ZESZYTY NAUKOWE POLITECHNIKI ŚLĄSKIEJ 1998
Seria: ELEKTRYKA z. 162 Nr kol. 1395

Krzysztof SKRZYPEK
Ryszard BOGACZ
Politechnika Śląska

# ADAPTACJA METOD REKURENCYJNYCH DLA PROGRAMÓW PISANYCH W JĘZYKU ZORIENTOWANYM MASZYNOWO

**Streszczenie.** W pracy zamieszczono opis metody rekurencyjnej możliwej do zastosowania w języku asemblera wybranego mikroprocesora zgodnie z ogólną definicją podaną w pracy [1]. Na przykładzie sześciu procedur przedstawiono wyniki analiz trzech metod, w tym: dwóch rekurencyjnych i iteracyjnej. Wskazano wady i zalety omawianych metod, a także zalecenia ograniczające zakres ich stosowania. Ponieważ jako przykładowy mikroprocesor został wybrany mikrokontroler serii Intel MCS51, zamieszczone procedury są napisane z uwzględnieniem specyfiki procesorów tej serii.

# ADAPTATION OF RECURSIVE METHODS TO LOW LEVEL PROGRAMMING LANGUAGES

**Summary.** Description of the recursive method which can be applied to assembler of the chosen microprocessor according to the general definition given in the work [1] is presented in the paper. Analysis results of three methods: two of them recursive ones and an iteration one are given basing on the example of six procedures. Advantages and disadvantages of the discussed methods as well as limitations of their use are shown. As an example the Intel MCS51 family processor has been chosen, so the presented procedures have been written taking into account this family specific qualities.

## 1. WSTĘP

Procedura rekurencyjna to taka procedura, która podczas działania wykorzystuje jako podprogram (tj. wywołanie w formie skoku ze śladem powrotu) część samej siebie. Według prof. N. Wirtha ogólniej „obiekt zwany jest rekurencyjnym, jeżeli częściowo składa się z siebie samego lub jego definicja odwołuje się do niego samego" [1]. Definicja w takiej formie będzie dalej stosowana jako przydatna do określenia rekurencji w programach pisanych w językach zorientowanych maszynowo, takich jak np. asemblery.

Zastosowanie rekurencji w programach daje, przy zachowaniu pewnych ogólnych zasad postępowania i oceny efektywności metody, zwykle dobre lub bardzo dobre efekty. Sama

rekurencja jako metoda alternatywna dla iteracji tworząca algorytm procedury nie powinna jednak być nadużywana. Są sytuacje, w których daje ona lepsze efekty, ale są też takie, w których jest tak samo dobra lub nawet gorsza od rozwiązania iteracyjnego. Prawidłowy dobór metody powinien polegać na ocenie według kilku kryteriów. Najważniejszymi kryteriami porównawczymi w dalszej części będą zajętość pamięci programu i czas jego wykonania.

W językach wyższego poziomu rekurencja jest uznawana za narzędzie o bardzo dużej skuteczności w wielu trudnych i prostszych problemach, jednak w językach zorientowanych maszynowo (j. niższego poziomu) występują pewne ograniczenia, o których nie ma mowy w językach zorientowanych problemowo (j. wyższego poziomu). Będzie tu rozpatrywany przypadek szczególny zastosowań idei rekurencji w asemblerze, w systemach opartych na mikrokontrolerach na przykładzie serii Intel MCS51. Ma on własną specyfikę polegającą głównie na kilku cechach:

- obszar stosu programowego znajduje się w wewnętrznej pamięci (on-chip memory),
- wskaźnik stosu jest zwykle 8-bitowy, co daje całkowity i nieprzekraczalny rozmiar stosu równy 256 bajtów,
- po impulsie resetującym mikrokontroler wskaźnik stosu jest ustawiany na określoną pozycję (np. $07, czyli wskazuje ósmy bajt stosu), co pomniejsza jeszcze efektywny rozmiar stosu,
- brak mechanizmu wywołania procedury z parametrami formalnymi (jest to cecha większości, lecz nie wszystkich, języków zorientowanych maszynowo),
- operacje na stosie są ograniczone do podstawowych, takich jak zrzucenie słowa statusu, czy pojedynczych rejestrów (i późniejsze zdjęcie).

