Семантичен анализ

Синтактичният анализ проверява дали програмата се състои от лексеми, подредени в синтактично коректна последователност.

Семантичният анализ изследва смисъла на входната програма (разпознато изречение). Семантиката се явява изображение от *програма* на езика към *модел* на нейното поведение.

Примери за необходими условия при семантично коректна програма:

- всички променливи, функции, класове да бъдат подходящо дефинирани и да бъдат използвани в съответната област на видимост;
- променливи и изрази трябва да бъдат използвани с подходящи типове данни;
- контрол на достъпа (напр. до методи на клас).

На етапа на семантичния анализ приключва анализа (т.е. проверката за коректност) на програмата и започва генериране на код.

Основни цели на семантичния анализ:

Проверка на правилата за използване на всички конструкции, идентификатори (променливи, функции и др.) и константи на базата на:

- Правила за видимост в езика (блоковата структура на езика);
- Съвместимост на типове
- Ограничения на конкретната реализация
- Допълнителни изисквания

Важна част от семантичния анализ е обработване на декларациите на променливи/функции/типове и проверка на типове данни. В много езици идентификаторите трябва да бъдат декларирани преди да бъдат използвани. Когато семантичният анализатор срещне нова декларация (напр. на променлива или функция от някакъв тип данни), той записва информация в символната таблица за типа на данните. Когато в останалата част от програмата анализаторът срещне същия идентификатор, той проверява дали типът данни на идентификатора позволява да бъдат изпълнени съответните операции.

Примери за проверки, извършвани на етапа на семантичния анализ:

- Типът на данните в дясната страна на оператор за присвояване трябва да съвпада с типа на данните в лявата страна, а идентификаторът в лявата страна трябва да има подходяща декларация.
- Формалните параметри на функция трябва да съответстват по брой и по тип на параметрите, с които е извикана функцията (фактически параметри).
- Някои езици изискват имената на идентификаторите да са уникални. Следователно семантичният анализатор трябва да извежда грешка при срещане на два глобално декларирани идентификатора с еднакви имена.
- Операндите в аритметичните изрази трябва да са от числов тип – някои езици изискват дори еднакъв тип на операндите (т.е. напр. без int-to-double преобразуване)

Типове и декларации

Тип данни представлява множество от стойности и операции върху тях. В повечето програмни езици съществуват три групи типове:

- Базови типове **int, float, double, char, bool, etc.** Това са основни типове данни, които се предоставят във всички езици. Може да има възможност за дефиниране на потребителски тип данни на базата на основните типове (напр. C **enums).**
- Съставни типове **arrays, pointers, records, structs, unions, classes** и др. These types are constructed as aggregations of the base types and simple compound types.
- Динамични типове данни **lists, stacks, queues, trees, heaps, tables**

В много езици първо трябва да бъдат декларирани име и тип на данните (на променлива, функция, тип и др.). Освен това декларациите определят област на видимост. *Декларацията* представлява оператор, който изпраща съответната информация към компилатора. Основно декларацията се състои от име и тип, но в много езици тя включва модификатори за определяне на видимост (напр. **static** в C**, private** в Java**).** Някои езици позволяват също инициализиране на променливи, при което с един оператор се извършва декларация и задаване на стойност. Пример за декларации в C програма:

```
double   calculate(int a, double b);       // function prototype
int  x = 0;                          // global variables available throughout
double  y;                            // the program
int  main()
{
int  m[3];                            // local variables available only in main
char  *n;
...
}
```

Проверка на типове

Проверката на типовете е процес на сверяване дали всяка операция, зададена в програмата, отговаря на изискванията на езика. Това изисква операндите във всеки израз да бъдат подходящ брой и от подходящ тип.

Голяма част от семантичния анализ се състои в проверка на типовете. Понякога правилата за операциите са дефинирани в части от кода (напр. прототипи на функции), а понякога тези правила са част от дефинициите на езика (напр. правилото „и двата операнда в бинарен аритметична операция трябва да са от един и същ тип“). При откриване на нарушено правило, напр, опит за събиране на стойност на символ със стойност на реално число, анализаторът извежда съобщение за *несъвместимост на типове.*

Даден програмен език е *силно типизиран*, ако всяка грешка, свързана с типовете данни, се открива по време на компилиране.

Проверките с типове данни могат да бъдат направени по време на компилиране (*статична проверка*), по време на изпълнение (*динамична проверка*) или да бъдат разделени между тези два етапа.

Статичната проверка на типове се извършва по време на компилация. Необходимата информация се получава от декларациите в програмата и се съхранява в символната таблица.

