НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ
УКРАЇНИ

Теорія розпізнавання образів та класифікації в системах штучного інтелекту

Тема №7. Семантичний аналіз текстів формальної мови

Київ - 2025

Зміст

1. Семантичний аналіз.

2. Семантичний аналізатор.

Список використаних джерел

1. Серебряков - Мови програмування: http://infonet.cherepovets.ru/citforum/programming/theory/serebryakov

2. Вільна енциклопедія - Вікіпедія http://ru.wikipedia.org/wiki/% D0% A2% D1% 80% D0% B0% D0% BD% D1% 81% D0% BB% D1% 8F% D1% 82% D0 % BE% D1% 80

На етапі контекстного аналізу виявляються залежності між частинами програми, які не можуть бути описані синтаксисом контекстно-вільної мови. Полягає в основному зв'язку, асаме - «опис-використання», а саме: аналіз типів об'єктів, аналіз областей видимості, відповідність параметрів, мітки та інші. У процесі контекстного аналізу таблиці об'єктів поповнюються інформацією про описи (властивості) об'єктів.

Основним формалізмом, що використовується при контекстному аналізі, є апарат атрібутних граматик.

Результатом контекстного аналізу є атрибутовані дерево програми. ІнформаціяI про об'єкти може бути як розосереджена в самому дереві, так і зосереджена в окремих тубличних об'єктах.

У процесі контекстного аналізу також можуть бути виявлені помилки, пов'язані з неправильним використанням об'єктів.

Потім програма може бути переведена у внутрішнє представлення. Це робиться для цілей оптимізації та/або зручності генерації коду.

Ще однією метою перетворення програми у внутрішнє представлення є використання переносимого компілятора. В такому разі, тільки остання фаза (генерація коду) є машинно-залежною. В якості внутрішнього подання може використовуватися префіксниї або постфіксній запис, орієнтований граф, трійки, четвірки та інші.

Основні принципи роботи синтаксичного аналізатора

Синтаксичний аналізатор (синтаксичний розбір) - це частина компілятора, яка відповідає за виявлення основних синтаксичних конструкцій вхідної мови. У завдання синтаксичного аналізу входить: знайти і виділити основні синтаксичніі конструкції в тексті вхідної програми, встановити тип і перевірити правильність кожної синтаксичній конструкції і, нарешті, представити синтаксичні конструкції у вигляді, зручному для подальшої генерації тексту результуючої програми.

В основі синтаксичного аналізатора лежить распознаватель тексту вхідної програми на основі граматики вхідної мови. Як правило, синтаксичні конструкції мов програмування можуть бути описані за допомогою КВ-граматик, рідше зустрічаються мови, які, можуть бути описані за допомогою регулярних граматик. Найчастіше регулярні граматики застосовні до мов асемблера, а мови високого рівня побудовані па основі синтаксису КВ-мов. Розпізнавач дає відповідь на питання про те, належить чи не належить ланцюжок вхідних символів заданому в мові.

Як і у випадку лексичного аналізу, завдання синтаксичного розбору не обмежишся лише перевіркою приналежності ланцюжка заданому мови. Необхідно виконати всі перераховані вище завдання, які повинен вирішити синтаксичний аналізатор. У такому варіанті аналізатор вже не є різновидом МП-автомата – його функції можна трактувати ширше. Синтаксичний аналізатор повинен мати якиусь вихідну мову, за допомогою якогї він передає наступним фазам компіляції не тільки інформацію про знайдені і розібрані синтаксичні структури.

У такому випадку він є перетворювачем з магазинною пам'яттю - МП-перетворювачем.

Синтаксичний розбір - це основна частина компілятора на етапі аналізу.

Усі завдання з перевірки синтаксису вхідного мови можуть бути вирішені на етапі синтаксичного розбору.

Сканер тільки дозволяє позбавити складний за структурою синтаксичний аналізатор від рішення примітивних завдань з виявлення та запам'ятовування лексем вихідної програми.

Виходом лексичного аналізатора є таблиця лексем (або ланцюжок лексем). Ця таблиця утворює вхід синтаксичного аналізатора, який досліджує тільки один компонент кожної лексеми - її тип. Решта інформації про лексемах використовується на більш наступних етапах компіляції при семантичному аналізі, підготовці до генерації і генерації коду результуючої програми.

Синтаксичний аналіз (розбір) - це процес, в якому досліджується таблиця лексем і встановлюється, чи задовольняє вона структурним умов, явно сформульованим і визначених синтаксисом мови.

