11. Wzmianka o metodach definiowania semantyki

– przypomnijmy ***definicję matematyczną*** języka formalnego:

L = <Σ, **Syn**, **DSem**, **Sem**>

Σ – ***alfabet***, skończony niepusty zbiór symboli (znaków)

Syn – ***syntaktyka*** (składnia, syntaksa), zbiór napisów języka zbudowanych z symboli alfabetu

DSem – ***dziedzina znaczeń***, zbiór bytów semantycznych, przypisywanych napisom języka

Sem – ***relacja semantyczna*** (**Sem** $\subseteq$ **Syn** $\times$ **DSem**), definiująca związki między napisami języka a elementami dziedziny znaczeń (relacja ta w zastosowaniach praktycznych jest ***funkcją semantyki***)

– rozważamy ***języki programowania***

– co stanowi tym wypadku dziedzinę semantyczną **DSem**?

a) reprezentacja ***środowiska obliczeniowego***

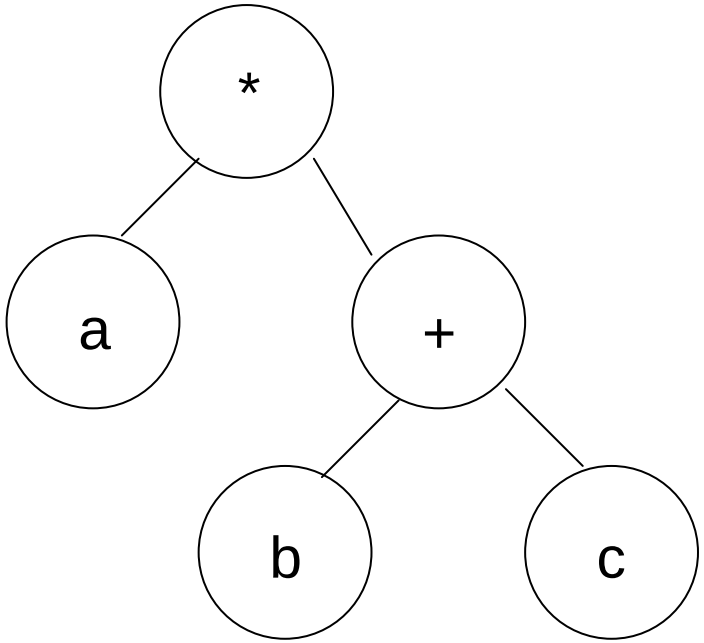
b) zdania logiczne opisujące ***stwierdzenia o programach***.

– podstawowe podejścia: ***operacyjne***, ***denotacyjne*** (funkcyjne) i ***aksjomatyczne***

– ***metoda operacyjna:***

a) napisy **Syn** języka programowania są reprezentowane w postaci tzw. „składni abstrakcyjnej” (drzewo rozbioru składniowego), np.:

`a*(b+c)`



b) dziedzina **DSem** jest w tym wypadku uniwersum ***operacji***, jakie wykonuje pewna abstrakcyjna maszyna podczas interpretacji napisów **Syn** języka; stąd, metoda operacyjna nazywana jest także „interpretacją abstrakcyjną”

c) funkcja **Sem** przypisuje każdemu napisowi należącemu do **Syn** ciąg operacji z dziedziny **DSem**

d) przykładem jest metoda wiedeńska (ang. *Vienna Definition Language*)

11.1 Metoda wiedeńska VDL

– programy, reprezentowane za pomocą składni abstrakcyjnej są interpretowane przez niedeterministyczną maszynę W

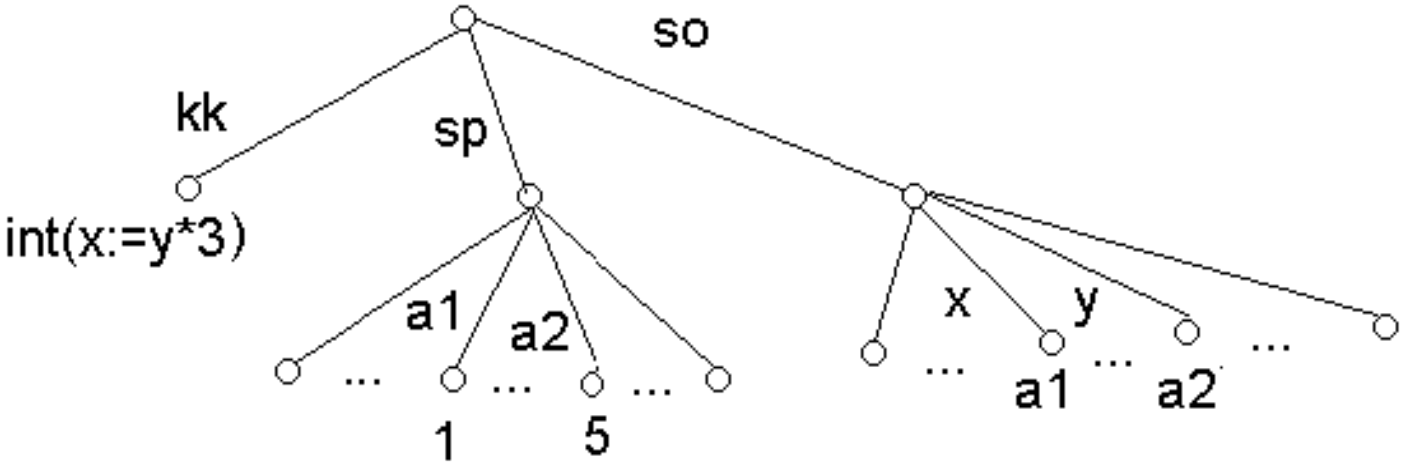
– każdy ***stan maszyny*** W jest zbiorem obiektów abstrakcyjnych trojakiego rodzaju:

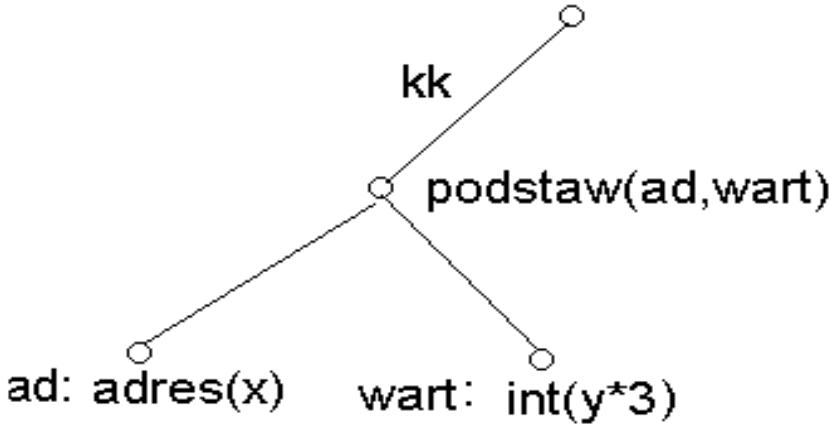
a) ***składowa kontrolna kk*** (reprezentująca wyrażenie **Syn** podlegające interpretacji)

