# *Методи на Транслация*

Семантичен анализ.
Символна таблица.

*доц. д-р Александър Пенев*

# *Семантичен Анализ*

# *Семантичен анализ*

Синтактичният анализ (парсерът) проверява само синтактичната вярност на програма. Например: `a + b`

е синтактично коректен израз в повечето ЕП, но може да няма смисъл (ако „**a**“ е име на тип) или да не е коректен (ако „**a**“ и „**b**“ са променливи от несъвместими за операцията събиране типове).

**Допълнителните контекстно-зависими правила се проверяват от т.нар. семантични действия. Аналогично генерацията на код може да бъде реализирана или инициирана от семантични действия.**

Семантичните действия са вградени в парсера и се описват с атрибутни граматики (ATG).

# *Семантичен анализ*

**Задачи на семантичния анализ:**

- Управление на символната таблица
  - поддържане на информация за декларираните имена;
  - поддържане на информация за типовете;
  - поддържане на области на видимост;
- Проверка на контекстни условия
  - правила за област на видимост;
  - проверка на типовете;
- Извикване на функции (методи) за генерация на код;

# *Семантични действия*

## До сега: анализ на входа

```
Expr = Term { ‘+’ Term } .
```

*Парсерът проверява дали входът е синтактично правилен*

## Сега: *превод* на входа (семантичен анализ)

*Например, трябва да се преброят термите (Term) в израз*

```
Expr =
      Term          (. int n = 1; .)
      { ‘+’ Term    (. n++; .)
      } .           (. Console.Writeline(n); .)
```

*За да се преброят термите трябва да се добави допълнителен код към анализиращите методи на парсера*

Резултата от работата на парсера в този случай е:

| **Израз** | **Брой терми** |
|---|---|
| 1 + 2 + 3 | 3 |
| 47 + 1 | 2 |

# *Атрибутни граматики*

**Нотации за описание на процеса на превод**

*Състои се от три части:*

**1. Произведения в EBNF**

```
Expr = Term { ‘+’ Term } .
```

**2. Атрибути (параметри на синтактичните символи)**

```
Term<↑int val>

Expr <↓bool print>
```

***изходни*** ( ↑ ) *атрибути (синтезирани) – предоставят резултат*
***входни*** ( ↓ ) *атрибути (наследени) – предоставят контекст от извикващия*

**3. Семантични действия, записани най-често на реализационния език на парсера**

# *Реализиране на ATG в парсера*

**Произведение:**

```
Expr<↑int val>                (. int val1; .)
= Term <↑val>
  { '+' Term <↑val1>          (. val += val1; .)
   | '–' Term <↑val1>         (. val -= val1; .)
  } .
```

**Анализиращ метод:**

***входни ( ↓ ) атрибути*** *като параметри*
***изходни ( ↑ ) атрибути*** *като изходни параметри*
***семантични действия*** *като вграден код (с червен цвят)*

*Терминалните символи нямат входни атрибути.*
*В примерния компилатор те нямат и изходни параметри, защото всеки обект има поле value*

```
bool IsExpr(out int val) {
  int val1;
  if (!IsTerm(out val)) Error("...");
  while (true) {
    if (CheckSpecialSymbol("+")) {
      if (!IsTerm(out val1)) Error("...");
      val += val1;
    } else if (CheckSpecialSymbol("-")) {
      if (!IsTerm(out val1)) Error("...");
      val -= val1;
    } else break;
  }
  return true;
}
```

# *Символна Таблица*

# *Отговорности на символната таблица*

**1. Съхранява всички декларирани имена и техните атрибути като:**

- Тип;
- Стойност (за константи);
- Адреси (за локални и глобални променливи, аргументи на методи, ...);
- Параметри (за методи, функции) и др.;

**2. Използва се, за да се получават атрибутите на дадено име**

- Съпоставяне (mapping): име → (тип, стойност, адрес, ...)

