Analiza semantică

Limbaje formale şi translatoare (Compilatoare)

Analiza semantică

- Analiza lexicală
- – detectează atomii lexicali care nu aparţin limbajului respectiv
- Analiza sintactică
- – detectează dacă atomii lexicali sunt scrişi într-o ordine eronată
- Analiza semantică
- – se ocupă de restul erorilor, şi anume de erorile care privesc sensul atomilor lexicali
- – reprezintă ultima fază a componentei front-end a compilatorului

Analiza semantică

- Analiza semantică este necesară ca o etapă separată deoarece analiza sintactică nu este capabilă să identifice toate erorile posibile.
- Cauza este faptul că unele construcţii ale limbajului sunt dependente de context.

- Exemplu:
- string a = “1234”; a++;
- Lex: KW ID OP_EQ CTS SEP_PV ID OP_INC SEP_PV
- Yacc: corectă din punct de vedere sintactic
- Sem: incorectă din punct de vedere semantic, deoarece operatorul incrementare se poate aplica numai dacă variabila este de tip întreg

Analiza semantică

- Analiza semantică constă dintr-o serie de verificări care au ca şi scop stabilirea dacă secvenţa de atomi lexicali din arborele sintactic are sens în contextul existent.
- Verificări posibile:
- – **nu se folosesc identificatori** (nume de variabile, nume de funcţii, nume asociate unor constante, nume de tipuri etc.) **care nu au fost definiţi**
- Excepţie: – **non typed variabiles** (ASP)
- Ex: var x = 10; (după atribuire x primeşte tipul int – tip stabilit static, la compilare)
- – **Anonymous variables** (C#)
- Ex: var x = 10; (variabila x rămâne fără tip chiar şi după atribuire, în continuare putând fi folosită pentru a i se atribui orice alt tip de valoare – tip stabilit dinamic, la rulare)

- Tipul se deduce astfel din context.

Analiza semantică

- **- tipurile identificatorilor folosiţi în cadrul unei expresii sunt compatibile**
- Ex: 5 * “ATM”
- Limbaje strongly typed: o variabilă poate fi folosită numai în conformitate cu tipul cu care a fost definită (C#)
- Ex: int a; Console.Writeline(“{0}”, a);
- Limbaje weakly typed: o variabilă poate fi folosită şi altfel de cum a fost definită (C, C++)
- Ex: int a; printf(“%c”, a);
- **- identificatorii sunt folosiţi în conformitate cu structura lor**
- Ex: a.b (a este un tip compus şi are ca şi membru pe b)
- V[10] (v este o variabilă multidimensională)
- f(5,”ATM”) (f este o funcţie care acceptă doi parametrii, primul de tip întreg, iar al doilea de tip şir de caractere)
- **- relaţiile de moştenire sunt corecte**
- **- clasele, structurile, enumerările au fost definite o singură dată**

Analiza semantică

- **– Metodele din cadrul claselor au fost definite o singură dată**
- **– funcţiile din afara claselor au fost definite o singură dată**
- **– identificatorii sunt accesaţi în conformitate cu modul de acces declarat**
- Ex: public/private, const

- Toate aceste verificări depind de limbajul pentru care se face analiza semantică şi de aceea nu se poate elabora un algoritm generic.

Domeniul de valabilitate (Scope)

- Are în vedere potrivirea care trebuie să existe între declaraţia unui utilizator şi utilizarea acestuia.
- **Domeniul de valabilitate al unei variabile** reprezintă porţiunea din program în care variabila respectivă este accesibilă.
- Acelaşi identificator (acelaşi nume) se poate referi la entităţi semantice diferite, în porţiuni diferite din program. Singura condiţie este ca domeniile de valabilitate ale acestor entităţi identificate cu acelaşi nume să nu se suprapună.
- Ex: utilizarea aceluiaşi nume pentru variabile de tip diferit, pentru variabile şi funcţii sau metode, pentru variabile şi nume de clase, ş.a.