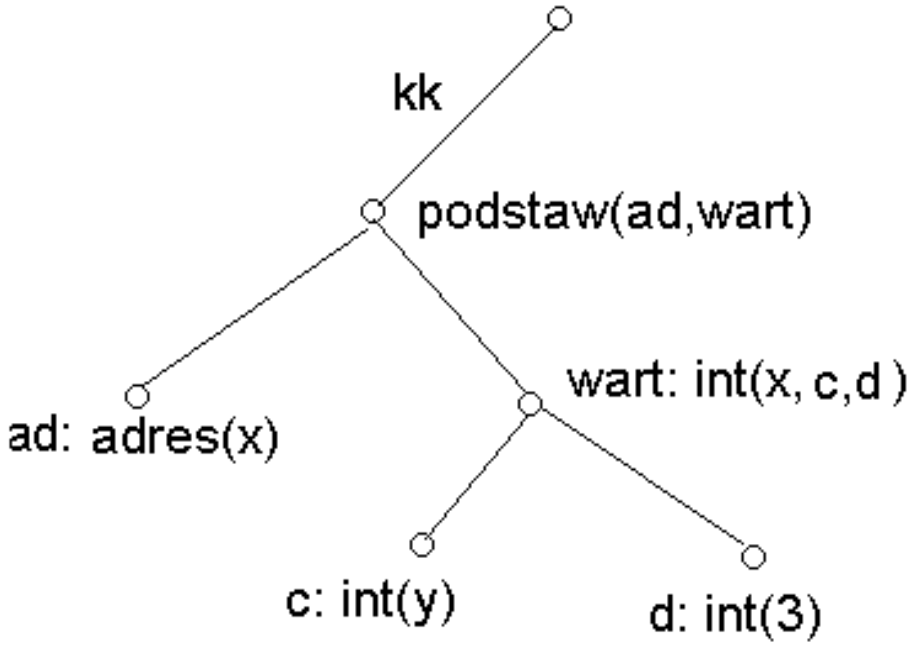
b) ***składowa stanu pamięci sp*** (powiązania adresów komórek pamięci z zawartymi w nich wartościami) i

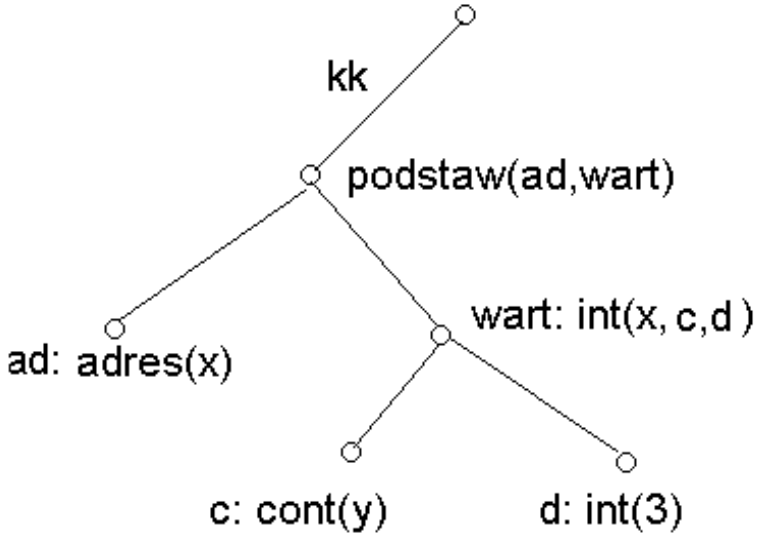
c) ***składowa otoczenia so*** (słownik wiążący identyfikatory z ich znaczeniami, np. nazwy zmiennych z adresami komórek pamięci).

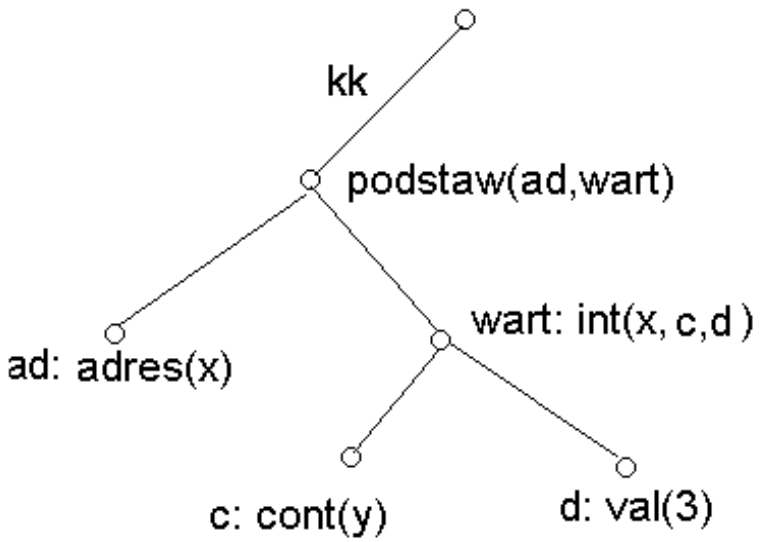
– **<u>przykład</u>** (znaczenie wyrażenia x := y * 3 w maszynie W)

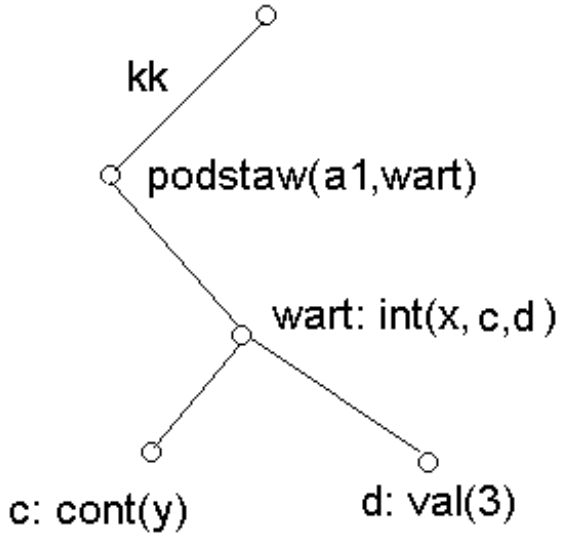


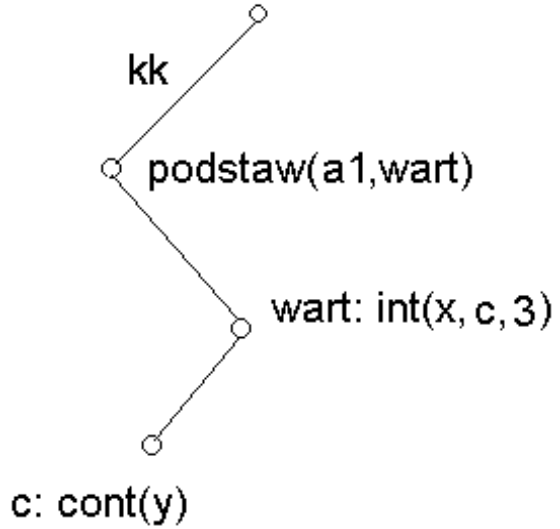

kk
podstaw(ad,wart)
ad: adres(x)
wart: int(y*3)