Używane tu pojęcie stosu dotyczy wydzielonej części pamięci maszyny o organizacji LIFO, używanej głównie do przechowywania adresu powrotnego z wołanego podprogramu. Taka organizacja docelowej maszyny realizującej algorytm nie uniemożliwia stosowania rozwiązań rekurencyjnych, jednak wprowadza konieczność dodatkowego sprawdzenia, czy zapotrzebowanie na stos pojedynczych procedur oraz całego programu nie przekracza możliwości maszyny. W językach wyższego poziomu podwyższenie efektywności działania algorytmu oraz zwykle zwiększenie jego czytelności najczęściej wystarcza jako bodziec do korzystania z metod rekurencyjnych.

W dalszej części tekstu terminy "procedura" i "podprogram" będą używane zamiennie, ponieważ nie ma między nimi formalnych różnic w zastosowaniu do języka zorientowanego maszynowo.

## 2. OGÓLNA CHARAKTERYSTYKA PROBLEMU

Bardzo często stosowane w językach wyższego poziomu techniki rekurencyjne wykorzystują parametr formalny lub zmienną globalną w warunku opuszczenia rekurencji w postaci procedury

zagnieżdżonej w samej sobie. Takie techniki w omawianych tu sytuacjach zbyt często mogą prowadzić do przepełnienia stosu – ze względu na jego niewielką pojemność. Zawsze prowadzą do komplikacji algorytmu w języku bez mechanizmu przekazywania parametrów formalnych w procedurze. Należy zatem nie wykorzystywać takich technik, lub ewentualnie robić to niezmiernie ostrożnie.

O wiele bardziej przydatne są tu techniki wykorzystujące podaną wcześniej definicję rekurencji w sposób nieco inny, jednak również bardzo dosłowny. Podział zadania, które ma być realizowane przez projektowaną procedurę na zadania cząstkowe pozwoli zdefiniować procedurę w postaci zaetykietowanej sekwencji rozkazów z jednym lub więcej punktami wskoku, tj. etykietami zaznaczającymi te części procedury, które można wykorzystać do realizacji jej zadań cząstkowych. Te dodatkowe etykiety pozwolą na taką definicję procedury, w której będą występowały wywołania odpowiednich części (przez punkty wskoku) oraz ewentualnie pojedyncze rozkazy ustawiające rejestry lub komórki potrzebne w zadaniach cząstkowych. Wykorzystanie stosu w takiej metodzie jest dużo lepsze niż w poprzedniej, a ponadto zawsze z góry można dokładnie określić, ile wpisów na stos nastąpi w ciągu realizacji procedury. Jednocześnie należy zauważyć, że stan stosu nie zwiększa się lawinowo w trakcie działania procedury, przeciwnie – powrót z realizacji zadania cząstkowego powoduje uwolnienie stosu od śladu powrotu z tego zadania, dzięki czemu stos nie rośnie o wartość większą niż jeden lub kilka wpisów adresów powrotnych. Poniższy przykład ilustruje tę sytuację:

```
proc: mov   R0,   #3
ptl:  mov   A,    R0
      movc  A,    @A+DPTR
      rl    A
      acall do_smth
      djnz  R0,   ptl
      ret
```

Procedura 1. Implementacja algorytmu w postaci iteracyjnej

```
proc: mov   R0,   #0
jsr:  push  R0
      inc   R0
      mov   A,    R0
      xrl   A,    #3; A=3?
      jz    rts
      acall jsr
      pop   R0
rts:  mov   A,    R0
      movc  A,    @A+DPTR
      rl    A
do_smth:          ; tekst procedury do_smth
      [...]
      ret
```

Procedura 2. Implementacja w postaci klasycznej rekurencji

```
proc: mov   R0,   #3
      acall do
      acall do
do:   mov   A,    R0
      dec   R0
      movc  A,    @A+DPTR
      rl    A
do_smth:          ; tekst procedury do_smth
      [...]
      ret
```

