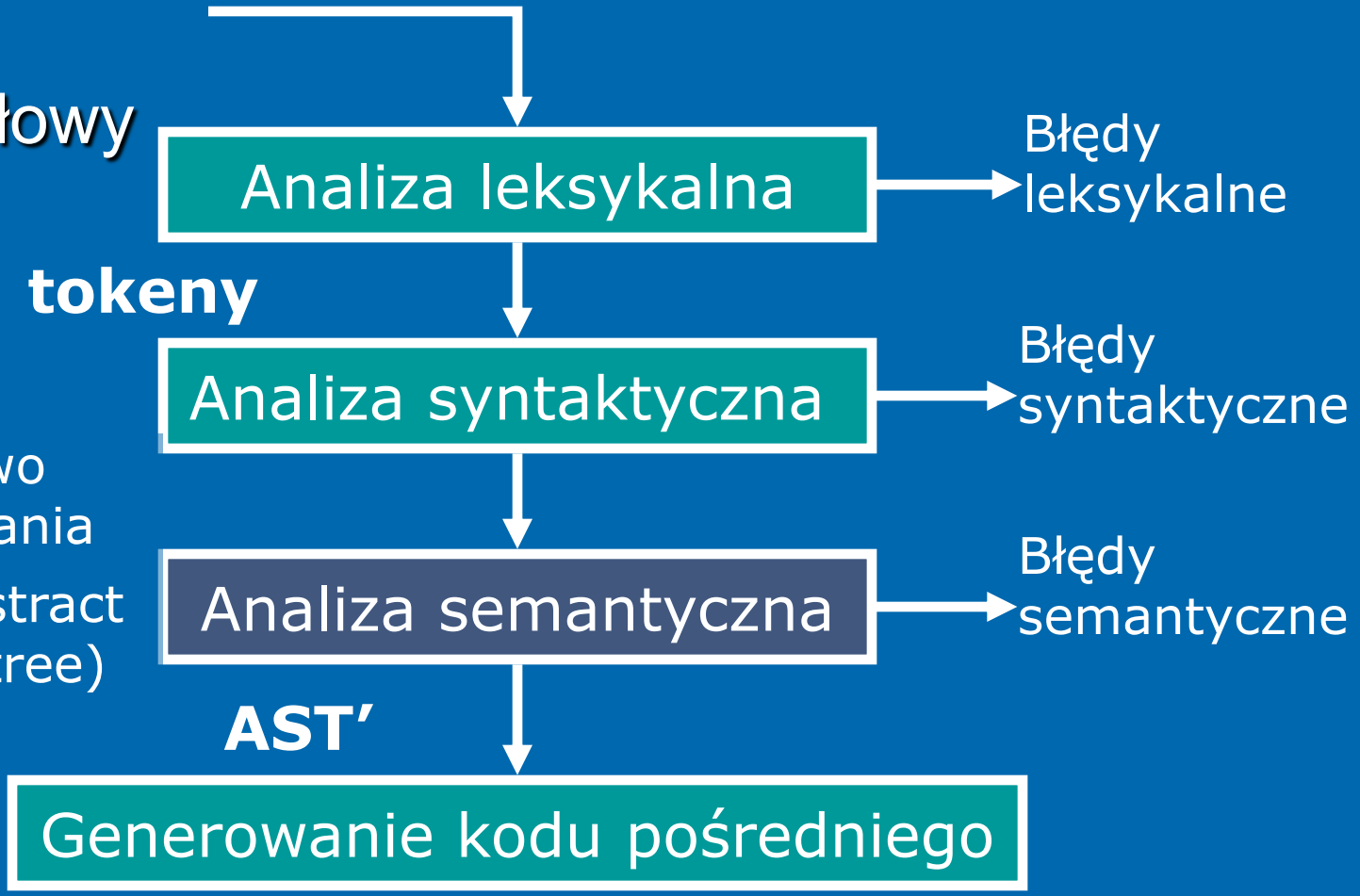
Wykład 12
Analiza Semantyczna,
2 godziny

Analiza semantyczna

- Analiza leksykalna
  - Wykrywa nielegalne tokeny
  - Na przykład: main$();
- Analiza syntaktyczna
  - Sprawdza poprawność syntaktyczną programu
    - Na przykład: brak średnika
- Analiza semantyczna
  - Ostatnia faza analizy:
  - Wykrywa wszystkie pozostałe błędy

Analiza semantyczna

- Kod źródłowy



Analiza semantyczna

Co jest nie tak z tym kodem?

(*brak błędów syntaktycznych*)

```
foo(int a, char * s){ … }

int bar() {
  int f[3];
  int i, j, k;
  char *p;
  float k;
  foo(f[6], 10, j);
  break;
  i->val = 5;
  j = i + k;
  printf("%s,%s.\n",p,q);
  goto label23;
}
```

Analiza semantyczna

Co jest nie tak z tym kodem?