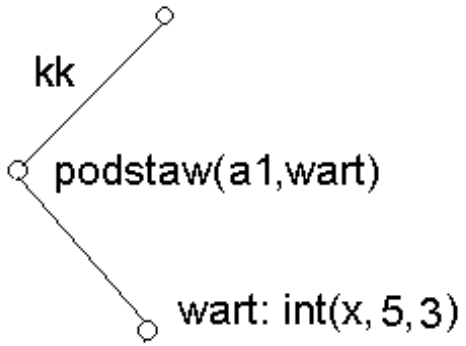

kk
podstaw(ad,wart)
ad: adres(x)
wart: int(x, c,d)
c: int(y)
d: int(3)

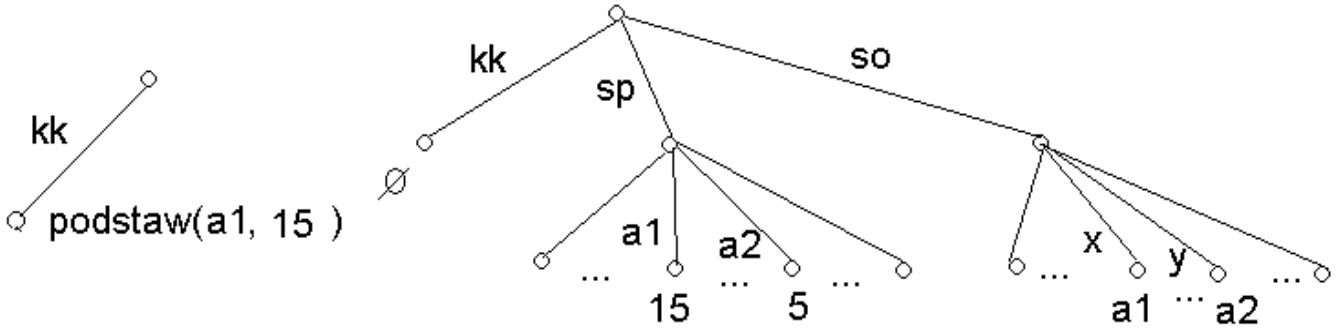

kk
podstaw(ad,wart)
ad: adres(x)
wart: int(x, c,d)
c: cont(y)
d: int(3)


kk
podstaw(ad,wart)
ad: adres(x)
wart: int(x, c,d)
c: cont(y)
d: val(3)


kk
podstaw(a1,wart)
wart: int(x, c,d)
c: cont(y)
d: val(3)


kk
podstaw(a1,wart)
wart: int(x, c, 3)
c: cont(y)


kk
podstaw(a1,wart)
wart: int(x, 5, 3)



– w ***metodzie denotacyjnej*** (matematycznej, funkcyjnej) ***znaczeniem*** napisu **Syn** ***jest obiekt semantyczny*** nie związany bezpośrednio z pojęciem obliczenia lecz ***charakteryzujący*** abstrakcyjne ***środowisko obliczeniowe*** (stanowiące reprezentację **Dsem**)

– funkcja semantyczna **Sem** przypisuje bezpośrednio znaczenia wyrażeniom elementarnym języka, natomiast dla wyrażeń złożonych stosuje operatory umożliwiające składanie znaczeń (znaczenie konstrukcji złożonej jest kombinacją znaczeń konstrukcji składowych)

11.2 Metoda denotacyjna

– przyjmiemy, że ***znaczeniem*** wyrażeń języka programowania są ***funkcje*** określone ***w zbiorze stanów*** abstrakcyjnego ***środowiska obliczeniowego***

– należy ustalić: ***dziedzinę syntaktyczną*** (składnia abstrakcyjna programów i ich składowych), ***dziedziny semantyczne***, ***funkcje bazowe***, ***operatory funkcyjne***

– **przykład** (semantyka denotacyjna prostego języka programowania)

a) ***dziedziny syntaktyczne*** **Syn** – Op: operatory arytmetyczne, relacyjne i logiczne, Exp: wyrażenia arytmetyczne i logiczne, Stat: instrukcje,

b) ***dziedziny semantyczne*** **DSem**

$TF = \{t, f\}$	wartości logiczne,
Z	zbiór liczb całkowitych,
$D = TF \cup Z \cup \{\perp\}$	dziedzina wszystkich wartości,
$C = [Ide \rightarrow D]$	zbiór stanów środowiska (Ide to nazwy zmiennych)

$[C \to D]$ dziedzina „wartości w stanie” (dla wyrażeń)

$[C \bullet \to C]$ dziedzina interpretacji (funkcja częściowa)

c) ***funkcje bazowe*** **DSem** (tworzą elementarne byty semantyczne) – funkcje arytmetyczne, relacje i operacje logiczne, określone na dziedzinie D

d) ***operatory funkcyjne*** **DSem** (tworzą złożone byty semantyczne) – złożenie funkcji częściowych, warunkowy wybór, podstawienie i operator punktu stałego

e) ***semantyka*** **Sem** wyrażeń Exp (funkcja value: $Exp \to [C \to D]$)

$value(true)(c) = t$

$value(false)(c) = f$

$value(n)(c) = \underline{n}$

$value(x)(c) = c(x)$, x jest nazwą zmiennej

$value(E_1\ \Delta_2\ E_2)(c) = value(E_1)(c)\ \underline{\Delta_2}\ value(E_2)(c)$

value(Δ_1 E)(c) = $\underline{\Delta_1}$ value(E)(c)

f) ***semantyka*** **Sem** instrukcji Stat (funkcja SD: Stat $\rightarrow$ [C $\bullet$ $\rightarrow$ C])

SD(x:=E)(c) = c(x/value(E)(c))

SD(skip) = Id

SD(if R then S_1 else S_2 fi)(c) = (value(E) $\rightarrow$ SD(S_1), SD(S_2))

SD(while E do S od)(c) = Y(*F*),

gdzie $F(\xi)$ = (value(E) $\rightarrow$ SD(S)$^\circ$ ξ, Id)

SD(S_1;S_2)(c) = SD(S_1) $^\circ$ SD(S_2)

SD(begin DL; S end)(c) = SD(S)

– w ***metodach aksjomatycznych*** formułuje się zdania o programach (na gruncie określonych systemów formalnych) i dowodzi ich prawdziwości

a) zdania mogą dotyczyć np. ***zależności między problemem a jego rozwiązaniem*** w postaci programu (poprawność programu, równoważność programów itp.)

