ZESZYTY NAUKOWE POLITECHNIKI ŚLĄSKIEJ 1993

Seria: INFORMATYKA z. 24 Nr kol. 1222

Dariusz ZONENBERG

# ODWROTNA NOTACJA POLSKA W ZAPISIE KANONICZNEJ FORMY ALGORYTMÓW I SYNTEZIE SYSTEMÓW STEROWANYCH PRZEPŁYWEM ARGUMENTÓW

Streszczenie. W opracowaniu zaprezentowano możliwy sposób generacji kodu przez kompilatory języków wysokiego poziomu jako zdań w Odwrotnej Notacji Polskiej (ONP). Następnie zostało wskazane, że zdanie w ONP jest zapisem algorytmu w postaci kanonicznej. Jako wnioski zaprezentowano przykładowy system sterowany przepływem argumentów o ograniczonych zasobach, oparty na realizacji zdania ONP. Opisano jego strukturę pamięci, metodę organizacji przepływu danych oraz wskazano problemy implementacji.

# THE POSTFIX NOTATION IN A CANONICAL ALGHORITM FORM DESCRIPTION AND ARGUMENT FLOW SYSTEM SYNTHESIS

Summary. The article presents a method of generating code by high level language compilers as a postfix notation statement. Next, it was pointed out that prefix notation is a canonical alghoritm form description. As a conclusion, the article presents an example argument flow system with limited resources based on postfix notation execution. Memory structure, data flow organization and implementation problems were also mentioned.

# LA NOTATION INVERSE POLONAISE DANS DESCRIPTION DE LA FORME CANONIQUE DES ALGORITHMES ET DANS LA SYNTHESE DES SYSTEMES CONTROLES PAR LE FLUX DES ARGUMENTS

Resumé. L'article présente une méthode de la génération d'un code par les compilateurs de languages de haut niveau comme des phrases dans la notation polonaise. On démontre qu'une phrase dans la notation polonaise peut engendrer la description d'un algorithme dans sa forme canonique. Un système controlé par le flux des arguments décrit dans cette notation est presenté. On analyse la structure de son mémoire, l'organisation du flux de données et on indique les problèmes de son implémentation.

## 1. Wprowadzenie

W pracy [1] wprowadzono pojęcie kanonicznej formy algorytmu i wykazano, że może ona stanowić dogodną podstawę do syntezy realizującego ten algorytm systemu sterującego przepływem argumentów.

W niniejszym opracowaniu wskazuje się na to, że ONP (Odwrotna Notacja Polska), którą stosuje się zazwyczaj do zapisu wyrażeń arytmetycznych, może być też wykorzystana do formalnego zapisu algorytmu w postaci kanonicznej, a utworzony zapis do syntezy realizującego ten algorytm systemu sterowanego przepływem argumentów.

Dla wprowadzenia, przedstawione zostaną na początku niektóre podstawowe pojęcia Odwrotnej Notacji Polskiej. Przez ONP rozumie się język formalny generowany następującą gramatyką:

$$< Z >::= \underbrace{< Z_p >, < Z_p >, \ldots, < Z_p >}_{m}, < O_m > \mid < Z_p >$$

$$< Z_p >::= \underbrace{< A >, < A >, \ldots, < A >}_{n}, < O_n > \mid < A >$$

gdzie

$Z$ — zdanie złożone,

$Z_p$ — zdanie proste,

$A$ — argument,

$O_m, O_n$ — operacje $m, n$-argumentowe ($m, n = 1, 2, \ldots$)

Język ONP jest stosowany do zapisu wyrażeń arytmetycznych jako bardziej wygodny niż język wyrażeń nawiasowych, ze względu na brak priorytetu poszczególnych operacji. Przykładowo

$$(a + b) * c$$

w ONP zapisujemy jako

$$a, b, +, c, *$$

Zauważmy, że w trakcie wyliczania wartości wyrażenia przyjmowane jest niejawne założenie, że każdy argument lub wynik wyrażenia prostego będzie użyty tylko raz. Wynika to z właściwości wyrażeń arytmetycznych zapisanych ONP i wiąże się z wykorzystywaniem stosu do wyliczania wyrażenia w ONP (wg [3], [4], [5]). Wyrażenie ONP może być następnie rozwinięte w kod 4-argumentowy lub kod 3-argumentowy (wg [2], [4], [5]), na potrzeby optymalizacji kodu wynikowego.

