**PRZETWORNIK DCG**

**C - ASM**

**DOKUMENTACJA**

Opracował: Damian Lewandowski

1. **Wstęp:**

Do projektowania systemów informatycznych bardzo potrzebnym a czasami wręcz niezbędnym może okazać się narzędzie jakim jest Kompilator.

Program ten potrafi ułatwić napisanie przez programistę programów wykonywanych przez urządzenie komputerowe. Dostępnych jest wiele języków wysokiego poziomu które są łatwiejsze w opanowaniu niż język maszynowy który jest „łatwy” dla maszyny – w tym wypadku dla komputera. Od strony technicznej kompilator jest programem komputerowym przetwarzającym wejście napisane w konkretnym języku do wyjścia które jest napisane w innym języku niż był to język wejściowy, ale interpretacja kodu wyjściowego względem wejściowego jest taka sama.

Do napisania kompilatora przez programistę potrzebna jest wiedza w dziedzinie języków formalnych, a konkretniej gramatyk bezkontekstowych.

Powszechnie stosowane języki programowania imperatywnego zazwyczaj nie posiadają odpowiednich narzędzi ani możliwości aby można było stworzyć w sposób łatwy taki przetwornik. Z pomocą przychodzą języki deklaratywne a konkretniej język programowania Prolog. Środowisko programistyczne SWI-Prolog służące do programowania w Prologu posiada narzędzie do tworzenia tzw. Gramatyki DCG.

Gramatyki DCG umożliwiają w sposób łatwy napisanie przez programistę kompilatora przy wykorzystaniu wiedzy jaką on posiada. W tym wypadku został stworzony program przetwarzający wejście w postaci pliku zawierającego kod napisany w języku programowania C, do kodu niższego poziomu jakim jest język programowania Asembler. Program kompilatora jest zbiorem reguł przetwarzania wejścia oraz instrukcji wyjścia. Obszerność programu w porównaniu z alternatywnym programem napisanym języku programowania imperatywnego sprawia że języki deklaratywne są dedykowane wręcz do takich programów.

1. **Przyjęte założenia:**

Przy tworzeniu programu przyjęliśmy sobie następujące założenia:

- Program działający tylko w środowisku programistycznym Prolog.

- odczyt pliku wejściowego przez ten program,

- wyświetlanie na ekranie wyjścia programu,

Praca jest pracą grupową, Otrzymane zostały polecenia napisania reguł dla pętli wykonywanych w programie oraz dokonania poprawek generowania kodu wyjściowego.

1. **Składnia języka wyjściowego:**

Językiem programowania tekstu wyjściowego jest język programowania Asembler. Składnia tego język jest zgodna z asemblerem NASM. Ponadto w języku tym obsługiwane będą rejestry 32-bitowe.

1. **Zbiór obowiązków do wykonania:**

* Pętla „for”,
* Petla „do - while”,
* Pętla „while”.
* Obsługa zagnieżdżania pętli w instrukcjach

1. **Ogólna zasada działania programu:**



Zasada działania programu od strony użytkowej jest bardzo prosta Na ilustracji powyżej przedstawiono wejście i wyjście programu które obydwa można interpretować tak samo.

1. **Zasada działania pętli „while”**

Jedną z pętli który można używać podczas programowania jest pętla „while”. Przykładowe instrukcje oraz wyjście do asemblera wygląda następująco:

while(x < 15)

{

x = x + 1;

printf(„HelloWorld”);

}

Zasada działania instrukcje polega na sprawdzeniu w pierwszej kolejności czy warunek znajdujący się w nawiasach jest spełniony. Jeżeli tak to wykonywana jest instrukcja znajdująca się w nawiasach klamrowych po czym następuje powrót do miejsca sprawdzania warunków.

Kod Asemblera równoważny z powyższym kodem wyglądać powinien następująco:

loop:

mov EAX, [x]

cmp EAX, 15

jg endloop

mov EAX [x]

inc EAX

mov [X], EAX

mov AH, 09

mov DX, Hello

INT 21h

jmp loop

endloop:

Pierwsze trzy linijki kodu poniżej etykiety loop odpowiadają sprawdzeniu warunków.

Jeżeli warunki są prawdziwe wykonywane są instrukcje poniżej tego warunku. Iinstrukcje te są kończone skokiem do ponownego sprawdzenia warunków,

W przeciwnym wypadku następuje skok poza obręb tego kodu (w tym wypadku do etykiety endloop).

