Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie

Kontrola typów

Teoria kompilacji

Dr inż. Janusz Majewski
Katedra Informatyki

Analiza semantyczna

Kompilator musi sprawdzić, czy w programie zachowane są zarówno zasady syntaktyczne, jak i semantyczne. To ostatnie sprawdzanie czasem nosi nazwę kontroli statycznej (static checking). Analiza semantyczna obejmuje następujące zagadnienia:

(1) kontrolę typów (type checking) – kompilator musi przykładowo sprawdzić, czy operatory nie są zastosowane do nieodpowiednich operandów,

```
(1)     var
        ss : array [char] of char;
        rr : real;
        …
        ss:=ss+rr;        → niezgodność typów operandów
                            dla operatora '+'
        …
```

Analiza semantyczna

(2) kontrolę przebiegu sterowania (flow-of-control-checking) – kompilator musi sprawdzić, czy sterowanie w programie jest przekazywane w odpowiednie miejsce; (przykład – czy nie jest wykonywany skok spoza pętli do jej wnętrza),

```
(2)     int fun (i, j, k)
        int i, j, k;
        {
            if(j==k) return i;
            break;       → gdzie przekazać sterowanie?
            if(j<k) return j;
            break;       → gdzie przekazać sterowanie?
            if(j>k) return k;
        }
```

Analiza semantyczna

(3) kontrolę unikalności (uniqueness checking) – kompilator powinien sprawdzić, czy pewne obiekty są definiowane w programie tylko jeden raz, przykładowo etykiety w kompilowanym module (bloku) muszą być unikalne, kategorie w typach enumeratywnych muszą być unikalne,

```
(3)     type
        colors = (violet, indigo, magenta,
        cyan)
        kolory = (yellow, red, gray, violet,
        orange)
```

`violet` → **elementy typu wyliczeniowego nie mogą się powtarzać**

Analiza semantyczna

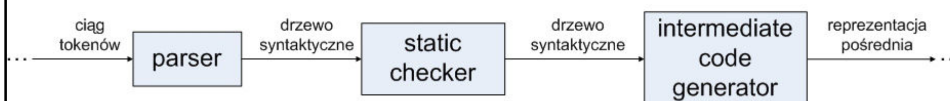
(4) kontrolę powtarzalności nazw (name-related checking) – kompilator powinien sprawdzić, czy w pewnych konstrukcjach określona nazwa pojawia się więcej niż jeden raz.

```
(4)     dane SEGMENT WORD PUBLIC
        zmienna DW 13
        komunikat DB 'Gdzie jest błąd,
  kochanie?'
        adres DD komunikat
        extra ENDS
```

dane, extra → **nazwa musi się powtórzyć**

Analiza semantyczna

Kontrola statyczna, a w szczególności także kontrola typów może być przeprowadzana (przynajmniej częściowo) równolegle z parsingiem i//lub generacją kodu pośredniego albo niezależnie, jak na schemacie poniżej. Wykorzystywany jest przy tym mechanizm gramatyk atrybutywnych (definicji kierowanych składnią).



System typów (przykładowy)

Typ konstrukcji językowej określony jest przez „wyrażenie typu”. Wyrażenie typu jest albo typem podstawowym, albo jest tworzone z wyrażenia typu poprzez zastosowanie „konstruktora typu”.

Wyrażenie typu może być zdefiniowane w następujący sposób:

(1) Typ podstawowy jest wyrażeniem typu. Poza typami podstawowymi jak *boolean*, *cha*r, *integer*, *real* wprowadza się specjalne typy podstawowe: *void* dla konstrukcji nie wymagających sprecyzowania typu oraz *type_error* dla konstrukcji błędnych z punktu widzenia kontrolera typów.

(2) Jeżeli typy mogą mieć własne nazwy – nazwa typu jest (a właściwie może być) wyrażeniem typu.

System typów (przykładowy)

(3) Konstruktor typu zastosowany do wyrażenia typu jest wyrażeniem typu.

(a) Jeśli *T* jest wyrażeniem typu, to *array(I, T)* jest wyrażeniem typu określającym typ tablicy o elementach typu *T* i typie indeksowym *I*

Przykład: deklaracja

```
var A:array [char] of integer;
```

kojarzy wyrażenie typu *array(char, integer)* ze zmienną *„A”*.