Struktury sterujące przepływem argumentów zdefiniowane w [1] dla pewnej klasy wykonywanych operacji można zapisać za pomocą powyżej opisanej ONP. Klasa ta, to operacje mające jeden wynik (kwant informacji szczególny dla danej architektury — słowo, wektor, pakiet danych), który jest operandem następnej operacji. Wtedy, przyjmując przykładowe operacje 2-argumentowe, $O^1$ i $O^2$ takie, że

$$O^1 : wynik = O^1(a, b) = aO^1b = a, b, O^1$$

$$O^2 : wynik = O^2(x, y) = xO^2y = x, y, O^2$$

to wyrażenie

$$(aO^1b)O^2c$$

można zapisać w ONP jako

$$a, b, O^1, c, O^2$$

czyli wynik jednego działania jest argumentem następnego — tylko jednego. W układzie sterowanym przepływem argumentów obecność argumentów wejściowych operacji powoduje jej realizację, a implementacja linii danych wymaga dołączenia do wartości statusu aktywności danej:

```
<argument> ::= <linia-danej> ::= <dana> <status>
```

Można by przyjąć tę klasę operacji jako szczególną i jedną z wielu innych możliwych, niemniej jest ona regułą dla algorytmów obliczania wyrażeń arytmetycznych w kodzie generowanym przez kompilatory języków wysokiego poziomu (przed optymalizacją kodu 3- lub 4-argumentowego) i jako taka stanowi podstawę automatycznej generacji kodu maszynowego. Realizacje szczegółowe zostaną przedstawione w jednym z następnych rozdziałów.

Uwaga: Brak elementarnych operacji typu

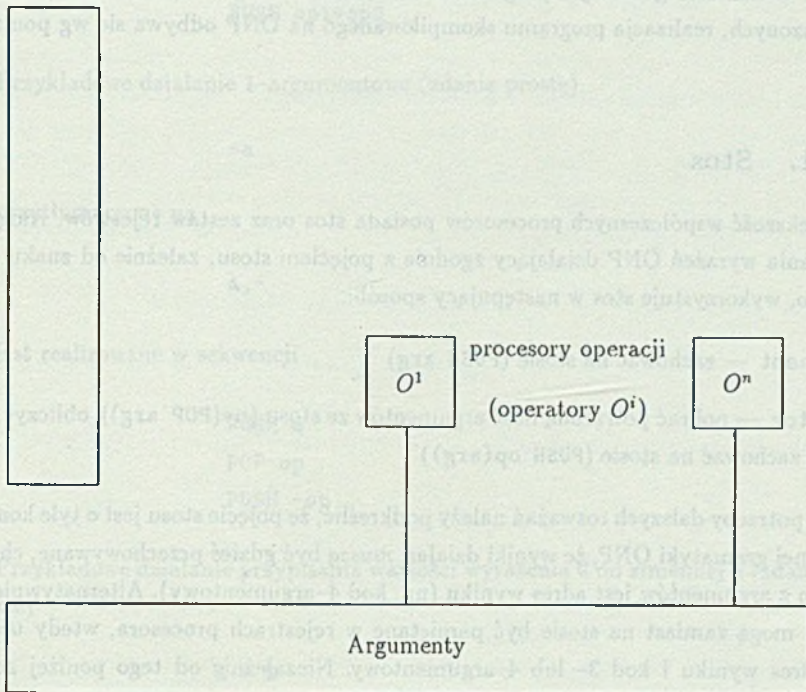
$$(w_1, w_2, \ldots) = O(a, b)$$

nie jest warunkiem koniecznym realizacji kodu o filozofii ONP. Sytuacja ta jest raczej warunkiem zwyczajowym, wynikającym z typowo wykorzystywanej przez użytkowników arytmetyki, czyli warunkiem wynikającym z typowych zastosowań. Można jednak podać szczególne zastosowania (np. procesory wektorowe lub procesory liczb zespolonych), w których wyniki muszą być wieloelementowe. W poniższym opracowaniu skupimy się na sytuacjach typowych.

## 2. Tekst programu

Przyjmując, że maszyna cyfrowa jest wykorzystywana przez przeciętnego użytkownika, kodującego algorytm w czytelnym i ergonomicznym języku programowania, a tekst programu abstrahuje od architektury procesora (choć nie zawsze jest to regułą), każdy algorytm wpierw jest zapisywany w postaci tekstu czytelnego dla programisty. Tekst ten zawiera opis kolejnych czynności algorytmu. Następnie tekst jest tłumaczony na kod maszynowy, który będzie wykonywany przez określony procesor lub zbiór procesorów — dopiero na tym etapie rozważymy, czy algorytm jest realizowany przez maszynę na zasadzie przepływu operacji, czy przepływu argumentów. Niezależnie jednak od typu procesora, wynik tłumaczenia będzie nadal pewnym zbiorem danych o określonej strukturze i sposobie uporządkowania.