Синтаксичний аналізатор сприймає вихід лексичного аналізатора і розбирає його відповідно до граматики вхідної мови.

Дерево розбору. Перетворення дерева розбору в дерево операцій

Результатом роботи розпізнавача КС-граматики вхідної мови є послідовність правил граматики, застосованих для побудови вхідний ланцюжка. За знайденою послідовністю, знаючи тип розпізнавача, можна побудувати ланцюжок виводу або дерево виводу. У цьому випадку дерево виведення виступає в якості дерева синтаксичного розбору і являє собою результат роботи синтаксичного аналізатора в компіляторі.

Ні ланцюжок виводу, ні дерево синтаксичного розбору не є метою роботи компілятора. Дерево висновку містить масу надлишкової інформації, яка для подальшої роботи компілятора не потрібна. Ця інформація включає в себе всі нетермінальні сисволи, які містяться і вузлах дерева, Після побудови дерева нетермінальні символи не несуть ніякого смислового навантаження і не використовуються.

Для повного уявлення про тип і структурі знайденої і розібраної синтаксичної конструкції вхідного мови достатньо знати послідовність номерів правил граматики, що використовуються для її побудови. Форма подання цієї достатньої інформації може бути різною як залежно від реалізації самого компілятора, так від фази компіляції. Ця форма напивається внутрішнім поданням програми.

У синтаксичному дереві внутрішні вузли (вершини) відповідають операціям, а листя представляють собою операнди. Як правило, листя синтаксичного дерева співпадають з записами в таблиці ідентифікаторів.

Структура синтаксичного дерева відображає синтаксису мови програмування, на якому нашкапа похідна програма.

Синтаксичні дерева можуть бути побудовані компілятором для будь-якої частини вхідної програми. Не завжди синтаксичному дереву повинен відповідати фрагмент коду результуючої програми, наприклад, можлива побудова синтаксичних дерев для декларативної частини мови. У цьому випадку операції, наявні в дереві, не вимагають породження об'єктного коду, але несуть інформацію про дії, які повинен виконати сам компілятор над відповідними елементами. У випадку, коли синтаксичному дереву відповідає деяка послідовність операцій, що тягне породження фрагмента об'єктного коду, говорять про дерево операцій.

Дерево операцій можна безпосередньо побудувати з дерева виведення, породженого синтаксичним аналізатором. Для цього достатньо виключити з дерева виведення ланцюжка нетермінальних символів, а також вузли, що не несуть семантичного навантаження при генерації коду.

Прикладом таких вузлів можуть бути різні дужки, які змінюють порядок виконання операції і операторів, але після побудови дерева ніякого смислового навантаження не несуть, гак яким не відповідає ніякий об'єктним код.

Алгоритм перетворення дерева семантичного розбору і дерево операції

Крок 1. Якщо у дереві більше не міститься вузлів, помічених нетермінальним символами, то виконання алгоритму завершено; інакше - перейти до кроку 2.

Крок. 2. Вибрати крайній лівий вузол лерена, що позначений нетермінальним символом граматики і зробити його поточним. Перейти до кроку 3.

Крок 3. Якщо поточний вузол має тільки один ниже розташований вузол, то поточний вузол необхідно видалити з дерена, а пов'язаний з ним вузол приєднати до вузла вищого рівня (виключити з лерена ланцюжок) і повернутися до кроку 1; інакше - перейти до кроку 4.

Крок 4. Якщо поточний вузол має нижележащий вузол (лист дерева), позначений термінальним символом, який не несе семантичного навантаження, тоді цей лист потрібно видалити з дерева і повернутися до кроку 3, інакше - перейти до кроку 5.

Крок 5. Якщо поточний вузол має один нижележащий вузол (лист дерева), позначений термінальним символом, що позначає знак операції, а інші вузли позначені як операнди, то лист, позначений знаком операції, треба видалити з дерева, поточний вузол позначити цим знаком операції і перейти до кроку 1; інакше - перейти до кроку 6.

Крок 6. Якщо серед нижчих вузлів для поточного вузла є вузли, помічені нетермінальним символами граматики, то необхідно вибрати крайній лівий серед цих вузлів, зробити його поточним і перейти до кроку 3: інакше - виконання алгоритму завершено.

1. Семантичний аналіз

КВ-граматики, за допомогою яких описують синтаксис мов програмування, не дозволяють задавати контекстні умови (КУ), що є в будь-якій мові. Перевірку КУ називають **семантичним аналізом**.