(*brak błędów syntaktycznych*)

```
foo(int a, char * s){ … }

int bar() {
  int f[3];
  int i, j, k;
  char *p;
  float k;
  foo(f[6], 10, j);
  break;
  i->val = 5;
  j = i + k;
  printf("%s,%s.\n",p,q);
  goto label23;
}
```

f[6] powoduje błąd

Cele analizy semantycznej

- Kompilator musi zrobić więcej niż rozpoznać czy zdanie należy do języka.
- Znajdowanie pozostałych błędów, które sprawiają, że program jest niepoprawny
  - Niezdeklarowane zmienne, typy
  - Błędy, które mogą zostać wykryte statycznie

Tworzenie danych przydatnych dla późniejszych faz translacji

- Typy wszystkich wyrażeń
- Układ danych

Cele analizy semantycznej

- Terminologia:
- kontrola statyczna - wykonana przez kompilator

- Kontrola dynamiczna - realizowana w czasie wykonywania programu

Rodzaje kontroli

- Kontrola wyjątkowości:
  - Niektóre nazwy muszą być unikalne
  - Wiele języków wymaga deklaracji zmiennych

Przepływ kontroli sterowania

- Dopasowanie operatorów sterowania do dozwolonych struktur
- Przykład: operator "break" zastosowany na zewnątrz konstrukcji for/switch

Rodzaje kontroli

Kontrola typów

- kontrola zgodności operatorów i operandów

Kontola logiki programu

- Program jest składniowo i semantycznie poprawny, ale produkuje zły wynik

Przykłady błędów

- Niezdefiniowany identyfikator
- Wielokrotnie zadeklarowany identyfikator
- Zmienna iteracyjna pętli jest poza granicami
- Błędna liczba argumentów funkcji
- Niezgodne typy operandów operatora
- Operator „break” jest na zewnątrz instrukcji switch/for
- Brak etykiety w instrukcji "goto",

Kontrola programu

- *Zadania:*
  - Wykrywanie błędów: *f[6]* w analizowanym wcześniej przykładzie powoduje błąd
  - Zgłoszenie błędów do programisty
  - Wsparcie programisty w celu zweryfikowania jego zamiaru

Kontrola programu

- Jak ta kontrola jest pomocna?
  - Przydziela odpowiednią ilość miejsca dla zmiennych
  - Wybiera odpowiednie operatory w oparciu o operandy
  - Wybiera właściwe struktury kontrolne operatorów

Kontrola programu

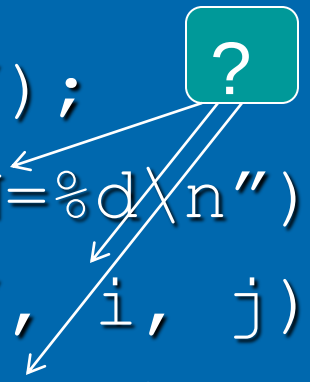
- Czy możemy wykryć wszystkie błędy?

```
void main()
{
    int i=21, j=42;
    printf("Hello World\n");
    printf("Hello World, N=%d\n");
    printf("Hello World\n", i, j);
    printf("Hello World, N=%d\n");
    printf("Hello World, N=%d\n");
}
```

Kontrola programu

- Czy możemy wykryć wszystkie błędy?

```
void main()
{
    int i=21, j=42;
    printf("Hello World\n");
    printf("Hello World, N=%d\n");
    printf("Hello World\n", i, j);
    printf("Hello World, N=%d\n");
    printf("Hello World, N=%d\n");
}
```



Kontrola programu

Możliwe jest zidentyfikowanie błędów w kodzie z poprzedniego slajdu, jeżeli jest określona semantyka analizowanej funkcji.

Kontrola typów i generowanie kodu

- Można kontrolować typy i wygenerować kod w ramach akcji semantycznych:

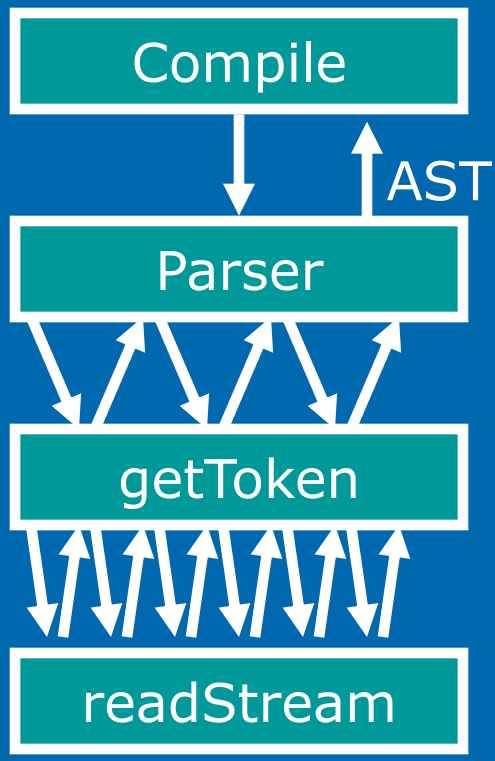
```
expr : expr PLUS expr {
        if ($1.type == $3.type &&
          ($1.type == IntType ||
            $1.type == RealType)) $$.type = $1.type
        else error("+ applied on wrong type!");
        GenerateAdd($1, $3, $$);
      }
```

Problemy

- Akcje mogą być trudne do odczytania
- Kompilator musi przeanalizować cały program celem znalezienia błędów

Alternatywne podejście

```
void Compile() {
   AST tree = Parser(program);
   if (TypeCheck(tree))
      IR ir =
        GenIntermedCode(tree);
      EmitCode(ir);
   }
}
```



Typowe błędy semantyczne

- Wielokrotne deklaracje: zmienna musi być zadeklarowana (w tym samym zakresie) co najwyżej raz
- Zmienna niezdeklarowana: zmienna nie może być stosowana bez jej deklaracji
- Niezgodność typów: typ zmiennej po lewej stronie instrukcji przypisania musi pasować do typu wyrażenia prawej strony
- Błędne argumenty: funkcja musi być wywołana z poprawnymi liczbą i typami argumentów

Prosty analizator semantyczny

- Pracuje w dwóch fazach – przechodzi przez drzewo parsowania utworzone przez parser

Dla każdego zakresu w programie

Przetwarza deklaracje i instrukcje:

- dodaje nowe wpisy do tablicy symboli i
  zgłasza zmienne, które są wielokrotnie deklarowane
- znajduje niezdeklarowane zmienne
- aktualizuje węzły, odpowiadające nazwom w programie, dodając ich wejścia w tablicy symboli.