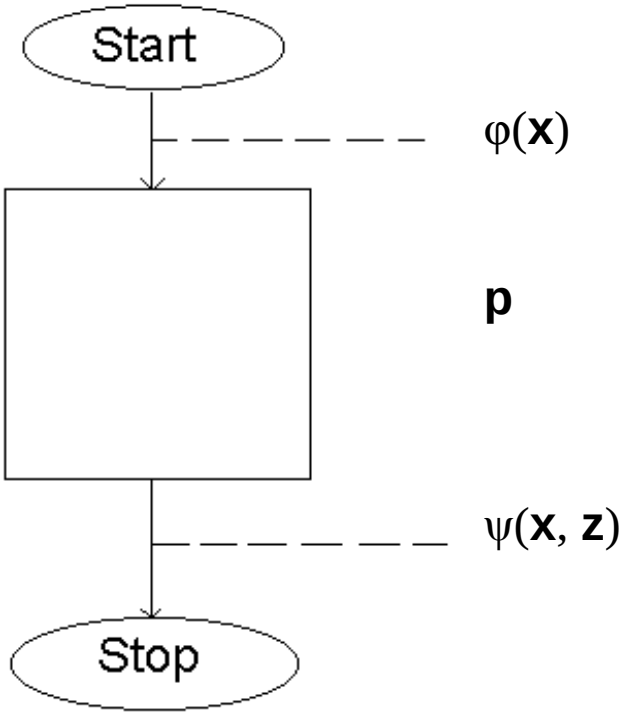
b) przykładami systemów są: ***system wnioskowania C.A.R. Hoare'a***, logiki programów: temporalna, dynamiczna, algorytmiczna i in.

11.3 System wnioskowania Hoare'a

– weryfikowanie zdań dotyczących zgodności rozwiązania (programu **p**) z zadaną specyfikacją (warunkami nałożonymi na dane i funkcję pomiędzy danymi i wynikami): poprawność częściowa programu

– program **p**, wektor danych wejściowych **x**, wektor danych roboczych **y**, wektor danych wyjściowych **z**

– predykat wejściowy $\phi(\mathbf{x})$, predykat wyjściowy $\psi(\mathbf{x}, \mathbf{z})$



– ***poprawność częściowa*** programu **p** ze względu na predykat wejściowy φ i predykat wyjściowy ψ:

$$\varphi(\mathbf{x}) \wedge \mathbf{p} \text{ zatrzyma się} \rightarrow \psi(\mathbf{x}, \mathbf{z})$$

– metoda weryfikacji poprawności częściowej programów w notacji „pascalopodobnej”

– program jest skończonym ciągiem instrukcji, rozdzielonych średnikami:

B_0; B_1; B_2; …; B_n,

B_0 jest instrukcją startu START

y := f(**x**)

– pozostałe instrukcje mogą przyjąć postać:

y := g(**x**, **y**), *instrukcja przypisania*

if t(**x**, **y**) then B else B’ lub

if t(**x**, **y**) do B, *instrukcja warunkowa* (B i B’ są dowolnymi instrukcjami)

while t(**x**,**y**) do B, *instrukcja iteracyjna*

z := h(**x**, **y**)

STOP, *instrukcja stopu*

begin B_1; B_2; …; B_n end, *instrukcja złożona*.

– C.A.R. Hoare wprowadził następującą notację:

{p(**x**, **y**)} B {q(**x**, **y**)}

oznaczającą, że jeśli predykat p(**x**, **y**) jest prawdziwy bezpośrednio przed wykonaniem segmentu B programu, to predykat q(**x**, **y**) będzie prawdziwy bezpośrednio po wykonaniu tego segmentu;

– następujące wyrażenie rozważanej postaci wyraża poprawność częściową programu **p** ze względu na predykat wejściowy φ(**x**) i predykat wyjściowy ψ(**x**, **z**):

{φ(**x**)} **p** {ψ(**x**, **z**)}

– Hoare zdefiniował rachunek logiczny, w którym podał aksjomat i reguły wnioskowania umożliwiające przeprowadzanie dedukcji dotyczących programów

– aksjomat i reguły przyjmują postać:

1) aksjomat dla instrukcji przypisania

{p(**x**, g(**x**, **y**))} **y** := g(**x**, **y**) {p(**x**, **y**)}

2) reguły dla instrukcji warunkowej

$\{p \wedge t\}\ B_1\ \{q\},\ \{p \wedge \neg t\}\ B_2\ \{q\}$

$\{p\}$ if t then B_1 else B_2 $\{q\}$

$\{p \wedge t\}\ B\ \{q\},\ \{p \wedge \neg t\} \rightarrow \{q\}$

$\{p\}$ if t do B $\{q\}$

3) reguła dla instrukcji iteracyjnej

$\{p \wedge t\}\ B\ \{p\}$

$\{p\}$ while t do B $\{p \wedge \neg t\}$

4) reguła dla instrukcji złożonej

$$\frac{\{p\}\ B_1\ \{q\},\ \{q\}\ B_2\ \{r\}}{\{p\}\ B_1; B_2\ \{r\}}$$

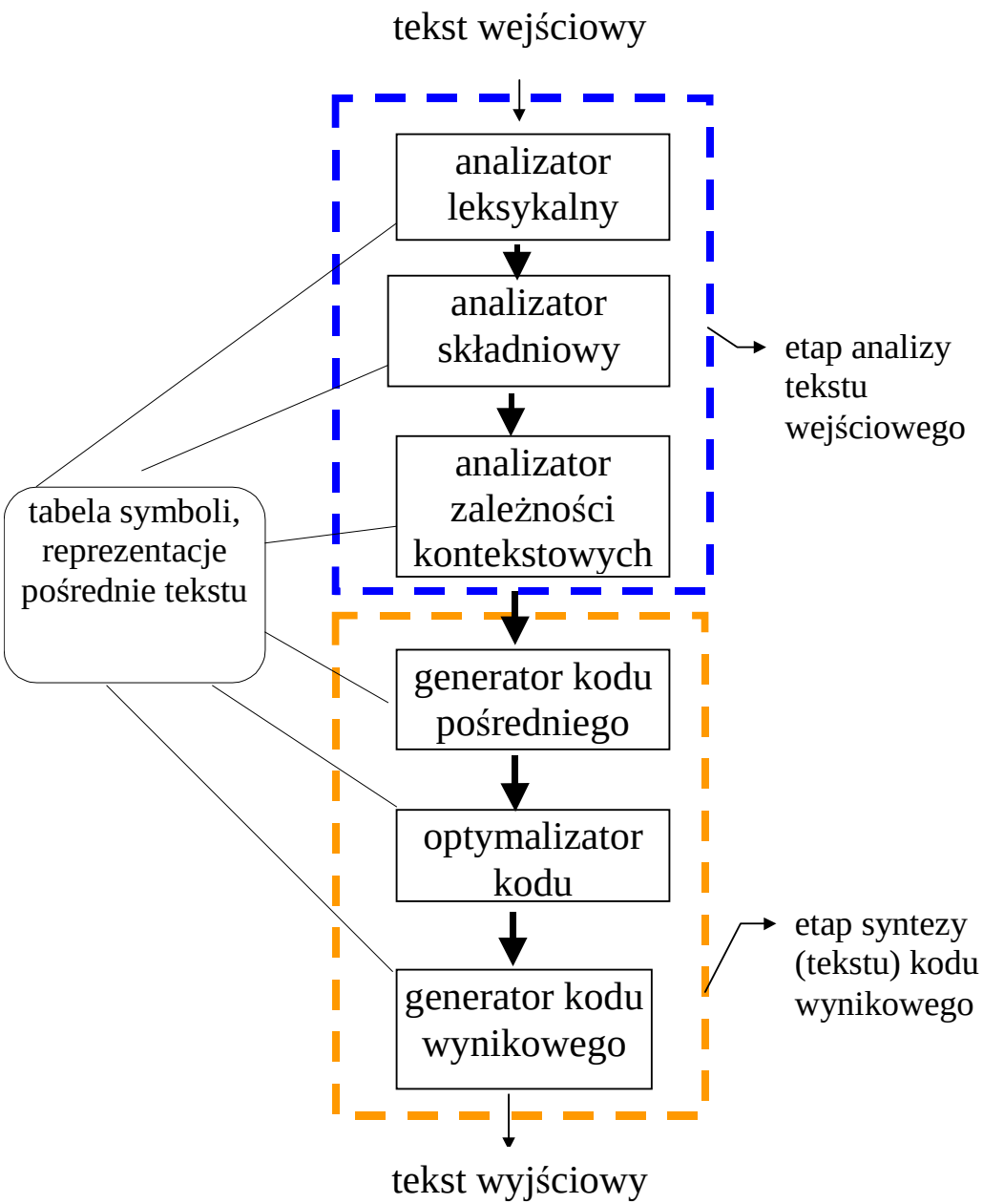
5) reguły logiczne

$$\frac{p \rightarrow q,\ \{q\}\ B\ \{r\}}{\{p\}\ B\ \{r\}}$$

$$\frac{\{p\}\ B\ \{q\},\ q \rightarrow r}{\{p\}\ B\ \{r\}}$$

– metoda weryfikacji częściowej poprawności programu wg Hoare'a: dany jest program **p**, predykat wejściowy $\varphi(\mathbf{x})$ i predykat wyjściowy $\psi(\mathbf{x}, \mathbf{z})$; jeśli przez stosowanie reguł 1) – 5) można wydedukować formułę $\{\varphi(\mathbf{x})\}$ **p** $\{\psi(\mathbf{x}, \mathbf{z})\}$, to program **p** jest częściowo poprawny ze względu na φ i ψ.

– wśród metod pochodnych można wyróżnić ***semantykę akcyjną*** P.D. Mossesa, a także semantyki algebraiczne;


tekst wejściowy
analizator leksykalny
analizator składniowy
analizator zależności kontekstowych
etap analizy tekstu wejściowego
tabela symboli, reprezentacje pośrednie tekstu
generator kodu pośredniego
optymalizator kodu
generator kodu wynikowego
etap syntezy (tekstu) kodu wynikowego
tekst wyjściowy

12 Analiza zależności kontekstowych

12.1 Wprowadzenie

- ***analiza zależności kontekstowych*** można wykonać na etapie kompilacji programu (***statyczna weryfikacja zgodności typów*** (ang. *static checking*)) lub na etapie jego wykonania (***dynamiczna weryfikacja zgodności typów*** (ang. *dynamic checking*))
- na etapie kompilacji wykonuje się:
  - badanie ***zgodności typów*** (ang. *type checks*), np. x + y
  - badanie ***kontekstu występowania*** pewnych ***instrukcji*** (ang. *flow-of-control checks*), np. `break` w języku C
  - badanie ***unikatowości*** deklaracji ***identyfikatorów*** (ang. *uniqueness checks*)
  - badanie ***powiązanych wystąpień nazw*** (ang. *name-related checks*), np. konieczność użycia tej samej nazwy na początku i na końcu bloku w języku Ada
- statyczny weryfikator zgodności typów można zaprogramować posługując się translacją sterowaną składnią

12.2 Weryfikator zgodności typów

- ***weryfikacja zgodności typów*** polega na sprawdzeniu, czy ***typy*** poszczególnych ***konstrukcji językowych*** odpowiadają ***typom oczekiwanym*** w ***kontekście*** ich występowania
- należy zdefiniować ***system typów*** oraz z ***reguły przypisywania*** typów konstrukcjom języka
- do oznaczania typów konstrukcji językowych służy ***wyrażenie typowe*** (ang. *type expressions*)
- definicja wyrażeń typowych:

1) ***identyfikator typu podstawowego*** jest wyrażeniem typowym, np. ***boolean***, ***char***, ***integer***, ***real***, typ specjalny ***type_error*** oraz typ zastępczy ***void***

2) ***nazwa typu*** jest wyrażeniem typowym

3) ***konstruktor typu złożonego*** zastosowany do wyrażeń typowych jest wyrażeniem typowym:

   – konstruktor tablicy ***array***; jeśli T jest wyrażeniem typowym, a I oznacza pewien zbiór indeksowy, to ***array***(T, I) jest wyrażeniem typowym

   – konstruktor produktu kartezjańskiego $\times$; jeśli T_1 oraz T_2 są wyrażeniami typowymi, to ich produkt kartezjański $T_1 \times T_2$ jest także wyrażeniem typowym

   – konstruktor rekordu ***record***; jeśli k jest krotką, składającą się z nazw pól i stowarzyszonych z tymi nazwami typów, to ***record*** k ***end*** jest także wyrażeniem typowym

   – konstruktor referencji ***pointer***; jeśli T jest wyrażeniem typowym, to ***pointer***(T) jest wyrażeniem typowym, oznaczającym referencję do obiektu typu T

   – konstruktor funkcji ***function***, odwzorowującej elementy jednego zbioru (***dziedziny*** D) w inny zbiór (***przeciwdziedzinę*** R); typ funkcji oznaczamy za pomocą wyrażenia typowego $D \rightarrow R$

4) wyrażenia typowe mogą zawierać zmienne, wartościowane w zbiorze wyrażeń typowych.

