Формални Езици и Езикови Процесори
ТУ, кат. КС, летен семестър 2012

Лекция 8

Тема:

Лексически Анализ

Съдържание:

- Модели на Компилатор
- ЛА като етап, фаза на компилатор
- Лексически единици (лексеми)
- Токени (дескриптори)
- Режими на работа на ЛА
- Лексиката на ПЕ се описва с РИ
- Проектиране на ЛА-тор като КА

ЛА – базови понятия

Лексическият Анализ или още (linear analysis or scanning) като фаза (етап) в процеса на компилация.

ЛА – базови понятия

Дефиниция на лексически единици (лексеми) и описатели (токени)

Релация лексема – токен

ЛА – базови понятия

Примери за лексеми в ПЕ:

- Идентификатори;
- Запазени думи;
- Константи (цели, реални, символни);
- Разделители (операции и специални знаци).

ЛА
според
TS

ЛА според TS

- Една първична програма е низ от символи като букви, цифри, специални знаци +, −, (,) etc.
- Една първична програма съдържа елементарни езикови конструкти като имена на променливи, етикети, служебни (запазени) думи, константи, и операции. Желателно е компилаторът да идентифицира изброените конструкти като отделни категории.
- Всеки ПЕ съдържа определение на тези конструкти.

ЛА според TS

- Първичната програма е вход за ЛА, чиято цел е да нацепи входа на отделни части, наречени tokens като
  - идентификатори (променливи и/или етикети),
  - константи,
  - служебни (запазени) думи и
  - операции (разделители).
- На всеки токен се дава число, буква или група от числа, букви, с които еднозначно се идентифицира всяка лексема.

```
test: if a > b then x=y; endif
```

test: if a > b then x = y;

`test`	Идентификатор - етикет
`:`	Разделител
`if`	Запазена дума
`a`	Потреб. Деф. име
`>`	Разделител rel операция
`b`	Потреб. Деф. име
`then`	Запазена дума
`x`	Потреб. Деф. име
`=`	Разделител
`y`	Потреб. Деф. име
`;`	Разделител

ЛА според TS

- Доп. Дейности по време на ЛА:
  - Елиминиране на незначещи символи,
  - Игнориране на коментари,
  - Преобразуване на числени (numeric) константи във вътрешно представяне с фиксирана запетая (fixed point) или плаваща запетая (floating point).

ЛА
според
ASU

ЛА според ASU

- ЛА, или линеен анализ, чете (сканира) входния поток (низ от символи) от ляво на дясно (L−>R) и го прегрупира в токени или поредици от символи с групово значение (collective meaning)
- pos = init + rate * 60;

pos	=	init	+	rate	*	60	;
<id1>	<del=>	<id2>	<del+>	<id3>	<del*>	<const>	<del;>

ЛА според ASU

- Незначещи разделители като интервал и табулация се игнорират от ЛА.
- ЛА следва да обработва и многоредови оператори (multi-line statements), т.е символ ‘\n’ нов ред също следва да се игнорира.

ЛА – три важни термина

- ***Token***. Във входа за ЛА има множество от низове, за които се създава един и същ токен.
- ***Pattern***. За всяко множество символи има правило, наречено шаблон (pattern), свързан с всеки токен. Така шаблонът служи да разпознава низове от входния поток.
- ***Lexeme***. Лексическа единица (или лексема) е последователност (низ) от символи във входния поток, която се разпознава чрез шаблон за токен (описател).

ЛА като фаза(етап) на К.

- Първична програма конвертирана в D-език
- Типични лексеми:
  - Идентификатори
  - Запазени думи
  - Константи (цели, реални, символни)
  - Разделители (операции и специални знаци)
- Структура на описател:
  - Тип и Код
  - Тип и Указател (към позиция в таблица)

Структура данни за токен

```
struct token
        {
            int type, code;
            char buf[SIZE];
        };
```

ЛА – режими на работа

- Като отделен пас
- Като ПП, подчинена /викана/ от парсера
  - `Lexic (type, code, buf);`
  -
```
struct  token
        {    int type, code;
             char buf[SIZE];
        };
token Lexic();
```

ЛА

Проектиране и реализация на сканер.