Przetwarza deklaracje i instrukcje jeszcze raz

wykorzystując informacje w tablicy symboli, określa typ każdego wyrażenia oraz wyszukuje błędy typów.

Zakres zmiennych

- W większości języków, ta sama nazwa może być użyta wielokrotnie jeśli jej deklaracje występują w różnych zakresach.

Java: można używać tej samej nazwy dla:

- klasy
- pola klasy
- metody klasy
- lokalnej zmiennej metody

```
class Test {
     int Test;
     void Test( ) { double Test; }
}
```

Przeciążenie

- Java i C++:
  - można użyć tej samej nazwy dla więcej niż jednej metody pod warunkiem, że liczba i / lub typy parametrów są unikalne

```
int add(int a, int b);
float add(float a, float b);
```

Zakresy: Zasady ogólne

- Zasady określenia zakresu języka:
  - Określ, które deklaracje obiektu, mającego nazwę, odpowiadają każdemu użyciu obiektu
  - Zasady określania zakresu mapują użycie obiektów do ich deklaracji
- C++ i Java używają zakresów statycznych (*static scoping*):
  - Mapowanie odbywa sie w czasie kompilacji
- C++ wykorzystuje regułę "najściślej zagnieżdżone„
  - zastosowanie zmiennej x odpowiada deklaracji w zakresie najściślej otaczającego bloku
  - deklaracja zmiennej poprzedza jej użycie

Scope levels

Każda funkcja ma dwa lub więcej zakresów:

- Jeden dla ciała funkcji
  - Czasami parametry mają osobne zakresy!
  - (nie w języku C)

```
void f( int k ) { // k is a parameter
  int k = 0;      // also a local variable
  while (k) {
    int k = 1;  // another local var, in a
    loop
  }
}
```

- Dodatkowe zakresy funkcji:
  - dla pętli
  - dla bloku zagnieżdżonego

Punkt kontrolny #1

- **dopasuj każde użycie do jej deklaracji lub znajdź użycie dla którego brakuje deklaracji**
-
```
int k=10, x=20;
void foo(int k) {
  int a = x; int x = k; int b = x;
  while (...) {
   int x;
   if (x == k) {
      int k, y;
      k = y = x;
   }
   if (x == k) { int x = y; }
  }
}
```

Zakresy dynamiczne

- Nie wszystkie języki używają zakresu statycznego
- Lisp, APL, Snobol używają zakresu dynamicznego (*dynamic* scoping)

Zakresy dynamiczne

- Zakres dynamiczny oznacza:

  Użycie zmiennej, która nie ma jednej stałej deklaracji, jej deklaracja odpowiada deklaracji w ostatnim (względem czasu) wywołaniu wciąż aktywnej funkcji.

Przykład

- na przykład, rozważmy kod poniżej:

```
int i = 1;
void func() {
    cout << i << endl;
}
int main () {
    int i = 2;
    func();
    return 0;
}
```

Jeśli C++ zastosował zakres dynamiczny, to zostanie wyświetlona wartość 2, nie 1

Punkt kontrolny #2

- Zakładając, że dynamiczne zakresy są dozwolone, określ co jest wyświetlane według następującego programu?

```
void main() { int x = 0; f1(); g(); f2(); }

void f1() { int x = 10; g(); }

void f2() { int x = 20; f1(); g(); }

void g() { print(x); }
```

Śledzenie

- Potrzebny jest sposób, aby śledzić wszystkie typy identyfikatorów w każdym zakresie

```
{
    int i, n = …;
    for (i=0; i < n; i++)
        boolean b= …
}
```

i → int
n → int

i → int
n → int
b → boolean

?

Tablice symboliczne

- Cel:
  - Śledzenie nazw zdeklarowanych w programie przez wpisy w tablicy symboli:
    - **typ nazwy**(variable, class, field, method, …)
    - **typ** (int, float, …)
    - **poziom zagnieżdżenia**
    - adres w pamięci

Tablice symboliczne

- Funkcje:
- Typ wyszukiwania(np. łańcuch , *string*)
- Void Add(String id, Type binding)
- Wiązania: pary nazwa-typ {a $\rightarrow$ string, b $\rightarrow$ int}

Środowisko

Reprezentuje zbiór odwzorowań w tablicy symboli

- function f(a:int, b:int, c:int) =
- (print_int(a+c);
- let var j := a+b
- var a := “hello”
- in print(a); print_int(j)
- end;
- print_int(b)
-)