- ***system typów*** jest zbiorem reguł przypisywania wyrażeń typowych różnym częściom składowym programu
- weryfikator zgodności typów stanowi implementację systemu typów
- w ***silnym systemie typów*** weryfikacja odbywa się jedynie na etapie kompilacji
- **przykład** prostego weryfikatora zgodności typów zrealizowanego w paradygmacie translacji sterowanej składnią

Fragment analizatora leksykalnego:

```
ident [a-zA-Z][0-9]*
%%
"var"   {return(var);}
{ident}{yylval.text=strdup(yytext);
         return(id);
        }
[0-9]+ {yylval.integer=atoi(yytext);
         return(num);
        }
"'".+"'" {return(literal);}
":="     {return(ass);}
...
```

Tabela symboli:

```
#include <stdio.h>
#include "typexpr.h"
typedef struct t1 {char     *name;
                   TypeExpr *type_;
                   t1       *next;
                  }Descrip;


Descrip *Vars=NULL;
TypeExpr* VarLookup(char *n)
    {
     …
    }
char AddVar(char *n, TypeExpr *t)
   {
    …
   }
```

Analizator składniowy i weryfikator zgodności typów:

```
VD: id ':' T ';'              {if (!AddVar($1,$3)) ERROR;}
  | VD id ':' T ';'           {if (!AddVar($2,$4)) ERROR;}
  ;

T : charac                    {$$=NewBasic("char");}
  | '^' T                     {$$=NewPointer($2);}
  | array '[' num ']' of T    {$$=NewArray($3,$6);}
  ;


E : literal                   {$$=NewBasic("char");}
  | num                       {$$=NewBasic("integer");}
  | id                        {$$=VarLookup($1);}
  | E mod E                   {if (IsInt($1) && IsInt($3)) $$=$1;
                               else ERROR;}
  | E '[' E ']'               {if (IsArray($1) && IsInt($3))
                                  $$=ElemType($1);
                               else ERROR;}
  | E '^'                     {if (IsPointer($1))
                                  $$=ElemType($1);
                               else ERROR;}
  | E '=' E                   {if (IsChar($1) && IsChar($3) ||
                                    IsInt($1) && IsInt($3))
                                     $$=NewBasic("Boolean");
                               else ERROR;}
  ;
```

```
S : id ass E          {if (!Equiv(VarLookup($1),$3)) ERROR;}
  | _if E then S      {if (!IsBoolean($2)) ERROR;}
  | begin SSeq end
  ;

Sseq
  : S
  | Sseq ";" S
  ;


%%
Boolean Equiv(t1, t2)
  TypeExpr *t1, *t2;
{if (IsBasic(t1) && IsBasic(t2))
  return (!strcmp(NameOf(t1), NameOf(t2)));
 else if (IsPointer(t1) && IsPointer(t2))
  return (Equiv(ElemType(t1),ElemType(t2)));
 else if (IsArray(t1) && IsArray(t2))
  return (ArrayIndx(t1)==ArrayIndx(t2) &&
          Eqiuv(ElemType(t1),ElemType(t2)));
 else
  return (False);}
```

12.3 Konwersje typów

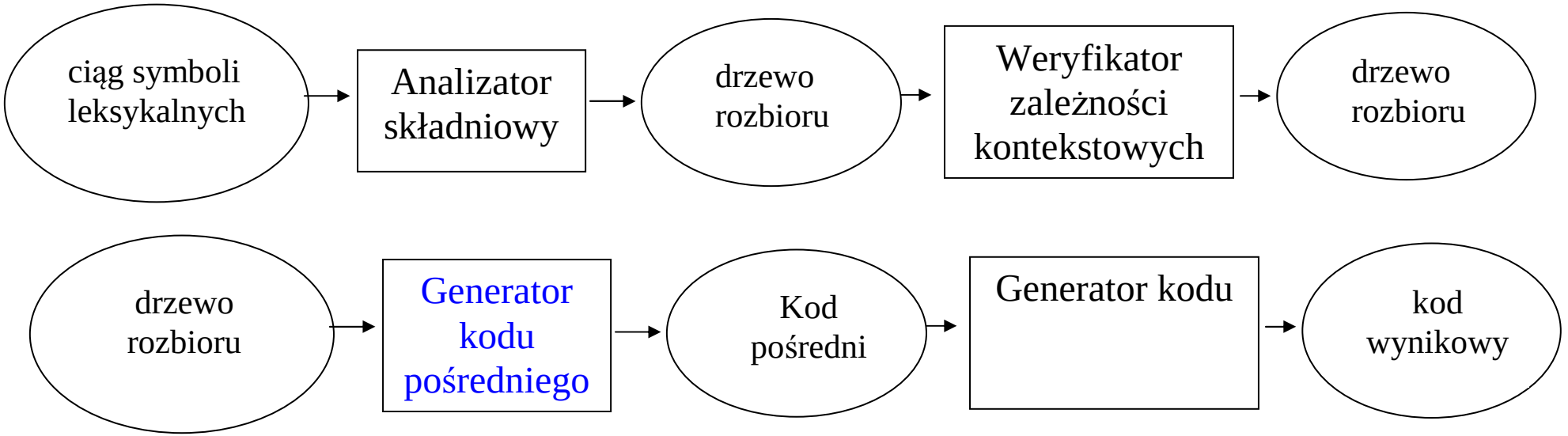
- ze względu na ***różną reprezentację maszynową liczb*** typów całkowitego i rzeczywistego oraz zróżnicowanie instrukcji maszynowych działających na liczbach całkowitych i liczbach rzeczywistych, ***kompilator modyfikuje typy pewnych*** szczególnych ***operandów***
- dotyczy to, między innymi, przypadku wyrażenia o postaci `x + i`, w którym x reprezentuje zmienną typu rzeczywistego, a i – zmienną typu całkowitego
- w celu poprawnego zrealizowania operacji dodawania koniecznym jest, by oba operandy były tego samego typu; w rozważanym wyrażeniu zmienna `i` powinna zostać zmodyfikowana do postaci zmiennej typu rzeczywistego
- omówiona modyfikacja typu operandu nazywa się ***konwersją typów*** (ang. *type conversion*)
- konwersje typów mogą mieć charakter ***jawny*** (ang. *explicit*), bądź ***niejawny*** (ang. *implicit*); w pierwszym przypadku definiuje je programista (np. jawne konwersje w języku Ada lub C); konwersje niejawne, zwane ***koercjami*** są automatycznie realizowane przez kompilator.