(b) Jeżeli T_1 i T_2 są wyrażeniami typu, to ich „produkt kartezjański” $T_1 \times T_2$ jest wyrażeniem typu (operacja „×” jest lewostronnie łączna).

System typów (przykładowy)

(c) Konstruktor *record(...)* zastosowany do produktu wyrażeń typu pól rekordu jest wyrażeniem typu. Wyrażenie typu dla pola rekordu ma postać:
(nazwa_pola × *typ_pola*)

Przykład:

```
type row = record
     address : integer;
     lexeme : array[byte] of char
          end;
var table : array[1..100] of row;
```

Ze zmienną *"table"* skojarzone jest wyrażenie typu:
*array(1..100, record((*address × *integer)* × *(*lexeme × *array(byte, char))))*

System typów (przykładowy)

(d) Jeśli *T* jest wyrażeniem typu, to *pointer(T)* jest wyrażeniem typu określającym wskaźnik do obiektu typu *T*.

Przykład: deklaracja

```
var p : ↑integer;
```

powoduje skojarzenie wyrażenia typu: *pointer(integer)* ze zmienną *„p"*.

System typów (przykładowy)

(e) Jeśli D jest wyrażeniem typu odpowiadającym liście argumentów funkcji, a R wyrażeniem typu określającym typ wyniku funkcji, to zapis $D \rightarrow R$ jest wyrażeniem typu skojarzonym z tą funkcją (operacja „$\rightarrow$” jest prawostronnie łączna i ma niższy priorytet od operacji „$\times$”)

Przykład: deklaracja

```
function f(a, b : char) : ↑integer;
```

kojarzy wyrażenie typu: $char \times char \rightarrow pointer(integer)$ z funkcją f.

Niektóre języki, np. LISP dopuszczają konstrukcje o typie: $(integer \rightarrow integer) \rightarrow (integer \rightarrow integer)$

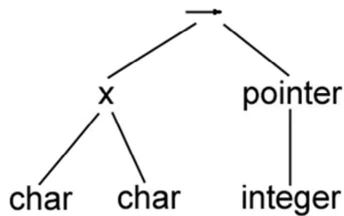
System typów (przykładowy)

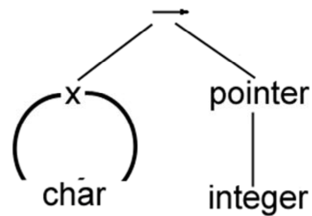
(4) Wyrażenie typu może zawierać zmienne, wartościami których są wyrażenia typu.

System typów (przykładowy)

Wyrażenia typu mogą być reprezentowane np. w postaci grafów lub drzew.

Przykład: Wyrażenie typu: *char* × *char* → *pointer(integer)*





(drzewo) (dag)

System typów

Systemem typów nazywamy zbiór reguł przypisujących wyrażenia typu do różnych konstrukcji programu kompilowanego. Kompilator powinien więc implementować system typów, aby mieć możliwość kontroli poprawności użycia typów przez programistę i poprawnie dokonać tłumaczenia.

Kontrolę typów wykonywaną przez kompilator nazywamy statyczną, natomiast sprawdzanie podczas działania programu wynikowego – sprawdzaniem dynamicznym. Właściwie każde sprawdzanie może być zrobione dynamicznie, jeżeli kod wynikowy zawiera typ elementu wraz z jego wartością.

Statyczna i dynamiczna kontrola typów

Z reguły nie wszystkie kontrole związane z typami mogą być przeprowadzone statycznie (w trakcie kompilacji).

Przykład

```
var table : array[0..255] of char;
      i : integer;
```

Poprawna w Pascalu konstrukcja:

```
table[i]
```

nie pozwala kompilatorowi zagwarantować, że w trakcie wykonywania programu wartość zmiennej indeksowej „i" będzie mieścić się w przedziale 0..255.

Statyczna i dynamiczna kontrola typów

Język programowania nazywamy „ścisle typowanym" (strongly typed) jeżeli jego kompilator może zagwarantować, że zaakceptowany przez niego program wykona się bez „błędów typów".