$\sigma 0$

$\sigma 1 = \sigma 0 + a \rightarrow int$

$\sigma 2 = \sigma 1 + j \rightarrow int$

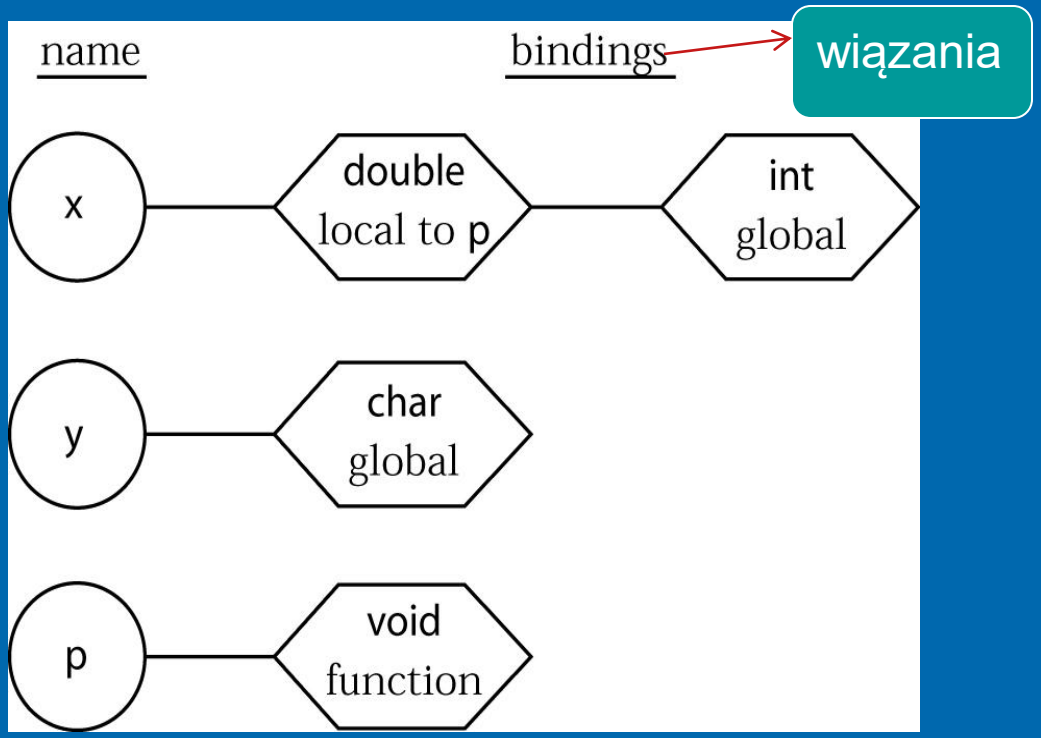
Zastosowanie tablic symboli (1)

```
int x;
char y;

void p(void)
{ double x;
    …
  { int y[10];
    …
  }
    …
}

void q(void)
{ int y;
    …
}

main()
{ char x;
  …
}
```



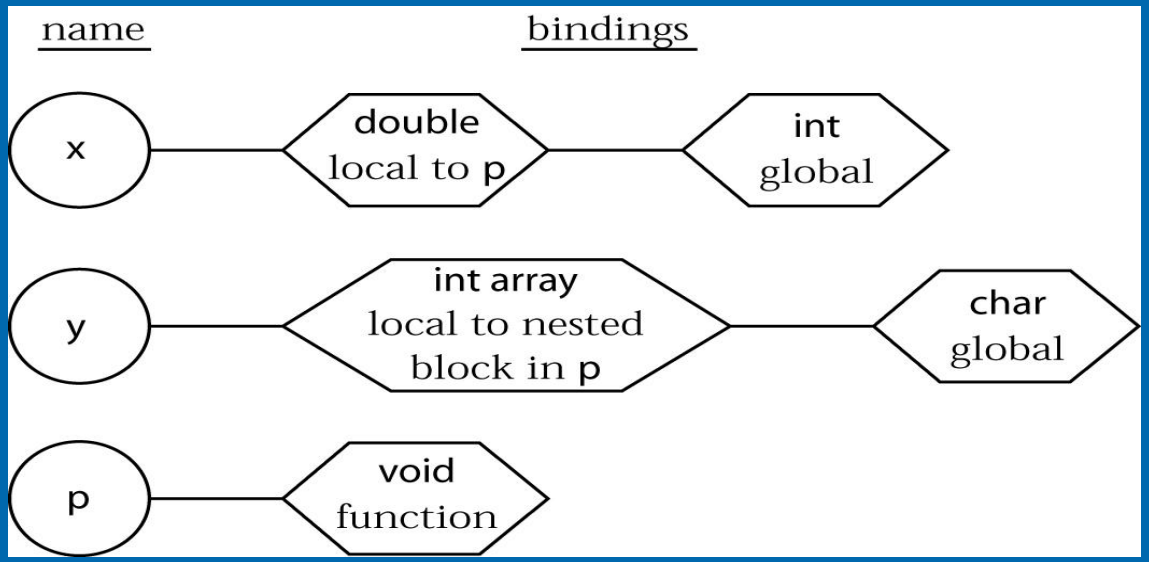
Zastosowanie tablic symboli (2)

```
int x;
char y;

void p(void)
{ double x;
    …
  { int y[10];
    …
  }
   …
}

void q(void)
{ int y;
   …
}

main()
{ char x;
  …
}
```