13 Generacja kodu pośredniego

13.1 Charakterystyka problemu

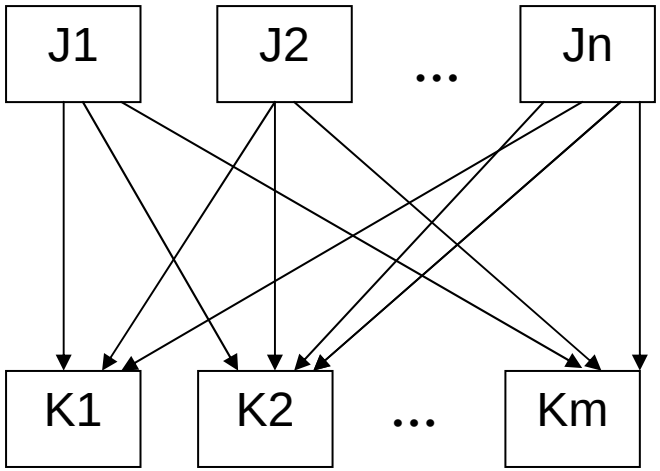
- ***w podstawowym modelu kompilatora*** wszystkie jego ***przebiegi*** dzielą się na ***analizujące*** (ang. *front end*) i ***generujące*** (ang. *back end*)
- pomiędzy analizatorami a generatorem występuje zazwyczaj pewien ***program*** zapisany w języku formalnym zwanym ***językiem pośrednim*** (ang. *intermediate language*)
- chociaż można konstruować ***kompilatory jednoprzebiegowe*** (tłumaczące kod źródłowy na kod maszyny docelowej), to kompilator ***wieloprzebiegowy***, oparty na ***maszynowo niezależnym języku pośrednim*** ma zalety polegające na:
  a) łatwości jego przeniesienia na inne maszyny docelowe
  b) możliwości zaprojektowania efektywnego, maszynowo niezależnego optymalizatora kodu
- translację drzewa rozbioru na zapis w języku pośrednim można zaprogramować za pomocą translacji sterowanej składnią

13.2 Generator kodu pośredniego

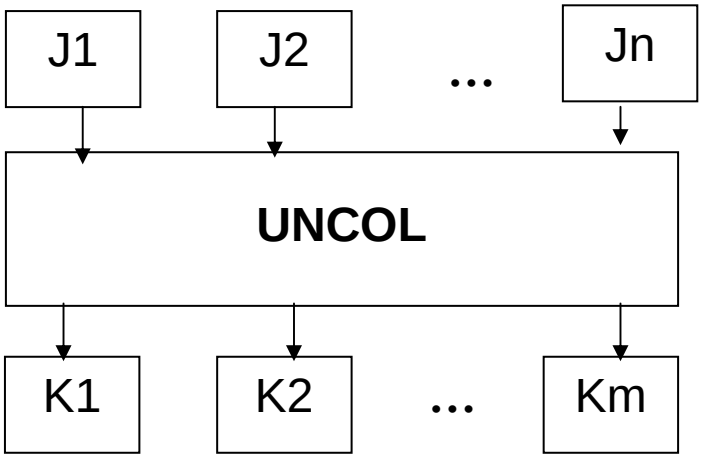


13.3. Języki pośrednie

Koncepcja uniwersalnego języka pośredniego UNCOL (ang. *Universal Computer Oriented Language*)



m × n translacji



m + n translacji

13.3.1. Kategoryzacja języków pośrednich

- klasyfikacja języków ze względu na stopień ich ***uniwersalizmu***:

– języki maszynowo niezależne, zorientowane na język źródłowy

– języki pseudo-uniwersalne, tzn. ukierunkowane na grupę języków źródłowych i klasę maszyn docelowych

– języki źródłowo niezależne, maszynowo uwarunkowane

- klasyfikacja języków ze względu na ich ***poziom*** – stopień szczegółowości dostępnych w nich operacji:
  - języki wysokiego poziomu, zbliżone do języka źródłowego
  - języki niskiego poziomu, zbliżone do assemblera
- klasyfikacja języków ze względu na ***sposób*** przyjętej w nich ***reprezentacji programu***:
  - języki o notacji opisowej, np. o postaci drzew składniowych
  - języki o notacji imperatywnej, np. o postaci: ***linearnej postfiksowej***, ***kodu trójadresowego*** (trójkowego lub czwórkowego)

13.3.2. Kryteria oceny języka pośredniego

– niezawodność

– efektywność mierzona czasem wykonywania kompilowanego programu

– wysokość kosztów przenoszenia oprogramowania

– efektywność mierzona czasem kompilacji programu

13.3.3. Kod trójadresowy

– ***kod trójadresowy*** jest językiem ***imperatywnym***, ***maszynowo niezależnym***, o ***niskim*** poziomie abstrakcji

Typy najczęściej używanych instrukcji kodu trójadresowego:

1) ***instrukcja podstawienia***, o postaci **x := y op z**, gdzie **op** oznacza binarny operator arytmetyczny lub logiczny, np. +, *, AND, OR

2) ***instrukcja podstawienia***, o postaci **x := op y**, gdzie **op** oznacza unarny operator arytmetyczny lub logiczny, np: – , NOT

3) ***instrukcja kopiowania***, o postaci **x := y**, oznaczająca podstawienie wartości **y** za zmienną **x**

4) ***instrukcja skoku bezwarunkowego***, o postaci **goto L**, oznaczająca nakaz wykonania w następnej kolejności instrukcji oznaczonej etykietą **L**

5) ***instrukcja skoku warunkowego***, o postaci **if x relop y goto L**, gdzie **relop** oznacza binarny operator relacyjny, np: >=, <>

6) ***instrukcje obsługi podprogramów***, o postaci **param x**, **call p, n** oraz **return y**, gdzie **p** oznacza nazwę podprogramu, a **n** liczbę jego parametrów aktualnych; typowe użycie wymienionych instrukcji to sekwencja o postaci:

param x1
param x2
...
param xn
call p, n

stanowiąca przekład instrukcji **p(x1, x2, ..., xn)** pochodzącej z języka źródłowego;
instrukcja **return y** ma charakter opcjonalny; powinna pojawić się w przekładzie podprogramu będącego funkcją

7) ***instrukcje kopiowania indeksowanego***, o postaci **x := y[i]** oraz **x[i] := y**, oznaczające podstawienie, odpowiednio:

– wartości przechowywanej w **i**-tej jednostce pamięci następującej za jednostką początkową, zarezerwowaną do reprezentacji **y** – za zmienną **x** oraz

– wartości przechowywanej w jednostce **y** do **i**-tej jednostki pamięci następującej za jednostką początkową, zarezerwowaną do reprezentacji zmiennej **x**;

8) ***instrukcje kopiowania***, manipulujące adresami, o postaci **x := &y**, **x := *y** oraz ***x := y**, oznaczające podstawienie, odpowiednio:

- adresu jednostki pamięci używanej do reprezentacji zmiennej **y** – za zmienną **x**
- wartości przechowywanej w jednostce pamięci o adresie **y** – za zmienną **x**
- wartości **y** – do jednostki pamięci, której adres jest reprezentowany zmienną **x**.

