A szemantikus elemzés feladatai

Fordítóprogramok előadás (A, C, T szakirány)

A szemantikus elemzés helye

A szemantikus elemzés feladatai

A szemantikus elemzés jellemzően a környezetfüggő ellenőrzéseket valósítja meg.

- deklarációk kezelése
  - változók
  - függvények, eljárások, operátorok
  - típusok
- láthatósági szabályok
- aritmetikai ellenőrzések
- a program szintaxisának környezetfüggő részei
- típusellenőrzés
- stb.

A szemantikus elemzés erősen függ a konkrét programozási nyelvtől!

Deklarációk és láthatósági szabályok

Többszörös deklaráció

```
int x = 1;
cout << x;
int x = 2;
cout << x;
```

Elfedés

```
int x = 1;
cout << x;
{
  int x = 2;
  cout << x;
}
```

Adatelrejtés

```
class sajat
{
  public:
   int x;
  private:
   int y;
}
...
sajat s;
cout << s.x;
cout << s.y;
```


Deklarációk és láthatósági szabályok

- Jól megválasztott szimbólumtáblával megoldható:
  - verem szerkezetű szimbólumtábla (+ keresőfa vagy hash-tábla)
  - minden blokkhoz egy új szint a veremben
  - a keresés a verem tetejéről indul
- Lásd az előző előadás anyagát!

Aritmetikai ellenőrzések

- pl. a *nullával osztás* esetenként kiszűrhető:

Konstanssal osztás

```
a = b / 0;
// Itt lehet hibát vagy
// figyelmeztetést adni!
```

Változóval osztás

```
cin >> c;
a = b / c;
// Itt nem...
```

- hasonlóan kiszűrhető konstansok esetén a *túl- vagy alulcsordulás*

A szintaxis környezetfüggő részei

Példa: Ada eljárásdefiníciók

```
procedure eljarasom is
  ...
end eljarasom
```


A szintaxis környezetfüggő részei

Példa: Ada eljárásdefiníciók

```
procedure eljarasom is
  ...
end eljarasom
```

A környezetfüggetlen nyelvtan nem tudja leírni a szabályt:

$$EljDef \rightarrow \underline{procedure}\ Azonosito\ \underline{is}\ Program\ \underline{end}\ Azonosito$$

A szintaxis környezetfüggő részei

Példa: Ada eljárásdefiníciók

```
procedure eljarasom is
  ...
end eljarasom
```

A környezetfüggetlen nyelvtan nem tudja leírni a szabályt:

$$EljDef \rightarrow \underline{procedure}\ Azonosito\ \underline{is}\ Program\ \underline{end}\ Azonosito$$

Ez megengedi ezt a programszöveget is:

Hibás eljárásdefiníció

```
procedure egyik is
  ...
end masik
```

Ezt a hibát a szemantikus elemzésnek kell megtalálnia.

Típusellenőrzés

- első közelítésben a kifejezések típusozhatóságának ellenőrzése

$s.length() + 1$

```
s               :: string
s.length()      :: int
1               :: int
s.length() + 1  :: int
```

s + 1

```
s      :: string
1      :: int
s + 1  :: hiba
```


Típusellenőrzés

- első közelítésben a kifejezések típusozhatóságának ellenőrzése

$s.length() + 1$

```
s               :: string
s.length()      :: int
1               :: int
s.length() + 1  :: int
```

s + 1

```
s      :: string
1      :: int
s + 1  :: hiba
```

- mára nagyon kifejező típusrendszerek léteznek
  - sablon (generikus) típusok
  - altípusok
  - öröklődés
  - minél több információ tárolása a típusban (pl. string hossza, gráf párossága stb.)

A típusok szerepe

- időnként a típushibás utasításokhoz nem lehet értelmesen kódot generálni
  - pl. különböző méretű rekordok közötti értékadás
- a típusellenőrzés számos elírást felderít

```
string s;
if( s = ''hello'' )
// ...
```

Statikus vs. dinamikus típusozás

- Statikus: a kifejezésekhez *fordítási időben* a szemantikus elemzés rendel típust
  - az ellenőrzések fordítási időben történnek
  - futás közben csak az értékeket kell tárolni
  - futás közben „nem történhet baj"
  - előny: biztonságosabb
  - pl.: Ada, C++, Haskell ...
- Dinamikus: a típusellenőrzés *futási időben* történik
  - futás közben az értékek mellett típusinformációt is kell tárolni
  - minden utasítás végrehajtása előtt ellenőrizni kell a típusokat
  - típushiba esetén futási idejű hiba keletkezik
  - előny: hajlékonyabb
  - pl.: Lisp, Erlang ...