Недостатък:

Не всички грешки, свързани с типовете, могат да бъдат открити по време на компилиране. Например, ако на две променливи **a** и **b** от тип **int** са присвоени големи стойности, тяхното произведение **a * b** може да надхвърли диапазона на представимите цели числа или частното на две цели числа може да породи грешка деление на нула. Такива грешки не могат да бъдат открити по време на компилиране.

Динамичната проверка на типове данни се извършва по време на изпълнение.

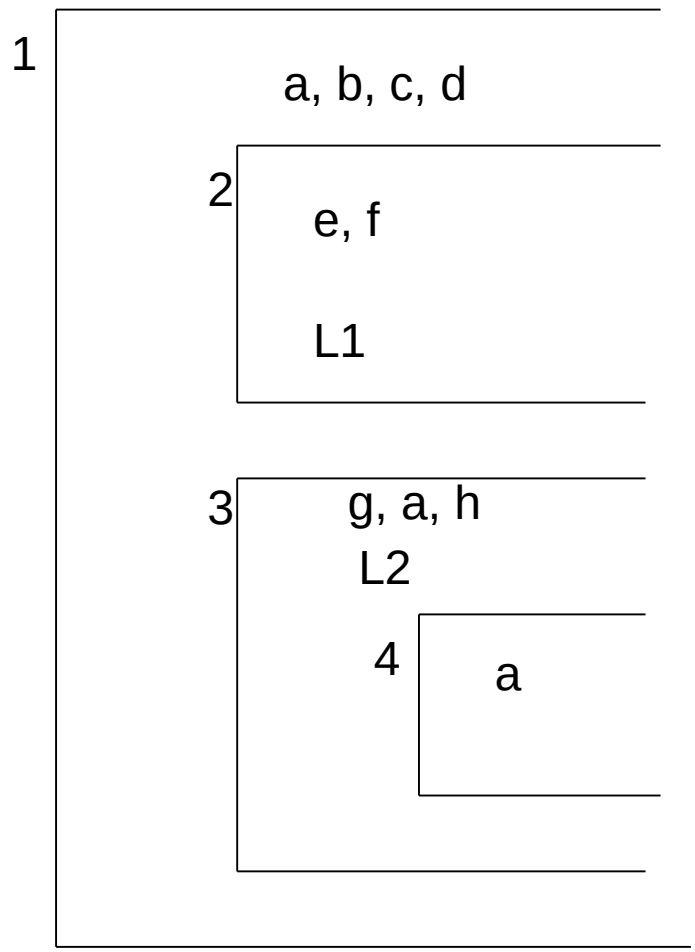
Например запис в символната таблица за променлива от тип *double* ще съдържа освен стойността на променливата и информация за типа (double). Изпълнението на всяка операция с тази променлива първо ще проверява информацията за типа на променливата. Операцията ще се изпълнява само, ако типът на променливата позволява.

Например при извикване на операция за събиране първо се проверява дали типовете на операндите са съвместими. LISP е пример за език, при който проверката на типовете се изпълнява динамично. В програма на LISP не е задължително да се задава тип на данните при деклариране на променлива. Поради това компилаторът не може да изпълни никакъв анализ на използваните типове данни. Проверката се извършва по време на изпълнение.

Динамичната проверка на типове прави изпълнението по-бавно, но открива грешки, които не могат да бъдат открити при компилиране.

Блокова структура на програма и област на видимост на променливи

```
Program  BLOCK;
  var  a, b, c, d : …
    procedure P1;
      var  e, f : …
       . . .
      begin … L1: … end;
    procedure  P2;
      var  g, a, h : …
        procedure  P3;
          var  a:…
          begin  … end;
      begin … L2: … end;
  begin …  end;
```

