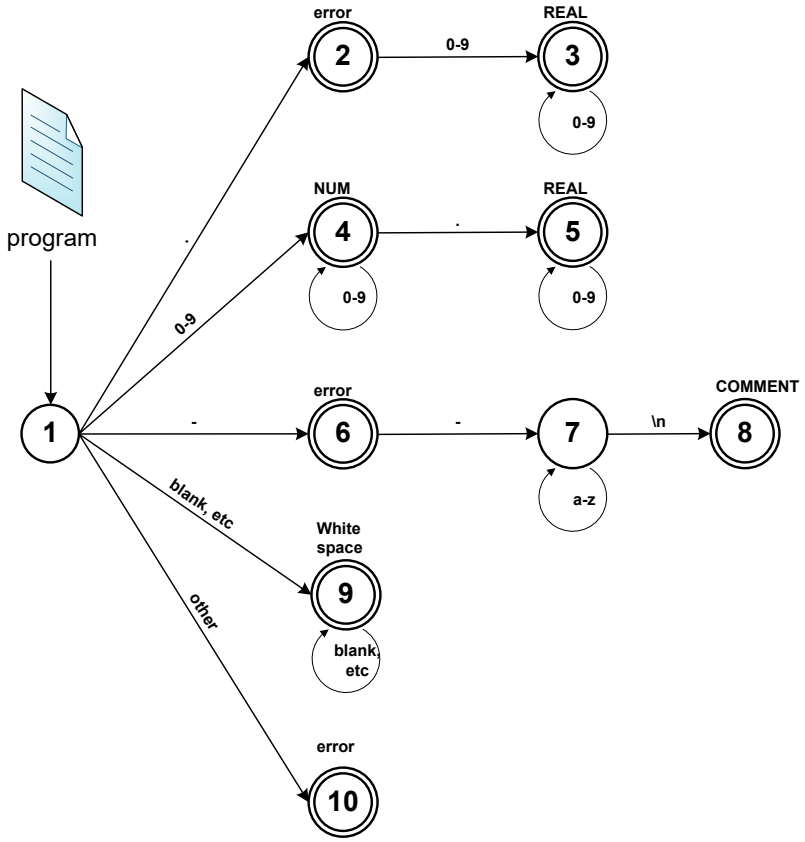
ЛЕКСИЧКИ АНАЛИЗАТОР

Лексички анализатор је реализован помоћу коначног детерминистичког аутомата чији је циљ да препозна све симболе у програму. Пример једног таквог аутомата дат је на следећој слици:



Слика 1 Пример једног коначног детерминистичког аутомата

Почетно стање је означено бројем 1. Стања у којима се завршава претрага једног симбола називају се финална стања и означена су двоструким кругом (стања 2, 3, 4, 5, 6, 8, 9 и 10). Из датотеке се редом читају знакови и шаљу у коначни аутомат, одакле се, у зависности од знака, прелази у наредно стање. Критеријуми за прелазак из једног стања у друго написани су на стрелицама које спајају стања. Тако, на пример, да би се препознао реалан број, први знак треба да буде тачка (.) или цифра (0-9). Уколико је први знак био тачка (.), из стања 1 прелази се у финално стање 2. Из финалног стања 2 је могуће: (1) прећи само у финално стање 3, уколико је наредни знак цифра (0-9), или (2) завршити рад коначног аутомата, уколико знак није цифра (0-9). У финалном стању 3 се очекују цифре (0-9). Када се појави знак који није из овог опсега, пријавиће се откривање симбола REAL (.12345, 1.2345, итд).

Приликом откривања симбола је могуће користити 2 правила: (1) правило најдуже речи и (2) правило приоритета.

<u>Правило најдуже речи</u>

Под овим правилом се подразумева читање знакова из програма и прелазак из једног у наредно стање аутомата све док се не пријави грешка лексичког анализатора. Нека се у датотеци програма налази *x+++y*, тада би лексички анализатор ово могао препознати као (x++) + y.

Правило приоритета

Под овим правилом се подразумева читање знакова из програма и прелазак из једног у наредно стање аутомата тако да се пријављује симбол чим аутомат пређе у финално стање. Овакав лексички анализатор би текст *x+++y* тумачио као x + (++y).