Gramatyki obsługi dla pętli „while” wyglądają następująco:

|  |
| --- |
| while(Z) --> "while", whitespace, "(",whitespace, chars(A), whitespace, cond\_op\_equal\_to,whitespace, integer\_number(B), whitespace, ")", whitespace, "{", whitespace, loop\_exp(Za), whitespace, "}", {concat\_atom(['\nwhloop:\nmov eax, [', A, ']\ncmp eax, ', B, '\njne whendloop\n', Za, '\njmp whloop\nwhendloop:'], Z)}.  while(Z) --> "while", whitespace, "(",whitespace, chars(A), whitespace, cond\_op\_not\_equal\_to,whitespace, integer\_number(B), whitespace, ")", whitespace, "{", whitespace, loop\_exp(Za), whitespace, "}", {concat\_atom(['\nwhloop:\nmov eax, [', A, ']\ncmp eax, ', B, '\nje whendloop\n', Za, '\njmp whloop\nwhendloop:'], Z)}.  while(Z) --> "while", whitespace, "(",whitespace, chars(A), whitespace, cond\_op\_less,whitespace, integer\_number(B), whitespace, ")", whitespace, "{", whitespace, loop\_exp(Za), whitespace, "}", {concat\_atom(['\nwhloop:\nmov eax, [', A, ']\ncmp eax, ', B, '\njnl whendloop\n', Za, '\njmp whloop\nwhendloop:'], Z)}.  while(Z) --> "while", whitespace, "(",whitespace, chars(A), whitespace, cond\_op\_greater,whitespace, integer\_number(B), whitespace, ")", whitespace, "{", whitespace, loop\_exp(Za), whitespace, "}", {concat\_atom(['\nwhloop:\nmov eax, [', A, ']\ncmp eax, ', B, '\njng whendloop\n', Za, '\njmp whloop\nwhendloop:'], Z)}.  while(Z) --> "while", whitespace, "(", whitespace, "true", whitespace, ")", whitespace, "{", whitespace, loop\_exp(Za), whitespace, "}", {concat\_atom(['\nwh:', Za, '\n jmp wh'], Z)}. |

Przedstawione są 4 reguły dla pętli „while”:

* Dla przypadku kiedy warunek zawiera operator równości
* Dla przypadku kiedy warunek zawiera operator różności
* Dla przypadku kiedy warunek zawiera operator mniejszości
* Dla przypadku kiedy warunek zawiera operator większości
* Dla przypadku wykonywania pętli nieskończonej.

**7.Zasada działania pętli „do - while”:**

Kolejną pętla jaką można używać w programowaniu jest pętla „do – while”. Przykładowe instrukcje oraz wyjście do asemblera wygląda następująco:

Do

{

x = x + 1;

printf(„HelloWorld”);

}

while(x < 15);

Zasadniczą różnicą między działaniem pętli „while” a pętli „do-while” jest to że w powyższym kodzie w pierwszej kolejności nie są sprawdzane warunki lecz wykonywany jest kod. Po wykonaniu tego kodu w nawiasach klamrowych dokonywane jest badanie warunków.

Kod Asemblera równoważny z powyższym kodem wyglądać powinien następująco:

etykieta:

mov EAX, [X]

inc EAX

mov [X], EAX

mov AH, 09

mov DX, Hello

int 21h

mov EAX, [X]

cmp EAX, 15

jnl etykieta

Na podstawie powyższego kodu widać jest że najpierw wykonywana jest inkrementacja zmiennej znajdującej się w rejestrze EAX, a następnie wykonywana jest instrukcja wyjścia, Po tych instrukcjach następuje sprawdzenie warunków.

Gramatyki obsługi dla pętli „do - while” wyglądają następująco:

|  |
| --- |
| dowhile(Z) --> "do", whitespace, "{", whitespace, loop\_exp(Za), whitespace, "}", whitespace, "while", whitespace, "(", whitespace, dwcond(Zb), whitespace, ")", whitespace, ";", {concat\_atom(['\ndowhloop:\n', Za, Zb], Z)}.  dwcond(Z) --> chars(A), whitespace, cond\_op\_equal\_to, whitespace, integer\_number(B), {concat\_atom(['mov eax, [', A, ']\ncmp eax, ', B, '\n je dowhloop'], Z)}.  dwcond(Z) --> chars(A), whitespace, cond\_op\_not\_equal\_to, whitespace, integer\_number(B), {concat\_atom(['mov eax, [', A, ']\ncmp eax, ', B, '\n jne dowhloop'], Z)}.  dwcond(Z) --> chars(A), whitespace, cond\_op\_less, whitespace, integer\_number(B), {concat\_atom(['mov eax, [', A, ']\ncmp eax, ', B, '\n jl dowhloop'], Z)}.  dwcond(Z) --> chars(A), whitespace, cond\_op\_greater, whitespace, integer\_number(B), {concat\_atom(['mov eax, [', A, ']\ncmp eax, ', B, '\n jg dowhloop'], Z)}. |

Przedstawione są cztery reguły które są analogiczne jak w pętli „while”, natomiast nie ma w niej reguły dla warunków pętli nieskończonej.

