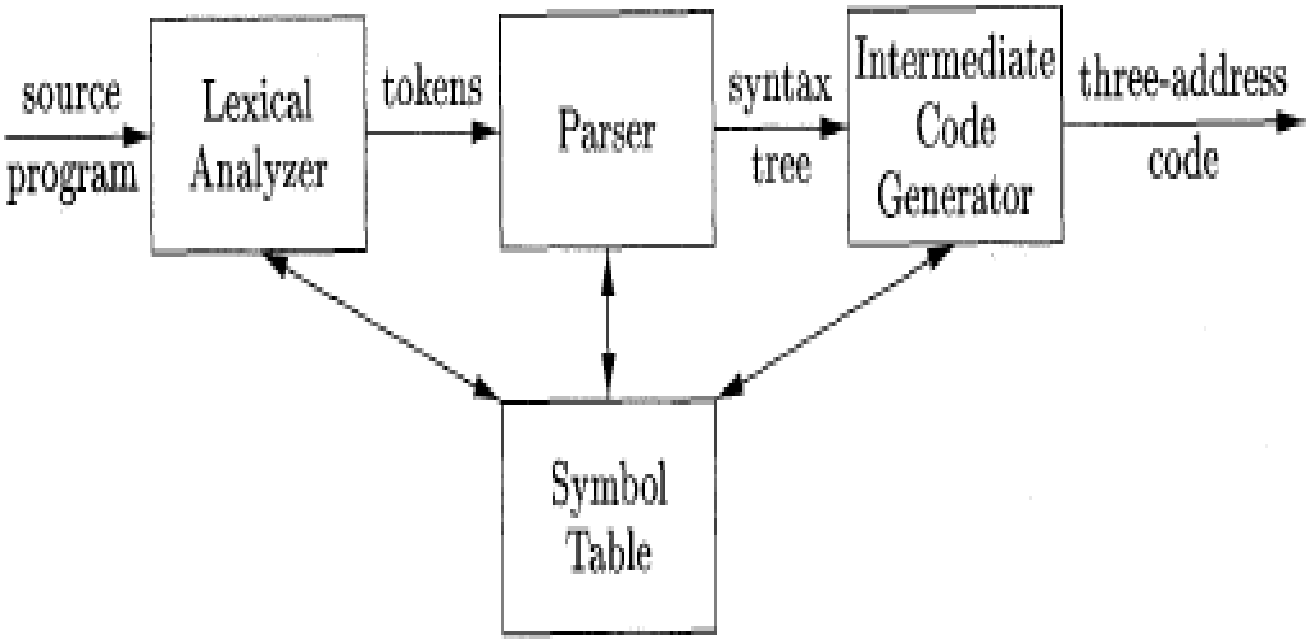
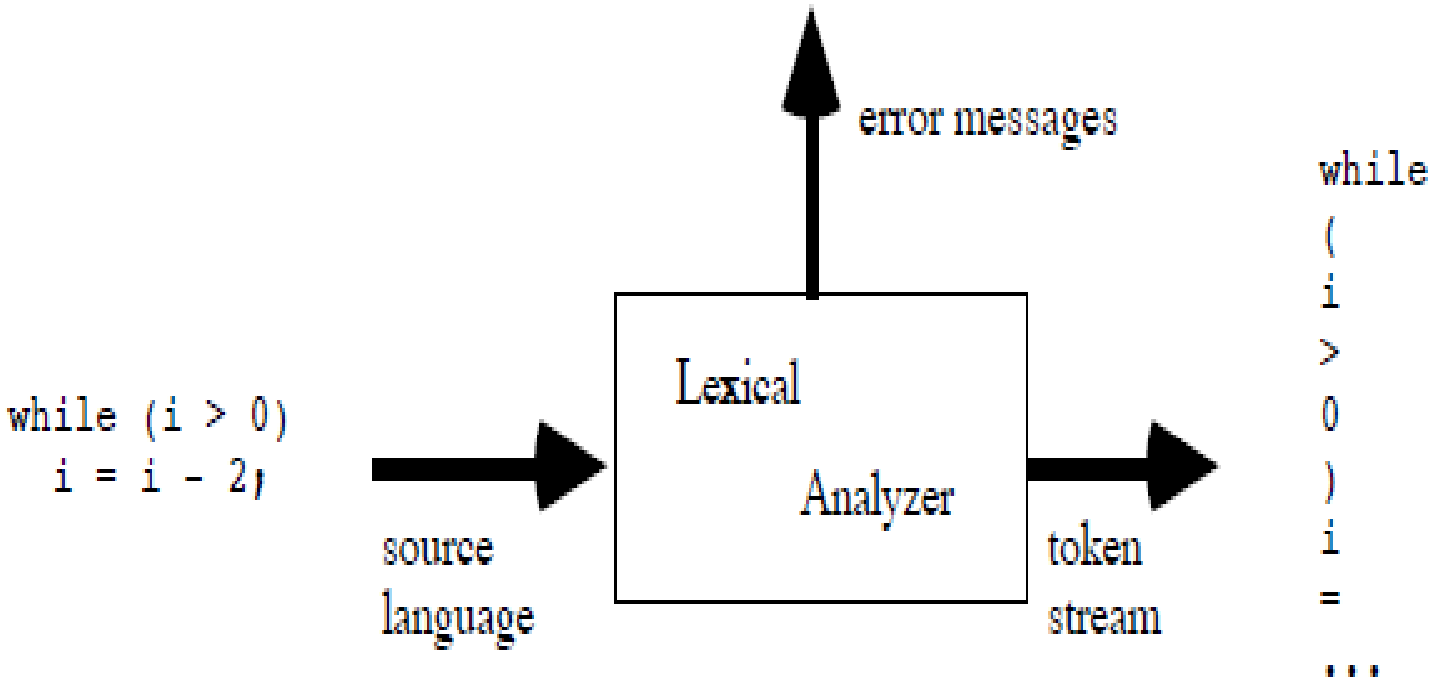
Лексически анализ