Przyjmijmy na wstępie, że jednostka realizująca kod maszynowy jest zbiorem wyspecjalizowanych procesorów — każdy z nich potrafi realizować jedną z podstawowych operacji, na jakie tłumaczony jest tekst zapisany w języku wysokiego poziomu. Przyjmijmy również, że istnieje program maszynowy, czyli tekst (kod) zrozumiały z kolei dla jednostki realizującej program, który zawiera informacje o przepływie argumentów oraz załóżmy, że został on oparty na ONP — gramatyce typowo wykorzystywanej przez współczesne kompilatory. Zostało to zaprezentowane na rysunku 1.




Rys. 1. Schemat analizowanego systemu
Fig. 1. Structure of analized system

## 3. Podstawowe struktury produkowane przez kompilator języka wyższego poziomu

Na podstawie [2], [3] oraz prac planowych Zespołu Systemów Wielomikroprocesorowych w zakresie generacji oprogramowania podstawowego i systemowego środowisk rozproszonych, realizacja programu skompilowanego na ONP odbywa się wg poniższych reguł:

### 3.1. Stos

Większość współczesnych procesorów posiada stos oraz zestaw rejestrów. Alogorytm wyliczania wyrażeń ONP działający zgodnie z pojęciem stosu, zależnie od znaku terminalnego, wykorzystuje stos w następujący sposób:

argument — zachować na stosie (`PUSH arg`)

operator — pobrać potrzebną ilość argumentów ze stosu ($n*$(`POP arg`)), obliczyć wynik i zachować na stosie (`PUSH op(arg)`)

Na potrzeby dalszych rozważań należy podkreślić, że pojęcie stosu jest o tyle konieczne dla samej gramatyki ONP, że wyniki działań *muszą* być gdzieś przechowywane, chyba że jednym z argumentów jest adres wyniku (np. kod 4-argumentowy). Alternatywnie więc, wyniki mogą zamiast na stosie być pamiętane w rejestrach procesora, wtedy używany jest adres wyniku i kod 3- lub 4-argumentowy. Niezależnie od tego poniżej zostanie zaprezentowany sposób realizacji wyrażeń z użyciem stosu, ze względu na swą czytelność.

### 3.2. Wyrażenia arytmetyczne

Zasada obliczania wartości wyrażeń arytmetycznych wynika wprost z gramatyki ONP i tak

1. Przykładowe działanie 2-argumentowa (zdanie proste)

```
a+b
```

przetłumaczone na

```
a,b,+
```

jest realizowane w sekwencji

```
PUSH a
PUSH b
POP op1
POP op2
PUSH op1+op2
```

2. Przykładowe działanie 1-argumentowe (zdanie proste)

```
-a
```

przetłumaczone na

```
a,-
```

jest realizowane w sekwencji

```
PUSH a
POP op
PUSH -op
```

3. Przykładowe działanie przypisania wartości wyrażenia W do zmiennej a (zdanie proste)

```
a = W
```

przetłumaczone na

```
adres(a), ONP(W), =
```

gdzie ONP(W) — rozwinięcie wyrażenia w notacji ONP
jest realizowane w sekwencji

```
PUSH adres(a)
ONP: wyliczenie wartosci wyrazenia W (wynik na stosie)
POP op1
POP op2
(op2) := op1
```

co wskazuje na konieczność operowania również adresami operandów oraz możliwość generacji zdań ONP, pozostawiających pusty stos.

Biorąc pod uwagę strukturę sterowaną przepływem argumentów (wg [1]) w przypadku argumentów zarówno jednokrotnie używanych (nazwijmy je liniami), jak i wielokrotnie używanych (nazwijmy je n–liniami) bardzo prosto zastąpić operację PUSH wysłaniem danej na linię, a operację POP pobraniem danej z linii.

## 3.3. Sterowanie wykonaniem programu

Język programowania, oprócz instrukcji podstawienia wartości wyrażeń, wymaga implementacji struktur decyzyjnych. Poniżej podane zostaną struktury programowe i ich rozwinięcia w ONP spełniające ten wymóg:

1. Przykładowa instrukcja warunkowa dla wyrażenia logicznego W

```
if (W) then S1 else S2
```

może być przetłumaczona na

```
(etykiety)      (kod wynikowy)

                ONP(W), etS2, jmpB,
etS1:           S1, etend, jmp,
etS2:           S2
etend:
```

gdzie kod wynikowy (zdanie ONP) wymaga wyróżnienia pewnych szczególnych miejsc (*etykiety*). Przyjęto powyżej zapis dwukolumnowy, w którym jeśli dane miejsce zdania wymaga wyróżnienia, to wiersz przed nim jest łamany, a następny wiersz zawiera etykietę danego miejsca i kontynuację. Lewa kolumna zawiera etykiety, a prawa zdanie ONP — ewentualne etykiety dotyczą pierwszego znaku w nowym wierszu.