Specyfikacja przykładowego kontrolera typów

P → D ; S	
D → D ; D	
D → id : T	{addtype(id.entry, T.type)}
T → char	{T.type ← char}
T → integer	{T.type ← integer}
T → boolean	{T.type ← boolean}
T → ↑T_1	{T.type ← pointer(T_1.type)}
T → array [num] of T_1	{T.type ← array(1..num.val, T_1.type)}
T → function id (T_1) : T_2	{T.type ← (T_1.type → T_2.type); addtype(id.entry, T.type)}

Specyfikacja przykładowego kontrolera typów

E → literal	{E.type ← char}
E → num	{E.type ← integer}
E → true	{E.type ← boolean}
E → false	{E.type ← boolean}
E → id	{E.type ← lookup(id.entry)}
E → E_1 + E_2	{E.type ← if E_1.type = integer and E_2.type = integer then integer else type_error}
E → E_1 mod E_2	{E.type ← if E_1.type = integer and E_2.type = integer then integer else type_error}
E → E_1 [E_2]	{E.type ← if E_2.type = integer and E_1.type = array(s,t) then t else type_error}
E → E_1↑	{E.type ← if E_1.type = pointer(t) then t else type_error}
E → E_1 (E_2)	{E.type ← if E_2.type = s and E_1.type = (s → t) then t else type_error}

Specyfikacja przykładowego kontrolera typów

$S \rightarrow$ <u>id</u> := E	{$S.type \leftarrow$ <u>if</u> lookup(<u>id</u>.entry) = $E.type$ <u>then</u> void <u>else</u> type_error}
$S \rightarrow$ <u>if</u> E <u>then</u> S_1	{$S.type \leftarrow$ <u>if</u> $E.type$ = boolean <u>then</u> $S_1.type$ <u>else</u> type_error}
$S \rightarrow S_1 ; S_2$	{$S.type \leftarrow$ <u>if</u> $S_1.type$ = void <u>and</u> $S_2.type$ = void <u>then</u> void <u>else</u> type_error}

Równoważność wyrażeń typu

Reguły semantyczne w poprzednim przykładzie miały postać:
„<u>if</u> dwa wyrażenia typu są równe (równoważne)
    <u>then</u> zwróć określony typ
    <u>else</u> zwróć ‘type_error’”

Pytanie: „kiedy dwa wyrażenia typu są równoważne?”

Równoważność strukturalna

(1) Równoważność strukturalna (structural equivalence)
W przypadku, gdy wyrażenia typu są zbudowane wyłącznie z typów podstawowych i konstruktorów (tzn. nie zawierają nazw typów) naturalnym sposobem określenia równoważności typów jest równoważność strukturalna. Dwa wyrażenia typu są równoważne strukturalnie, gdy są albo tymi samymi typami podstawowymi, albo są utworzone przez zastosowanie tego samego konstruktora do równoważnych strukturalnie wyrażeń typu (jest to definicja rekurencyjna). Innymi słowy dwa wyrażenia typu są strukturalnie równoważne, wtedy i tylko wtedy, gdy są one identyczne.

Równoważność strukturalna

Przykład:
„pointer(integer)” jest równoważny tylko wyrażeniu *„pointer(integer)”* $\Rightarrow$ zastosowano ten sam konstruktor *pointer(...)* do równoważnych wyrażeń typu (tutaj do identycznych typów podstawowych - *integer*)

Równoważność strukturalna

Przykład:

```
type
   cell = array[char] of integer;
   link = ↑cell;
var     next : link;
   last : link;
   p : ↑cell;
   q, r : ↑cell;
```

Strukturalna równoważność typów nie dopuszcza nazw typów. W związku z tym:
link oznacza *pointer(array(char, integer))*
↑*cell* oznacza *pointer(array(char, integer))*
czyli w sensie równoważności strukturalnej zmienne: *next, last, p, q, r* mają ten sam typ.