Обща организация на компилатор



Основни понятия

- *Лексически анализ* или *сканиране* е процес, при който входната програма се прочита отляво надясно и се формират символни последователности (нар. *tokens*).
- *Tokens* са последователности от символи, които са значещи в програмния език. Даден програмен език типично съдържа няколко групи такива последователности: константи (цели числа, реални числа, символи, символни низове и др.), оператори (аритметични, логически, за сравнение и др.), разделители, ключови (резервирани) думи.


error messages
while (i > 0)
i = i - 2;
Lexical
Analyzer
source
language
token
stream
while
(
i
>
0
)
i
=
...

Лексическият анализатор чете входна програма, а на изхода се получава поток от символни последователности. Примери за символни последователности, които се разпознават на етапа на лексическия анализ:

3 или 255 е целочислена константа

"Fred" или **"Wilma"** е символен низ

numTickets или **queue** е име на променлива

Стойностите на горните последователности се нар. *лексеми.*

За някои типове лексеми има само по една съответстваща лексема (напр. >), а други имат множество лексеми (напр. целочислени константи).

Целта на лексическия анализатор е да раздели входната програма на валидни последователности на входния език, но не може да определи дали всяка лексема е на правилното място.

- По време на лексическия анализ могат да бъдат открити малко на брой типове грешки:
  - невалидни (или неразпознати) лексеми
  - неправилни лексеми (напр. грешен брой апострофи в символна константа, недовършени коментари и др.)

По време на лексическия анализ не могат да бъдат открити лексеми, които не са на правилно място, нито недекларирани идентификатори, сгрешени ключови думи, несъвместими типове и др.