Procedura 3. Implementacja w postaci proponowanej rekurencji

Przykładowe procedury realizują tę samą funkcję, tj. odczytują kolejne komórki tablicy zawartej w pamięci programu mikrokontrolera (serii MCS51) i po pomnożeniu przez dwa wywołują procedurę 'do_smth', która dokonuje dalszej obróbki (jej działanie nie ma znaczenia dla rozważań, wystarczy tylko założenie, że nie powoduje zmian w rejestrze R0). Widać, że pod względem długości kodu wszystkie trzy procedury są podobne. Rozwiązanie iteracyjne wydaje się najprostsze, zajętość pamięci programu jest najmniejsza (zestawienie w tabeli 1), jednak jeśli chodzi o szybkość – przegrywa zdecydowanie: 8 cykli maszynowych więcej niż jej odpowiednik rekurencyjny zawarty w procedurze 3. Na uwagę zasługuje fakt, że możliwe jest umieszczenie dodatkowej procedury (tutaj jest to 'do_smth') jako części składowej omawianej procedury ('proc') bez ujemnego efektu uniemożliwienia korzystania z niej przez resztę programu głównego – pozostaje ona podprogramem z początkiem w punkcie 'do_smth' i kończącym się rozkazem 'ret', mimo że wewnątrz procedury 'proc' nie ma jawnego odwołania do etykiety 'do_smth' przez rozkaz 'acall do_smth' lub 'lcall do_smth'.

Tabela 1

Zestawienie parametrów porównawczych dla przykładowych procedur 1÷3

| | Zajętość pamięci programu w bajtach | Zajętość czasu maszyny w cyklach masz. | Zapotrzebowanie na stos w bajtach |
|---|---|---|---|
| Procedura 1 | 10 | 28 | 4 |
| Procedura 2 | 17 | 41 | 11 |
| Procedura 3 | 11 | 20 | 4 |

Wartości podane w tabeli nie uwzględniają oczywiście ani czasu wykonania, ani rozmiaru procedury 'do_smth'. Wartości w kolumnie „Zapotrzebowanie na stos" mają charakter kontrolny, ich porównanie pokazuje, że zajętość stosu dla porównywanych metod jest podobna, a dla metody klasycznej rekurencji wyraźnie się powiększa.

Jak widać, rekurencja w klasycznym wydaniu przegrywa porównanie z procedurą iteracyjną, tak samo jak z rekurencją proponowaną w procedurze 3. Główne trudności w jej stosowaniu to konieczność zachowywania wartości parametrów razem z wywoływaniami właściwej części procedury na stosie (w tym prostym przypadku był to tylko jeden rejestr roboczy R0). Jej działanie polega na odłożeniu na stosie wywołań podprogramu realizującego wymaganą czynność wraz z ewentualnymi wymaganymi przez ten podprogram parametrami. Końcowe wykonanie rozkazu 'ret' rozpoczyna dopiero uwalnianie stosu ściągając szereg rozkazów 'acall' oraz odpowiednich wartości dla rejestru R0. Takie kłopoty oczywiście nie występują w językach wyższego poziomu, z uwagi na to, że programy realizowane są w środowisku maszyn wirtualnych budowanych przez odpowiednią implementację danego języka. Dzięki temu programista nie musi się martwić np. o fizyczny rozmiar stosu czy też o sposób przekazywania parametrów formalnych, ponieważ jest to realizowane programowo przez odpowiedni blok kompilatora języka. Wyższość ta jednak znika w sytuacji mikrokomputerów o wcześniej omówionej strukturze, tzn. zbudowanych na bazie mikrokontrolera o niewielkiej (i najczęściej w postaci trwałej – ROM) pamięci programu, oraz – bardzo często – krytycznie małej ilości czasu przeznaczonej na wykonanie odpowiednich operacji.

Procedura 4 jest rozwiązaniem zadania procedur 1-3 w języku wyższego poziomu. Jest ona odpowiednikiem procedury 2, czyli przedstawia klasyczną rekurencję. Jej zwartość jest tak duża dzięki możliwości przekazania parametru, która jest jedną z charakterystycznych cech tej klasy języków. Dzieje się to kosztem zwiększonego zapotrzebowania na stos, co jednak dla maszyn wykonujących programy pisane w tych językach nie jest nadmiernym obciążeniem.

```
procedure proc (R0:integer);
 begin
      if R0<3 then proc (R0+1);
      do_smth (R0*2)
 end;
```

Procedura 4. Rozwiązanie zadania procedur 1÷3 w języku wyższego poziomu

Widać w porównaniu do procedury 2, jak mocno wpłynęła specyfika maszyny docelowej na postać programu, jego czytelność i efektywność.