**3. Съдържание на символната таблица**

- Символни записи: информация за декларираните имена
- Структурни записи: информация за структурата на типовете

# *Символната таблица – реализация*

*Най-често се реализира като динамична структура от данни:*

- Линеен списък;
- Двоично дърво;
- Хеш таблица;
- **Стек от хеш таблици;**
- Дърво от хеш таблици;
- Хеш таблица със списъци (стекове) на синонимите;
- др.;

# *Цел на символната таблица*

**1. Да предостави помощна структура, която**

- Съхранява данни, получени от синтактичния анализатор (променливи, методи, ...);
- Манипулира се по време на семантичния анализ;
- Спомага за откриване на семантични и контекстни грешки (използване на променлива преди да е декларирана; съвместимост на типове; ...);
- Подпомага генерацията на код;

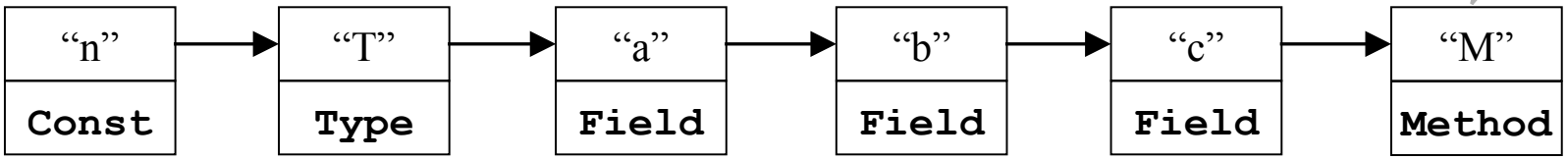
# *Символната таблица: линеен списък*

## Декларации

```
const int n = 10;
class T { ... }
int a, b, c;
void M () { ... }
```

За всяко декларирано име има символен запис

## получава се следният линеен списък



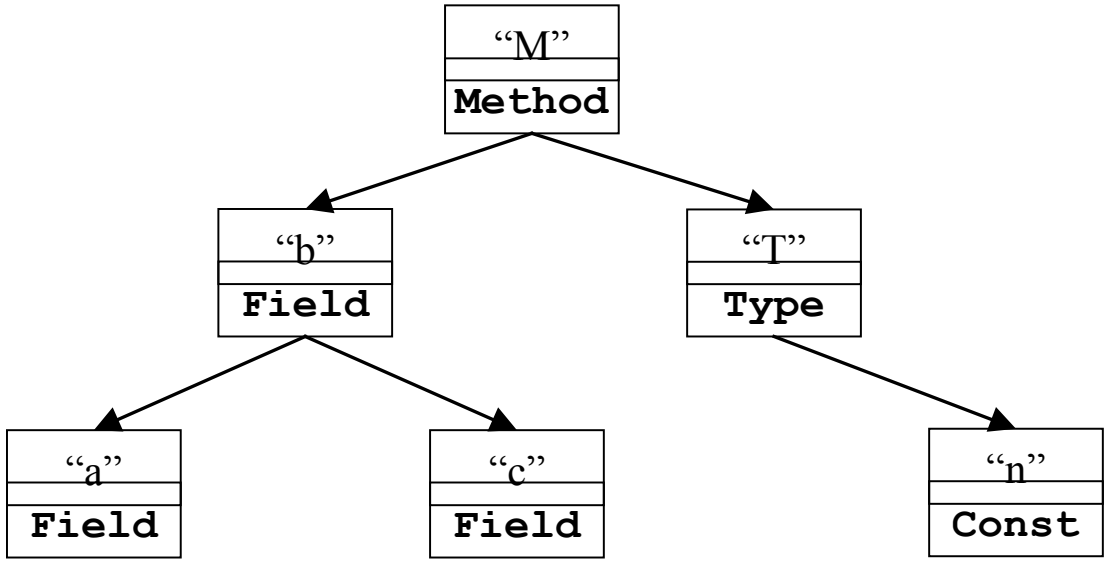

- *прост за реализиране;*
- *бавен при много декларации;*
- *затруднена поддръжка на езици с вложена блочна структура;*
- *редът на деклариране се запазва (важен само когато адресите се дават след попълване на списъка);*