Równoważność strukturalna

Procedura (funkcja rekurencyjna) do testowania równoważności strukturalnej:

<u>function</u> SEQUIV(s, t) : boolean;
<u>begin</u>
  <u>if</u> s oraz t są identycznymi typami podstawowymi <u>then</u>
    SEQUIV := true
  <u>else</u> <u>if</u> s=array(s_1, s_2) <u>and</u> t=array(t_1, t_2) <u>then</u>
    SEQUIV := SEQUIV(s_1, t_1) <u>and</u> SEQUIV(s_2, t_2)
  <u>else</u> <u>if</u> s=$s_1 \times s_2$ <u>and</u> t=$t_1 \times t_2$ <u>then</u>
    SEQUIV := SEQUIV(s_1, t_1) <u>and</u> SEQUIV(s_2, t_2)
  <u>else</u> <u>if</u> s=pointer(s_1) <u>and</u> t=pointer(t_1) <u>then</u>
    SEQUIV := SEQUIV(s_1, t_1)
  <u>else</u> <u>if</u> s=$s_1 \rightarrow s_2$ <u>and</u> t=$t_1 \rightarrow t_2$ <u>then</u>
    SEQUIV := SEQUIV(s_1, t_1) <u>and</u> SEQUIV(s_2, t_2)
  <u>else</u> SEQUIV := false
end;

Uwaga: dla uproszczenia pominięto konstruktor *record(...)*.

Równoważność strukturalna

(…)
else if s=array(s_1, s_2) and t=array(t_1, t_2) then
SEQUIV := SEQUIV(s_1, t_1) and SEQUIV(s_2, t_2) //(*)
(…)

(*) – Uwaga: W praktyce niekiedy równoważność strukturalna bywa rozumiana mniej rygorystycznie, Jeśli np. tablice są przekazywane jako parametry procedur lub funkcji, może nas nie interesować zakres zmienności indeksu tablicy, a tylko typ jej elementu. Wówczas odpowiedni zapis w procedurze SEQUIV może mieć postać:

(…)
else if s = array(s_1, s_2) and t = array(t_1, t_2) then
SEQUIV := SEQUIV(s_2, t_2)
(…)

Równoważność strukturalna

Przykład:
W kompilatorze języka C napisanym przez Ritchie'ego stosowano m.in. następujące konstruktory:
pointer(t) – wskaźnik na obiekt typu *t*
freturns(t) – funkcja pewnych argumentów zwracająca obiekt typu *t*
array(t) – tablica (długość nieistotna) elementów typu *t*

Wówczas:
array(pointer(freturns(char))) – oznacza tablicę wskaźników na funkcje zwracające obiekty typu znakowego.

Równoważność strukturalna

Przy zastosowaniu kodowania:

konstruktor	kod
pointer	01
array	10
freturns	11
…	…

typ podstawowy	Kod
boolean	0000
char	0001
integer	0010
real	0011
……	……

Równoważność strukturalna

…można wyrażenia typu oznaczać ciągiem bitów:

wyrażenie typu	kod
char	0000000001
freturns(char)	0000110001
pointer(freturns(char))	0001110001
array(pointer(freturns(char)))	1001110001

Wówczas dwie różne sekwencje bitów nie mogą reprezentować tego samego typu. Oczywiście dwie jednakowe sekwencje bitów mogą reprezentować różne typy, gdyż wielkości tablic ani argumenty funkcji w takim zapisie nie są reprezentowane.

Równoważność przy dopuszczeniu nazw typów

(2) Równoważność przy dopuszczeniu nazw typów (name equivalence)

Jeśli w pewnych językach dopuszcza się nadawanie typom nazw, to można zgodzić się aby nazwy typów pojawiały się w wyrażeniach typu. Traktuje się każdą nazwę typu jako odrębny typ. Przy takich założeniach dwa wyrażenia typu są równoważne wtedy i tylko wtedy, gdy są one identyczne.

Działanie takiego systemu typów zależy od implementacji…

Równoważność przy dopuszczeniu nazw typów

Przykład:

```
type link = ↑cell;
var  next : link;
     last : link;
     p : ↑cell;
     q, r : ↑cell;
```

Zmienna	Wyrażenie typu
next	*link*
last	*link*
p	*pointer(cell)*
q	*pointer(cell)*
r	*pointer(cell)*

Zgodnie z tak rozumianą równoważnością (przy dopuszczeniu nazw):

a) równoważne są typy zmiennych: „*next*” i „*last*”
b) równoważne są typy zmiennych: „*p*”, „*q*”, „*r*”
c) nie są równoważne typy zmiennych np.: „*next*” i „*p*”

Równoważność przy dopuszczeniu nazw typów

Przykład: Wiele implementacji Pascala zakłada niejawną nazwę typu dla deklaracji zmiennych, w których typ nie został nazwany.