1. **Zasada działania pętli „for”**

W porównaniu z poprzednimi pętlami, pętla ta jest praktycznie najchętniej używana przez programistów w programach wykonywanych iteracyjnie.

Pętla ta zawiera w sobie niezbędne warunki jej wykonania, co sprawia że nie trzeba dopisywać do instrukcji znajdujących się w nawiasach klamrowych rozkazów inkrementacji zmiennej będącej licznikiem.

Ogólna zasada działania pętli „for” jest następująca:

for(wyrażenie1 ; wyrażenie2 ; wyrażenie3)

{

//wykonywany kod

}

* Wyrażenie1 – miejsce inicjalizacji zmiennej będącej licznikiem
* Wyrażenie2 – określa warunek końcowy
* Wyrażenie3 – określa instrukcję wykonywaną po wykonaniu kodu znajdującego się między klamrami.

Przykładowy kod wygląda następująco:

for(i = 0 ; i < 15 ; i = i + 1)

{

printf(„HelloWorld”);

}

W porównaniu z poprzednimi instrukcjami kod wygląda bardziej przejrzyście. W Asemblerze kod ten wygląda następująco:

mov eax, [x]

mov eax, 0

mov [x], eax

loop:

mov AH, 9

mov DX, Hello

int 21h

mov EAX, [X]

inc EAX

mov [X], EAX

mov EAX, [X]

cmp EAX, 15

jl loop

Można zauważyć że powyżej etykiety loop znajduje się instrukcja przypisania do zmiennej x wartości zero.

Gramatyki dla tej pętli wyglądają następująco:

|  |
| --- |
| for(Z) --> "for", whitespace, "(", whitespace, for1cond(Za), whitespace, ";", whitespace, for2cond(Zb), whitespace, ";", whitespace, for3cond(Zc), whitespace, ")", whitespace, "{", whitespace, loop\_exp(Zd), whitespace, "}", {concat\_atom([Za, '\nfor:\n', Zd, Zc, Zb], Z)}.  for1cond(Z) --> chars(A), whitespace, "=", whitespace, integer\_number(B), {concat\_atom(['\nmov eax, [', A, ']\nmov eax, ', B, '\nmov [i], eax'], Z)}.  for2cond(Z) --> chars(A), whitespace, cond\_op\_equal\_to, whitespace, integer\_number(Y), {concat\_atom(['\nmov eax, [', A, ']\ncmp eax,', Y, '\nje for'], Z)}.  for2cond(Z) --> chars(A), whitespace, cond\_op\_less, whitespace, integer\_number(Y), {concat\_atom(['\nmov eax, [', A, ']\ncmp eax,', Y, '\njl for'], Z)}.  for2cond(Z) --> chars(A), whitespace, cond\_op\_greater, whitespace, integer\_number(Y), {concat\_atom(['\nmov eax, [', A, ']\ncmp eax,', Y, '\njg for'], Z)}.  for3cond(Z) --> chars(A), whitespace, "=", whitespace, chars(A), whitespace, "+", whitespace, integer\_number(C), {concat\_atom(['\nmov eax, [', A, ']\nadd eax, ', C, '\nmov [', A, '], eax'], Z)}.  loop\_exp(Z) --> exp(Za), whitespace, exp(Zb), {concat\_atom([Za, Zb], Z)}.  loop\_exp(Z) --> exp(Za), {concat\_atom([Za], Z)}. |

1. **Obsługa zagnieżdżenia instrukcji:**

Aby możliwe było zagnieżdżenie danej pętli w drugiej pętli należało stworzyć jeszcze dodatkową gramatykę:

|  |
| --- |
| exp(Z) --> while(Za), exp(Zb), {concat\_atom([Za, Zb], Z)}.  exp(Z) --> while(Za), {concat\_atom([Za], Z)}.  exp(Z) --> dowhile(Za), exp(Zb), {concat\_atom([Za, Zb], Z)}.  exp(Z) --> dowhile(Za), {concat\_atom([Za], Z)}.  exp(Z) --> for(Za), exp(Zb), {concat\_atom([Za, Zb], Z)}.  exp(Z) --> for(Za), {concat\_atom([Za], Z)}. |

Regułą exp opisuje się pojedyncze linijki kodu, może to być deklaracja zmiennej, instrukcja arytmetyczna, bądź pętle. Powyższe instrukcje umożliwiają obsługę zagnieżdżenia jednej pętli w drugiej.

