Синтактичен анализ

На етапа на синтактичен анализ компилаторът проверява дали последователността от лексеми (tokens), получени при лексическия анализ, представляват валидно изречение в граматиката на програмния език.

Съществуват два основни метода за синтактичен анализ:

- *Отгоре-надолу (top-down* parsing) - започваме от стартовия символ и прилагаме правилата до достигане на търсения символен низ.
- *Отдолу-нагоре (bottom-up* parsing) - започваме от символния низ и го редуцираме чрез прилагане на правилата до достигане на стартовия символ.

Примери

Дадена е граматика G, която разпознава низове, съдържащи произволен брой **a**, следвани от поне едно (или повече) **b**:

S –> AB

A –> aA | ε

B –> b | bB

Пример: отгоре-надолу (top-down)

Анализ отгоре-надолу за пораждане на низ **aaab**:

Започваме със стартовия символ и на всяка стъпка заместваме един от нетерминалните символи в низа с лявата страна на някое от правилата. Повтаряме това до получаване на низ, съставен само от терминални символи. При анализа отгоре-надолу низът (изречението) се получава отляво надясно.

S	
AB	S –> AB
aAB	A –> aA
aaAB	A –> aA
aaaAB	A –> aA
aaaεB	A –> ε
aaab	B –> b

Пример: отдолу-нагоре (bottom-up)

Анализът отдолу-нагоре работи в обратен ред. Започваме с изречение от терминални символи и на всяка стъпка прилагаме правило на обратно (т.е. отдясно-наляво), замествайки част от низа с нетерминален символ в лявата част на правилото. Продължаваме до получаване на стартовия символ в граматиката. При този анализ изречението се получава отдясно наляво.

aaab	
aaaεb	(insert ε)
aaaAb	A –> ε
aaAb	A –> aA
aAb	A –> aA
Ab	A –> aA
AB	B –> b
S	S –> AB

При създаване на синтактичен анализатор в компилатора обикновено са наложени ограничение за обработване на входа. В разгледаните примери е лесно да решаваме кое правило да приложим на всяка стъпка, защото виждаме целия низ aaab.

На практика, обаче, входният низ се получава като поток от символи, т.е. символ по символ. **Обикновено сме ограничени да виждаме само един символ напред.** Виждаме само следващия символ във входния поток (*lookahead*). Това ограничение създава проблем при реализацията на синтактичен анализатор.

Например в разглежданата граматика, ако анализаторът „вижда“ само пореден прочетен символ **b,** той не може да „избере“ точно кое правило да приложи.

B –> b or B –> bB.

Връщане назад (backtracking)

Едно от възможните решения за реализация на синтактичен анализ е връщане назад (*backtracking).* При него анализаторът избира да приложи едно правило. Ако след този избор се окаже, че заместването на символите не може да продължи (т.е. няма подходящо правило), анализаторът се връща по входния низ и започва отново, избирайки различно правило.

Пример:

Дадена е следната проста граматика:

S –> bab | bA

A –> d | cA

Разглеждаме синтактичен анлиз на изречение bcd.

Лявата колона показва полученият низ на всяка стъпка, в средната колона е оставащият входен поток, а в дясната колона в правилото, което се прилага на съответната стъпка.

S	bcd	Опитваме S –> bab
bab	bcd	съвпада b
ab	cd	блокировка (dead-end), връщане назад
S	bcd	опитваме S –> bA
bA	bcd	съвпада b
A	cd	опитваме A –> d
d	cd	dead-end, връщане назад
A	cd	опитваме A –> cA
cA	cd	съвпада c
A	d	опитваме A –> d
d	d	съвпада d

Успех!

При всяко срещане на dead-end се връщаме до последното приложено правило, където прилагаме различно правило. Ако всички алтернативни правила са били проверени и водят до dead-end, се връщаме до предишното правило и т.н. Това продължава до получаване на входния низ или до изчерпване на всички комбинации от правила без успех.

Анализ отгоре-надолу с предсказване

Фокусираме се върху анализ отгоре-надолу. Ще разгледаме начин за реализация на синтактичен анализатор без връщане назад, който се нарича *предсказващ*.

Предсказващият анализатор избира кое правило да приложи само на базата на следващия входен символ и на текущо обработвания нетерминал. За да бъде възможно това обаче, граматиката трябва да има определена форма.

Граматика с такава форма се означава като *LL(1).* Първата буква *"L“* означава, че входният поток се сканира отляво-надясно; второто "L" означава прилагане на правилата отляво-надясно; а „1“ означава поглеждане с един символ напред във входния поток.