# *Символната таблица: двоично дърво*

## Декларации

```
const int n = 10;
class T { ... }
int a, b, c;
void M () { ... }
```

## получава се следното двоично дърво




- *бързо;*
- *дървото може да дегенерира, ако не е балансирано;*
- *по-голяма консумация на памет;*
- *загуба на реда на деклариране;*

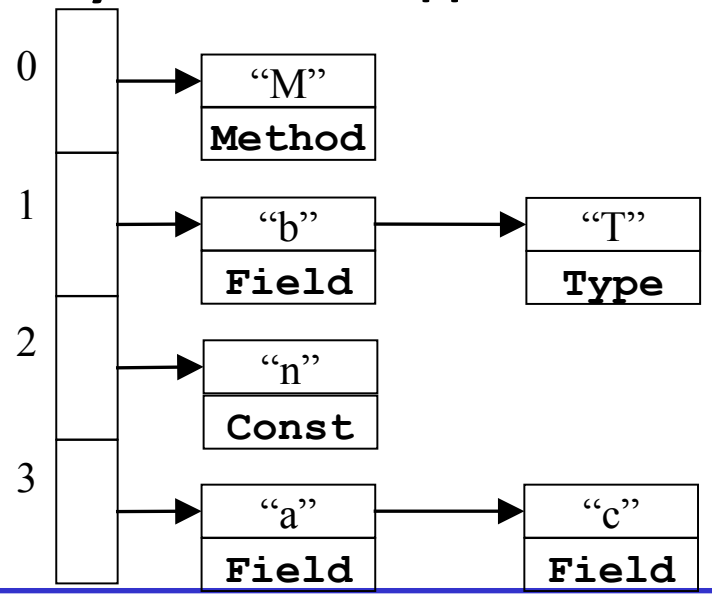
***Подходът е използваем, когато има много декларации***

# Символната таблица: хеш таблица

## Декларации

```
const int n = 10;
class T { ... }
int a, b, c;
void M () { ... }
```

## получава се следната хеш таблица




- *бързо;*
- *по-сложно от линейния списък;*
- *по-трудно добавяне на елемент;*
- *загуба на реда на деклариране;*

***На практика това е хеш таблица със списъци на синонимите.***

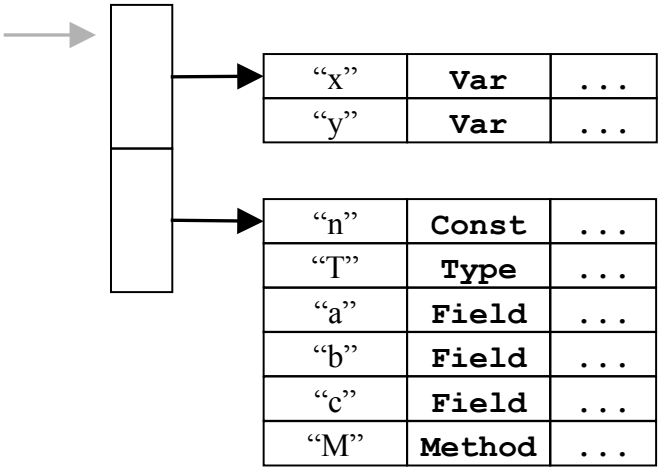
***В примера се използва стек за всяка различна хеш стойност.***

# Символната таблица: стек от хеш таблици

## Декларации

```
const int n = 10;
class T { ... }
int a, b, c;
void M () { ... }
```

## получава се следната стек от хеш таблици




- *бързо търсене;*
- *лесно добавяне на нов елемент;*
- *лесна обработка на вложена блочна структура;*
- *загуба на реда на деклариране;*

# *Символни записи*

**Всяко декларирано име се съхранява в символен запис.**

**Видове символи:**

- Константи;
- Глобални променливи;
- Полета;
- Аргументи на метод/функция;
- Локални променливи;
- Типове / класове;
- Методи / функции;
- Програми;
- и др.

