Metody Realizacji Języków Programowania

Analiza semantyczna

Marcin Benke

MIM UW

15 listopada 2010

Analiza semantyczna

- Analiza nazw
  - Czy *x* jest zadeklarowane przed użyciem?
  - Która deklaracja *x* obowiazuje w danym miejscu programu?
  - Czy jakieś nazwy są zadeklarowane a nie używane?
- Analiza zgodności typów
  - Czy wyrażenie *e* jest poprawne typowo?
  - Jakiego typu jest *e*?
  - Czy funkcja zawsze zwraca wartość typu zgodnego z zadeklarowanym?
- Identyfikacja operacji
  - Jaką operację reprezentuje $+$ wyrażeniu $a + b$?

Odpowiedzi na te pytania mogą wymagać informacji nielokalnych — *kontekstowych*. Nie są to własności bezkontekstowe.

Tablica symboli

- Opis wszystkich bytów (zmienych, funkcji, typów, klas, atrybutów, metod,. . .) wystepujących w programie.
- Musi mieć narzuconą strukturę (mechanizm wyszukiwania), odzwierciedlającą reguły wiązania identyfikatorów w danym języku.
- Opis bytu:
  - rodzaj definicji
  - inne informacje zależne od rodzaju
- Byty mogą być wzajemnie powiązane.

Struktura języka a struktura tablicy symboli

Niektóre konstrukcje językowe narzucające strukturę tablicy symboli:

- zagnieżdzanie (struktura blokowa)
- dziedziczenie
- sumowanie (moduły)

```
import java.util.*
class A {
  int a,b;
  class B extends A {
    B() {}
    int f() {
      String a;
    }
}
```

Zasięg i zakres

Zasięg definicji identyfikatora to obszar programu, w którym możemy użyć identyfikatora w zdefiniowanym znaczeniu. Nie musi być ciągły.

Zakres to konstrukcja składniowa, z którą mogą być związane definicje identyfikatorów (funkcja, blok, itp.)

Przykład

```
void f() {
   int a;
   a = g();
   {
     string a;
     b = a;
   }
   h(a,b);
}
```

Zasięg deklaracji `int a` jest zaznaczony na czerwono. Jest ona związana z zakresem funkcji f.

Struktura blokowa

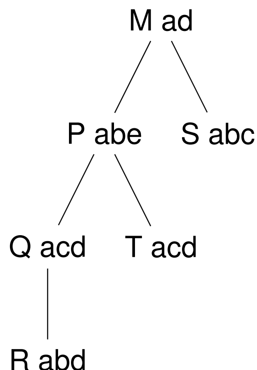
```
module M;
  var a,d : int;
  type t = ...
  procedure P;
    var a,b,e : int
    procedure Q;
      var a,c,d : t;
      procedure R;
        var a,b,d : real;
    end
    procedure T;
      var a,c,d : int;
    end
  procedure S;
    var a,b,c : int;
    d := e+1; // Gdzie są definicje d i e?
  ...
```

Drzewo zagnieżdżeń

Problem: analizujemy węzeł drzewa struktury, np przypisanie `d:=e+1`. Gdzie są definicje `d` i `e`?

Drzewo zagnieżdżeń

Problem: analizujemy węzeł drzewa struktury, np przypisanie `d:=e+1`. Gdzie są definicje `d` i `e`?



Metoda I: stos tablic symboli

Wyszukiwanie:

- przeszukaj zakresy od bieżącego do znalezienia lub do końca,
- jeżeli nie znaleziono, to dodaj fikcyjną definicję dla uniknięcia kaskady błędów.

Wejście do zakresu:

- połóż na stos nową tablicę symboli,
- umieść w niej definicje związane z tym zakresem

Wyjście z zakresu:

- zdejmij ze stosu ostatnią tablicę symboli

Metoda II: tablica stosów

Dla każdego identyfikatora tworzymy osobny stos odwołań do jego definicji

Niezmiennik: w trakcie analizy, dla każdego identyfikatora na szczycie stosu jest odsyłacz do aktualnej definicji (lub stos pusty).

Wejście do zakresu: przechodzimy listę definicji związanych z zakresem i wkładamy odsyłacze do nich na odpowiednie stosy.

Wyjście z zakresu: przechodzimy ponownie listę definicji i zdejmujemy odsyłacze ze stosów.

W porównaniu z Metodą I nieco więcej pracy na granicach zakresów, ale za to szybsze wyszukiwanie.

Zagadka

```
class A {
char a;
A() { a = 'A';}
}
class B {
    char a;
    B() { a = 'B'; }
    class C extends A {
        public char c;
        C() { c = a; }
    }
    C C() { return new C(); }
}
...
B b = new B(); B.C c = b.C();
```

Jaką wartość ma `c.c` ?

Dziedziczenie

- Jak widać z powyższego przykładu, dziedziczenie nieco komplikuje wyszukiwanie.
- Przy pojedynczym dziedziczeniu możemy przy wchodzeniu do zakresu podklasy wkładać na stos(y) definicje z nadklasy.
- Innym rozwiązaniem jest modyfikacja metody I: zamiast stosu - graf acykliczny tablic symboli.
- Każda tablica ma dowiązanie do tablic ewentualnych nadklas i zakresu obejmującego.