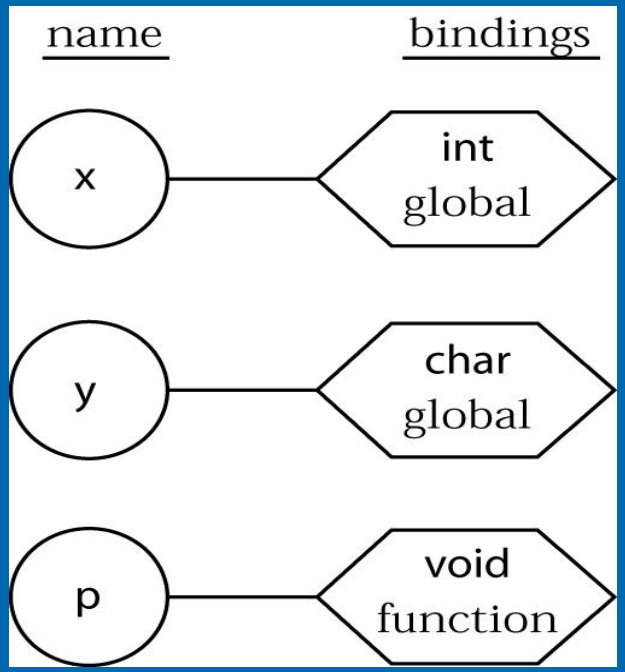
Zastosowanie tablic symboli(3)

```
int x;
char y;

void p(void)
{ double x;
    …
  { int y[10];
    …
  }
   …
}

void q(void)
{ int y;
   …
}

main()
{ char x;
  …
}
```



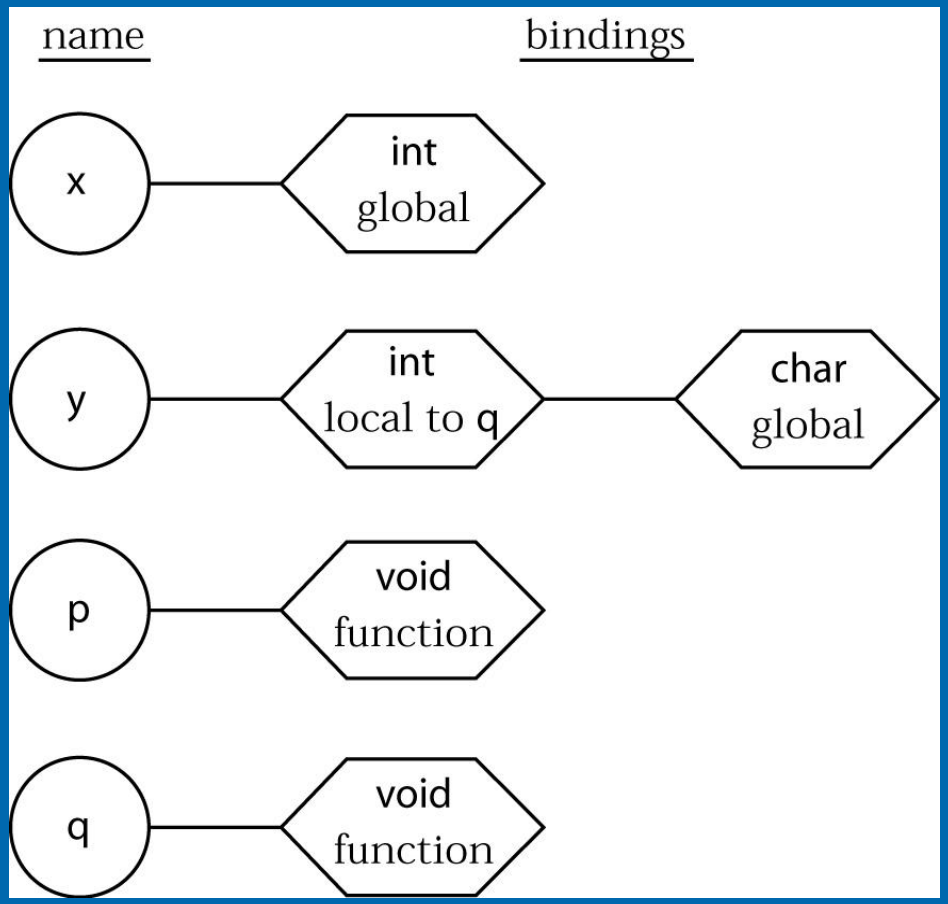
Zastosowanie tablic symboli(4)

```
int x;
char y;

void p(void)
{ double x;
    …
  { int y[10];
    …
  }
    …
}

void q(void)
{ int y;
    …
}

main()
{ char x;
  …
}
```



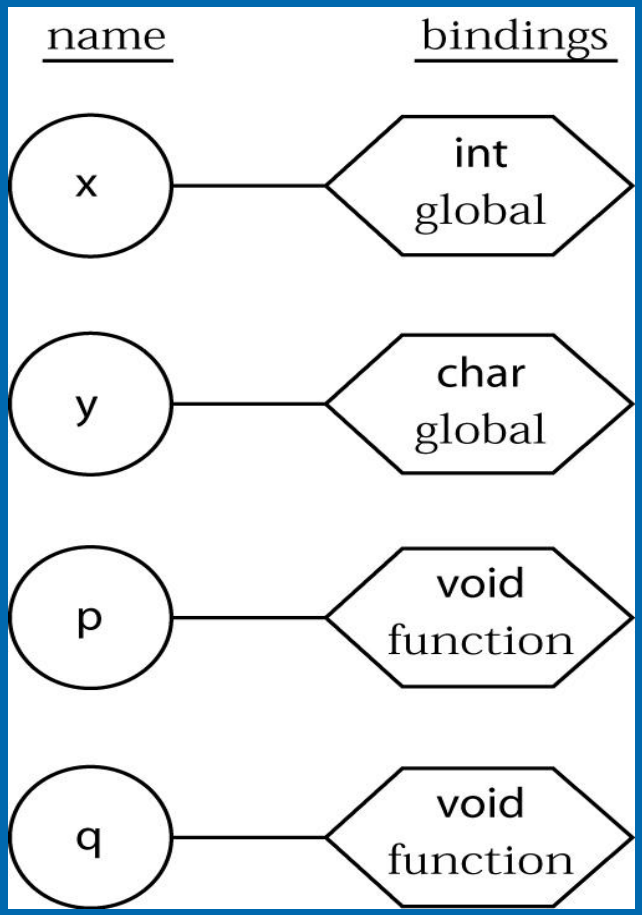
Zastosowanie tablic symboli(5)

```
int x;
char y;

void p(void)
{ double x;
    …
  { int y[10];
    …
  }
   …
}

void q(void)
{ int y;
   …
}

main()
{ char x;
  …
}
```



