
}
```

G_1:

$S \rightarrow ABd$

$A \rightarrow a \mid cA$

$B \rightarrow bA$

```
void B ()
{
   if ( c =='b' )
   {
      cout << "B-->bA, ";
      gc ();
      A ();
   }
   else
      throw c;
}
```

G_1:

$S \rightarrow ABd$

$A \rightarrow a \mid cA$

$B \rightarrow bA$

G_1:

$S \to ABd$

$A \to a \mid cA$

$B \to bA$

```
int main ()
{
   try
   {
      gc ();
      S ();
      if ( c != '⊥' ) // проверяем, что достигнут конец
                      //  цепочки
         throw c;
      cout << "SUCCESS !!!" << endl;
      return 0;
   }
   catch ( int c )
   {
      cout << "ERROR on lexeme" << c << endl;
      return 1;
   }
}
```

Достаточное условие применимости метода рекурсивного спуска

Для применимости метода рекурсивного спуска достаточно, чтобы каждое правило в грамматике имело вид:

(а) либо $X \rightarrow \alpha$,

где $\alpha \in (T \cup N)^*$ и это единственное правило вывода для этого нетерминала;

(б) либо $X \rightarrow a_1\alpha_1 \mid a_2\alpha_2 \mid \ldots \mid a_n\alpha_n$,

где $a_i \in T$ для всех $i = 1, 2, \ldots, n$; $a_i \neq a_j$ для $i \neq j$; $\alpha_i \in (T \cup N)^*$, т. е. если для нетерминала X правил вывода несколько, то они должны начинаться с терминалов, причем все эти терминалы должны быть различными;

Это условие не является необходимым.

Найдем критерий применимости

РС-метод примен*и*м, если и только если левый вывод (или дерево нисходящим способом) можно построить, начиная с начального символа S, так, что на каждом шаге вывода решение о том, какое правило (альтернативу) применять для замены левого нетерминала, безошибочно принимается по первому символу из непрочитанной части входной цепочки (т. е. по «текущему» символу).

Рассмотрим примеры

G_2:

$S \to aA \mid B \mid d$

$A \to d \mid aA$

$B \to aA \mid a$

G_2 неоднозначна, РС-метод неприменим.

нельзя дать однозначный прогноз, что делать на первом шаге при анализе цепочки, начинающейся с символа a (т. е. по текущему символу a невозможно сделать однозначный выбор: $S \to aA$ или $S \to B$)

G_3 однозначна, но РС-метод неприменим

G_3:

$S \to A \mid B$

$A \to aA \mid d$

$B \to aB \mid b$