## 3. PORÓWNANIE PROPONOWANEJ METODY REKURENCYJNEJ Z ITERACYJNĄ

### 3.1. Zestawienie procedur realizujących obydwie metody

```
; procedura testujaca porty I/O - przygotowanie tabeli wynikow testow
test_re:    mov   R6,   #tab_O-1
            mov   P1,   #4
            acall test_r
            mov   P1,   #0
            acall test_r
            mov   20h,  R4
            mov   P1,   #8
            acall test_r
            mov   P1,   #0Ch
test_r:     inc   R6; aktualizacja - ostatni raz: R6=tab_O.O4 (U6)
            mov   R7,   #tab_I
            acall test_we
            acall test_we
            acall test_we
test_we:    acall test; procedura testu, wykorzystuje R5, R6, R7,
                      ; zostawia status testu w bicie C (carry)
b_wej:      mov   A,    R4; rejestr statusow testow
            rlc   A; dolaczenie kolejnego statusu
            mov   R4,   A
            inc   R7; nastepny rej. wej.
            mov   21h,  R4; jesli ost. przejscie - drugie 2 wiersze tabeli
            ret
```

Procedura 5. Przykładowe zastosowanie proponowanego rozwiązania rekurencyjnego w procedurze realizującej zadanie dające się podzielić na więcej zadań cząstkowych

Procedura ma następujące cechy:

- przeprowadza 16 testów cząstkowych dla układów peryferyjnych (dostęp przez port P1),
- testy cząstkowe są wykonywane w 4 grupach: 4 testy dla każdego z 4 układów,
- dostęp do poszczególnych układów wymaga ustawienia odpowiedniego adresu dostępu wpisywanego do portu P1; adresy te nie są kolejnymi liczbami – nie jest więc możliwe zastosowanie stanu licznika bezpośrednio jako adresu,
- idea algorytmu nie wymaga żadnego licznika,
- wyniki testu są zapisywane w tabeli 16-bitowej (2 bajty pamięci) przez dopisywanie kolejnych bitów po wykonaniu testu cząstkowego (podprogram 'test').

Zastosowanie rozwiązania wyłącznie iteracyjnego spowodowałoby konieczność znacznego rozbudowania w sensie objętości, jak również dołączenia dodatkowej tabeli zawierającej odpowiednie stałe oraz użycia dodatkowych liczników w postaci rejestrów wewnętrznych

procesora. Ponadto musiałaby się ona rozczłonkować na kilka „mniejszych" podprogramów realizujących zadania cząstkowe.

Niezdefiniowany w przykładzie podprogram 'test' nie jest istotny dla tematu rozważań, tak jak zawartość tabel 'tab_O' i 'tab_I'. Łatwo zauważyć, że głównym czynnikiem zwiększającym czas wykonania procedury 1 w stosunku do procedury 3 jest kombinacja dwóch rozkazów 'acall do_smth' i 'djnz r0, ptl'. W algorytmie procedury 5 zapisanym w postaci iteracyjnej (p. procedura 6) liczba rozkazów tworzących pętle, takich jak 'djnz', musiałaby być o wiele większa. Jak widać, w procedurze 6 konieczne stało się dodatkowe utworzenie dwóch tabel pomocniczych: jednej ze stałymi dla portu P1, drugiej z adresami tabeli zawierającej wyniki działania procedury. Jednocześnie dla obsługi iteracji i tych tabel konieczne było założenie dodatkowych trzech liczników oraz jednego rejestru adresującego (R3) w rejestrach ogólnego przeznaczenia (R0, R1, R2, R3). Aby uzyskać dostęp do poszczególnych podzadań tej procedury ('test_r', 'test_we', 'p_wej', 'b_wej'), co w programie, z którego pochodzi procedura 5, jest potrzebne, należałoby rozbić tekst procedury 6 na kilka części, co dodatkowo zwiększyłoby liczbę wywołań (rozkazów typu 'acall') i jeszcze bardziej wydłużyło czas działania tej procedury.

```
test_re:    mov   R6,   #tab_O-1
            mov   R0,   #4
            mov   R1,   #4
            mov   R2,   #3
ptl:        mov   A,    R0
            mov   DPTR, #tabc1-1
            movc  A,    @A+DPTR
            mov   P1,   A
test_r:     inc   R6
            mov   R7,   #tab_I
test_we:    acall test
b_wej:      mov   A,    R4
            rlc   A
            mov   R4,   A
            inc   R7
            djnz  R1,   test_r
            mov   A,    R2
            mov   DPTR, #tabc2
            movc  A,    @A+DPTR
            mov   R3,   A
            mov   A,    R4
            mov   @R3,  A
            dec   R2
            djnz  R0,   ptl
            ret
tabc1:      db    0Ch, 08h, 00h, 04h
tabc2:      db    21h, 21h, 20h, 20h
```