Например лексическият анализатор няма да открие грешки в следната входна последователност, защото той все още „не знае“ правилата за подредба на лексемите. Тези грешки се откриват на по-късния етап на синтактичен анализ.

int a double } switch b[2] =;

Освен това лексическият анализатор „не знае“ как трябва да бъдат групирани разпознатите лексеми. В горната последователност ще бъдат открити лексеми **b**, **[**, **2** и **]**, т.е. четири отделни лексеми, които всъщност заедно формират достъп до елемент на масив.

Лексическият анализатор е подходящ за изпълнение и на някои други задачи като например:

- Премахване на коментари, интервали, табулации;
- Макроси, условно компилиране.

Лексически анализ

```
void main () {
        bool NotEndOfSource = true;
        do {
                // read next input character
                // clasify character by symbol group
                // build symbol
                // write intermediate code
         } while (NotEndOfSource);
}
```

Лексически анализ

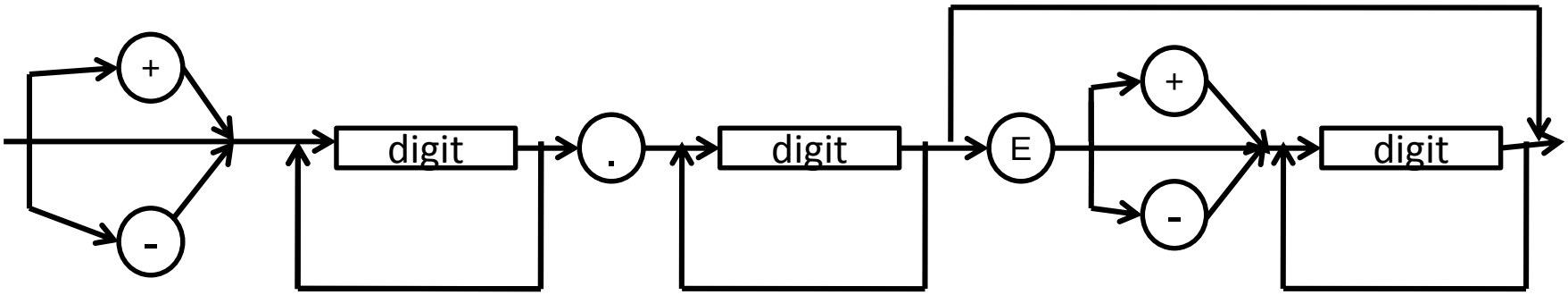
```
#include  <stdio.h>
void  main () {
   bool  NotDone = true
   char  ch;
   do  {
            ch  =  getchar ();
             switch  (ch)  {
             case  ‘a’: case ‘b’: /* . . . */ case ‘z’ :
                      /* scan identifier */
                       break;
             case ‘0’: case ‘1’ : /* . . . */ case ‘9’:
                       /* scan number */
                        break;
```

Лексически анализ

```
        case ‘<’:
                /* scan operator < */
                break;
                /* . . . */
        case ‘ ‘ :
                /* scan white spaces */
                break;
        case ‘.’ :
                 NotDone = false;
                 break;
        };
    }  while  (NotDone);
}
```

Премахване на интервали

```
#include  <stdio.h>
void  main() {
     char    ch;
     /* skip white spaces */
     while ( (ch = getchar() ) == ‘’;
      /*  . . .  */
}
```


+
-
digit
.
digit
E
+
-
digit

Разпознаване на дробна част в реални числа

```
#include  <stdio.h>
void  main() {
     char    ch;
      ch = getchar();
      while ( (ch>=‘0’) || (ch <= 9) ) {
          // collect the integer part
       };
if ( ch == ‘.’ ) {
          // collect the fractional part
        };

       if ( ch == ‘e’ ) {
          // collect the exponent part
        };
     // build the number
     // write intermediate code
}
```

Разпознаване на цяла част в реални числа

```
#include  <stdio.h>
extern  “C” void error (char *);
void  main() {
     float  number = 0.0;
     char  ch
     const  float  float_limit = float (0xFFFFFFFF);
      ch = getchar ();
      while ( ( ch = ‘0’ ) || ( ch <= ‘9’ ) ) {
                if ( number < float_limit ) {
                     number = number * 10 + ( ch – ‘0’ );
                     ch = getchar ();
                } else {
                      error (“float  exceeded \n”);
                }
       }
}
```

Символни таблици

Символните таблици са структури от данни, които се използват от компилаторите за съхраняване на информация за използваните конструкции във входната програма. Тази информация се формира постъпателно по време на анализа и се използва по време на синтеза за генериране на изходен код.