Определение: *множество first* (α) — это множество терминальных символов, которыми начинаются цепочки, выводимые из цепочки α в грамматике $G = \langle T, N, P, S \rangle$, т. е.

$$first(\alpha) = \{ a \in T \mid \alpha \Rightarrow a\alpha', \text{ где } \alpha \in (T \cup N)^+,\ \alpha' \in (T \cup N)^* \}.$$

Например: $first(A) = \{ a, d \}$, $first(B) = \{ a, b \}$. Пересечение этих множеств непусто: $first(A) \cap first(B) = \{ a \} \neq \varnothing$, и поэтому метод рекурсивного спуска к G_3 неприменим.

Итак, **наличие в грамматике правил вида $X \to \alpha \mid \beta$, таких что $first(\alpha) \cap first(\beta) \neq \varnothing$, делает метод рекурсивного спуска неприменимым**.

Рассмотрим еще несколько примеров.

G_4:	
$S \to aA \mid BDc$	$first(aA) = \{ a \}$, $first(BDc) = \{ b, c \}$;
$A \to BAa \mid aB \mid b$	$first(BAa) = \{ a, b \}$, $first(aB) = \{ a \}$, $first(b) = \{ b \}$;
$B \to \varepsilon$	$first(\varepsilon) = \varnothing$;
$D \to B \mid b$	$first(B) = \varnothing$, $first(b) = \{ b \}$.

Метод рекурсивного спуска неприменим к грамматике G_4, так как $first(BAa) \cap first(aB) = \{ a \} \neq \varnothing$.

G_5:

$S \rightarrow aA$

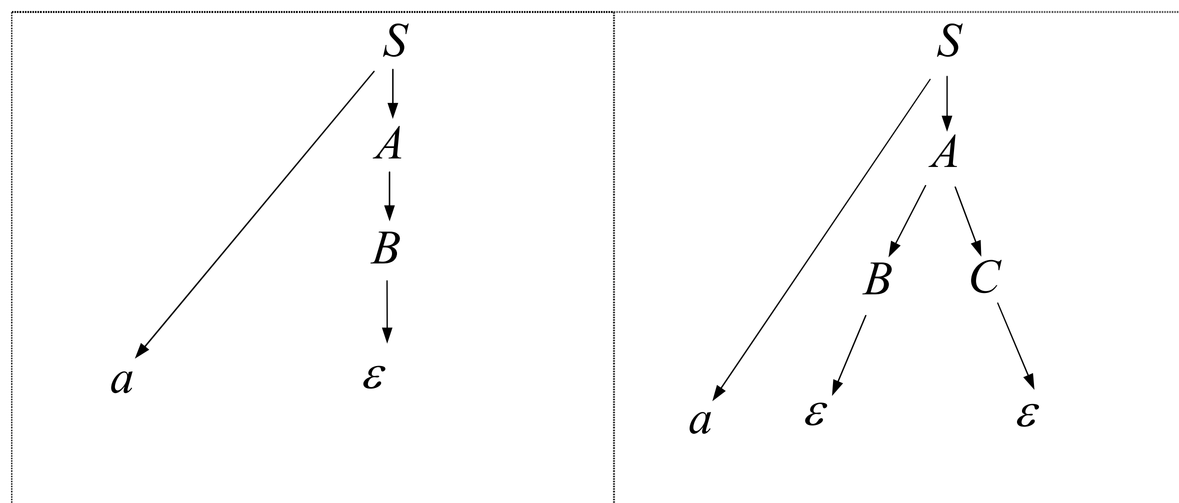
$A \rightarrow BC \mid B$

$C \rightarrow b \mid \varepsilon$

$B \rightarrow \varepsilon$

Пересечение множеств *first* пусто, но РС-метод непримен*и*м.

Действительно, $BC \Rightarrow \varepsilon$ и $B \Rightarrow \varepsilon$. Цепочка a имеет два различных дерева вывода



Таким образом, **если в грамматике для двух различных правил $X \to \alpha \mid \beta$ выполняются соотношения $\alpha \Rightarrow \varepsilon$ и $\beta \Rightarrow \varepsilon$, то метод рекурсивного спуска неприменим**.

Рассмотрим примеры с единственной альтернативой, из которой выводится ε.

G_6:

$S \to cAd \mid d$

$A \to aA \mid \varepsilon$

Метод применим: если текущий символ a , то выбираем альтернативу $A \to aA$ иначе — $A \to \varepsilon$

G_7:

$S \to Bd$

$B \to cAa \mid a$

$A \to aA \mid \varepsilon$

Неприменим, т.к. для A невозможно правильно выбрать альтернативу без «заглядывания» на символ вперед.

Определение: *множество follow(A)* — это множество терминальных символов, которые следуют за цепочками, выводимыми из A в грамматике $G = \langle T, N, P, S \rangle$, т. е.