Operacja 1–argumentowa jmp jest realizowana w sekwencji

```
POP label
realizuj przetwarzanie od etykiety "label"
```

a operacja 2-argumentowa jmpB jest realizowana w sekwencji

```
POP label
POP bool
if(bool==false)
        realizuj przetwarzanie od etykiety "label"
```

Jak interpretować warunek? — Jest to decyzja o przetwarzaniu danych opisanych od określonego miejsca tekstu programu, o ile wyrażenie przyjęło pożądaną wartość. Można tą sytuację traktować jako linię danej o nieistotnej wartości, na którą oczekuje jeden z procesorów (S1 lub S2), aby rozpocząć swoje operację.

2. Przykładowa instrukcja pętli

```
while (W) do S;
```

przetłumaczona na

```
(etykiety)   (kod wynikowy)

  etbeg:     ONP(W), etend, jmpB,
             S, etbeg, jmp
  etend:
```

Jak interpretować pętle? — Jest to decyzja o przetwarzaniu danych opisanych określonym przedziałem tekstu programu, jak długo wyrażenie W przyjmuje pożądaną wartość.

3. Przykładowe wywołanie funkcji (bezparametrowej, nierekursywnej i nie pozwalającej na wielokrotne wejście).

```
fun()
```

przetłumaczone na

```
(etykiety)   (kod wynikowy)

             fun,()

 etpowr:
```

operacja () jest realizowana w sekwencji

```
POP fun
PUSH etpowr
realizuj przetwarzanie od etykiety "fun"
```

a 1-argumentowa operacja powrotu z funkcji (`ret`) będzie realizowana w sekwencji

```
POP label
realizuj przetwarzanie od etykiety "label"
```

Przekazywanie parametrów, zwracanie przez funkcję wartości oraz organizacja wielokrotnych wywołań funkcji (w tym rekursji) wymaga rozwinięcia mechanizmów gospodarki pamięcią systemu (alokacje pamięci). Na danym etapie analizy problemu kwestie te zostaną pominięte, natomiast ich przykładowe realizacje w kodach opartych na ONP można znaleźć w [4] i [5].

Każda instrukcja przetłumaczona na ONP jest sekwencją operacji prostych. Każda operacja prosta jest działaniem generującym określony wynik z argumentów (również operacja przypisania, instrukcja warunkowa, pętli oraz wywołania funkcji). Zatem program przetłumaczony na tekst w gramatyce ONP jest kanoniczną postacią algorytmu, sekwencją operacji przypisań, o postaci

$$wynik = O(a,b) = aOb = a,b,O$$

zgodnie z postulatem zawartym w [1] w celu zrealizowania systemu sterowanego przepływem argumentów.

# 4. Tekst programu procesora sterowanego przepływem argumentów

Na wstępie zastanowimy się nad znaczeniem pojęcia *etykieta* dla tekstu programu. W przypadku procesora sterowanego przepływem operacji etykieta jest wyróżnionym miejscem w tekście opisującym przepływ sterowania. Poprzez analogię, dla procesora sterowanego przepływem argumentów etykieta powinna być wyróżnionym miejscem w tekście opisującym przepływ argumentów, czyli *wyróżnionym miejscem w zdaniu gramatyki ONP.*

Tekst programu określa przepływ do procesorów operacji określonych argumentów. Argumenty mogą być różne:

- pewne argumenty istnieją od początku (dane wejściowe algorytmu),
- pewne argumenty pojawiają się w trakcie obliczeń (linie wyrażeń arytmetycznych oraz n-linie — jawne dane pośrednie algorytmu),
- pewne argumenty pojawiają się w miarę realizacji algorytmu (np. stałe w zdaniu ONP).

**Zastanówmy się nad następującym problemem:** Procesor $O_k$ ma już część argumentów gotowych do wykonania operacji i nastąpiło obliczenie wartości wyrażenia, którego operandem była etykieta (skok w tekście programu). — **Rozwiązanie wydaje się następujące:** Operacje wskazane tekstem przed skokiem powinny ulec zakończeniu, a operacje wskazane za skokiem nie powinny oczekiwać na argumenty. **Wniosek:** Istnieje pojęcie *wskazania* operacji — zaadresowanie procesora operacji, aby oczekiwał argumentów.