Domeniul de valabilitate

- Un identificator poate avea un domeniu de valabilitate restrâns.
- Ex: - variabilele locale unei funcţii pot fi accesate numai în cadrul codului funcţiei
- - în schimb, variabilele globale pot fi accesate de oriunde din cadrul programului în care au fost declarate
- - atributele şi metodele definite într-o clasă sunt accesibile numai pentru instanţele clasei respective

Domeniul de valabilitate

- Domeniul de valabilitate poate fi:
- – **static** – depinde numai de textul programului şi nu este influenţat de comportamentul din momentul rulării
- Legătura dintre date şi identificator (în cazul variabilelor), şi dintre cod şi identificator (în cazul metodelor şi funcţiilor) este statică.
- Ex: majoritatea limbajelor de programare
- – **dinamic** – depinde de execuţia programului
- Ex: Lisp, SNOBOL
- O variabilă cu domeniu de valabilitate dinamic se referă la cea mai apropiată legătură din execuţia programului.

Domeniul de valabilitate

- Legăturile dintre identificatori şi date sau cod sunt definite de către:
- – declaraţiile claselor (rezultă numele claselor)
- – definiţiile metodelor (rezultă numele metodelor)
- – declaraţiile de variabile (rezultă identificatorii variabilelor)
- – parametrii formali din definiţiile funcţiilor (rezultă identificatorii variabilelor locale)
- – definiţiile atributelor unei clase/structuri (rezultă identificatorii membrilor clasei/structurii)

Contexte

- Contextele păstrează definiţiile şi declaraţiile curente necesare analizei semantice:
- – pentru variabile: numele, tipul, dacă au fost sau nu iniţializate, valoarea iniţială
- – pentru clase/structuri: numele şi structura noilor tipuri
- – pentru funcţii: numele, tipul datei returnate, numărul şi tipul parametrilor de intrare, precum şi modalitatea de transmitere a acestora
- – ş.a. în funcţie de limbaj

Contexte

- Pe măsură ce se întâlnesc instrucţiunile de declarare şi definiţie a variabilelor/tipurilor/funcţiilor ş.a. se adaugă contextului curent informaţiile despre fiecare în parte.
- La întâlnirea instrucţiunilor care folosesc variabile/tipuri/funcţii, se face analiza semantică a acestora pe baza informaţiilor memorate în contextul curent.

Contexte

- Contextele sunt implementate prin intermediul tabelei de simboli.
- Cu alte cuvinte, tabela de simboli este cea care memorează legăturile curente existente între identificatori şi date, respectiv cod.
- În general aceasta este o tabelă de dispersie asupra căreia se pot realiza următoarele operaţii:
  - deschidere context nou,
  - adăugarea unui nou identificator împreună cu informaţiile de rigoare,
  - căutarea unui identificator (verificarea dacă un identificator a fost definit în contextul curent),
  - extragerea informaţiilor privind un identificator existent,
  - închiderea unui context inclusiv cu ştergerea identificatorilor din contextul respectiv.

Tipuri

- Definiţia tipurilor variază de la limbaj la limbaj, dar partea comună a acestor definiţii este: tipul este un set de valori împreună cu operaţiile care se pot efectua asupra acestora.
- Clasificare:
- – tipuri simple, de bază (int, char, float, double, bool, union) – sunt suportate chiar de către procesor – nu au corespondenţă în lumea reală
- – tipuri compuse (variabile multidimensionale, pointeri, structuri, clase) – încearcă să ofere o corespondenţă cu lumea reală
- – tipuri complexe (liste, arbori) – nu sunt suportate direct de către limbaj, dar pot fi implementate

Tipuri de bază

- Nu au informaţie semantică suplimentară.
- Excepţie face tipul enum.
- Deoarece sunt deja implementate în hardware, ele nu trebuie create.
- Variabilele declarate pentru un tip de bază trebuie să aibă o referinţă către tipul lor.