$$follow(A) = \{\ a \in T \mid S \Rightarrow \alpha A \beta,\ \beta \Rightarrow a\,\gamma,\ A \in N,\ \alpha, \beta, \gamma \in (T \cup N)^{*} \}$$

Тогда, **если в грамматике есть пара правил $X \rightarrow \alpha \mid \beta$, таких что $\beta \Rightarrow \varepsilon$, $first(\alpha) \cap follow(X) \neq \varnothing$, то метод рекурсивного спуска неприменим к данной грамматике.**

Утверждение. Пусть G — КС-грамматика. Метод рекурсивного спуска применим к G, если и только если для любой пары альтернатив вида $X \rightarrow \alpha \mid \beta$ выполняются следующие условия:

(1) $first(\alpha) \cap first\ (\beta) = \varnothing$;

(2) справедливо не более чем одно из двух соотношений:

$\alpha \Rightarrow \varepsilon,\ \beta \Rightarrow \varepsilon$;

(3) если $\beta \Rightarrow \varepsilon$, то $first(\alpha) \cap follow(X) = \varnothing$.

Канонический вид для РС-метода

(1) либо $X \to \alpha$,
где $\alpha \in (T \cup N)^*$ и это единственное правило вывода для этого нетерминала;

(2) либо $X \to a_1\alpha_1 \mid a_2\alpha_2 \mid ... \mid a_n\alpha_n$,
где $a_i \in T$ для всех $i = 1, 2,..., n$; $a_i \neq a_j$ для $i \neq j$;
$\alpha_i \in (T \cup N)^*$, т. е. если для нетерминала X правил вывода несколько, то они должны начинаться с терминалов, причем все эти терминалы должны быть попарно различными;

(3) либо $X \to a_1\alpha_1 \mid a_2\alpha_2 \mid ... \mid a_n\alpha_n \mid \varepsilon$,
где $a_i \in T$ для всех $i = 1, 2,..., n$; $a_i \neq a_j$ для $i \neq j$;
$\alpha_i \in (T \cup N)^*$, и $first(X) \cap follow(X) = \varnothing$.

Этот вид удобен для построения рекурсивных процедур, но он дает только достаточное условие применимости.

Вопрос*: если грамматика не удовлетворяет критерию применимости РС-метода, то существует ли эквивалентная КС-грамматика, для которой метод рекурсивного спуска примен**и**м?*

К сожалению, нет алгоритма, отвечающего на этот вопрос для произвольной КС-грамматики*,* т.е. это ***алгоритмически неразрешимая проблема.***

Модификация метода для грамматик с итераторами:

Для правил вида $L \to a \{,a\}$

(*эквивалент* $L \to a \mid a, L$)

рекурсию заменяем итерацией:

```
void L()
{ if (c != 'a') throw c;
  gc();
 while (c == ',')
   {gc(); if (c != 'a') throw c; else
                                   gc();}
}
```

Важно, чтобы в любой сентенциальной форме после L не было запятой, иначе L прочитает «не свою» запятую. (Вместо запятой в данном примере может быть любой другой символ.)

Пример, когда анализатор по вышеприведенной схеме не дает корректный ответ:

G:

$S \to LB\bot$

$L \to a \{, a\}$

$B \to ,b$

Если для этой грамматики написать анализатор, действующий РС-методом, то цепочка *a,a,a,b* будет признана им ошибочной, хотя $a,a,a,b \in L(G)$.

В языках программирования после повторяющихся конструкций обычно идет какой-нибудь новый символ, так что подобных проблем не возникает:

var *a,b,c,d : integer;* или ***int*** *a,b,c,d;*

Если грамматику переписать без итератора { }

$S \to LB\perp$

$L \to a\ M$

$M \to ,\ a\ M \mid \varepsilon$

$B \to ,\ b$

то нетрудно видеть, что $first\ (,\ a) \cap follow\ (M\ ) = \{\ ,\ \} \neq \varnothing$, и поэтому метод рекурсивного спуска неприменим.

Эквивалентные преобразования для КС-грамматик, которые могут помочь перестроить исходную грамматику так, чтобы РС-метод был примен**и**м.

1) Если в грамматике есть нетерминалы, правила вывода которых **леворекурсивны**, т.е. имеют вид

$$A \to A\alpha_1 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m,$$

где $\alpha_i \in (V_T \cup V_N)^+$, $\beta_j \in (V_T \cup V_N)^*$, $i = 1, 2, \ldots, n; j = 1, 2, \ldots, m,$

то левую рекурсию всегда можно заменить правой:

$$A \to \beta_1 A' \mid \ldots \mid \beta_m A'$$
$$A' \to \alpha_1 A' \mid \ldots \mid \alpha_n A' \mid \varepsilon$$

Будет получена грамматика, эквивалентная исходной.

2) Если в грамматике есть нетерминал, у которого несколько правил вывода начинаются **одинаковыми терминальными символами**, т.е. имеют вид

$$A \to a\alpha_1 \mid a\alpha_2 \mid \ldots \mid a\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m,$$
$$\text{где } a \in V_T; \;\; \alpha_i, \beta_j \in (V_T \cup V_N)^*,$$