Procedura 6. Implementacja iteracyjna zadań procedury 5

Tabela 2

Zestawienie parametrów porównawczych dla procedur 5 i 6

| | Zajętość pamięci programu w bajtach | Zajętość czasu maszyny w cyklach masz. | Zapotrzebowanie na stos w bajtach |
|---|---|---|---|
| Procedura 5 | 38 | 179 | 10 |
| Procedura 6 | 44 | 204 | 6 |

W zestawieniu w tabeli 2 widać już wyraźnie wyższość rozwiązania w postaci proponowanej rekurencji nad rozwiązaniem iteracyjnym. Procedura iteracyjna zajmuje o 25 cykli maszynowych (przy pojedynczym wywołaniu procedury) oraz o 6 bajtów pamięci programu więcej. Podczas działania jej rekurencyjnego odpowiednika stos narasta tylko o 4 bajty więcej, czyli zajętość stosu także jest bardzo podobna. Po przeformułowaniu procedury 6 (rozwiązanie iteracyjne) tak, aby odpowiadała w pełni funkcjonalnie procedurze 5, obciążenie stosu, jak również zajętość czasu i pamięci programu zwiększy się jeszcze bardziej na niekorzyść procedury 6.

Jak widać, w sytuacji gdy zadanie realizowane przez algorytm z założenia nie wymaga użycia licznika, wtedy bardzo często rozwiązanie iteracyjne okaże się dużo mniej efektywne od rozwiązania rekurencyjnego w opisywanej postaci.

### 3.2. Ograniczenia proponowanej metody rekurencyjnej

- Krotność wykonań zadania cząstkowego; każde wykonanie zadania cząstkowego przez procedurę rekurencyjną wiąże się z operacją na stosie (rozkaz typu 'acall') – im większa liczba wywołań, tym więcej czasu jest potrzebne na dodatkowe operacje na stosie (tu: odłożenie śladu powrotu); algorytm iteracyjny wymaga tylko kontroli licznika i dokonania odpowiedniej ilości skoków zamykających pętlę, w związku z tym dla krotności większych może się okazać, że rozwiązanie iteracyjne jest szybsze, jednak porównanie takie jest zależne od zadań stawianych algorytmowi – można je ocenić tylko indywidualnie, lub oszacować ilości odpowiednich rozkazów charakterystycznych dla metody: w przypadku iteracji – 'djnz', dla rekurencji – 'acall'.
- Organizacja programu głównego; zastosowanie metod rekurencyjnych wymaga podziału programu na mniejsze podzadania wywoływane jako podprogramy; rozwiązania iteracyjne nie wprowadzają takiej konieczności – program może być pojedynczą sekwencją bez rozgałęzień (nie licząc pętli).
- Procedura 7 przedstawia sytuację skrajną, w której prawdopodobnie najlepszym (poza wykorzystaniem układu czasowo-licznikowego) rozwiązaniem jest iteracja; próby sformułowania jej w innej postaci niż ta nie mogą doprowadzić do właściwych efektów.

```
delay:      mov   R0,   #100
ptl:        djnz  R1,   ptl
            djnz  R1,   ptl
            djnz  R0,   ptl
            ret
```