# *Символни записи*

**Необходима информация за обектите:**

| | |
|---|---|
| ❖ За всички символи | име, вид символ, тип; |
| ❖ Константи | стойност; |
| ❖ Аргументи на метод | адреси (ред на декларация); |
| ❖ Локални променливи | адрес (ред на декларация); |
| ❖ Методи | брой на арг. и лок. променливи, ...; |
| ❖ ... | |

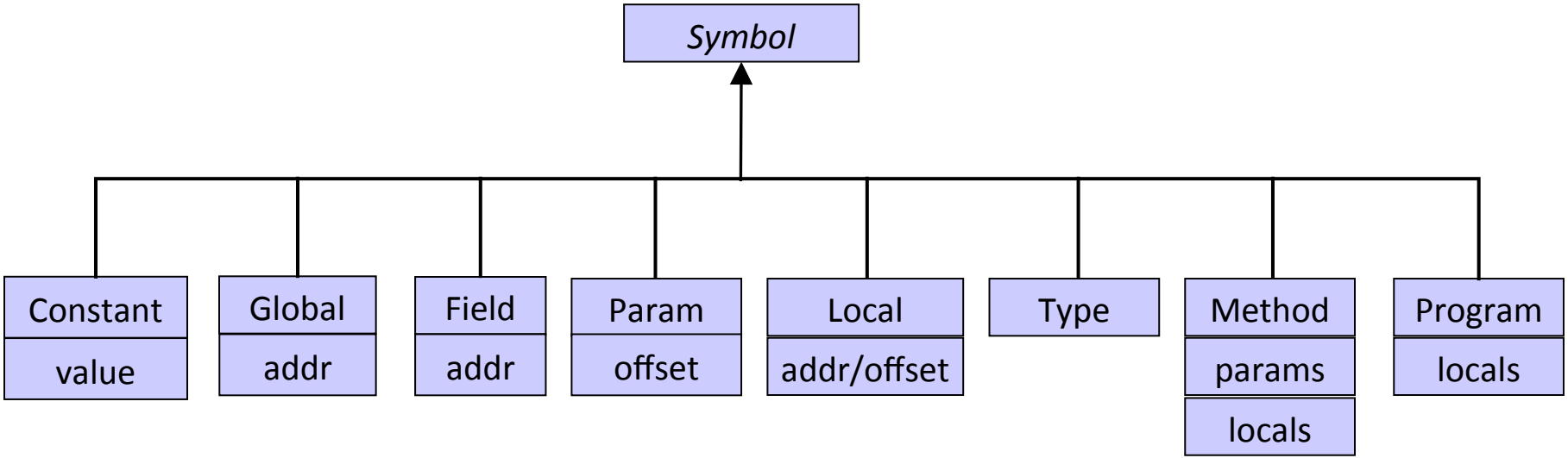
# *Структура на символната таблица*

**Важно е да се обърне внимание, че структурата на символната таблица зависи от:**

- Синтактичната структура на входния език;
- Семантиката на входния език;
- Начинът на реализация на компилатора;
- Генерацията на код от ниско ниво;

**Ще бъдат разгледани основните концепции, използвани в структурирането на символните таблици, както и символната таблица на демонстрационния проект.**

# Възможна ОО йерархия




Тази организация предполага преобразуване на типове. Например:

```
Symbol s = Table.Find("x");
if (s is Argument) ((Argument)s).addr = ...;
else if (s is Method) ((Method)s).args = ...;
   ...
}
```

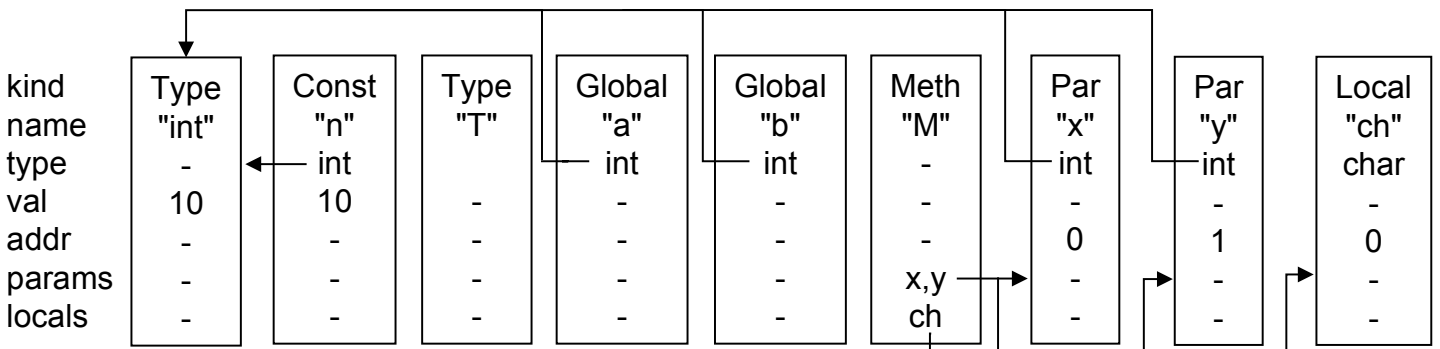
Може да се използва нейерархична структура: цялата информация се съхранява в един клас/структура.