то можно преобразовать правила вывода данного нетерминала, объединив правила с общими началами в одно правило:

$$A \to aA' \mid \beta_1 \mid \ldots \mid \beta_m$$
$$A' \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$$

Будет получена грамматика, эквивалентная исходной.

3) Если в грамматике есть нетерминал, у которого **несколько** правил вывода, и среди них есть правила, **начинающиеся нетерминальными символами**, т.е. имеют вид

$$A \to B_1\alpha_1 \mid \dots \mid B_n\alpha_n \mid a_1\beta_1 \mid \dots \mid a_m\beta_m$$

$$B_1 \to \gamma_{11} \mid \dots \mid \gamma_{1k}$$

. . .

$B_n \to \gamma_{n1} \mid \dots \mid \gamma_{np}$, *где* $B_i \in N;$ $a_j \in T;$ $\alpha_i, \beta_j, \gamma_{ij} \in (T \cup N)^*$,

то можно заменить вхождения нетерминалов B_i их правыми частями из правил вывода:

$$A \to \gamma_{11}\alpha_1 \mid \dots \mid \gamma_{1k}\alpha_1 \mid \dots \mid \gamma_{n1}\alpha_n \mid \dots \mid \gamma_{np}\alpha_n \mid a_1\beta_1 \mid \dots \mid a_m\beta_m$$

Будет получена грамматика, эквивалентная исходной.

4) Если в грамматике есть правила

$A \rightarrow \alpha_1 A \mid \ldots \mid \alpha_n A \mid \beta_1 \mid \ldots \mid \beta_m \mid \varepsilon$

$B \rightarrow \alpha A \beta$

и $FIRST(A) \cap FOLLOW(A) \neq \varnothing$ (из-за вхождения A в правило вывода для B), то можно преобразовать их в такие:

$B \rightarrow \alpha A'$

$A' \rightarrow \alpha_1 A' \mid \ldots \mid \alpha_n A' \mid \beta_1 \beta \mid \ldots \mid \beta_m \beta \mid \beta$

Полученная грамматика будет эквивалентна исходной.

Задача разбора (синтаксический анализ) для неоднозначных грамматик

две постановки задачи:

(1) *Даны КС-грамматика G и цепочка x. Требуется проверить:* $x \in L(G)$*? Если да, то построить все деревья вывода для x (или все левые выводы для x, или все правые выводы для x)*

Для решения этой задачи можно обобщить метод рекурсивного спуска, чтобы он работал с возвратами, пробуя различные подходящие альтернативы.

(2) *Даны КС-грамматика G и цепочка x. Требуется проверить:* $x \in L(G)$*? Если да, то построить одно дерево вывода для x (возможно, «наиболее подходящее» в некотором смысле).*