В ГРАМАТИКА LL(1) НЕ МОЖЕ ДА ИМА *ЛЯВО РЕКУРСИВНИ* ПРАВИЛА. Това е необходимо, но не достатъчно условие за LL(1) граматика. Съществуват граматики без ляво рекурсивни правила, които не са LL(1). Също така съществуват граматики, които не могат да бъдат преобразувани в LL(1). В такива случаи трябва да бъдат използвани други методи за синтактичен анализ или да бъдат реализирани специални правила.

Рекурсивно спускане

Първата техника за реализация на синтактичен анализатор с предсказване се нар. *рекурсивно спускане*. Анализатор с рекурсивно спускане се състои от множество малки функции, по една за всеки нетерминален символ в граматиката. При разпознаване на изречение извикваме функциите, съответстващи на нетерминала от лявата страна на правилото, което прилагаме. Ако правилото е рекурсивно, функцията се извиква рекурсивно.

Нека са дадени правила от граматиката на опростен програмен език. В този програмен език всички функции започват с ключова дума FUNC:

program –> function_list

function_list –> function_list function | function

function –> FUNC identifier (parameter_list) statements

- Примерна реализация на C за разпознаване на функция в съответствие с горната граматика има следното действие: очаква първо лексема FUNC, след това идентификатор (име на функцията), последван от отваряща скоба и т.н. Всеки прочетен символ от входния поток се сравнява с очаквания според съответното правило. Ако не съответства, се извежда съобщение за грешка.
- За всеки нетерминален символ от правилото се извиква съответна функция, която го разпознава.

```
void  ParseFunction() {
if (lookahead != T_FUNC) {     // всеки символ, различен от FUNC е грешка
      printf("syntax error \n");
      exit(0);
}
else
      lookahead = nextoken();        // глобална пром. 'lookahead' съдържа
                                     // следващата лексема
ParseIdentifier();
if (lookahead != T_LPAREN) {
      printf("syntax error \n");
      exit(0);
}
else
       lookahead = nextoken();
ParseParameterList();
if (lookahead!= T_RPAREN) {
      printf("syntax error \n");
      exit(0);
}
else
       lookahead = nextoken();
ParseStatements();
}
```

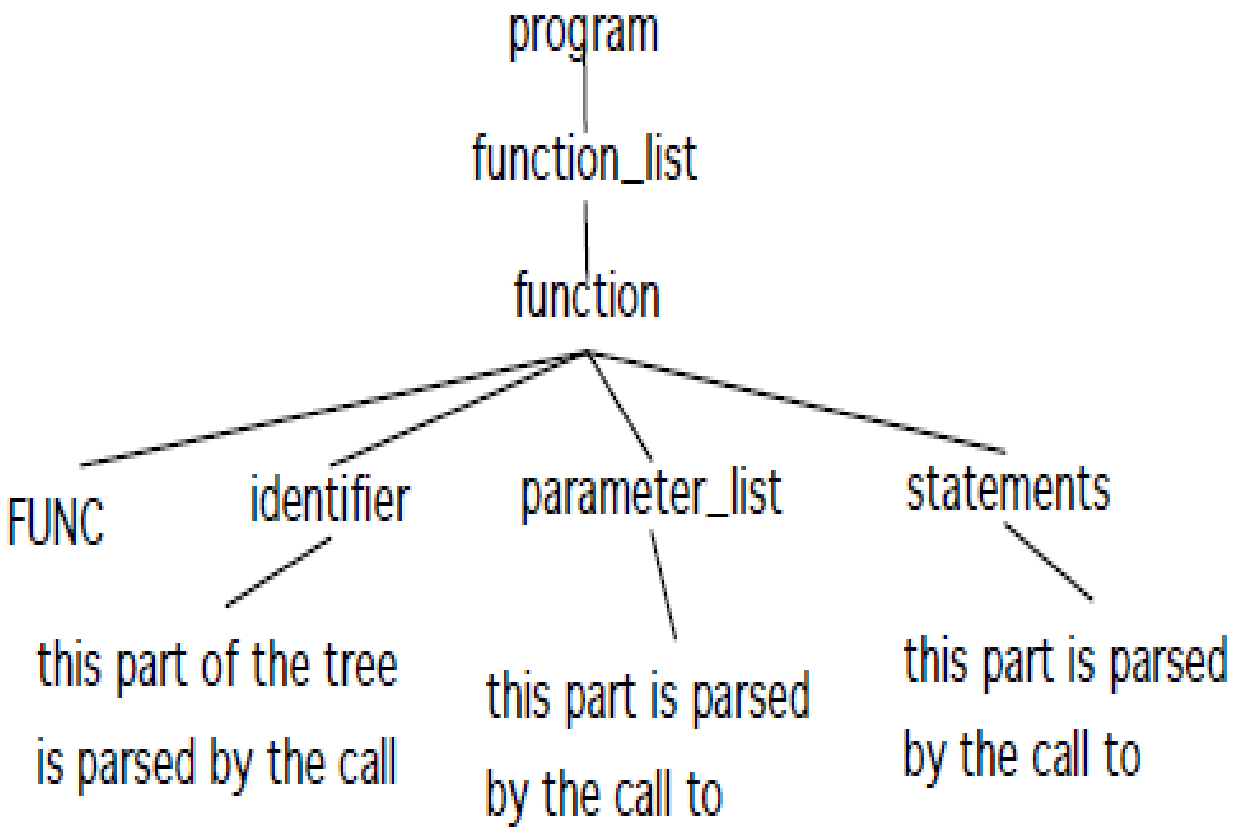
За по-ясно представяне на показаната реализация въвеждаме помощна функция, която проверява дали поредният прочетен символ съвпада с очаквания в правилото и извежда грешка в противен случай. Тази функция ще бъде използвана във всички функции на анализатора.

```
void MatchToken(int expected)
{
if (lookahead != expected) {
    printf("syntax error, expected %d, got %d\n",
            expected,lookahead);
    exit(0);
}   else // ако съответства, чете следващата лексема
lookahead = nextoken();
}
```