Zapis:

```
type
     link = ↑cell;
var
     next : link;
     last : link;
     p : ↑cell;
     q, r : ↑cell;
```

jest interpretowany jako:

```
type
     link = ↑cell;
     ntyp_p = ↑cell;
     ntyp_q_r = ↑cell;
var
     next : link;
     last : link;
     p : ntyp_p;
     q : ntyp_q_r;
     r : ntyp_q_r;
```

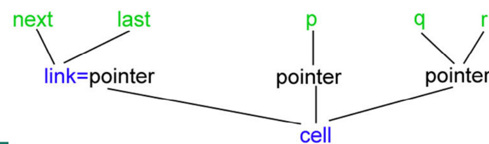
Równoważność przy dopuszczeniu nazw typów

```
type
     link = ↑cell;
     ntyp_p = ↑cell;
     ntyp_q_r = ↑cell;
var
     next : link;
     last : link;
     p : ntyp_p;
     q : ntyp_q_r;
     r : ntyp_q_r;
```

Wobec tego:

(a) równoważne typy mają zmienne:
  (-) *"next"* oraz *"last"*
  (-) *"q"* oraz *"r"*

(b) każda ze zmiennych *"next"*, *"p"* oraz *"q"* ma inny typ.



Cykle w reprezentacji typów

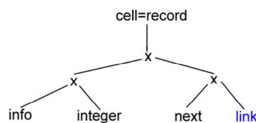
Cykle w reprezentacji typów

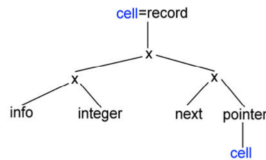
Cykle w wyrażeniach typu mogą pojawiać się w zasadzie tylko i wyłącznie w konstrukcjach: *pointer(record((...))*.

Cykle w reprezentacji typów

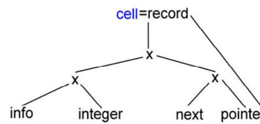
Przykład:

```
type
    link = ↑cell;
    cell = record
        info : integer;
        next : link;
    end;
```





Reprezentacja acykliczna



Reprezentacja cykliczna

Cykle w reprezentacji typów

Przykład:

Analogiczna definicja w języku C ma postać:

```
struct cell {
    int info;
    struct cell *next;
};
```

W języku C nazwa rekordu (nazywanego tutaj strukturą) stanowi część nazwy typu. Poza tym język C wymaga aby nazwa była zadeklarowana przed jej użyciem.

W związku z tym język C wykorzystuje reprezentację acykliczną konstrukcji rodzaju *pointer(record(...))*. Wszystkie potencjalne cykle sprowadzają się do konstrukcji *pointer(record(...))*. Ponieważ nazwa rekordu jest częścią jego typu, sprawdzanie równoważności jest kończone gdy po konstruktorze *pointer(...)* spotykany jest konstruktor *record(nazwa_rekordu, ...)*. Wówczas typy są równoważne, gdy rekordy porównywane mają takie same nazwy, w przeciwnym przypadku sygnalizowana jest nierównoważność typów.

Niejawne konwersje typów

Przykład:

```
var  x : real;
     i : integer;
......
x := x+i;
......
```

x+i – tłumaczenie to do np. odwrotnej notacji polskiej powinno być następujące:

x *i* *inttoreal* *real+*

Niejawne konwersje typów

Przykład: ustalanie typu wyrażenia

$E \rightarrow$ *num*	{*E.type* ← *integer*}
$E \rightarrow$ *num* . *num*	{*E.type* ← *real*}
$E \rightarrow$ *id*	{*E.type* ← *lookup*(*id.entry*)}
$E \rightarrow E_1$ *op* E_2	{*E.type* ← *if* E_1.*type* = *integer and* E_2.*type* = *integer then integer* *else if* E_1.*type* = *integer and* E_2.*type* = *real then real* *else if* E_1.*type* = *real and* E_2.*type* = *integer then real* *else if* E_1.*type* = *real and* E_2.*type* = *real then real* *else type_error*}

O ile jest to możliwe konwersje (niejawne) stałych powinny być dokonywane na etapie kompilacji, a nie wykonania.