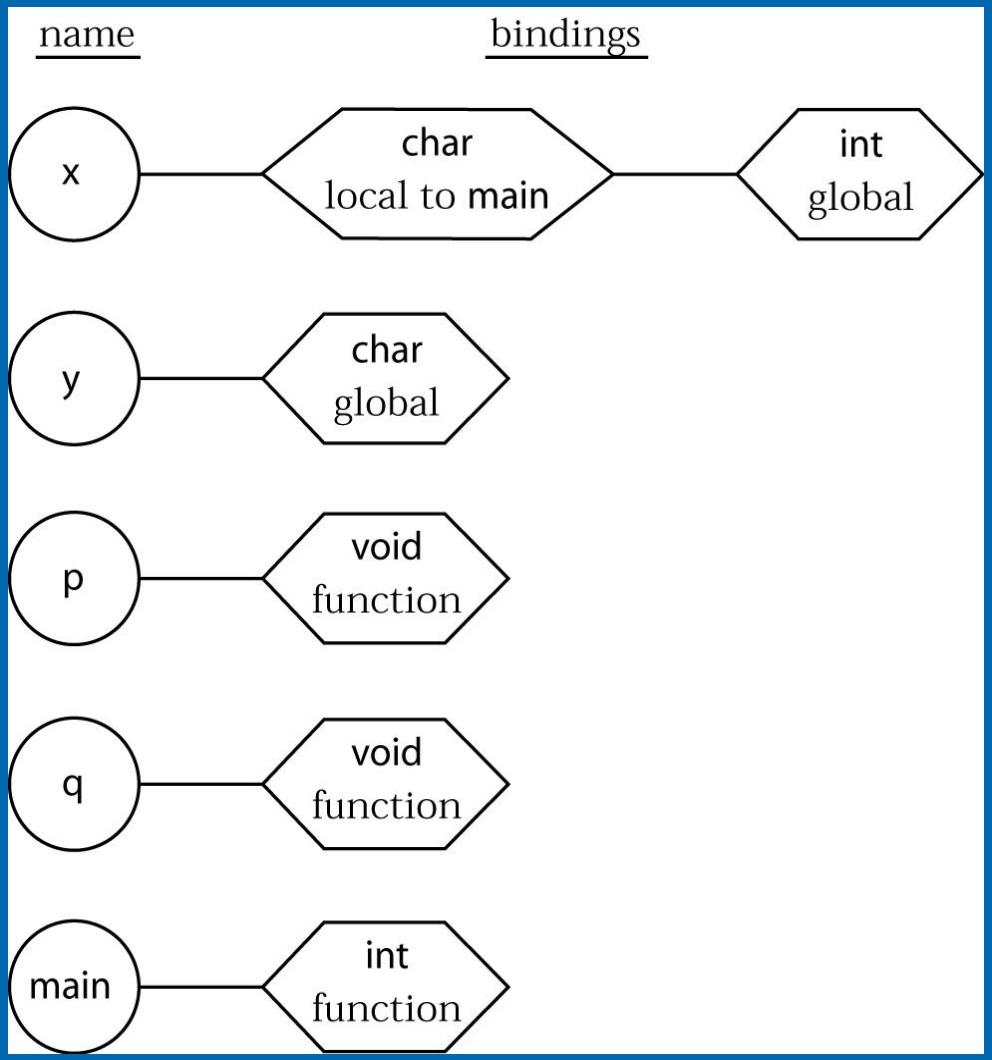
Zastosowanie tablic symboli(6)

```
int x;
char y;

void p(void)
{ double x;
    …
  { int y[10];
    …
  }
   …
}

void q(void)
{ int y;
   …
}

main()
{ char x;
  …
}
```