Найчастіше зустрічаються контекстні умови:

- кожен ідентифікатор, що використовується в програмі, повинен бути описаний, але не більше одного разу на одній зоні опису;
- при виклику функції число та тип фактичних параметрів повинні відповідати числу та типам формальних параметрів;
- зазвичай у мові накладаються обмеження на:
  - типи операндів будь-якої операції, визначеної у цій мові;
  - типи лівої та правої частин в операторі присвоєння;
  - Тип параметра циклу;
  - тип умови в операторах циклу та умовному операторі тощо.

Перевірка КУ проводитиметьсяв залежності від того, як синтаксичний аналізатор розпізнає конструкцію, на компоненти якої накладені деякі обмеження, тобто. на етапі синтаксичного аналізу повинні виконуватись деякі додаткові дії, які здійснюють семантичний контроль.

Вставка дій у КС-граматику

Якщо для синтаксичного аналізу використовується **метод рекурсивного спуску (РС),** то контролю КУ в тіла процедур РС-метода необхідно вставити виклики додаткових " семантичних " процедур (семантичні дії).

Спочатку семантичні дії вставляються в синтаксичні правила, а потім за цими розширеними правилами будуються процедури РС-методу.

Щоб відрізняти виклики семантичних процедур від інших символів граматики, вони полягають у кутових дужках.

Фактично розширюється поняття КВ-грамматики.

Нехай у граматиці є правило

A→ a<D1 >B<D1;D2> |b C<D3 >, де

$A,B,C \in N; a,b \in T$; <Di >- є виклик процедуриDi,i= 1, 2, 3.

За таким правилом граматики процедуру для РС-методу буде такою:

```
void A ( ) {
if (c == 'a') { gc( ); D1( ); B( ); D1( ); D2( ); }
else
 if (c == 'b') { gc( ); C( ); D3( ); }
  else throw c;
```

Приклад

Написати граматику, яка породжує мову

$L = \{\alpha \in (0,1)^{+}\perp | \alpha$ містить рівну кількість 0 і 1$\}$.

Вирішити це завдання можна

- чисто синтаксичними засобами - описати ланцюжки, які мають необхідну властивість;
- за допомогою синтаксичних правил описати довільні ланцюжки з 0 та 1, а потім вставити дії для відбору ланцюжків з рівною кількістю 0 та 1.

S→ < k0 = k1 = 0; >A ⊥

A→0< k0++; >B | 1< k1++;> B

B→A |ε< if (k0! = k1) throw "ERROR !!!"; >

Семантичний аналізатор для М-мови

Контекстні умови, виконання яких треба контролювати у програмах на М-мові, такі:

- Будь-яке ім'я, що використовується в програмі, має бути описано і лише один раз.
- В операторі присвоювання типи змінної та виразу повинні збігатися.
- В умовному операторі та в операторі циклу як умови можливе лише логічне вираження.
- Операнди операції відносини мають бути цілими.
- Тип вираження та сумісність типів операндів у виразі визначаються за звичайними правилами (як у Паскалі).

Для перевірки КУ М-мови в синтаксичні правила граматики вставимо виклики процедур, які здійснюють необхідний контроль, а потім перенесемо в процедури рекурсивного спуску.

Обробка описів

У синтаксичні правила для описів потрібно вставити дії, за допомогою яких можна запам'ятати**типи**змінних та контролювати**єдиність їх опису**.

і-ий рядок таблиціTIDвідповідає ідентифікатору-лексемі виду (LEX_ID,i).

Лексичний аналізатор заповнив поле **name**; значення полів **declare** і **type** заповнюватимемо на етапі семантичного аналізу.

Розділ описів має вигляд

D→I { ,I }: [int | bool],

тобто. імені типу (int або bool) передує список ідентифікаторів.

Ці ідентифікатори (номери відповідних рядків таблиці) TID) треба запам'ятати, наприклад, у стеку цілих чисел

Stack<int, 100 >st_int ,

а коли буде проаналізовано ім'я типу, треба заповнити поля declare і type у відповідних рядках.

Функція ***void Parser::dec(type_of_lextype)***:

- зчитує зі стека номера рядків таблиці TID,
- заносить у них інформацію про тип відповідних змінних та про наявність їх описів та
- контролює повторне опис змінних.

```
parser::dec ( type_of_lex type ) {
int i;
while (! st_int.is_empty ( )) {
i = st_int.pop ();
if ( TID [ i ].get_declare ()) throw "twice";
else {
TID [ i ].put_declare ();
TID [ i ].put_type (type);
 }
}
}
```

З урахуванням наявних функцій правило виведення з діями обробки описів буде таким:

```
D→  < st_int.reset () >I<st_int.push (c_val)>
{ , I<st_int.push (c_val)> }:
[ int<dec (LEX_INT)>| bool<dec (LEX_BOOL)>]
```

Контроль контекстних умов у виразі

Типи операндів і позначення операцій зберігатимемо в стеку***Stack<type_of_lex, 100>st_lex***.