**Problem:** Wyrażenie arytmetyczne (a+b)*(c+d) stworzyło kłopotliwą sytuację o następujących cechach:

- Dwukrotne użycie operacji '+'!
- Czy operacja '+' wymaga dwu lub więcej procesorów?
- Przyjmijmy, że jest więcej niż 1 procesor tej operacji. Niechaj będzie ich $N$. W konsekwencji:
  - zakazujemy użytkownikowi wykonywania więcej niż $N$ dodawań w programie,
  - któreś z procesorów będą używane wielokrotnie. co ostatecznie wraca nas do punktu wyjścia — dwukrotne użycie operacji jednego procesora.
- po rozwinięciu w sieć połączeń realizacja operacji w czasie będzie następująca:

```
linia p1 = a + b
linia p2 = c + d
wynik w  = p1 * p2
```

Nasuwają się następujące wątpliwości:

a) Po czym procesor '+' pozna, gdzie przesłać wynik (p1 czy p2)?

b) Po czym procesor '+' pozna, skąd wziąć argumenty (para (a,b) czy para (c,d))?

c) Czy od razu dostarczać wszystkie argumenty, które już są? Jak się mieszczą 4 argumenty na 2 liniach danych?

Należy rozwinąć używaną notację:

Ad a) Zdanie `a,b,+` należy rozwinąć na zdanie 3-argumentowe `a,b,p1,+` wskazujące oprócz argumentów i operacji również adres jej wyniku (kod 4-argumentowy). Niepotrzebne są więc już operandy domniemane (i stos), czyli całe wyrażenie powinno być zapisane jako

```
a,b,p1,+,c,d,p2,+,p1,p2,*
```

Dotychczas nie podejmowaliśmy próby rozwiązania problemu wielodostępu do stosu przez procesory operacji elementarnych — jak się okazuje, nie jest to potrzebne. Brak domniemanego miejsca wyniku (konieczność jego podania) umożliwia ponadto rozszerzenie operacji elementarnych na wielowynikowe, jeśliby tylko zaszła taka potrzeba.

Ad b) Wskazanie procesorowi "+" argumentów i samej operacji, kiedy dochodzimy do danego miejsca tekstu programu.

Ad c) Wskazanie procesorowi *następnej* operacji dozwolone tylko wtedy, gdy procesor zakończył poprzednią — oczekiwanie po to, żeby nie cofać się w tekście programu! Jest to bardzo podobny mechanizm do *synchronicznego przekazywania komunikatów* w niektórych językach programowania systemów równoległych (np. OCCAM-2), główna jego zasada, to *czekaj aż odbiorca gotów do odebrania*:

```
syncsend(Op, argumenty)
```

Wniosek: ONP może zapisywać tekst algorytmu dla systemu sterowanego przepływem argumentów o ograniczonych zasobach pracującego wg następujących założeń:

- Adres wyniku jest jednym z argumentów każdego procesora operacji elementarnych.
- Tempo realizacji algorytmu jest synchronizowane zajętością procesorów operacji elementarnych.
- Adresy etykiet są wskaźnikami wyszczególnionych miejsc w zdaniu ONP.
- Realizacja programu polega na wskazywaniu poszczególnym procesorom operacji adresów argumentów i adresów wyników.

- Procesory operacji generują zwrotny sygnał o ewentualnej zajętości poprzednią operacją.
- Tekst programu jest sekwencyjnie analizowany przez nadrzędny egzekutor tekstu ONP (4-argumetowy, bez stosu), wysyłający do poszczególnych procesorów zlecenia usług lub czekający na zakończenie poprzednich, wg poniższego schematu:

```
powtarzaj (do konca tekstu programu) {
        czytaj argumenty(args) i operator (Op) z tekstu
        syncsend(Op, args)
        popatrz na nastepne zdanie proste
}
```

  a rozkazy jmp, jmpB, call i ret dotyczą egzekutora tekstu, który na potrzeby ich realizacji może pracować jako maszyna stosowa.
- W tak zdefiniowanej notacji kolejne zdania proste $Z_p$ są wyliczane, natomiast nie są już argumentami dla następnych operacji. Zatem poszczególne wystąpienia $Z_p$ decydują tylko o kolejności wskazania poszczególnych operacji $O_p$.

## 5. Dane programu

Tekst programu napisanego w języku wysokiego poziomu, to poszczególne operacje przypisań lub operacje sterujące wykonaniem programu, czyli instrukcje języka. W ramach jednej instrukcji wyniki określonej operacji prostej są przekazane do następnej *jednej* operacji, zgodnie z właściwością stosowej realizacji obliczeń wyrażenia ONP (niezoptymalizowanego). Dane przekazujące wartości w ramach jednej instrukcji programu nazwano uprzednio liniami danych. Są one potrzebne na czas wykonania instrukcji.

W skali całego programu jedna instrukcja może obliczać wartość wykorzystywaną przez kilka innych operacji. Dane o tym charakterze nazwano n-liniami, aby zaznaczyć ich n-krotne używanie. Możliwe są dwie sytuacje związane z nimi:

a) Dana zmienia wielokrotnie wartość — Czy dany procesor wykonuje operację na poprawnej wartości, czy jeszcze na starej? Kiedy argument jest gotowy dla konkretnego procesora?