# Нейерархична структура

```
class Symbol {
  public enum Kinds {Const, Global, Field, Par, Local, Type, Meth, Prog }
  Kinds       kind;
  string      name;
  Symbol      type;
  int         val;        // за Const: стойност
  int         addr;       // за Par, Local: адрес / отместване
  Symbol[]    params;     // Method: формални параметри (за удобство те също
                          // се включват в таблицата)
  Symbol[]    locals;     // Method: параметри & лок. пром.;
                          // Prog: символна таблица на програмата
}
```

Пример

```
const int n = 10;
class T { ... }
int a, b;
void M (int x, int y)
   char ch;
{ ... }
```



# *Предекларирани (вградени) имена*

**Какво може да се предекларира:**

- Променливи (осносно глобални);
- Класове;
- Типове;
- Константи;
- Методи;
- Могат да се предекларират елементи, които езика не позволява да бъдат дефинирани;
- и други, които биха били полезни за използване;

# *Предекларирани (вградени) имена*

**Предекларирането се използва за да:**

- “олекоти” граматиката на входния език;
- даде базисно ниво, което директно е готово за използване;
- предостави елементи, които не е необходимо програмистът да дефинира преди да използва;

**Предекларираните имена могат да се обработват:**

- като елементи на символната таблица от ниво 0 (Universe);
- като ключови думи;

# *Обработка на предекларирани имена*

**Предекларираните имена като ключови думи**

*Изискват специална обработка:*

```
Type<↑type> = 'int'          (. type = Table.IntType; .)
            | 'string‘ (. type = Table.StringType; .)
            |...
            | Ident.  (. var s = (TypeSymbol) Table. GetSymbol(token.value);
              type = s.type; .)
```