Якщо у виразі зустрічається лексема-ціле число чи логічні константи***true***або***false***то відповідний тип відразу заноситься в стек.

Якщо операнд - лексема-змінна, необхідно перевірити, чи описана вона; якщо описана, її тип треба занести в стек.

Ці дії виконуються за допомогою функції check_id:

```
void parser::check_id ( ) {
 If (TID [ c_val ].get_declare ( ))
st_lex.push (TID [ c_val ].get_type ( ) );
 else throw "not declared";
}
```

Для контролю контекстних умов кожної трійки - "операнд-операція-операнд" використовується функція check_op:

```
void Parser::check_op ( ) {
type_of_lex t1, t2, op, t = LEX_INT, r = LEX_BOOL;
t2 = st_lex.pop ( );
op = st_lex.pop ( );
t1 = st_lex.pop ( );
if (op == LEX_PLUS | op == LEX_MINUS ||op == LEX_TIMES | == LEX_SLASH)
r = LEX_INT;
if (op == LEX_OR | | op == LEX_AND)
t = LEX_BOOL;
if (t1 == t2 && t1 == t) st_lex.push( r );
else throw "wrong types are in operation";
prog.put_lex (Lex (op));
}
```

Для контролю за типом операнда одномісної операції*not*будемо використовувати функцію check_not:

```
parser ::check_not ( ) {
if (st_lex.pop ( ) ! = LEX_BOOL)
throw "wrong type is in not";
else {
st_lex.push (LEX_BOOL);
```

Порівняємо граматики, що описують вирази, що складаються із символів + , * , (,) , i:

G1: E→E+E | E*E | (E) | i G4: E→T | E+T

T→F|T*F

G2:   E→E+T|E*T|T F→i | (E)

T→i | (E)

G3: E→T+E | T*E | T G5: E→T | T+E

T→i | (E) T→F | F*T

F→i | (E)

Правила виведення виразів модельної мови з діями для контролю контекстних умов

E→E1 | E1 [= |< |>] <st_char.push(TD[c_val])> E1<check_op()>

E1→T{[+| - | or]<st_char.push (TD[c_val])>T<check_op()>}

T→F {[* | / | and]<st_char.push (TD[c_val])>F<check_op()>}

F→I<check_id()>| N<st_char.push ("int")>|
[true | false]<st_char.push ("bool")> | not F<check_not()>| (E)

Контроль контекстних умов в операторах

S→I := E | if E then S else S | while E do S | B | read (I) | write (E)

Оператор присвоєння I:=E

Контекстна умова: в операторі присвоєння типи змінної I та виразу E повинні збігатися.

В результаті контролю контекстних умов виразу E в стеку залишиться тип цього виразу (як тип результату останньої операції).

При аналізі ідентифікатора I перевіряється, чи він описаний, і його тип заноситься в той же стек (з допомогою функції check_id()). При цьому достатньо в потрібний момент рахувати з стека два елементи та порівняти їх:

```
parser::eq_type () {
if ( st_lex.pop() != st_lex.pop() )
throw "wrong types are in :=";
}
```

Правило виведення для оператора присвоєння:

I<check_id ( )>:= E<eq_type ( )>

Умовний оператор, оператор циклу, оператор введення

if E then S else S | while E do S |read (I)

Контекстні умови:

- в умовному операторі та в операторі циклу як умова можливі лише логічні вирази.
- Операнд оператора введення повинен бути описаний.