Przeciążanie funkcji lub operatorów

Operatory lub funkcje mogą mieć „różne znaczenia” w zależności od kontekstu. Nazywamy je wtedy przeciążonymi.

Przykład: (Turbo Pascal)

```
var
     x, y : real;
     i, j : integer;
     s, t : string;
```

$x + y \rightarrow$ dodawanie zmiennopozycyjne
$i + j \rightarrow$ dodawanie stałopozycyjne
$s + t \rightarrow$ konkatenacja łańcuchów znaków

Operator ‘+’ nazywamy operatorem przeciążonym.

Przeciążanie funkcji lub operatorów

Przy założeniu, że podwyrażenia mają unikalny typ, kontrolę typu funkcji przeprowadzano w następujący sposób:

$E \rightarrow E_1\ (E_2)$	{*E.type* ← *if* E_2*.type* = *s* *and* E_1*.type* = (*s* → *t*) *then* *t* *else* *type_error*}

Jeżeli dopuścimy aby funkcje mogły być przeciążone, wówczas określenie typu podwyrażenia będącego wywołaniem funkcji będzie np. następujące:

$E \rightarrow E_1\ (E_2)$	$\{E.types \leftarrow \{t : \exists s \in E_2.types : (s \rightarrow t) \in E_1.types\}$

W wielu językach dopuszczających przeciążalność wymaga się, aby wyrażenie miało unikalny (pojedynczy) typ, podczas gdy podwyrażenia składowe tego warunku spełniać nie muszą.