Procesor rozpocznie pracę, kiedy wskaże mu się argumenty — tylko wtedy, kiedy wszyscy producenci argumentów rozpoczęli pracę (lub oczekiwanie na swoje argumenty). Mimo to możliwy jest swego rodzaju hazard: aktywny procesor dokonuje operacji na starych argumentach. Weźmy pod uwagę następujący przykład:

```
a = 1;
et:
a = a+5;
if (a<6) goto et;
```

równoważny

```
a = 1;
do a = a+5;
while (a<6);
```

Na wyjściu zmienna a przyjmie wartość 6 lub 11, zależnie od prędkości wykonania operacji '=', '+' i '<'. Jedyną gwarancją, że sprawdzenie warunku pętli nastąpi na nowo wyliczonej danej, jest przyjęcie, że po wskazaniu operacji '=' egzekutor tekstu przyjmie zapisywaną zmienną jako *zastrzeżoną*. Jeśli któraś z następnych operacji w tekście operuje na zmiennej *zastrzeżonej*, zlecanie obliczeń jest wstrzymane do czasu zakończenia operacji '='. Algorytm zastrzegania będzie więc następujący:

```
if (operacja == '=')
        zastrzezenie = adres_docelowy;
```

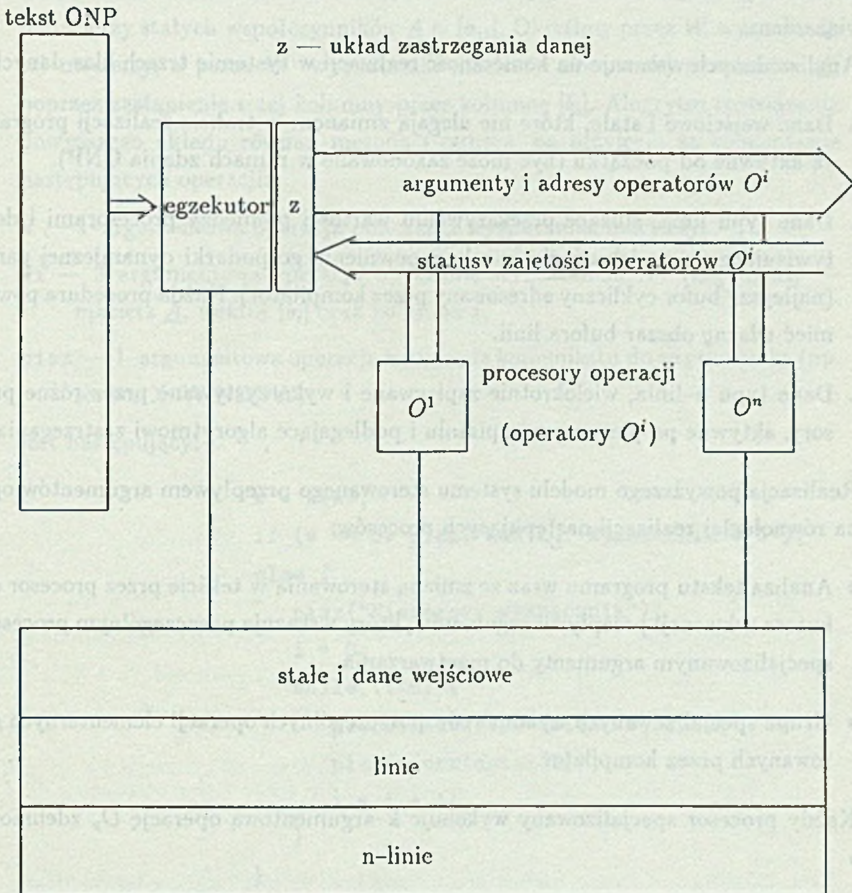
a w trakcie zlecania następnych operacji zastrzeżenie jest weryfikowane poprzez

```
if (argument == zastrzezenie) {
        while ('=' zajety);
        zastrzezenie = NULL;
}
```

Układ zastrzegania wstrzymuje zlecanie operacji przez egzekutor tekstu, izolując go od reszty systemu. Zlecenie operacji '=' następuje wcześniej niż '<', zatem operacja '<' poczeka na zakończenie przypisania. Potencjalne następne przypisanie przy jednym procesorze przypisania w systemie będzie oczekiwało na zakończenie poprzedniej oparacji, a dla $n$ procesorów '=' wymaga się $n$ rejestrów zastrzegających.

b) Dana zostanie odczytana jako argument przez kilka różnych procesorów – Czy użycie danej ma ją uczynić nieważną? Kiedy tak, a kiedy nie?