Statikus és dinamikus típusozás

Bizonyos feladatokhoz használni kell a dinamikus típusellenőrzés technikáit:

- objektumorientált nyelvekben a *dinamikus kötés*
- Java *instanceof* operátora

Típusellenőrzés vs. típuslevezetés

C++

```
int fac( int n )
{
  if( n == 0 )
    return 1;
  else
    return n * fac(n-1);
}
```

Haskell

```
fac n =
  if n == 0
  then 1
  else n * fac (n-1)
```


Típusellenőrzés vs. típuslevezetés

- Típusellenőrzés:
  - minden típus a deklarációkban adott
  - a kifejezések egyszerű szabályok alapján típusozhatók
  - egyszerűbb fordítóprogram, gyorsabb fordítás
  - kényelmetlenebb a programozónak

Típusellenőrzés vs. típuslevezetés

- Típusellenőrzés:
  - minden típus a deklarációkban adott
  - a kifejezések egyszerű szabályok alapján típusozhatók
  - egyszerűbb fordítóprogram, gyorsabb fordítás
  - kényelmetlenebb a programozónak
- Típuslevezetés, típuskikövetkeztetés:
  - a változók, függvények típusait (általában) nem kell megadni
  - a típusokat fordítóprogram „találja ki" a definíciójuk, használatuk alapján
  - bonyolultabb fordítóprogram, lassabb fordítás
  - kényelmesebb a programozónak

Automatikus típuskonverziók

```
void f1( double d ) {}
void f2( int i ) {}
int main()
{
  int i;
  double d;
  f1(i); // ez megy
  f2(i);
  f1(d);
  f2(d); // ez problémás ...
  return 0;
}
```

Warning

```
warning: passing 'double' for argument 1
         to 'void f2(int)'
```

Automatikus típuskonverziók

- C++ példa:
  - *int* $\rightsquigarrow$ *double* automatikusan konvertálódik
  - *double* $\rightsquigarrow$ *int* figyelmeztetést vált ki (elvesztjük a törtrészt)
- ha nincs automatikus típuskonverzió:
  - ki kell írni: `f2( (int)d );`
  - kényelmetlenebb, de biztonságosabb
- ha nagyon sok automatikus típuskonverzió van:
  - időnként meglepő eredmény születhet
  - kényelmesebb, de veszélyes lehet

Típuskonverziók és a fordítóprogram

- Típuskonverzió esetén:
  - a típusellenőrzés során át kell írni a kifejezés típusát
  - ha szükséges, akkor a tárgykódba generálni kell a konverziót elvégző utasításokat
    - az *int* és a *double* típusok reprezentációja különbözik!

Típusok fajtái

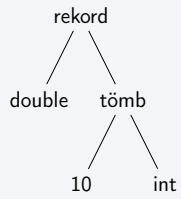
- alaptípusok
  - pl. int, double, char ...
- összetett típusok
  - tömb
  - rekord (osztály ...)
  - unió

Típusok fajtái

- alaptípusok
  - pl. int, double, char ...
- összetett típusok
  - tömb
  - rekord (osztály ...)
  - unió

Összetett típusok fa-struktúrája:

```
struct T
{
  double d;
  int t[10];
};
```



Típusok ekvivalenciája

Mikor tekintsünk két típust ekvivalensnek?

- szerkezeti (strukturális) ekvivalencia:
  „Két típus egyenlő, ha az őket leíró fa azonos."
  - bonyolultabb az ellenőrzés
  - nem mindig fejezi ki a programozói szándékot
    - `struct Ember { string nev; int eletkor; };`
    - `struct Uzenet { string szoveg; int azonosito; };`
- név szerinti ekvivalencia:
  „Két típus egyenlő, ha a nevük azonos."
  - egyszerűbb az ellenőrzés (==)
  - időnként kényelmetlen lehet a programozónak

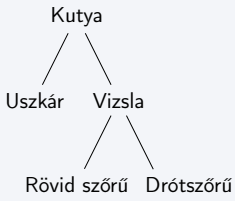
Altípusok, származtatott típusok

- altípus: általában valamilyen megszorítást tesz az alaptípusra
  - pl. a *természetes szám* típus altípusa az *egész szám* típusnak
- származtatott típus:
  - öröklődéssel jön létre az alaptípusból
  - a specializáció sokféle lehet:
    - új adattag
    - új metódus
    - meglévő metódus felüldefiniálása

Upcast: Altípus vagy származtatott típus mindenhol használható, ahol az alaptípus megengedett.
Downcast: Az alaptípus általában nem használható a származtatott vagy altípus helyett.

Altípusok, származtatott típusok

Az öröklődési vagy altípus kapcsolatok a szokásos módon ábrázolhatók fával (többszörös öröklődés esetén gráffal):



Altípusok, származtatott típusok

Upcast

```
void f( Kutya k )
{ ... }

int main()
{
  Vizsla v;
  f( v );  // gond nélkül megy
}
```


Altípusok, származtatott típusok

Upcast

```
void f( Kutya k )
{ ... }

int main()
{
  Vizsla v;
  f( v );  // gond nélkül megy
}
```

Típusellenőrzés: a fában felfelé kell keresni, hogy van-e olyan ősosztály (alaptípus), ami használható az adott helyen.