Елементите в символната таблица съдържат например следната информация за идентификатор: име (т.е. лексемата), тип на данните, позиция в паметта и др.

Символните таблици типично поддържат множество декларации на един идентификатор в програмата.

Елементи в символна таблица: Различни типове на имената

- Променливи (идентификатори)
  - име на променливата (лексема), може да има ограничение в броя на символите
  - тип на данните,
  - име и ниво на блок, в който е декларирана променливата
  - друга информация за права на достъп
  - базов адрес и отстъп в паметта

Записи в символната таблица: Различни типове данни

- Масиви
  - Размерности
  - Горна и долна граница на всяка дименсия
- Записи и структури
  - Списък от полета
  - Информация за всяко поле
- Функции и процедури
  - Брой и тип на параметрите
  - Тип на върнатата стойност

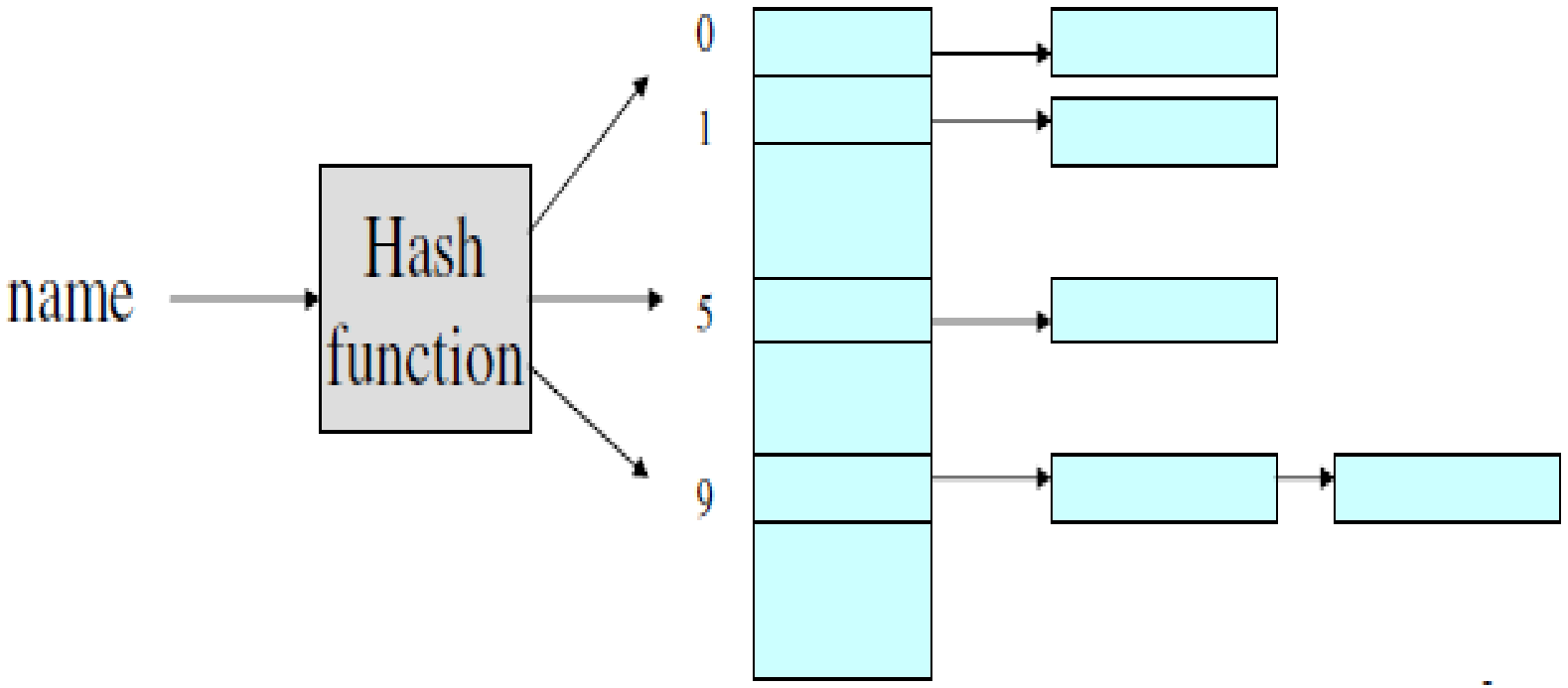
Операции със символната таблица

Двете основни операции със символната таблица са:

- Въвеждане на нов ред – т.е. запис на нов идентификатор (име на променлива, тип и др.)
- Търсене на запис – т.е. търсене на информация по зададено име

- Търсенето се извършва много по-често от въвеждането
- За представяне на символна таблица най-често се използват хеш таблици поради възможностите за ефективно търсене

Представяне на символна таблица



Последователно търсене

```
void main () {
      const N = 769;
      char  * a[N];
      char  * X = ‘’;
      int i;
      // build the identifier table a[]
      // put current identifier into “X”
      i = -1;
      do {
            i += 1;
      } while ( ( a[i] != X ) && ( I != N-1) );
      If ( a[i] != X ) { /* element not found */
}
```

Двоично търсене

```
#include <string.h>
void main () {
    const N = 769;
    char * a[N];
    char * X = “”;
    int  i, j, k;
    // build and sort identifier table a[]