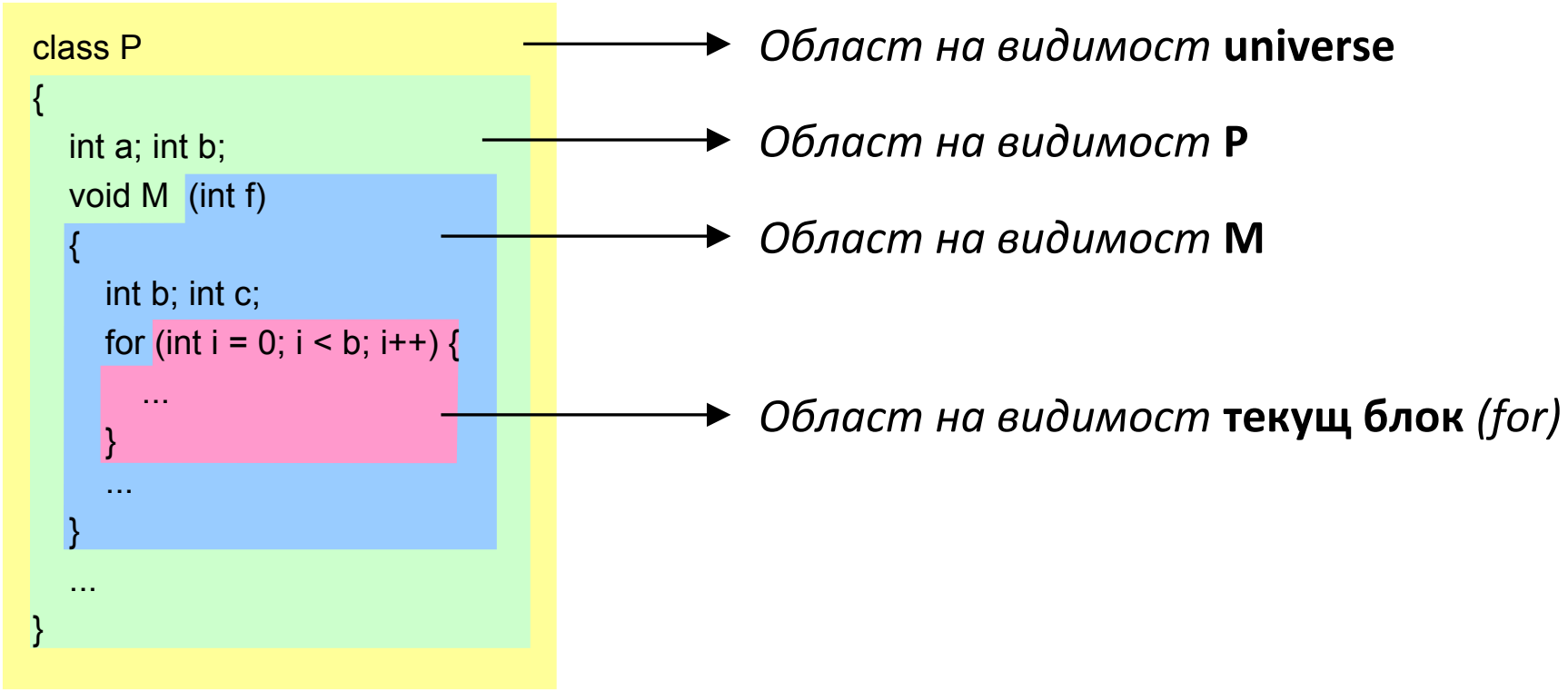
**Предекларираните имена като елементи на символната таблица**

```
Type<↑type> = Ident.        (. var s = (TypeSymbol) Table.GetSymbol(token.value);
                               type = s.type; .)
```

- не изискват специална обработка
- Типовете може да се припокриват (int може да се предекларира като потребителски

# *Област на видимост (Scope)*

Област на видимост на име е региона, в който името е валидно




*Области на видимост*

# Видове области на видимост

```
class P
{
   int a; int b;
   void M  (int f)
   {
      int b; int c;
      for (int i = 0; i < b; i++) {
         ...
      }
      ...
   }
   ...
}
```

Синтактично Вложени Области

```
interface
class P
{
   int a; int b;
   void M(int f);
   ...
}
implementation
void P::M(int f)
{
   int b; int c;
   for (int i = 0; i < b; i++) {
      ...
   } ...
}
```

Семантично Вложени Области

# *Видове области на видимост – статична*

```
// A C program to demonstrate static scoping.
//

#include<stdio.h>
int x = 10;

// Called by g()
int f()
{
    return x;
}
```

# *Видове области на видимост – статична*

```
// g() has its own variable
// named as x and calls f()
int g()
{
    int x = 20;
    return f();
}

int main()
{
  printf("%d\n", g());  // Print 10
  return 0;
}
```

# Видове области на видимост – динамични

```
// Since dynamic scoping is very uncommon in
// the familiar languages, we consider the
// following pseudo code as our example. It
// prints 20 in a language that uses dynamic
// scoping.

int x = 10;

// Called by g()
int f()
{
   return x;
}
```

# *Видове области на видимост – динамични*

```
// g() has its own variable
// named as x and calls f()
int g()
{
    int x = 20;
    return f();
}

main()
{
  printf(g()); // Print 20
}
```

# *Видове области на видимост – динамична*

```
# A perl code to demonstrate dynamic scoping
$x = 10;
sub f
{
   return $x;
}
sub g
{
   # Since local is used, x uses dynamic scoping.
   local $x = 20;
   return f();
}
print g()."\n"; # Print 20
```

# Въпроси?

apenev@uni-plovdiv.bg