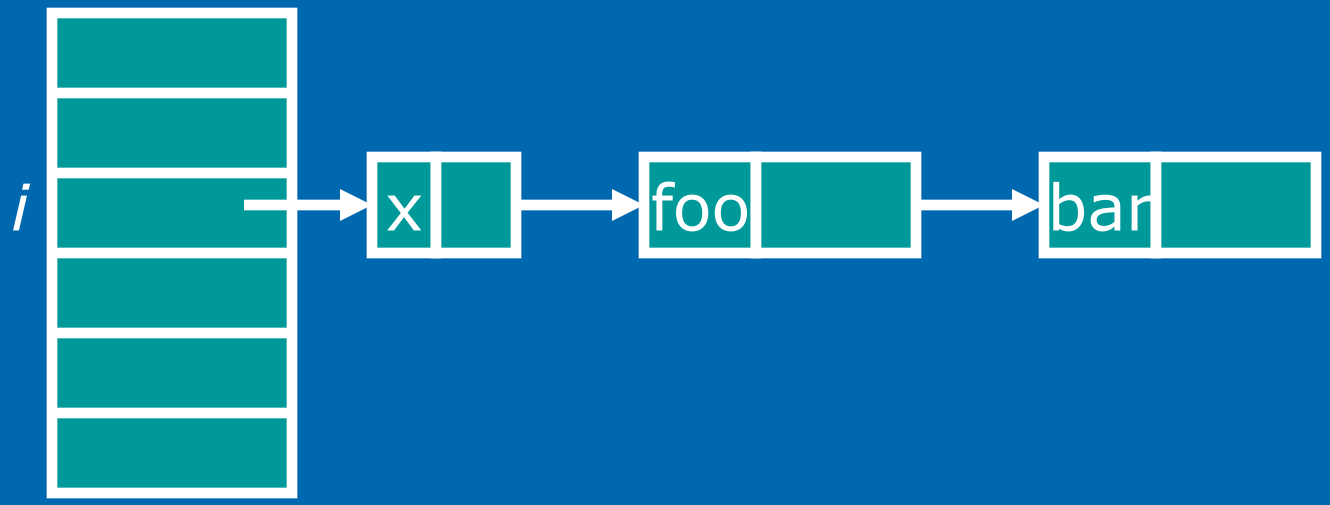


Implementacja tablicy symboli

Dwie struktury: tablica haszująca, stos zakresów

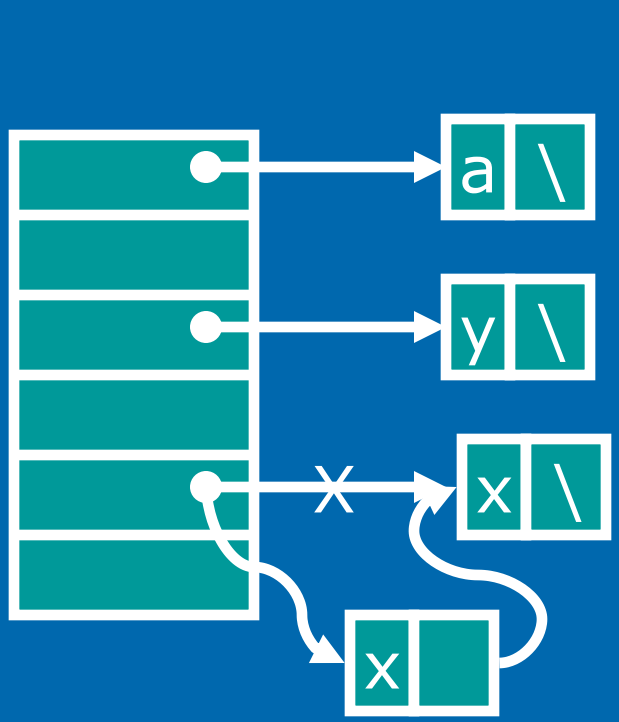
- Symbol = foo
- Hash(foo) = i

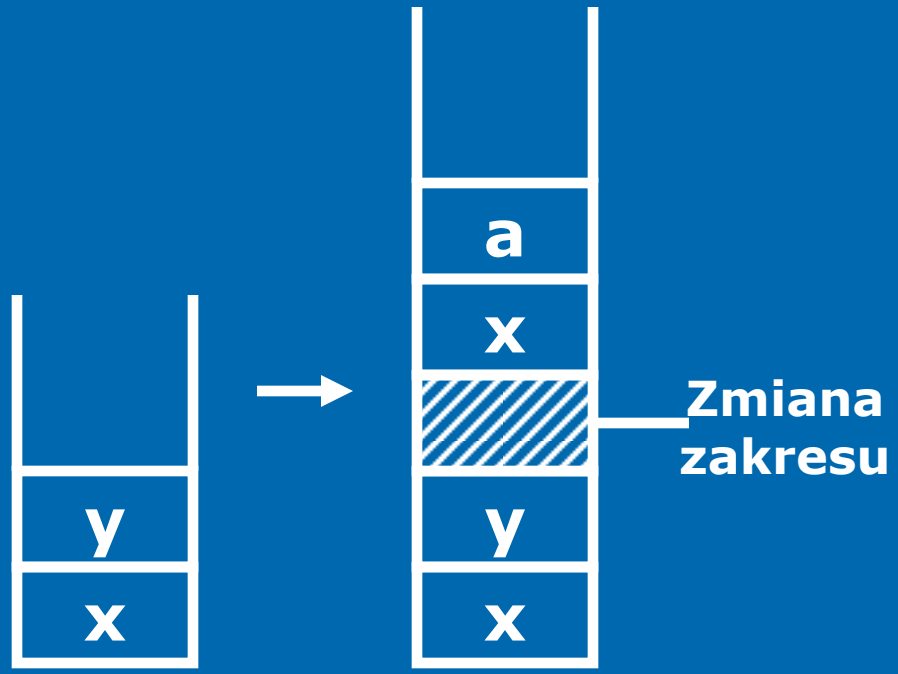
Tablica symboli



Zakres wejścia/wyjścia

- Potrzebujemy również stosu do śledzenia "poziomu zagnieżdżenia", w czasie przechodzenia przez drzewo





Zmienne i typy

- Często kompilatory tworzą osobne tablicy symboli dla typów Zmiennych / Funkcji.