13.3.4. Translacja instrukcji podstawienia na kod trójadresowy

S-atrybutowa definicja generacji kodu trójadresowego dla instrukcji podstawienia:

Produkcja	Reguła semantyczna
S → id := E	S.code := E.code || id.place ':=' E.place
E → E1 + E2	E.place := newtemp; E.code := E1.code || E2.code || E.place ':=' E1.place '+' E2.place
E → E1 * E2	E.place := newtemp; E.code := E1.code || E2.code || E.place ':=' E1.place '*' E2.place
E → – E1	E.place := newtemp; E.code := E1.code || E.place ':=' 'uminus' E1.place
E → (E1)	E.place := E1.place; E.code := E1.code
E → id	E.place := id.place; E.code := ''

Objaśnienia:

- syntetyzowany atrybut **code** reprezentuje kod trójadresowy dla przekładu: wyrażenia arytmetycznego (nieterminal E) oraz instrukcji podstawienia (nieterminal S)
- syntetyzowany atrybut **place** reprezentuje nazwę zmiennej służącej do przechowywania wartości wyrażenia arytmetycznego (nieterminal E)
- || oznacza operator sklejania kodu
- **newtemp** jest nazwą funkcji bezparametrowej, generującej nazwy zmiennych tymczasowych t1, t2, ..., tn w odpowiedzi na kolejne jej wywołania.

Schemat translacji w języku YACC, zdefiniowanej wyżej instrukcji podstawienia na kod trójadresowy; generacja kodu pośredniego jest w nim stowarzyszona z procesem parsingu:

```
E : id
  | E ‘+’ E    {printf("%s:= %s + %s;\n", $$=NewTemp(), $1, $3);}
  | ‘-’ E      {printf("%s:= - %s; \n",   $$=NewTemp(), $2);}
  | E ‘*’ E    {printf("%s:= %s * %s;\n", $$=NewTemp(), $1, $3);}
  | ‘(‘ E ‘)’  {$$=$2;}
  ;
S : id ass E   {printf("%s:= %s;\n", $1, $3);}
  ;

%%
int TempNo=0;
char *NewTemp()
     { char *s;
      s=(char*)calloc(7,sizeof(char));
      sprintf(s, "t%d",TempNo++);
      return(s);}
```

13.3.5.Translacja instrukcji sterujących na kod trójadresowy

S-atrybutowa definicja generacji kodu trójadresowego dla instrukcji warunkowej o postaci ***if e then s***, w której ***e*** oznacza wyrażenie logiczne zbudowane ze zmiennych i operatora relacyjnego ‘=‘, zaś ***s*** jest instrukcją podstawienia lub zagnieżdżoną instrukcją warunkową:

Produkcja	Reguła semantyczna
S → id := E	S.code := E.code ‖ id.place ‘:=‘ E.place
S → if E then S1	L := newlabel; S.code := E.code ‖ ‘if’ E.place = ‘0 goto’ L ‖ S1.code ‖ L ‘:’
E → E1 = E2	E.place := newtemp; L := newlabel; E.code := E1.code ‖ E2.code ‖ E.place ‘:= 1’ ‖ ‘if’ E1.place ‘=‘ E2.place ‘goto’ L ‖ E.place ‘:=‘ 0 ‖ L ‘:’
E → id	E.place := id.place; E.code := ‘’

Objaśnienia:

– syntetyzowany atrybut **code** reprezentuje kod trójadresowy dla przekładu: wyrażenia logicznego (nieterminal E) oraz instrukcji podstawienia lub warunkowej ***if e then s*** (nieterminal S),

– syntetyzowany atrybut **place** reprezentuje nazwę zmiennej służącej do przechowywania wartości wyrażenia logicznego (nieterminal E)

– wyrażenia logiczne są ewaluowane w zbiorze wartości numerycznych {0, 1}

– || oznacza operator sklejania kodu

– **newtemp** jest nazwą funkcji bezparametrowej, generującej nazwy zmiennych tymczasowych t1, t2, ..., tn w odpowiedzi na kolejne jej wywołania

– **newlabel** jest nazwą funkcji bezparametrowej, generującej etykiety l1, l2, ..., lm w odpowiedzi na kolejne jej wywołania.

Schemat translacji zdefiniowanej wyżej instrukcji warunkowej ***if e then s*** na kod trójadresowy; generacja kodu pośredniego jest w nim stowarzyszona z procesem parsingu:

```
E : id
  | E '=' E      {printf("%s:= 1;\n", $$=NewTemp());
                  printf("if %s= %s goto %s;\n",$1,$3, l=NewLab());
                  printf("%s:= 0;\n", $$);
                  printf("%s:\n", l);}
  ;
S : id ass E     {printf("%s:=%s; \n", $1, $3);}
  | _if E THEN S {printf("%s:\n", PopLab());}
  ;
THEN
  : then         {printf("if %s=0 goto %s;\n",$<text>0, l=NewLab());
                  PushLab(l);}
  ;

%%
int TempNo=0;
    LabNo=0;
char *NewTemp()
     { char *s;
       s=(char*)calloc(7, sizeof(char));
       sprintf(s, "t%d", TempNo++);
       return(s);}
char *NewLab()
     { char *s;
       s=(char*)calloc(7, sizeof(char));
```

```
      sprintf(s, ”l%d”, LabNo++);
      return(s);}
```

```
if a=2 then a:=1

t1:= 1
if a=2 goto l1
t1=0
l1:
if t1=0 goto l2       ----> l2 na stos
a:=1
l2:                   ----> l2 ze stosu
```

Objaśnienia:

- atrybutem wyrażenia logicznego jest miejsce przechowywania jego wartości
- instrukcje warunkowe mogą być zagnieżdżone, np.:

if e1 then
if e2 then
if e3 then id1 := id2 = id3

z tego powodu do ich poprawnej translacji niezbędny jest stos etykiet oznaczających końce poszczególnych instrukcji warunkowych; obsługę tego stosu zapewniają funkcje:

- bezargumentowa **PopLab**, zdejmująca szczytową etykietę ze stosu i przekazująca ją do miejsca wywołania funkcji
- jednoargumentowa **PushLab**, umieszczająca etykietę będącą argumentem jej wywołania na szczycie stosu.