Używanie danej typu n-linia nie powinno jej deaktywizować, podobnie jak to się dzieje w przypadku wirtualnych zmiennych w języku wysokiego poziomu, natomiast korzystna byłaby deaktywizacja danych jednorazowych typu linia, w celu zapewnienia gospodarki dynamicznej liniami (gospodarka w gestii kompilatora, ale dezaktywizacja w gestii systemu sterowanego przepływem argumentów).




Rys. 2. Schemat przykładowego systemu sterowanego przepływem argumentów opartego na realizacji ONP
Fig. 2. Structure of example argument flow system based on prefix notation execution

## 6. Podsumowanie

Funkcja tekstu programu zapisanego w ONP w celu synchronizacji realizacji programu w systemie sterowanym przepływem argumetów o ograniczonych zasobach polega na wskazywaniu poszczególnym procesorom elementarnych operandów oraz wstrzymywaniu tej procedury, jeśli któryś z procesorów nie ukończył jeszcze poprzedniej operacji. Pozostaje do przyszłego rozstrzygnięcia kwestia, czy oczekiwanie na gotowość danej operacji nie jest warunkiem zbyt silnie ograniczającym moc obliczeniową systemu — być może realizacja *kolejek* zleceń dla procesorów operacji elementarnych będzie efektywnym rozwiązaniem.

Analiza danych wskazuje na konieczność realizacji w systemie trzech klas danych:

1. Dane wejściowe i stałe, które nie ulegają zmianom w trakcie realizacji programu i są aktywne od początku (być może zakodowane w ramach zdania ONP).

2. Dane typu linia, służące przekazywaniu wartości pomiędzy procesorami i dezaktywizujące się po ich użyciu w celu zapewnienia gospodarki dynamicznej pamięci (najlepszy bufor cykliczny adresowany przez kompilator). Każda procedura powinna mieć *własny* obszar bufora linii.

3. Dane typu n-linia, wielokrotnie zapisywane i wykorzystywane przez różne procesory, aktywne po pierwszym zapisaniu i podlegające algorytmowi zastrzegania.

Realizacja powyższego modelu systemu sterowanego przepływem argumentów opiera się na równoległej realizacji następujących procesów:

- Analiza tekstu programu wraz ze zmianą sterowania w tekście przez procesor egzekutora sekwencji przepływu argumetów, który wskazuje poszczególnym procesorom specjalizowanym argumenty do przetwarzania.

- Grupa specjalizowanych wykonawców poszczególnych operacji elementarnych generowanych przez kompilator.

Każdy procesor specjalizowany wykonuje k-argumentową operację $O_p$ zdefiniowaną jako

$$w = O_p(a_1, \ldots, a_k)$$

i zapisaną w ONP jako

$$a_1, \ldots, a_k, w, O_p$$

Egzekutor z kolei jest odbiorcą sygnałów o zajętości procesorów specjalizowanych oraz wyniku wyrażeń logicznych w celu realizacji zmiany sterowania. W przypadku napotkania skoku egzekutor tekstu natychmiast pomija tekst za skokiem. Skok warunkowy jest synchronizowany pojawieniem się argumentu informującego o warunku.

Strukturę tego systemu ilustruje rysunek 2.

**Przykład:** Dany jest układ równań:

$$\sum_{j=1}^{n} a_{ij}x_j = b_i, \qquad i \in [1, n]$$

o macierzy stałych współczynników $\underline{A} = [a_{ij}]$. Określmy przez $W$ wyznacznik tej macierzy, a przez $W^i$ wyznaczniki macierzy utworzonej z macierzy $\underline{A}$, poprzez zastąpienie $i$–tej kolumny przez kolumnę $[b_i]$. Alogrytm rozwiązania powyższego układu równań metodą Cramera, po przyjęciu za elementarne następujących operacji:

`W` — 1-argumentowa operacja obliczania wyznacznika macierzy,

`Wi` — 3-argumentowa operacja obliczania wyznacznika $W^i$ (argumenty — macierz $\underline{A}$, wektor $[b_i]$ oraz kolumna $j$,

`pisz` — 1-argumentowa operacja wysyłania komunikatu do użytkownika (np. serwer X-WINDOW)

jest następujący:

```
w = W(A);
if (w == 0) pisz("KONIEC. Wyznacznik = 0");
else {
    pisz("Niezerowy wyznacznik");
    i = 0;
    while (i<n) {
        pierwiastek[i] = Wi(A, B, i)/w;
        pisz(pierwiastek[i]);
        i = i + 1;
    }
}
```