Для контролю КУ в**умовному операторі та операторі циклу**функціяeq_bool ():

```
void Parser::eq_bool () {
if ( st_lex.pop() != LEX_BOOL )
throw "expression is not boolean";
}
```

Для перевірки операнда**оператора введення**read (I) використовується таку функцію:

```
void Parser::check_id_in_read () {
if ( !TID [c_val].get_declare( ) ) throw "not declared";
}
```

Правила виведення для умовного оператора та операторів циклу та введення:

Граматика з діями для розділу описів М-мови


```
void Parser::D ( ) {
st_int.reset ( );
if (c_type! = LEX_ID) throw curr_lex;
else {
st_int.push (c_val);
gl ( );
while (c_type == LEX_COMMA) {
gl ( );
if (c_type! = LEX_ID) throw curr_lex;
else {
st_int.push (c_val); gl ( );
}
}
if (c_type != LEX_COLON) throw curr_lex;
else { gl ( );
if (c_type == LEX_INT) { dec (LEX_INT); gl ( ); }
else
if (c_type == LEX_BOOL) { dec (LEX_BOOL); gl ( ); }
else throw curr_lex;
}
}
}
```

2. Семантичний аналізатор

Семантичний аналізатор виконує такі основні дії:

- перевірку дотримання у вхідній програмі семантичних угодмови
- доповнення внутрішнього подання програми у компіляторі операторами та діями, неявно передбаченими семантикою вхідногомови

-перевірку елементарних семантичних (смислових) норм мов програмування, які безпосередньо не пов'язані з

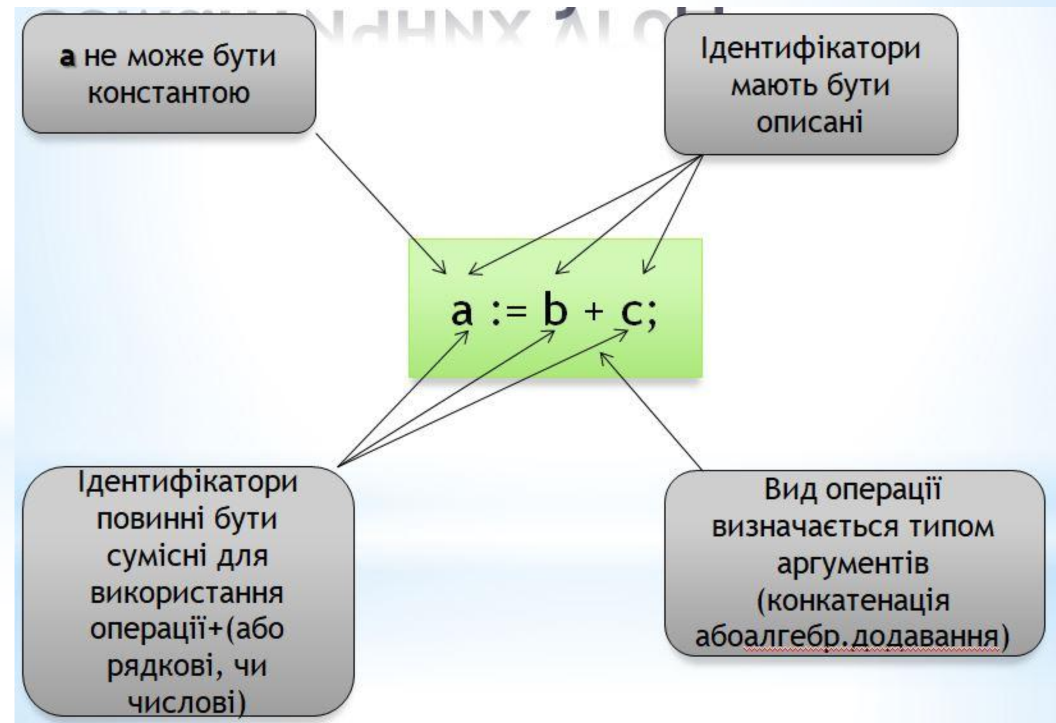
- кожна мітка, на яку є посилання, має один раз бути присутньою впрограмі

- ідентифікатор повинен бути описаний один раз і жоден ідентифікатор не може бути описаний більш ніж один раз (з урахуванням блокової структури описів)

- всі операнди у виразах та операціях повинні мати типи, допустимі для даного виразу абооперації

- типи змінних у виразах повинні бути узгоджені міжсобою

- при виклику процедур і функцій число та типи фактичних параметрів мають бути узгоджені з числом та типами формальнихпараметрів