Typy

- Czym jest typ?
  - To pojęcie różni się w różnych językach
- Konsensus:
  - Zbiór wartości
  - Zbiór operatorów dozwolonych na tych wartościach

Niektóre operatory są legalne dla każdego typu.

Nie jest sensownym dodawanie wskaźnika na funkcję i liczby całkowitej w C.

Sensownym jest aby dodać dwie liczby całkowite.
Ale obydwa dodawania mają taką samą realizację w asemblerze!

Systemy typów

- System typów języka określa, które operacje są ważne w zależności od typów operandów.
- Celem kontroli typu jest zapewnienie, że operacje są stosowane na dozwolonych typach operandów.
- Systemy typów określają zwięzłą formalizację reguł semantycznych sprawdzających poprawność typów dla operatorów.

Systemy typów

- Rozważmy instrukcję w asemblerze:
- addi $r1, $r2, $r3
- Jakie są typy dla $r1, $r2, $r3?

Sposoby kontroli typów

- Cztery rodzaje typów:
  - **Statyczne**: Wszystkie lub prawie wszystkie sprawdzania typów odbywają się w czasie kompilacji
  - **Dynamiczne** : Prawie wszystkie sprawdzania typów odbywają się w czasie wykonywania programu np. Perl, Ruby
  - **Model mieszany** : Java
  - **Brak typu** : brak kontroli typów(kod maszynowy)

Kontrola typów i wnioskowanie o typie

Kontrola typów jest to proces sprawdzania typów:

- Biorąc pod uwagę operator i operandy jakiegoś typu, określa czy ten operator jest dozwolony

Wnioskowanie o typie jest to proces wnioskowania jaki to jest typ

- Biorąc pod uwagę typy operandów, ustala
- Znaczenie operatora
  - Typ operatora
- Lub bez deklaracji zmiennej, określa jej typ na podstawie wywnioskowania w jaki sposób zmienna jest używana

Zasady wnioskowania

- Czy język ma system typów?
  - Języki mogą nie mieć systemu typów(np. Asembler)
- Kiedy są sprawdzane typy?
  - Typy statyczne : w czasie kompilacji
  - Typy dynamiczne: w czasie wykonywania programu
- Jak ściśle egzekwowane są reguły kontroli?
  - Typy silne: brak wyjątków
  - Typy słabe: ze ściśle określonymi wyjątkami

Zasady wnioskowania

Równoważność typów

- Kiedy dwa typy są równoważne?
  - Co oznacza równoważność?
- Kiedy można zastąpić jeden typ innym?

Składowe systemu typów

- Typy wbudowane
- Reguły do tworzenia nowych typów
  - Gdzie informacja o typach będzie się przechowywała
- Reguły do określenia typów równoważnych
- Zasady wnioskowania o typie wyrażeń

Typy wbudowane

- Całkowity
  - zwykłe operatory : arytmetyka standardowa
- Zmiennoprzecinkowy
  - zwykłe operatory : arytmetyka standardowa
- Znakowy
  - zbiór znaków zwykle jest uporządkowany w sposób leksykograficzny
  - Zwykłe operatory: porównanie leksykograficzne
- Boolowski
  - Zwykłe operatory: not, and, or, xor

Konstruktory typów

Tablice

- tablica(I,T) oznacza typ tablicy z elementami typu T i zbiorem indeksów I
- tablice wielowymiarowe są tablicami gdzie T jest również tablicą
- operacje: dostęp do elementu, przypisanie wartości elementom tablicy

Konstruktory typów

Łańcuchy

- Łańcuchy bitów, łańcuchy znaków
- operacje: łączenie, porównanie leksykograficzne

- Rekordy (struktury)
  - Grupy wielu obiektów różnych typów, w których elementy mają konkretne nazwy

Konstruktory typów

- Wskaźniki
  - adresy
  - operacje: arytmetyczne, dereferencji, referencji
- Typy funkcji
  - Funkcja "int add(real, int)" ma typ real×int→int

Równoważność typów

Równoważność typów

- Typy są równoważne tylko wtedy, gdy mają taką samą nazwę

Równoważność strukturalna

- Typy są równoważne tylko wtedy, gdy mają taką samą strukturę

Przykład

- Język C wykorzystuje równoważność strukturalną dla struktur i równoważność nazw dla tablic i wskaźników

Równoważność typów

- Wymuszenie typów
  - Jeśli x ma typ float, czy przypisanie x=3 jest dozwolone?
    - Nie jest dozwolone
    - Dozwolone i niejawnie 3 jest konwertowana na typ float
    - Dozwolone ale wymaga jawnego konwertowania przez programistę wartości 3 na typ float

Równoważność typów

- Wymuszenie typów
  - Jakie konwertowanie jest dozwolone?
    - float na int ?
    - int na float ?
    - Czy wielokrotne konwertowanie jest dozwolone?
      - Consider 3 + "4" + 4.1

Podsumowanie

- Kompilator musi zrobić więcej niż rozpoznać czy zdanie należy do języka:
  - znajduje niezdeklarowane zmienne i typy
  - zwraca błędy typów, które mogą zostać wykryte statycznie
  - przechowuje informacje przydatne dla późniejszych faz kompilacji
  - określa typy wszystkich wyrażeń

Dziękuję za uwagę