1. **Przykładowe wykonywanie programu:**

Przykładowe wykonywanie programu ma na celu sprawdzenie czy otrzymamy odpowiedz na wprowadzenie do pliku następującego wejścia:

* Wejście:

|  |
| --- |
| * int main() * { * int x = 0; * int y = 15; * for(x = 1 ; x > 15 ; x = x + 1) * { * x = x + 1; * do * { * int a = 12; * y = y - 1; * while(true) * { * a = a + 1; * y = y - a; * } * } * while(y > 12); * } * } |

Aby otrzymać wyjście należy wpisać w konsoli prologowej instrukcję „execute.”

Wyjście dla powyższego programu wygląda następująco:

Wyjście:

|  |
| --- |
| ?- execute.  org 100h  main:  x: dd 0  y: dd 15  mov eax, [x]  mov eax, 1  mov [i], eax  for:  mov eax, [x]  add eax, 1  mov [x], eax  dowhloop:  a: dd 12  mov eax, [y]  sub eax, 1  mov [y], eax  wh:  mov eax, [a]  add eax, 1  mov [a], eax  mov eax, [y]  sub eax, a  mov [y], eax  jmp whmov eax, [y]  cmp eax, 12  jg dowhloop  mov eax, [x]  add eax, 1  mov [x], eax  mov eax, [x]  cmp eax,15  jg for  true. |

1. **Wykonywanie konkretnego programu:**

Dany mamy następujący program w języku programowania C:

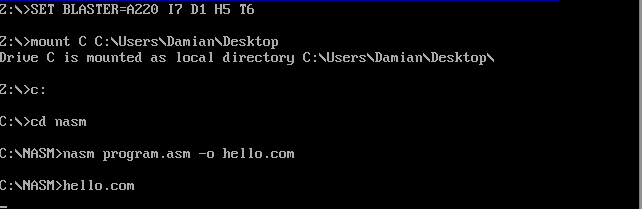
|  |
| --- |
| int main()  {  int i = 0;  int x = 100;  for(i = 0 ; i < 5 ; i = i + 1)  {  x = x - 1;  }  } |

Na podstawie tego kodu można wywnioskować że instrukcja dekrementacji zmiennej x będzie wykonywana 5 razy.

Po skompilowaniu otrzymujemy:

|  |
| --- |
| org 100h  main:  i: dd 0  x: dd 100  mov eax, [i]  mov eax, 0  mov [i], eax  for:  mov eax, [x]  sub eax, 1  mov [x], eax  mov eax, [i]  add eax, 1  mov [i], eax  mov eax, [i]  cmp eax,5  jl for |

Powyższy program zostaje zapisany w pliku program.asm i jest kompilowany:



Na podstawie kodu wejściowego można wywnioskować że program nie wypisze nic na ekran, gdyż kod ten nie zawiera instrukcji drukowania wartości na ekranie.

Na powyższym screenie przedstawione została procedura kompilacji kodu asemblera do pliku wykonywalnego „hello.com” oraz wykonanie tego programu.

Aby dowieźć tego że program faktycznie wykonuje pętle dokonane zostały zmiany w kodzie – w miejsce dekrementacji zmiennej x wprowadzone zostały rozkazy wyświetlenia napisu(w tym wypadku napis zostanie wyświetlony 5 razy). Po przerobieniu kod wygląda następująco:

|  |
| --- |
| org 100h  main:  i: dd 0  x: dd 100  mov eax, [i]  mov eax, 0  mov [i], eax  for:  mov ah, 09  mov dx, hello  int 21h  mov eax, [i]  add eax, 1  mov [i], eax  mov eax, [i]  cmp eax,5  jl for  hello db "HelloWorld!", 0ah, 0dh, "$" |

Po kompilacji i wykonaniu otrzymujemy następujące wyjście:



1. **Wnioski**

Program przetwarzający wejście w postaci pliku tekstowego zawierającego kod w języku C do języka Asembler w porównaniu do kompilatorów stosowanych w środowiskach programistycznych nie jest środowiskiem idealnym powodu dla którego tak jest można wywnioskować po samym kodzie wyjściowym. Procesor architektury x86 posiada szereg rejestrów – podstawowymi są rejestry eax, ebx, ecxi edx. Kod wyjściowy zawiera tylko rozkazy dla rejestru eax, oraz edx dla przypadku kiedy wykonywane są instrukcje wyjścia. Pomimo tych niedogodnień przedstawione zostały zasady działania gramatyki DCG w tym programie.