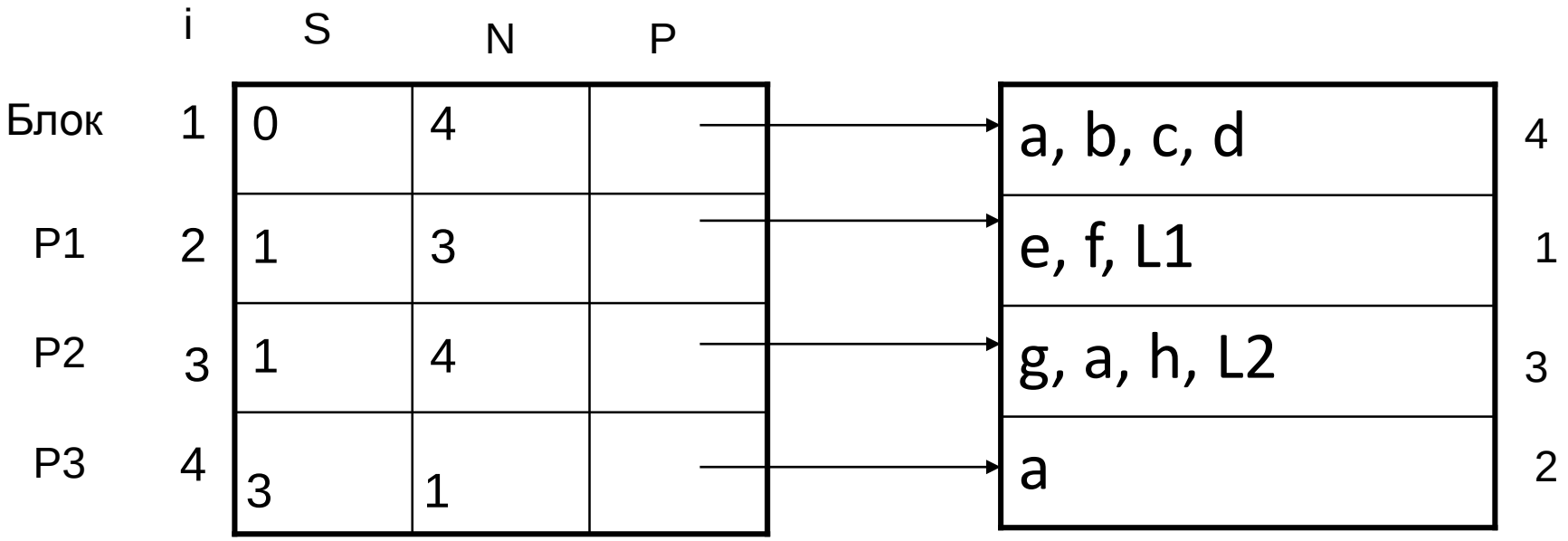


Последователност на търсене на идентификатор в блоковете

1. Търсене в текущия блок:
   - Ако идентификаторът е открит, премини към 3.
   - Ако идентификаторът не е открит, премини към 2.
2. Търсене в обхващащия блок:
   - Ако идентификаторът е открит, премини към 3.
   - Ако идентификаторът не е открит:
     - Ако блокът е най-външен, премини към 4
     - Ако блокът не е най-външен, премини към 2
3. Действия в зависимост от семантиката на езика
4. Грешка – недекларирaн идентификатор

Таблици с идентификатори

Идентификаторите се съхраняват в таблици.
Необходима е допълнителна таблица с указатели към списъка с идентификатори за всеки блок.



i – номер на блок
S – номер на обхващащ блок
N –брой идентификатори
P –указател към началото на списъка с идентификатори за блока

S[N] - символна таблица с N елемента;

B[M] – списък от блокове, зададени чрез структури с полета S, N, P

curr_bl – номер на текущия блок

last_bl – номер на най-външния блок (първоначално 0)

top_el - индекс на елемента във върха на стека (първоначално N + 1)

last_el – индекс на последния елемент в стека (първоначално 0)

При обработване на блок се изпълняват две процедури – едната се извиква в началото на блок (*open_block()*), а втората се извиква при изход от блок (*close_block()*).