С използване на тази допълнителна функция **ParseFunction** изглежда по следния начин:

```
void ParseFunction()
{
   MatchToken(T_FUNC);
   ParseIdentifier();
   MatchToken(T_LPAREN);
   ParseParameterList();
   MatchToken(T_RPAREN);
   ParseStatements();
}
```

Следващата диаграма илюстрира построяване на синтактичното дърво:



Какво се случва, ако имаме множество алтернативни клонове в дясната страна направило? Например:

statement –> assg_statement | return_statement | print_statement | null_statement | if_statement | while_statement | block_of_statements

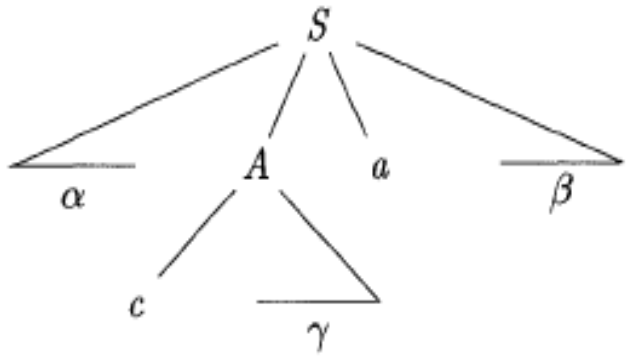
Проблемът при реализацията на функцията **ParseStatement** е коя от всичките седем възможности съответства на определен входен символ. Целта е да определим съответстващия клон от правилото без връщане назад и **поглеждане напред само с един символ**, т.е. изборът на клон от правило трябва да бъде направен с минимална информация.

Дефиниции

При построяване на синтактичен анализатор (независимо дали е отгоре-надолу или отдолу-нагоре) се използват две множества, *First* и *Follow*, свързани с граматиката *G*. **При анализ отгоре-надолу, *First* и *Follow* позволяват да изберем кое следващо правило да приложим на базата на следващия прочетен символ.** Тези две множества се използват и при възстановяване след синтактична грешка.

First(α), където **α** е низ от символи от граматиката, представлява множество от терминални символи, с които започва **α**. Ако **α** $\rightarrow^*$ **ε**, то **ε** принадлежи на **First (α)**.

Например, ако **A** $\rightarrow^*$ **cγ**, то **c** принадлежи на **First (A)**



First може да се използва при синтактичен анализ с предсказване.

Пример:

Дадено е правило за A с два клона: $A \rightarrow \alpha \mid \beta$, където First(**α**) и First(β) са непресичащи се множества.

$$First(\alpha) \cap First(\beta) = \emptyset \ldots\ldots\ldots\ldots \text{(правило 1)}$$

Изборът на правило може да се направи чрез следващия символ **a**, който може да принадлежи само на едно от множествата First (α) или First(β), но не и на двете.

Например, ако **a** принадлежи на First(**α**), избираме правилото $A \rightarrow$ **α**.

Нека е дадено правило за A с множество алтернативи: A –> u_1 | u_2 | ... . За да реализираме функция за синтактичен анализ с рекурсивно спускане, **е необходимо всички множества First(u_i) да не се пресичат**. Общата форма на такава функция е:

```
void ParseA() {
     // case below not quite legal C, need to list symbols individually
    switch (lookahead) {
    case First(u1):   // избор на правило A -> u1
                    /* код за разпознаване на u1 */
                    return;
    case First(u2): // избор на правилото A -> u2
                   /* код за разпознаване на u2 */
                    Return;
     ....
     default:
                    printf("syntax error \n");
                    exit(0);
   }
}
```

За изчисляване на First(X) за всички нетерминални символи X от граматиката, прилагаме следните правила докато нови терминални символи или ε могат да бъдат добавени към множеството First.