Рассмотрим пример. Грамматика неоднозначна. РС-метод неприменим.

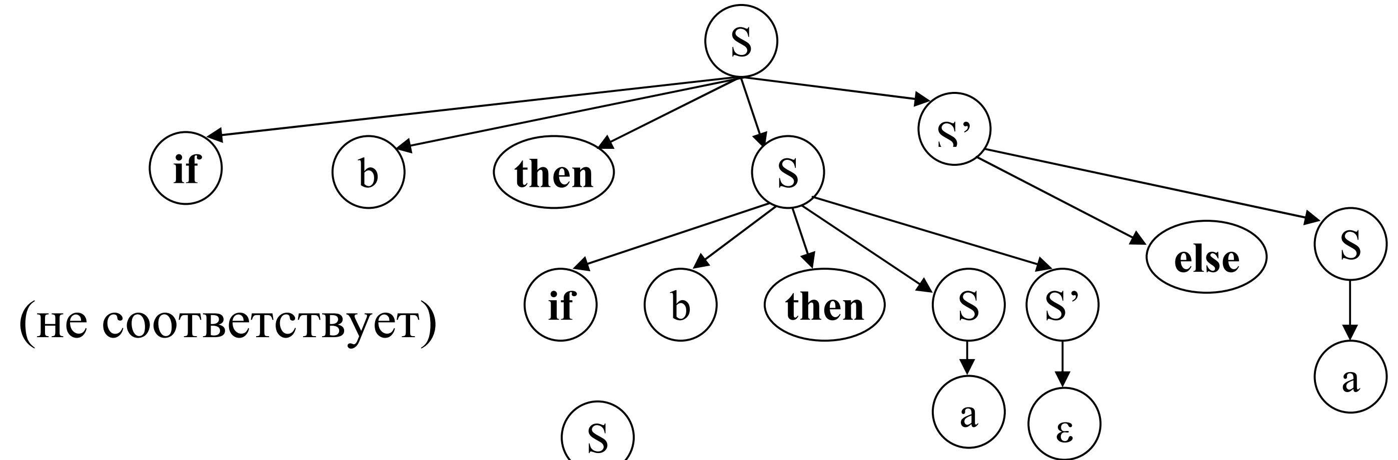
G = ({**if**, **then**, **else**, a, b}, {S}, P, S, S'),

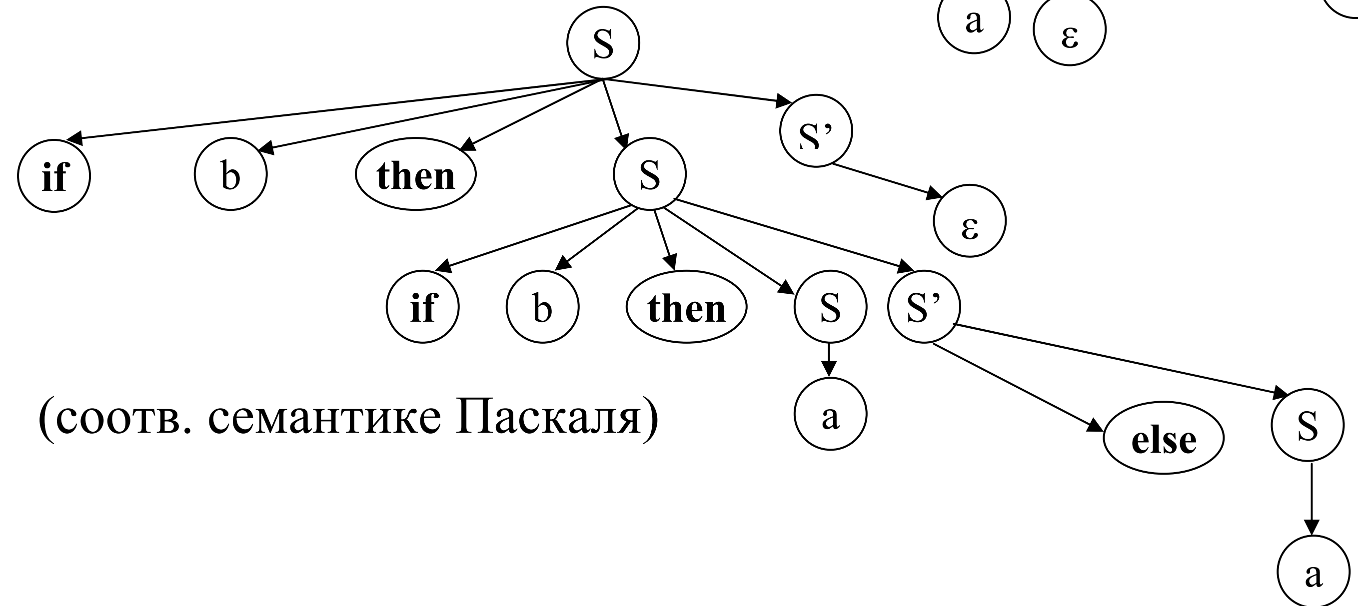
где P = {S → **if** b **then** S S' | a;

S' → **else** S | ε }.

В этой грамматике для цепочки **if** *b* **then** **if** *b* **then** *a* **else** *a* можно построить два различных дерева вывода:

Одно из них соответствует семантике Паскаля: **else** относится к ближайшему i**f**. Такое дерево можно получить, написав РС-процедуры, где S' по возможности отдает предпочтение непустой альтернативе.


S
if
b
then
S
S'
if
b
then
S
S'
else
S
a
ε
a
(не соответствует)


S
if
b
then
S
S'
ε
if
b
then
S
S'
a
else
S
a
(соотв. семантике Паскаля)

Синтаксический анализатор для М-языка

Будем считать, что синтаксический и лексический анализаторы взаимодействуют следующим образом: анализ исходной программы идет под управлением синтаксического анализатора; если для продолжения анализа ему нужна очередная лексема, то он запрашивает ее у лексического анализатора; тот выдает одну лексему и "замирает" до тех пор, пока синтаксический анализатор не запросит следующую лексему.

Соглашения:

- лексический анализатор — это функция-член класса Scanner — get_lex (), которая в качестве результата выдает лексемы типа (class) Lex;
- в переменной Lex curr_lex будем хранить текущую лексему, выданную лексическим анализатором, (а в переменной c_val – ее значение, в c_type – ее тип, это пригодится на этапе семантического анализа.)

Грамматика модельного языка

P → **program** D1; B⊥

D1 → **var** D {,D}

D → I {,I}: [**int** | **bool**]

B → **begin** S {;S} **end**

S → I := E | **if** E **then** S **else** S | **while** E **do** S | B | **read** (I) | **write** (E)

E → E1 [= | < | > | <= | >= | !=] E1 | E1

E1 → T {[+ | - | *or*] T}

T → F {[* | / | *and*] F}

F → I | N | L | *not* F | (E)

L → **true** | **false**

I → a | b| ...| z | Ia | Ib |...| Iz | I0 | I1 |...| I9

N → 0 | 1 |...| 9 | N0 | N1 |...| N9