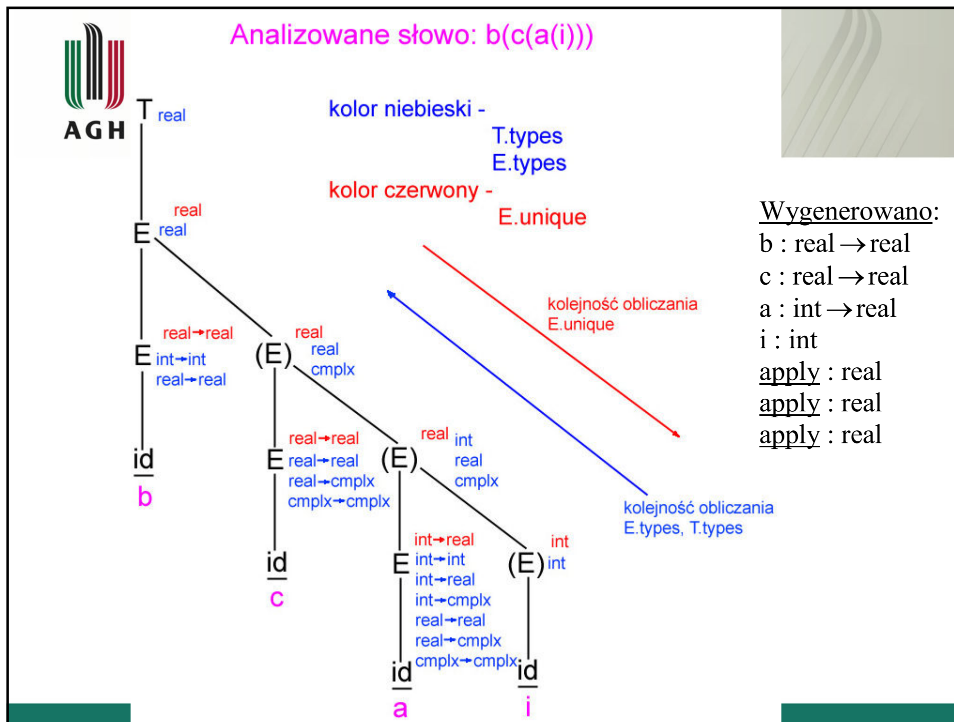
Przeciążanie funkcji lub operatorów

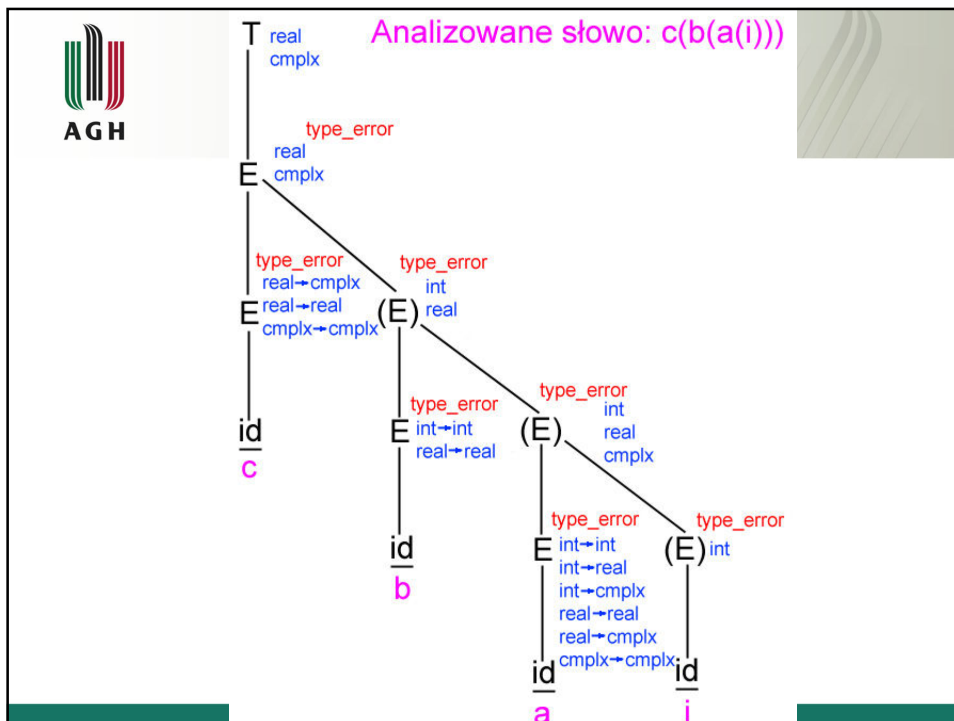
Przykład kontroli unikalności typu wyrażenia i generacji ONP

$T \to E$	$T.types \leftarrow E.types$ $E.unique \leftarrow \underline{if}\ T.types = \{t\}\ \underline{then}\ t\ \underline{else}\ type_error$ $T.code \leftarrow E.code$
$E \to \underline{id}$	$E.types \leftarrow lookup(\underline{id}.entry)$ $E.code \leftarrow gen(\underline{id}.lexeme\ ':'\ E.unique)$
$E \to E_1(E_2)$	$E.types \leftarrow \{s' : \exists s \in E_2.types : (s \to s') \in E_1.types\}$ $t := E.unique$ $S := \{s : s \in E_2.types\ \underline{and}\ (s \to t) \in E_1.types\}$ $E_2.unique \leftarrow \underline{if}\ S = \{s\}\ \underline{then}\ s\ \underline{else}\ type_error$ $E_1.unique \leftarrow \underline{if}\ S = \{s\}\ \underline{then}\ (s \to t)\ \underline{else}\ type_error$ $E.code \leftarrow E_1.code \parallel E_2.code \parallel gen('\underline{apply} :' E.unique)$

Przeciążanie funkcji lub operatorów

Identyfikator	Typ
i	int
a	$int \to int$ $real \to real$ $cmplx \to cmplx$ $int \to real$ $real \to cmplx$ $int \to cmplx$
b	$int \to int$ $real \to real$
c	$real \to cmlx$ $cmplx \to cmlx$ $real \to real$


Analizowane słowo: b(c(a(i)))
kolor niebieski -
T.types
E.types
kolor czerwony -
E.unique
kolejność obliczania
E.unique
kolejność obliczania
E.types, T.types
Wygenerowano:
b : real → real
c : real → real
a : int → real
i : int
apply : real
apply : real
apply : real
T real
E real real
E real→real int→int real→real
(E) real real cmplx
id b
E real→real real→real real→cmplx cmplx→cmplx
(E) real int real cmplx
id c
E int→real int→int int→real int→cmplx real→real real→cmplx cmplx→cmplx
(E) int int
id a
id i


Analizowane słowo: c(b(a(i)))
T real cmplx
E type_error real cmplx
E type_error real→cmplx real→real cmplx→cmplx
(E) type_error int real
id c
E type_error int→int real→real
(E) type_error int real cmplx
id b
E type_error int→int int→real int→cmplx real→real real→cmplx cmplx→cmplx
(E) type_error int
id a
id i