1. Ако X е терминал, то First(X) = {X}.
2. Ако X е нетерминал $X \rightarrow Y_1 Y_2 \ldots$, където Y_k е правило за *k >=1,* записваме **a** в First(X), ако за някоя стойност на **i**, **a** принадлежи на First(Yi), а **ε** е във всяко от множествата First(Y1) ., . , First(Y_{i-1}); т.е., $Y_1 \ldots Y_{i-1} \rightarrow \varepsilon$. Ако ε принадлежи на First(Y_j) за всяко j = 1,2, . . . , *k,* тогава добавяме ε към First(X).

   Например всеки символ от First(Y_1) със сигурност принадлежи на First(X). Ако Y_1 не извежда **ε**, тогава не добавяме нови символи към First(X), но ако $Y_1 \rightarrow \varepsilon$, тогава добавяме First($Y_2$) и т.н.
3. Ако $X \rightarrow \varepsilon$ е правило, тогава добавяме ε към First(X).

Изчисляването на First за всеки низ X_1X_2 , . . X_n става по следния начин.

Добавяме към First(X_1X_2 . . . X_n) всички символи различни от ε, които принадлежат на First(X_1).

Ако ε принадлежи на First(X_1), добавяме символите различни от ε, принадлежащи на First(X_2).

Ако ε принадлежи на First(X_1) и на First(X_2), добавяме символите различни от ε, принадлежащи на First(X_3).

и т.н.

Накрая добавяме ε към First(X_1X_2 . . X_n), ако за всяко i ε принадлежи на First(X_i).

Множеството FOLLOW за нетерминалния символ A се определя като множество от терминални символи, които могат да следват отдясно след A във валидно изречение.

За всяко валидно изречение S =>* uAv, където v започва с терминален символ, който принадлежи на Follow(A):

First(A) ∩ Follow(A) = Ø ……. (Правило 2)

Неформално множеството FOLLOW може да се разбира по следния начин: **A** може да се появява на различни места в едно валидно изречение. Множеството FOLLOW описва какви терминали могат да следват низа, който се разширява от **A**. Множеството FOLLOW определя десния контекст на даден нетерминал.

Използвайки тези две дефиниции (FIRST и FOLLOW), можем да обобщим как се обработва правило от вида A –> u_1 | u_2 | ..., в синтактичен анализатор с рекурсивно спускане. Във всички случаи трябва да бъде обработен всеки клон от First(u_i). Ако съществува нетерминал u_i , който извежда ε (т.е. празен низ), тогава трябва да бъдат обработени символите от Follow(A).

```
void ParseA() {
    switch (lookahead) {
    case First(u1):
                    /* код за разпознаване на u1 */
                    return;
    case First(u2):
                    /* код за разпознаване на u2 */
                    return;
            ...
    case Follow(A):    // избор на правило A->epsilon
                    /* usually do nothing here */
    default:
                    printf("syntax error \n");
                    exit(0);
    }
}
```

Изчисляване на множеството Follow

За всеки нетерминал в граматиката се изпълнява следното:

1. Записваме EOF в Follow(S), където S е стартовият символ, а EOF е маркер за край на входния поток. Маркерът може да бъде край на файл, нов ред или специален символ, който е очакван за край в използваната граматика. Ще използваме символ $ за край.
2. За всяко правило **A –> uBv ,** където **u** и **v** са низове от символи от граматиката, а **B** е нетерминал, всички символи от **First(v)** без **ε** се записват в **Follow(B)**.
3. За всяко правило **A –> uB** или **A –> uBv ,** където First(v) съдържа **ε** (т.е. v е празен) всички символи от Follow(A) се добавят към Follow(B).

Лява рекурсия

Граматика, която съдържа ляво рекурсивни правила, не е LL(1) граматика.

Пример за ляво рекурсивни правила:

A –> Aα

A –> α

Пример 2: Разглеждаме ляво рекурсивно правило за списък от една или повече функции:

function_list –> function_list function | function

function –> FUNC identifier (parameter_list) statement

Реализацията на това правило ще доведе до безкрайна рекурсия!

```
void ParseFunctionList()
{
ParseFunctionList();
ParseFunction();
}
```

Премахване на лява рекурсия

Необходимо е да премахнем лявата рекурсия. Тя може да бъде заменена с включване на допълнителни нетерминали или конвертиране в дясна рекурсия, т.е.:

A –> α A

A –> α

За да реализираме разпознаваща функция за правило function_list:

function_list –> function_list function | function

, заместваме :

function_list –> function function_list | function

, при което разпознаващата функция добива следния вид:

```
void ParseFunctionList()
{
ParseFunction();
ParseMoreFunctions();      // може да бъде отсъства(т.е.
                           //разширяваме с празен низ)
}
```