```
class Parser {
        Lex curr_lex;
        type_of_lex c_type;
        int c_val;
        Scanner scan;
        // Stack < int, 100 > st_int;
        // Stack < type_of_lex, 100 >  st_lex;
        void P(); void D1(); void D(); void B();void S();
        void E();  void E1();  void T();  void F();

         void gl (){
            curr_lex = scan.get_lex();
            c_type = curr_lex.get_type();
         // c_val = curr_lex.get_value();
        }
  public:
    // Poliz  prog;
Parser (const char *program):scan(program)//,prog (1000)
    {}
    void analyze();
};
```

```
void Parser::analyze () {
        gl();
        P();
    // prog.print();
        cout << endl << "OK" << endl;
}


void Parser::P () {
    if (c_type == LEX_PROGRAM) gl();
    else throw curr_lex;
    D1();
    if (c_type == LEX_SEMICOLON) gl();
    else throw curr_lex;
    B();
    if (c_type != LEX_FIN) throw curr_lex;
}
```

```
void Parser::D1 () {
    if (c_type == LEX_VAR){
        gl();
        D();
        while (c_type == LEX_COMMA){ gl(); D();}
    }
    else throw curr_lex;
}
...
```

Семантический анализ

Контекстно-свободные грамматики, с помощью которых описывают синтаксис языков программирования, не позволяют задавать контекстные условия, имеющиеся в любом языке.

Примеры наиболее часто встречающихся контекстных условий:

(а) каждый используемый в программе идентификатор должен быть описан, но не более одного раза в одной зоне описания;

(б) при вызове функции число фактических параметров и их типы должны соответствовать числу и типам формальных параметров;

(в) обычно в языке накладываются ограничения на типы операндов любой операции, определенной в этом языке, на типы левой и правой частей в операторе присваивания, на тип параметра цикла, на тип условия в операторах цикла и условном операторе и т.п.

Семантический анализатор для М-языка

Контекстные условия, выполнение которых надо контролировать в программах на М-языке:

1. Любое имя, используемое в программе, должно быть описано и только один раз.
2. В операторе присваивания типы переменной и выражения должны совпадать.
3. В условном операторе и в операторе цикла в качестве условия возможно только логическое выражение.
4. Операнды операции отношения должны быть целочисленными.
5. Тип выражения и совместимость типов операндов в выражении определяются по обычным правилам (как в Паскале).

Проверку контекстных условий совместим с синтаксическим анализом. Для этого в синтаксические правила вставим вызовы процедур, осуществляющих необходимый контроль, а затем перенесем их в процедуры рекурсивного спуска.

Обработка описаний

Лексический анализатор запомнил в таблице идентификаторов *TID* все идентификаторы-лексемы, которые были обнаружены в тексте исходной программы. Информация о типе переменных и о наличии их описания заносится в ту же таблицу.

i-ая строка таблицы *TID* соответствует идентификатору-лексеме вида (*LEX_ID*, *i*).

Лексический анализатор заполнил поле name; значения полей declare и type заполняются на этапе семантического анализа.

Раздел описаний имеет вид:

$D \rightarrow I\ \{,I\}: [int \mid bool]$,

т.е. имени типа (*int* или *bool*) предшествует список идентификаторов. Эти идентификаторы (вернее, номера соответствующих им строк таблицы *TID*) надо запоминать (мы будем их запоминать в стеке целых чисел ***Stack<int,100> st_int***), а когда будет проанализировано имя типа, надо заполнить поля declare и type в этих строках.

Функция ***void Parser::dec (type_of_lex type)*** считывает из стека номера строк таблицы *TID*, заносит в них информацию о типе соответствующих переменных, о наличии их описаний и контролирует повторное описание переменных.

```
void Parser::dec ( type_of_lex type ) {
    int i;
    while ( !st_int.is_empty()) {
        i = st_int.pop();
        if ( TID[i].get_declare() ) throw "twice";
        else {
            TID[i].put_declare();
            TID[i].put_type(type);
        }
    }
}
```

С учетом имеющихся функций правило вывода с действиями для обработки описаний будет таким:

D → *<st_int.reset()> I < st_int.push (c_val) >*
{, I < st_int.push (c_val) >}:
[int < dec (LEX_INT) > | bool < dec (LEX_BOOL) >]

Контроль контекстных условий в выражении

Типы операндов и обозначение операций будем хранить в стеке ***Stack<type_of_lex, 100> st_lex***.

Если в выражении встречается лексема-целое_число или логические константы ***true*** или ***false***, то соответствующий тип сразу заносится в стек.

Если операнд — лексема-переменная, то необходимо проверить, описана ли она; если описана, то ее тип надо занести в стек. Эти действия можно выполнить с помощью функции check_id:

```
void parser::check_id(){
    if(TID[c_val].get_declare())
        st_lex.push(TID[c_val].get_type());
    else throw "not declared";
}
```

Для контроля контекстных условий каждой тройки — "операнд-операция-операнд" (для проверки соответствия типов операндов данной двуместной операции) будем использовать функцию *check_op*:


```
void Parser::check_op () {
    type_of_lex t1, t2, op, t = LEX_INT, r = LEX_BOOL;
    t2 = st_lex.pop();
    op = st_lex.pop();
    t1 = st_lex.pop();
    if(op==LEX_PLUS ||op==LEX_MINUS ||op==LEX_TIMES
||op==LEX_SLASH)
        r = LEX_INT;
    if (op == LEX_OR || op == LEX_AND)
        t = LEX_BOOL;
    if (t1 == t2  &&  t1 == t) st_lex.push(r);
    else throw "wrong types are in operation";
    prog.put_lex (Lex (op) );
}
```

Для контроля за типом операнда одноместной операции *not* будем использовать функцию *check_not*:

```
void Parser::check_not () {
   if (st_lex.pop() != LEX_BOOL)
      throw "wrong type is in not";
   else {
      st_lex.push (LEX_BOOL);
      prog.put_lex (Lex (LEX_NOT));
   }
}
```

В грамматике модельного языка задано старшинство операций: наивысший приоритет имеет операция отрицания, затем в порядке убывания приоритета — группа операций умножения (*, /, and), группа операций сложения (+,-,or), операции отношения.

E → E1 | E1 [= | < | >] E1
E1 → T {[+ | - | or] T}
T → F {[* | / | and] F}
F → I | N | [true | false] | not F | (E)

Именно это свойство грамматики позволит провести синтаксически-управляемый контроль контекстных условий.

Правила вывода выражений модельного языка с действиями для контроля контекстных условий:

E→E1|E1 [= | < | >] <st_lex.push(c_type)>E1<check_op()>
E1→T { [+ | - | or] <st_lex.push (c_type)>T<check_op()>}
T→F { [* | / | and] <st_lex.push (c_type)>F<check_op()>}
F → I < check_id() > | N <st_lex.push (LEX_INT)> |
[true | false] <st_lex.push (LEX_BOOL)> | not F < check_not() > | (E)

Контроль контекстных условий в операторах

$$S \rightarrow I := E \mid if\ E\ then\ S\ else\ S \mid while\ E\ do\ S \mid B \mid read\ (I) \mid write\ (E)$$

1) **Оператор присваивания** *I := E*

Контекстное условие: в операторе присваивания типы переменной I и выражения E должны совпадать.

В результате контроля контекстных условий выражения E в стеке останется тип этого выражения (как тип результата последней операции); если при анализе идентификатора I проверить, описан ли он, и занести его тип в тот же стек (для этого можно использовать функцию check_id()), то достаточно будет в нужный момент считать из стека два элемента и сравнить их:

```
void Parser::eq_type () {
    if (st_lex.pop() != st_lex.pop())
        throw "wrong types are in :=";
}
```

Правило для оператора присваивания:

I <check_id()> := E <eq_type()>

2) Условный оператор и оператор цикла

if E then S else S | while E do S

Контекстные условия: в условном операторе и в операторе цикла в качестве условия возможны только логические выражения.

В результате контроля контекстных условий выражения E в стеке останется тип этого выражения (как тип результата последней операции); следовательно, достаточно извлечь его из стека и проверить:

```
void Parser::eq_bool () {
    if ( st_lex.pop() != LEX_BOOL )
        throw "expression is not boolean";
}
```

Правила для условного оператора и оператора цикла будут такими:

if E <eq_bool()> then S else S | while E <eq_bool()> do S

3) Для поверки операнда **оператора ввода** read (I) можно использовать следующую функцию:

```
void Parser::check_id_in_read () {
    if ( !TID [c_val].get_declare() ) throw "not
declared";
}
```

Правило для оператора ввода будет таким:

read (I < check_id_in_read()>).

В итоге получаем процедуры для синтаксического анализа методом рекурсивного спуска с синтаксически управляемым контролем контекстных условий, которые легко написать по правилам грамматики с действиями.

В качестве примера приведем функцию для нетерминала *D* (раздел описаний):

```
void Parser::D () {
    st_int.reset();
    if (c_type != LEX_ID) throw curr_lex;
    else {
        st_int.push ( c_val );
        gl();
        while (c_type == LEX_COMMA){
            gl();
            if (c_type != LEX_ID) throw curr_lex;
            else {
                st_int.push ( c_val ); gl();
            }
        }
        if (c_type != LEX_COLON) throw curr_lex;
        else {gl();
          if (c_type == LEX_INT) {dec ( LEX_INT ); gl();}
            else
              if (c_type == LEX_BOOL){dec ( LEX_BOOL );
                                                     gl();}
                else throw curr_lex;
}}}
```