Przykład

global:
type int
class Object,A,B

Object

A:
int a

B:
int a
con B()
class C
C C()

C:
int c
con C()

Tablica nazw

- Przechowywanie identyfikatorów jako napisów powoduje znaczące koszty (czasowe, pamięciowe)
- Aby tego uniknąć, każdy identyfikator zapisujemy w tablicy nazw (tylko raz!), w zamian dostajemy numerek.
- Tablica nazw = zbiór napisów + mechanizm wyszukiwania identyfikator $\rightarrow$ unikalny numerek
- Implementacja: np. pula napisów + tablica mieszająca (*hash table*)
- Najlepiej zrobić to na etapie analizy leksykalnej.
- W drzewie struktury identyfikator reprezentowany jest już przez numerek.

Pula napisów

- Duża tablica znaków + indeks pierwszego wolnego miejsca
- Wstawianie napisu *s*:
  - Sprawdzamy czy *s* jeszcze nie ma w puli (jak? o tym za chwilę)
  - Jeśli nie, wstawiamy na koniec
  - Przesuwamy znacznik
  - Wynik: pozycja wstawionego napisu w tablicy
- W puli nie wyszukujemy — dostęp tylko przez indeks.
- **W zasadzie nigdy nie usuwamy**

NB Java udostępnia pulę napisów za pośrednictwem metody `String.intern()`, ale niezbyt efektywną.

Funkcje mieszające

Funkcja mieszająca

$$h : napis \rightarrow [0..N-1]$$

taka, że:

1. można szybko obliczyć wartość dla danego napisu
2. małe prawdopodobieństwo kolizji

- Funkcje skrótu (MD5, SHA) mają małe prawdopodobieństwo kolizji ale są wolne.
- Dobre funkcje mieszające używają 3–4 instrukcji procesora na bajt napisu
- Zwykle dostępne w bibliotekach.

Prosta tablica mieszająca

- Tablica T indeksowana od 0 do $N - 1$, gdzie N jak w funkcji mieszającej.
- Elementami są rekordy (indeks w puli, numerek, następny)
- Inaczej: elementami tablicy są listy par (indeks, numerek)
- Możemy też pójść na skróty i uzywać adresów w puli jako numerków

Przykładowa implementacja (fragment)

```
const char* Hash::find(const char* s,int len) const
{
  unsigned h = hash(s,len);
  iterator i = T[h].begin();
  for(;i!= T[h].end(); i++)
    if(!strcmp(s,*i)) return *i;

  return 0;
}
const char* Hash::insert(const char* s, int len)
{
  unsigned h = hash(s,len);
   T[h].push_front(s);
  return s;
}
```

```
const char* NameTab::insert(const char* s, int len) {
  const char* ns;
  if(ns = hashTab->find(s,len)) {
    repeated++;
    return ns;
  }else{
    ns = stringPool->add(s,len);
    if(ns)
      hashTab->insert(ns,len);
    else
      return 0;
  }
}
```

Wariacje na temat

- *Otwarte adresowanie:* zamiast list, przechowujemy pojedyncze elementy, w razie kolizji (adres zajęty) wyliczamy nowy adres.
- Metoda młotka: jeśli w bibiotece mamy młotek (słownik), to używamy słownika $\mathit{napis} \mapsto \mathit{numerek}$. Efekt: oszczędność czasu programisty, ale większe (ok 2x) zużycie pamięci oraz (ok 4x) czasu procesora.
- Szczególne traktowanie krótkich (np. jednoznakowych) identyfikatorów — analizator leksykalny przydziela im numerek od razu, np.

  ```
  [A-Za-z]     { yylval = int(*yytext);
                 return IDENTYFIKATOR; }
  ```

  Warte rozważenia, zwłaszcza, że takie identyfikatory są częste.

Benchmark ($\sim 10^6$ linii kodu)

Pula napisów, tablica mieszająca

```
benke@students$ find /usr/include -name \*.h
              | xargs cat | /usr/bin/time ./pooled
Identifier lexems: 11992028
Repeats: 11239094
8.71user 0.26system 0:09.02elapsed 99%CPU
0inputs+0outputs (0major+7626minor)pagefaults 0swaps
```

Słownik, bez puli

```
benke@students$ find /usr/include -name \*.h
              | xargs cat | /usr/bin/time ./mapped
Identifier lexems: 11992028
Repeats: 11239094
38.09user 0.29system 0:38.44elapsed 99%CPU
0inputs+0outputs (0major+12426minor)pagefaults 0swaps
```

Kod benchmarku

```
void test() {
 NameTab nameTab;
 int identifiers = 0;
 int shorts = 0;
 while(yylex()) {
    identifiers ++;
    if(!nameTab.insert(yytext,yyleng))
      abort("FATAL: memory exhausted");
 }
 cout << "Identifier lexems: "
      << identifiers << endl;
 cout << "Repeats: " << nameTab.repeated << endl;
}
```

Ćwiczenie: zaimplementuj swoją tablicę nazw i przetestuj ją.