Лексички анализатор који је приложен као материјал уз ову вежбу је реализован коришћењем правила најдуже речи и налази се у библиотеци *LexLib_d.lib*. Излаз из лексичког анализатора је листа токена прочитаних из улазне датотеке. У овој листи се **не налазе** токени типа ***WHITE_SPACE***.

СИНТАКСНИ АНАЛИЗАТОР

Дефиниција појмова

Речник: скуп речи.

Реч: коначни скуп симбола из коначне азбуке. (код програма)

Азбука: симболи који се добијају као излаз из лексичког анализатора.

Граматика: скуп продукција које описују језик са једним или више симбола са десне стране.

`symbol → symbol symbol … symbol`

Сваки симбол може бити терминални или нетерминални. Терминални симболи су они који су преузети из азбуке језика. Нетерминални симболи су они који се налазе са леве стране исте продукције. Свака граматика поседује један нетерминални симбол који се означава као почетни симбол граматике. На следећој слици је дат пример једне граматике. Нетерминални симболи су: S, E и F, док су терминални: id, :=, -, ; и num.

S → id := E;

E → F –E

E → F

F → id

F → num

Слика 2 Граматика 1

S
id := E;
id := F-E;
id := F-F;
id := id-num;

Слика 3 Пример извођења за граматику 1

Да би се сазнало да ли је нека реченица граматички исправна одређује се њено порекло – обавља се извођење. Извођење се обавља тако што се креће од почетног симбола, након чега се у итерацијама уместо нетерминалних симбола смењују њихове продукције.

Примери улазног програма који задовољавају пример граматике са слике 2:

Пример 1:

```
x:=10;
```

Пример 2:

```
y:=1-3;
```

Пример 3:

```
z:=1-4-y-x;
```

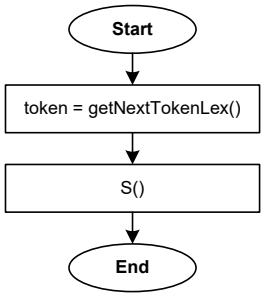
Алгоритам

Један од начина за реализацију синтаксног анализатора је помоћу алгоритма са рекурзивним спуштањем. Свака продукција се претвара у реченицу позивима рекурзивних функција. Алгоритам са рекурзивним спуштањем поседује по једну функцију за сваки нетерминални симбол и по један услов за сваку продукцију, тако да ће за граматику 1 бити потребне функције *S*, *E* и *F* у којима ће се испитивати постојање терминалних симбола id, :=, -, ; и num.

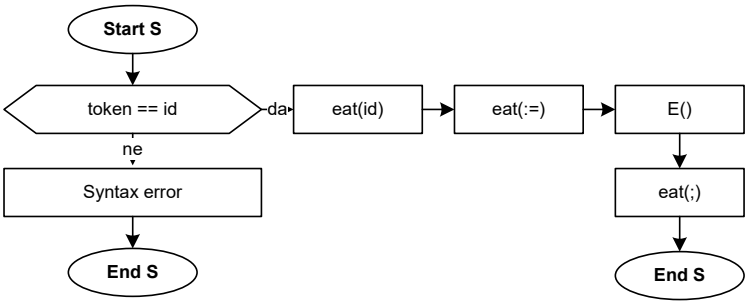
На почетку алгоритам добавља следећи (први) симбол и ставља га у глобалну променљиву *token*. Затим се позива функција *S* (слика 4). У функцији *S* (слика 5) се испитује да ли постоји правило у граматици

које почиње са добављеним симболом. Уколико не постоји овакво правило, пријављује се синтаксна грешка, док се у супротном ради извођење. Извођење се ради тако што се позива функција *eat* за све терминалне симболе и одговарајућа рекурзивна функција за нетерминалне симболе, у складу са граматиком. Функцији *eat* (слика 8) се прослеђује симбол који се очекује да ће бити следећи у програму (*param*). Уколико се очекивани симбол (*param*) не поклапа са новооткривеним симболом, потребно је пријавити синтаксну грешку.

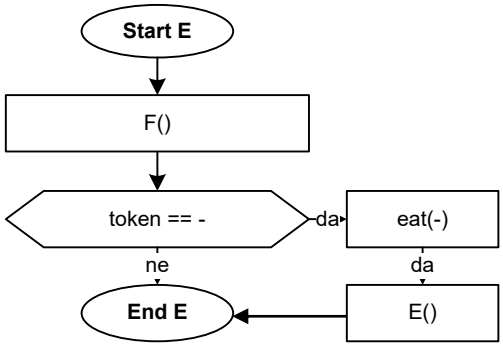
У функцијама *E* и *F* (слика 6 и 7) се испитује да ли постоји правило у граматици са тренутно актуелним симболом. Уколико постоји, обавља се извођење тог правила.