Czas realizacji algorytmu (zakładając pomijalny udział operacji arytmetycznych) dla typowego procesora von Neumana będzie następujący:

$$t = t(W) + t(pisz) + n * [t(W^i) + t(pisz)]$$

lub gdy wyznacznik $\underline{A}$ równy 0

$$t = t(W) + t(pisz)$$

W przypadku użycia powyższego modelu, zakładając typową sytuację

$$t(W^{\text{'}}) \gg t(\mathtt{pisz})$$

czas wykonania algorytmu wyniesie

$$t = t(W) + t(pisz) + k * t(W^{\text{'}}) + t(pisz)$$

Zlecenia realizacji algorytmu zostaną wstrzymane na instrukcji `if` oraz na zajętości operatorów $W^{\text{'}}$ oraz `pisz`. Operacja `pisz` oczekuje wyniku operacji $W^{\text{'}}$, natomiast po zleceniu operacji `pisz` od razu będzie obliczany następny wyznacznik. W przypadku gdy zasoby te są pojedyncze, $k = n$. Zakładając dużą złożoność obliczania wyznacznika, można postulować zwiększenie liczby operatorów $W^{\text{'}}$ do liczby $m$ oraz dostarczenie tej informacji egzekutorowi, wtedy (przyjmując np. $m$ procesorów operacji dzielenia i przypisania albo dodając operator `pierwiastek` lub redefiniując operator $W^{\text{'}}$ na `pierwiastek'`)

$$t = t(W) + t(pisz) + k * t(W^{\text{'}}) + l * t(pisz)$$

gdzie $k = \begin{cases} 1, & \text{gdy } m \geq n \\ int(\frac{n}{m}), & \text{gdy } m < n \text{, a int() — zaokrąglenie liczby w góre} \end{cases}$

natomiast $1 \leq l \leq n$. Operator `pisz` będzie typowo jeden (zwykle jedna konsola) — może on potencjalnie kolejkować zlecenia, ale czasy ich wystąpienia są zależne od $m$ oraz $t(W^{\text{'}})$. Istnieje również teoretyczna możliwość, że wcześniej zlecone do obliczenia wyznaczniki będą później dostępne, co należy wziąć pod uwagę przy projektowaniu wydruków algorytmu.

*Implementacja danych przykładu*, zależnie od środowiska wykonania danego algorytmu, może być następująca:

Sieć lokalna — W przypadku zastosowania do obliczeń sieci lokalnej należy dane przesyłać do operatorów, a tym samym sama transmisja danych może mieć wpływ na czas realizacji algorytmu. Wydaje się, że należy użyć trybu rozgłaszania, zamiast adresacji poszczególnych klientów sieci (wiele zastawów danych wejściowych algorytmu).

System wielomikroprocesorowy — Implementacja algorytmu wymaga wielobramowego dostępu do pamięci danych lub efektywnych algorytmów arbitrażu dostępu do pamięci. Implementacja pamięci lokalnych operatorów może wpłynąć na czas obliczeń ze względu na konieczność wstępnego przesyłania argumentów do operatorów.

Problemy gospodarki pamięcią systemu (gospodarka pamięcią dynamiczną procedur), gospodarki liniami (argumentami jednorazowymi) oraz n-liniami (rozwiązywanie kolizji dostępu, wielobramowe dostępy) nie zostały w powyższym opracowaniu rozwinięte, a jedynie wskazane jako kwestie potencjalnej implementacji. Sposób ich rozwiązania będzie silnie rzutował na prędkość pracy opisywanego przykładowego systemu.

## LITERATURA

[1] Stefan Węgrzyn, *Systemy sterowane przepływem operacji i systemy sterowane przepływem argumentów*, ZN Pol. Śl., ser. Informatyka, z. 24, Gliwice 1993.

[2] Stefan Węgrzyn, *Podstawy informatyki*, PWN, Warszawa 1982.

[3] Władysław M. Turski, *Propedeutyka informatyki*, PWN, Warszawa 1975.

[4] David Gries, *Konstrukcja translatorów dla maszyn cyfrowych*, PWN, Warszawa 1984.

[5] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, *Compilers. Principles, Techniques, and Tools*, Addison Wessley Publishing Company, March 1986.

Recenzent: dr hab. inż Stanisław Kozielski, prof. Politechniki Śląskiej

Wpłynęło do Redakcji 23 września 1993 r.

## Abstract

The article presented postfix notation as a argument flow operation and a method of code generation by high level language compilers as a postfix notation statement. Next, it was pointed out that prefix notation is a canonical alghoritm form description. The article analized a system with limited resources in the postfix notation context (figure 1). The analysis pointed out the need of elementary menagement of system resources and common data. As a conclusion, the article presented an example argument flow system with limited resources based on postfix notation execution (figure 2). Memory structure, data flow organization and data implementation problems were also mentioned.