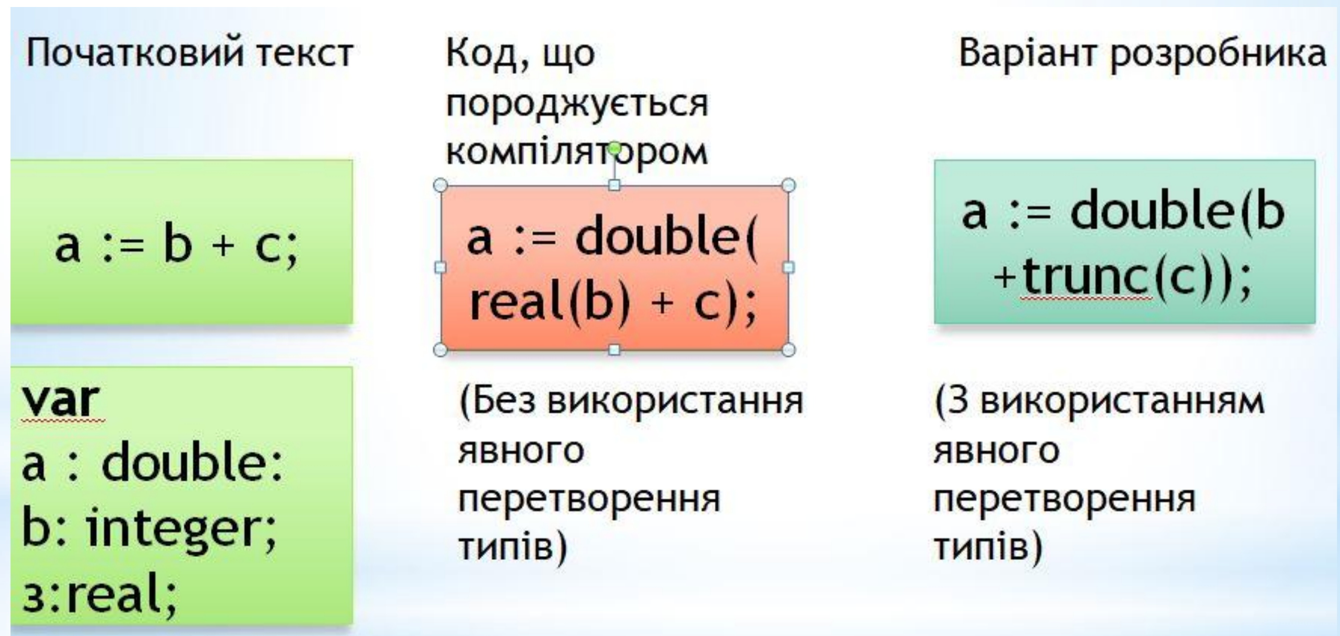


Перевірка дотримання семантичних угод

Зв'язано з додаваннямтекст програми операторівта дій, неявно передбачених семантикою вхідної мови:

- перетвореннятипів операндів у виразах та при передачі параметрів у процедури тафункції

- операції обчислення адреси, коли відбувається звернення до елементів складних структур даних



Доповнення внутрішнього представлення програми

Перевірка смислових норм мов програмування

компілятором угод, виконання яких пов'язане зі змістом як усієї вихідної програми в цілому, так і окремих їїфрагментів:

- кожна змінна або константа повинна хоча б один раз використовуватись у програмі;
- кожна змінна має бути визначена до її першого використання за будь-якого ходу виконання програми (першому використанню змінної має завжди передувати присвоєння їй будь-якого значення);
- результат функції має бути визначений за будь-якого ходуїївиконання
- кожен оператор у вихідній програмі повинен мати можливість хоча б один раз виконатись;
- оператори умови та вибору повинні передбачати можливість ходу виконання програми з кожної зі своїх гілок;
- оператори циклу мають передбачати можливість завершенняциклу
- …

Ідентифікація змінних, типів, процедур, функцій таін. лексичних одиниць мов програмування - це встановлення однозначної відповідності між лексичними одиницями та їх іменами у тексті вихідноїпрограми

імена лексичних одиниць неповинні збігатисяяк між собою, так і з ключовими словами синтаксичних конструкціймови

локальні змінні мають область видимості

```
int f_test(int a)
{         int b, c;
b = 0;
c = 0;
 if(b = 1){ return a;}
 c =a+b;
}
```

На етапі семантичного аналізу кожноїлексичній одиниці мовидається унікальнеім'я в межах усієї вихідної програми і потімвикористовується присинтезі результуючоїпрограми

- імена локальних змінних доповнюються іменами тих блоків (функцій, процедур), у яких ці змінніописані;

- імена внутрішніх змінних та функцій модулів вихідної програми доповнюються іменами самихмодулів;

- імена процедур та функцій, що належать об'єктам (класам) в об'єктно-орієнтованих мовах програмування, доповнюються найменуваннями типів об'єктів (класів), яким воналежать;

- імена процедур та функцій модифікуються в залежності від типів їх формальних аргументів та ін.