Лексемите се описват с РИ.

Задачата за реализацията на сканер е функционално еквивалентна на задачата за реализация на КА, разпознавател на низове, задавани с РИ.

ЛА и КА - отлики

- КА/FSA разпознава единичен токен.
- ЛА/LA разпознава множество от различни токени.

ЛА и КА - отлики

- КА/FSA е автомат с еднократно действие (once only) action machine.
- ЛА/LA е автомат с многократно действие (many times (repeated, reiterated) action machine).

ЛА и КА - отлики

- ЛА/LA изпълнява странични (extra time consuming activities) обработки като
  - Игнориране на интервали,
  - Елиминиране на коментари,
  - Преобразуване на данни от символно във вътрешно представяне.

ЛА и КА - отлики

Проблем: ПЕ с недетерминирана лексическа структура.

Проектиране на сканер

От РИ през граф на преходите на КА до блок схема на алгоритъм на сканер.

Идентификатори I = L . (L | D)*

Константи без знак C = D^+

Разделители R = + | −

Схема на токени

	тип	код
I	1	0
C	2	0
R	3	+ 11
		− 12

Проектиране на сканер

- За всеки токен се строи отделен КА
- Отделните КА се обединяват в общ КА
- Модифициране общия КА:
  - Добавяне обработка на незначещи разделители
  - Елиминиране на заключително състояние qf
- От граф на общия КА към блок схема на алгоритъм на лексически анализатор

Практика

C/C++ заглавен файл **<ctype.h>** или **<cctype>**

```
isalpha(), isalnum(), isdigit(),
isxdigit(), isspace ),...
```

Демо програми:

```
lex1a.cpp, lex1b.cpp,
lex1c.cpp, lex1d.cpp,
lexdefs.h.
```

Демо програми

Тук следва описание на три програми лексически анализатори, програмирани с техниките на тотално, структурно и обектно ориентирано програмиране. Различните версии са описани като първични и изпълними файлове със следните имена:

Демо програми

Lex1a: Scanner – Spaghetti code версия, една функция main() сканира вход и генерира токени

Lex1b: версия структурно програмиране, функции main(), Lexic(), GetNewChar()

Lex1c: Scanner – ООП версия, функция main(), клас Scanner с методи Run(), Lexic(), GetNewChar()

Демо програми

Техника/подход	Първичен текст	Изпълним код
Тотално програмиране	Lex1a.cpp, lexdefs.h	lex1a.exe
Структурно програмиране	Lex1b.cpp, lexdefs.h	Lex1b.exe
Обектно програмиране	Lex1c.cpp, lexdefs.h	Lex1c.exe
	Lex1dInterface.h, lexdefs.h, lex1dImplement.cpp, lex1dApplication.cpp	Lex1dApplicat.exe

Демо програми

И трите реализирани версии имат еднакво поведение при изпълнение – разпознават едно и също множество лексически единици, които са описани в таблицата на следващия слайд.

За всяка една от лексемите, срещната и локализирана във входния поток, се генерира унифициран дескриптор, описател или *токен (token)* със следната структура:

Структура на token

```
const int TokenBufSize=20;
struct token
{
 int type, code;
 char buf[TokenBufSize];
 char *tkname;
};
```

.

- Дескрипторът съдържа две полета от целочислен тип (*type, code*), които идентифицират лексемата еднозначно с тип и код за вътрешно представяне. Допълнително, структурата включва едномерен масив (*buf*), съхраняващ текущата разпознавана лексема в символен вид и поле указател (*tkname*), което сочи символен низ, обобщено име на лексическата единица.

.

Разпознаваните типове лексеми са:

- идентификатори,
- служебни думи,
- цели десетични, шестнадесетични и осмични константи,
- реални десетични константи,
- единични символи, символни низове,
- прости едно-символни и сложни дву-символни разделители.