Procedura 7. Przykład typowej iteracji - pętla opóźniająca

## 4. PODSUMOWANIE

Zamieszczone przykłady pokazują, że mimo pewnych ograniczeń, także w językach niskiego poziomu można i należy stosować nowoczesne koncepcje programistyczne. Przedstawiona metoda projektowania algorytmów rekurencyjnych jest przydatna w sytuacjach, gdy zachodzi konieczność wykonania powtarzalnej operacji. Jeśli krotność powtórzenia jest z góry określona i niezbyt wielka, zastosowanie przedstawionej metody rekurencyjnej najczęściej pozwoli zmniejszyć objętość programu, a także skrócić czas wykonania jego krytycznych fragmentów. Wprawdzie zapotrzebowanie na stos nie może być zwykle przedmiotem konkurencji proponowanego podejścia z podejściem iteracyjnym, jednak widać wyraźnie, że zastosowanie metody rekurencyjnej zgodnie z zamieszczonymi zaleceniami nie musi prowadzić do nadmiernego zapełnienia stosu. Głównymi kryteriami porównawczymi powinny tu być zajętości: czasu i pamięci programu.

Konieczność wcześniejszego rozbicia projektu programu, lub pojedynczej procedury, na cząstkowe zadania nie jest wymaganiem szczególnie kłopotliwym, tym bardziej że z reguły jest to jeden z pierwszych kroków projektowania. Jeśli jest możliwe rozłożenie programu na dostatecznie małe podzadania – redukcja objętości może dotyczyć nie tylko pojedynczych procedur, ale całego programu wynikowego.

Zamieszczone procedury przykładowe są właściwe dla mikrokomputera zbudowanego na bazie mikrokontrolera serii Intel MCS51. Jego specyfika (charakterystyczna lista rozkazów) wymusza pewne rozwiązania, które w przypadku innych mikroprocesorów byłyby nieco prostsze; widać to np. w strukturze procedury 2, jak i w końcowej sekwencji procedury 6. Jednak ogólna metoda postępowania jest możliwa do zastosowania w przypadku każdego mikroprocesora, a ogólne wyniki osiągane przez zestawione wyżej metody są powtarzalne, tzn. klasyczne podejście rekurencyjne będzie trudne lub niemożliwe do użycia, a jego efekty marne. Rozwiązanie iteracyjne w wielu sytuacjach będzie gorsze od rozwiązań rekurencyjnych w proponowanej postaci. Widać także, że ograniczona wielkość stosu nie wprowadza żadnych dodatkowych komplikacji, ponieważ obciążenie stosu jest porównywalne, choć zwykle nieco większe niż w przypadku zamienników iteracyjnych.

LITERATURA

1. Wirth N.: Algorytmy + struktury danych = programy. WNT, Warszawa 1989.
2. Jakubiec J.: Wprowadzenie do techniki mikroprocesorowej. Wyd. Pol. Śl., Gliwice 1994.
3. Waite M. W., Goos G.: Konstrukcja kompilatorów. WNT, Warszawa 1989.
4. Tanenbaum A. S.: Organizacja maszyn cyfrowych w ujęciu strukturalnym. WNT, Warszawa 1980.

Recenzent: Prof. dr hab. inż. Michał Szyper

Wpłynęło do Redakcji dnia 15 kwietnia 1998 r.

**Abstract**

A modified concept of recursive programming method is presented in the paper. It is designed to apply to low level programming languages, such like assemblers. As an example, the Intel MCS51 family processor has been used. Analysis of often applied methods of implementation of iterating programme fragments is given. Two parameters have been used as criteria, namely: program memory utilisation and procedure execution time. As an additional and control parameter, a stack utilisation need is presented for all the methods.

In the procedures 1, 2, 3 three various methods are shown (in proper order): iteration, classic recursion and proposition of modified recursion. In Table 1 the comparison of three procedures is given considering the criterion parameters. As it could be expected – the classic recursion has unacceptable characteristics in the described applications i.e. assembly languages.

Procedures 5 and 6 are solutions of the more complicated problem in two ways: iteration (proc. 6) and proposed modified recursion (proc. 5). Table 2 shows usability of the proposed concept of recursion.

There is also presented a set of simple restrictions and preferences of using the presented method. It is possible to obtain the following features when complying with these limitations:

- decreased needs for program-memory,
- shorten of the execution time (ET),
- no excessive needs for program-stack.