<block> ::= procedure <body> end

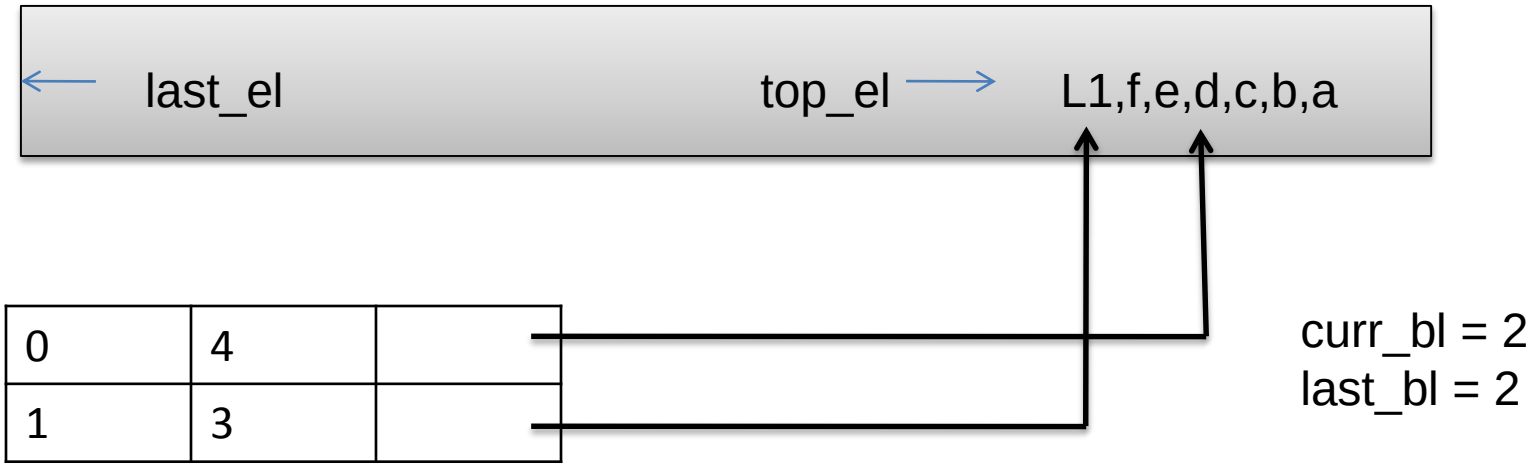
начало на блока

затваряне на блока

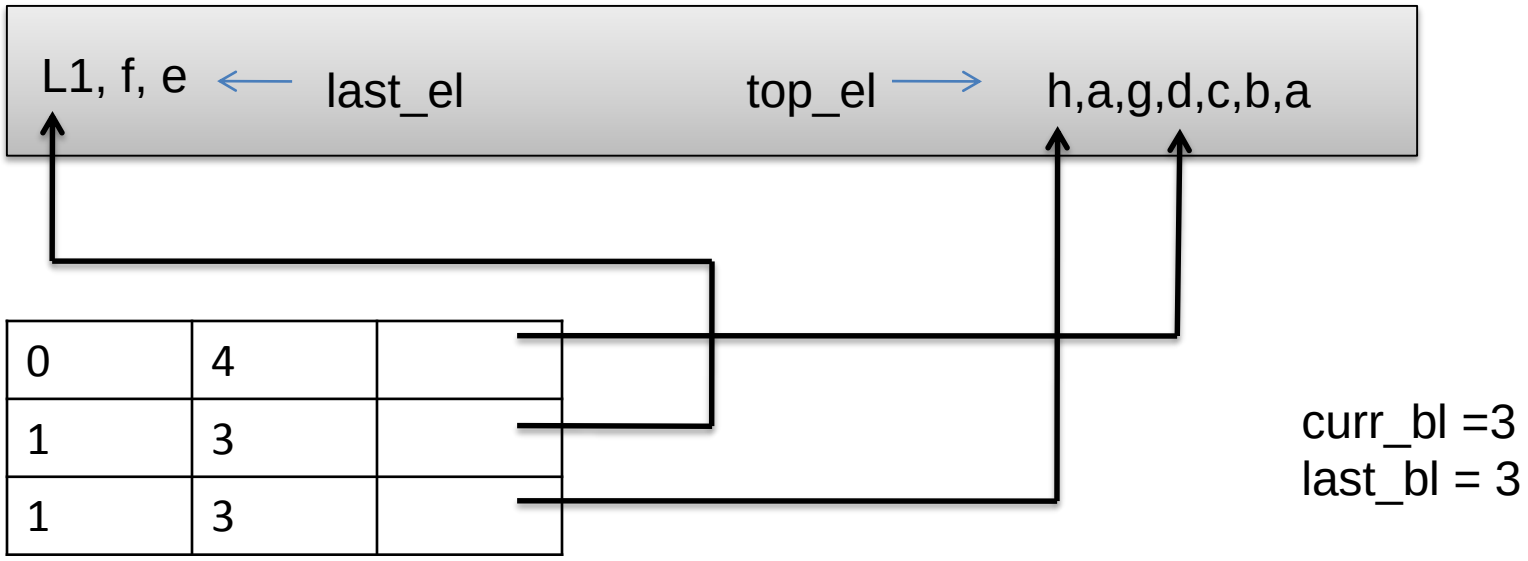
```
// semantic analysis
const N = 1000;
const M = 100;
char * S[N];
typedef struct {
	int S;
	int N;
	int P;
} block;
block B[M];
```

```
void  open_block () {
      last_bl = last_bl +1
      B[last_bl].S = curr_bl;
      B[last_bl].N = 0;
      B[last_bl].P = top_el;
      curr_bl = last_bl;
}
void  close_block () {
      int I;
      B[curr_bl].P = last_el + 1;
      for (i=1; B[curr_bl].N; i++) {
            last_el = last_el + 1;
            S[last_el] = S[top_e];
            top-el = top_el =1;
      }
      curr_el = B[curr_bl].S;
}
```

L1



h



L3