Предполага се C/C++ синтаксис за изброените лексеми.

.

Таблицата на лексическите единици съдържа описание и на три допълнителни лексеми.

Символите, които маркират *край на ред* и *край на входния поток* се третират като служебни лексеми и за тях се генерира съответен служебен дескриптор.

Случаят, когато във входния поток се прочете непознат символ, се регистрира с генериране на служебен дескриптор за невалиден знак.

Разпознавани лексеми

```
#define  EOFDATA          -50     // end of data indicator
#define  EOLINE           -70     // end of line indicator
#define  INVALIDCHAR      -90     // invalid character entered

#define  IDENT            100     // identifier
#define  KEYWORD          200     // key word

#define  INTDECCONST      300     // integer decimal constant D+
#define  INTHEXCONST      400     // integer hexadecimal constant 0(x!X)D16+
#define  INTOCTCONST      500     // integer octal constant 0D8+
#define  REALCONST        600     // real constant (D+.D* | D*.D+)

#define  SINGLECHAR       700     // single character 'X'
#define  CHARSTRING       800     // character string "X+"

#define  DELIMITER        900     // delimiter
```

Разпознавани лексеми

```
#define  KEYWORD     200      // key word
#define     BEGINKW          201
#define     DOKW             202
#define     ENDKW            203
#define     ENDIFKW          204
#define     ENDOKW           205
#define     FORKW            206
#define     IFKW             207
#define     WHILEKW          208
```

Разпознавани лексеми

```
#define  DELIMITER    900      // delimiter
#define     ADDDLM     901  //  +
#define     SUBDLM     902  //  –
#define     MULDLM     903  //  *
#define     DIVDLM     904  //  /
#define     MODDLM     905  //  %
#define     LTDLM      906  //  <
#define     LEDLM      907  //  <=
#define     GTDLM      908  //  >
#define     GEDLM      909  //  >=
#define     EQDLM      910  //  ==
#define     NEDLM      911  //  !=
#define     ASSGNDLM   912  //  =
```

Демо програми

Lex1a: Scanner – Spaghetti code версия, една ф-ия main() сканира вход и генерира токени

Програма *Lex1a.cpp* (тотална версия) е примитивната реализация на лексически анализатор. Принципът на тотално програмиране означава реализация на проект в една функция, в един модул, в една програма. Всички действия са кодирани в тялото на функция *main()*. Тя съдържа цикъл до изчерпване на входния поток. В тялото на цикъла са включени операторите за сканиране на входния поток, разпознаване на лексемите и генериране на дескриптор.

Демо програми

Lex1a: Scanner – Spaghetti code версия, само една ф-ия main() сканриа вход и генерира токени

```
#include "lexdefs.h"
void main( )
{  token tm;
   while ( (ptr=gets(niz)) != NULL)
       {
        . . .    // read input, generates tokens
        }
}
```

Demo program Lex1a

Lex1a: Scanner – Spaghetti code версия, само една ф-ия main() сканира вход и генерира токени

lexdefs.h

Lex1a.cpp

Lex1a.exe

Демо програми

Lex1b: версия структурно програмиране, функции main(), Lexic(), GetNewChar()

Принципът на структурното програмиране *Мислене чрез функции (Thinking in Functions)* е приложен при реализацията на втората версия на сканер. „Спагети“-кодът (*spaghetti code*) на програма *Lex1a.cpp* е преструктуриран в две функции *Lexic()* и *GetNewChar()*, които оформят ядрото на програма *Lex1b.cpp* (структурна версия). Преходът от тотален към структурен вариант на лексически анализатор се постига относително лесно с незначителни промени. Фрагменти от съществуващия код на тоталната версия, реализиращи действията по генериране на токен за разпозната лексема, както четене и предоставяне на текущия символ от входния поток са локализирани и обособени като функци *Lexic()* и *GetNewChar()*. Функцията *main()* съдържа цикъл до разпознаване на служебна лексема за край на входния поток, в тялото на който са включени само един оператор за извикване на функцията *Lexic()* и един оператор за извеждане на екран подробности за текущата разпознавана лексема