    // put current identifier into “X”
    i = 0;
    j = n -1;
```

Двоично търсене

```
    do {
        k = (i+j)/2;
        If ( strcmp (X, a[k])) < 0
            i = k + 1;
        else
            if ( strcmp (X, a[k])) > 0
                j = k -1;
    } while ( (a[k] != X) && (i <= j) );
    If ( a[k] != X ) { /* element not found */ };
}
```

Генериране на хеш ключ

```
#include <string.h>
const int N = 769;
int Hash (char * id) {
	unsigned int key, i;
	char  c;
	key = 1;
	i = -1;
	do {
		i += 1;
		c = id[i];
		if ( c != ‘\0’)
			key = (key + c) % N;
	} while ( ( c != ‘\0’ ) && ( i != strlen(id) )
	return (key);
}
```

Търсене с хеш ключ

```
void main () {
    int  index;
    char  * identifier = “asdasd”;

    /* . . . */

    index = Hash (identifier);

    /*  . . . */
}
```

Всяка променлива се декларира с област на действие даден фрагмент от програма (т.нар. scope). Често за всеки такъв фрагмент се реализира отделна символна таблица. Това означава, че даден фрагмент от програмата има собствена символна таблица, в която има ред за всяка декларация.

Блокове

- Областта на действие на име (идентификатор) е свързана с **блоковата структура** на повечето програмни езици:
  - Стандартни блокове (съставен оператор, оператор if)
  - Процедури и функции
  - Програма (глобален блок)
- Имената трябва да бъдат уникални в рамките на блока, в който са декларирани, т.е. не може да има две еднакви имена в един блок.
- Резолиране на имена: имена, декларирани във външен блок могат да бъдат използвани във вътрешни блокове.
- Същото се отнася и за други конструкции; например клас в обектно-ориентирана програма има собствена таблица с отделен ред за всяко поле и за всеки метод.

Свързани символни таблици

Вложени блокове се реализират чрез свързани символни таблици, т.е. таблицата на даден блок сочи към таблицата на обхващащия блок.