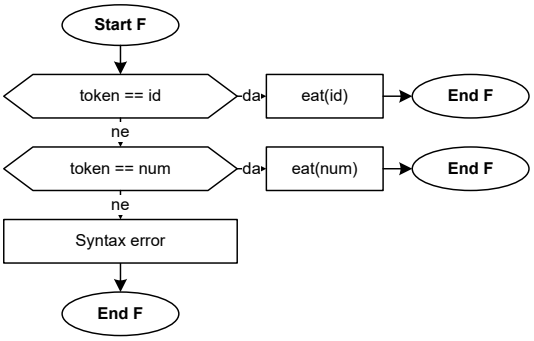
Слика 4 Синтаксни анализатор



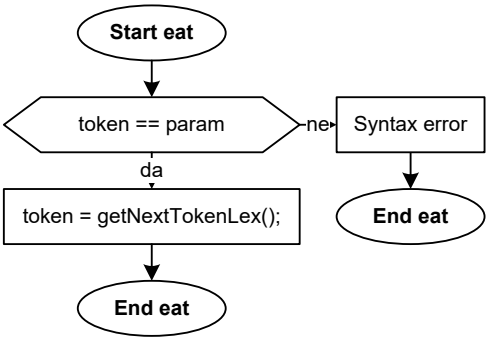
Слика 5 S функција



Слика 6 E функција



Слика 7 F функција



Слика 8 *eat* функција

РЕАЛИЗАЦИЈА СИНТАКСНОГ АНАЛИЗАТОРА

На почетку рада програма је потребно позвати лексички анализатор да обави лексичку анализу. Ово се постиже позивом методе *Do()*. Уколико је повратна вредност методе истинита, значи да се лексичка анализа завршила без грешке. У супротном ће се исписати извештај о грешци. Излаз из лексичке анализе је листа токена учитаних из улазне датотеке.

```
typedef std::list<Token> TokenList;

TokenList tokenList;
```

Ова листа не садржи *WHITE_SPACE* токене, јер они немају значај у датом примеру. Након лексичке анализе се позива синтаксна анализа. Приликом инстанцирања класе *SyntaxAnalysis* у конструктору се иницијализује итератор листе токена на почетак листе, након чега се позива метода *Do()* која означава извршење процеса синтаксне анализе.

```
TokenList::iterator tokenIterator;
```

Итератор служи за добављање наредног токена за анализу у синтаксном анализатору. То се постиже позивом методе:

```
Token getNextToken();
```

Типови подржаних токена су описани помоћу енумерације *TokenType* у датотеци *Common.h*.

Приликом реализације задатка користити методу:

```
void eat(TokenType t);
```

која служи за проверу синтаксне исправности тренутног токена. Уколико тренутни токен није очекиваног типа (прослеђеног типа), метода исписује поруку о синтаксној грешци.

За испис токена у конзолу на располагању је метода класе *SyntaxAnalysis*:

```
void printTokenInfo(Token token);
```

Или, у случају грешке, метода:

```
void printSyntaxError(Token token);
```

ЗАДАЦИ

1. Навести бар 3 коначна симбола који су излаз из аутомата са слике 1.
2. Који су терминални, а који нетерминални симболи из граматике 2 (Слика 9)?
3. Употребом програмског језика C++ и већ постојећег лексичког анализатора, реализовати синтаксни анализатор са рекурзивним спуштањем. Користити граматику 2 која је већ реализована у коду лексичког анализатора.

$Q \rightarrow$ begin S L $\quad$ $S \rightarrow$ if E then S else S $\quad$ $L \rightarrow$ end $\quad$ E $\rightarrow$ id = num

$S \rightarrow$ print E $\quad$ $L \rightarrow$ S L

Слика 9 Граматика 2

4. Приликом успешног откривања симбола исписати његову текстуалну вредност у конзолу.
5. Уколико је улазни програм лексички и синтаксно исправан, исписати у конзолу поруку „*Syntax OK*“, у супротном исписати грешку код првог симбола који није синтаксно или лексички исправан.