Демо програми

Lex1b: версия структурно програмиране, функции main(), Lexic(), GetNewChar()

```
#include “lexdefs.h”
token Lexic();       // returns token for the current scanned lexeme from input stream
int GetNewChar(char []);      // returns current scanned character from input stream
void main()
{     token tm;                          tm.type = −20;
      while (tm.type != EOFDATA)
      {
       tm = Lexic();
       printf("\nLex unit  %5d  %5d  %20s 24s", tm.type, tm.code, tm.buf, tm.tkname);
      }
}
token Lexic( )            //            generates token
      {        token t;      . . .          return t;     }
int GetNewChar(char LookAheadBuf[])// reads input stream char by char
      {          . . .                    }
```

Demo program Lex1b

Lex1b: версия структурно програмиране, функции main(), Lexic(), GetNewChar()

lexdefs.h

Lex1b.cpp

Lex1b.exe

Демо програми

Lex1c: Scanner – ООП версия, функция main(), клас Scanner с методи Run(), Lexic(), GetNewChar()

Принципът на обектно ориентираното програмиране *Мислене чрез обекти (Thinking in Objects)* е приложен при реализацията на следващата трета версия на сканер. Преходът от структурна към обектна версия на сканер се постига с лекота. Дефинира се клас *Scanner*. Функции *Lexic()* и *GetNewChar()* от структурната версия без промени са обособени като *public* методи на класа *Scanner* в състава на програма *Lex1c.cpp* (обектна версия). Допълнителен метод *Run()* е реализиран с цел да се подобри функционалността на класа *Scanner*. Като алгоритъм този метод е еквивалентен на операторите от тялото на функцията *main()* в състава на програмата сканер *Lex1b.cpp* (структурна версия). По тази причина функция *main()* на обектната версия съдържа само оператори за създаване на обект от клас *Scanner* и активиране на метода му *Run()*.

Демо програми

Lex1c: Scanner – ООП версия, функция main(), клас Scanner с методи Run(), Lexic(), GetNewChar()

```
#include “lexdefs.h”
class Scanner
{ private: . . .
  public:
     void Run( )   {      token tm;                  tm.type = −20;
                          while (tm.type != EOFDATA)
                          {
                             tm = Lexic();           printf("\nScanned lexeme . . . “);
                          }
                   }
     token Lexic( )       {          token t;
                                     . . .           // generates token
                                     return t;
                          }
     int GetNewChar(char LookAheadBuf[])      {
                                      . . .          // reads input stream char by char
                                                     }
}; // end of Scanner class definition
void main()  {
                 Scanner a;          a.Run();
              }
```

Demo program Lex1c

Lex1c: Scanner – ООП версия, функция main(), клас Scanner с методи Run(), Lexic(), GetNewChar()

lexdefs.h

Lex1c.cpp

Lex1c.exe

Сравнение на програми сканери

Описаните по-горе програми лексически анализатори са идентични по функционалност.

Те имат еднакво поведение при изпълнение.

Различават се по производителност и заемана памет.

Сравнение на програми сканери

- Обектната версия (програма *Lex1c*) заема най-много памет.
- Първичният код на трите версии има приблизително еднакъв размер (10 KB, 13 KB, 14 KB).
- Няма съществена разлика в кода, създаден от програмната среда Borland IDE. Размерът на обектния код (7 KB, 8KB, 10 KB) и изпълнимия код (18 KB, 19 KB, 20 KB) са приблизително еднакви за трите версии на лексическия анализатор.
- Кодът, генериран от програмната среда Microsoft Visual C++, заема повече дисково пространство в сравнение с кода на Borland IDE.
- Като правило, Release версията на Developer Studio IDE заема по-малко памет (1.5 пъти за обектен код и 4.5 пъти за изпълним код) в сравнение с Debug версията, създадена в същата среда.

Благодаря
За
Вниманието