Tipuri compuse

- Pentru fiecare tip compus trebuie salvată o listă de perechi nume-tip care să memoreze informaţiile despre membrii tipului (atribute şi metode).
- Aceste informaţii sunt salvate ca şi context, în tabela de simboli.

Variabile multidimensionale

- Trebuie să se memoreze:
- – numele variabile
- – tipul variabilei (care poate fi de bază sau compus)
- – dimensiunea/dimensiunile

Pointeri

- Trebuie memorat:
- – numele
- – tipul (tip de bază sau compus)

Verificarea de tip (type checking)

- Verifică dacă operaţiile folosesc ca şi operanzi tipuri potrivite.
- Această validitate a tipului versus operaţia curentă depinde de fiecare tip în parte, şi de limbaj.
- Orice nerespectare în această privinţă va ridica o eroare de tip.
- Dacă toate erorile de tip pot fi verificate de către compilator, atunci limbajul este strongly typed. Altfel, el se numeşte weakly typed.

Verificarea de tip

- Verificarea de tip se poate face:
- – static, adică la compilare (C, C++)
- – dinamic, adică la rularea programului (variabilele anonime din C#, Perl, Phyton, Ruby)
- – fără verificare de tip (limbajul de asamblare)
- Pro verificare statică:
- – descoperă foarte multe erori încă în etapa de compilare
- – elimină execuţia de cod suplimentar la rulare
- Pro verificare dinamică:
- – tipurile statice sunt restrictive

Verificarea de tip

- Verificarea de tip – verificarea programelor în care toate elementele au atribuit un tip
- Inferenţa de tip – completarea informaţiei privind tipul, informaţie care lipseşte, pornind de la context
- Sinteza de tip – determinarea tipului unei construcţii (expresii) pe baza tipurilor membrilor ei

- Concepte diferite, dar denumirile lor sunt adesea folosite interschimbabil.

Verificarea de tip

- Construcţiile limbajelor care au asociat un tip sunt:
- – constantele
- – variabilele
- – funcţiile
- – expresiile
- – instrucţiunile
  - – condiţia lui if, while, do-while în C# trebuie să aibă tipul bool
  - – valoarea pe care se face switch trebuie să aibă un tip întreg (C/C++), sau inclusiv string (C#)

Verificarea de tip

- Formalismul verificării de tip este reprezentat de către regulile logice ale inferenţei.
- Regulile de inferenţă au forma:
  - Dacă Ipoteza este adevărată, atunci şi concluzia este adevărată.
- Raţionamentul pentru verificarea de tip este:
  - Dacă E1 şi E2 au anumite tipuri, atunci E3 are un anumit tip
- Ex:
- int a,b=3; a = b + 3.4;

Verificarea de tip

- Există tipuri compatibile.
- Compatibilitatea în ambele direcţii vs. Compatibilitatea într-o singură direcţie
  - Ex: int este compatibil cu float, dar float nu este compatibil cu int
- Conversii de tip – cast
- Implicit cast
  - float a = 3;
- Explicit cast
  - Int a = (int)3/4;

Verificarea de tip

- Subtipuri – un subtip poate fi folosit totdeauna în locul tipului părinte.
- Ex:
- enumerările – tipul părinte este tipul întreg
- Moştenirea, polimorfismul

Acţiuni din analiza semantică

- Pentru declaraţii (variabile şi funcţii):
  - Adăugarea de informaţii în tabela de simboli
  - Verificarea redefinirii unui identificator
  - Verificarea definirii tipului folosit în declaraţia respectivă
  - Ş.a.
- Pentru instrucţiuni:
  - Verifică regulile specifice fiecărei instrucţiuni în parte (implică verificarea de tip)
- Pentru expresii:
  - Verificarea de tip, eventual sinteza şi/sau inferenţa de tip

Bibliografie

- http://cursuri.cs.pub.ro/~cpl/Curs/CPL-Curs04.pdf