Języki formalne i techniki translacji
Laboratorium - Projekt (wersja α)

Termin oddania: ostatnie zajęcia przed 26 stycznia 2025
Wysłanie do wykładowcy (MS TEAMS): przed 23:45 4 lutego 2025

Używając BISON-a i FLEX-a, lub innych narzędzi o podobnej funkcjonalności, napisz kompilator prostego języka imperatywnego do kodu maszyny wirtualnej. Specyfikacja języka i maszyny jest zamieszczona poniżej. Kompilator powinien sygnalizować miejsce i rodzaj błędu (np. druga deklaracja zmiennej, użycie niezadeklarowanej zmiennej, nieznana nazwa procedury, ...), a w przypadku braku błędów zwracać kod na maszynę wirtualną. Kod wynikowy powinien być jak najkrótszy i wykonywać się jak najszybciej (w miarę optymalnie, mnożenie i dzielenie powinny być wykonywane w czasie logarytmicznym w stosunku do wartości argumentów). Ocena końcowa zależy od obu wielkości.

Program powinien być oddany z plikiem `Makefile` kompilującym go oraz z plikiem `README` opisującym dostarczone pliki oraz zawierającym dane autora. W przypadku użycia innych języków niż C/C++ należy także zamieścić dokładne instrukcje co należy doinstalować dla systemu Ubuntu. Wywołanie programu powinno wyglądać następująco[1]

```
kompilator <nazwa pliku wejściowego> <nazwa pliku wyjściowego>
```

czyli dane i wynik są podawane przez nazwy plików (nie przez strumienie). Przy przesyłaniu do wykładowcy program powinien być spakowany programem `zip` a archiwum nazwane numerem indeksu studenta. Archiwum nie powinno zawierać żadnych zbędnych plików.

Prosty język imperatywny Język powinien być zgodny z gramatyką zamieszczoną w Tablicy 1 i spełniać następujące warunki:

1. Działania arytmetyczne są wykonywane na liczbach całkowitych, ponadto dzielenie przez zero powinno dać wynik 0 i resztę także 0; reszta z dzielenia powinna mieć taki sam znak jak dzielnik.
2. Deklaracja `tab[-10:10]` oznacza zadeklarowanie tablicy o 21 elementach indeksowanych od -10 do 10; identyfikator `tab[i]` oznacza odwołanie do `i`-tego elementu tablicy `tab`; deklaracja zawierająca pierwszą liczbę większą od drugiej powinna być zgłaszana jako błąd.
3. Procedury nie mogą zawierać wywołań rekurencyjnych, parametry formalne przekazywane są przez referencje (parametry IN-OUT), zmienne używane w procedurze muszą być jej parametrami formalnymi lub być zadeklarowane wewnątrz procedury; nazwa tablicy w parametrach formalnych powinna być poprzedzona literą *T*. W procedurze można wywołać tylko procedury zdefiniowane wcześniej w kodzie programu, a jako ich parametry formalne można podać zarówno parametry formalne procedury wywołującej, jak i jej zmienne lokalne.
4. Pętla `FOR` ma iterator lokalny, przyjmujący wartości od wartości stojącej po `FROM` do wartości stojącej po `TO/DOWNTO` kolejno w odstępie +1 lub odpowiednio w odstępie -1; liczba iteracji pętli `FOR` jest ustalana na początku i nie podlega zmianie w trakcie wykonywania pętli (nawet jeśli zmieniają się wartości zmiennych wyznaczających początek i koniec pętli); iterator pętli `FOR` nie może być modyfikowany wewnątrz pętli (kompilator w takim przypadku powinien zgłaszać błąd).
5. Pętla `REPEAT-UNTIL` kończy pracę kiedy warunek napisany za `UNTIL` jest spełniony (pętla wykona się przynajmniej raz).

[1] Dla niektórych języków programowania należy napisać w pliku `README` że jest inny sposób wywołania kompilatora, np. `java kompilator` lub `python kompilator`

```
program_all   -> procedures main

procedures    -> procedures PROCEDURE proc_head IS declarations BEGIN commands END
               | procedures PROCEDURE proc_head IS BEGIN commands END
               |

main          -> PROGRAM IS declarations BEGIN commands END
               | PROGRAM IS BEGIN commands END

commands      -> commands command
               | command

command       -> identifier := expression;
               | IF condition THEN commands ELSE commands ENDIF
               | IF condition THEN commands ENDIF
               | WHILE condition DO commands ENDWHILE
               | REPEAT commands UNTIL condition;
               | FOR pidentifier FROM value TO value DO commands ENDFOR
               | FOR pidentifier FROM value DOWNTO value DO commands ENDFOR
               | proc_call;
               | READ identifier;
               | WRITE value;

proc_head     -> pidentifier ( args_decl )

proc_call     -> pidentifier ( args )

declarations  -> declarations, pidentifier
               | declarations, pidentifier[num:num]
               | pidentifier
               | pidentifier[num:num]

args_decl     -> args_decl, pidentifier
               | args_decl, T pidentifier
               | pidentifier
               | T pidentifier

args          -> args, pidentifier
               | pidentifier

expression    -> value
               | value + value
               | value - value
               | value * value
               | value / value
               | value % value

condition     -> value = value
               | value != value
               | value > value
               | value < value
               | value >= value
               | value <= value

value         -> num
               | identifier

identifier    -> pidentifier
               | pidentifier[pidentifier]
               | pidentifier[num]
```

Tabela 1: Gramatyka języka

6. Instrukcja `READ` czyta wartość z zewnątrz i podstawia pod zmienną, a `WRITE` wypisuje wartość zmiennej/liczby na zewnątrz.
7. Pozostałe instrukcje są zgodne z ich znaczeniem w większości języków programowania.
8. `pidentifier` jest opisany wyrażeniem regularnym `[_a-z]+`.
9. `num` jest liczbą całkowitą w zapisie dziesiętnym (w kodzie wejściowym liczby podawane jako stałe są ograniczone do typu 64 bitowego, na maszynie wirtualnej nie ma ograniczeń na wielkość liczb, obliczenia mogą generować dowolną liczbę całkowitą).
10. Małe i duże litery są rozróżniane.
11. W programie można użyć komentarzy zaczynających się od # i obowiązujących do końca linii.

Maszyna wirtualna Maszyna wirtualna składa się z licznika rozkazów k oraz ciągu komórek pamięci p_i, dla $i = 0, 1, 2, \ldots$ (z przyczyn technicznych $i \leqslant 2^{62}$). Komórka p_0 pełni rolę akumulatora. Maszyna pracuje na liczbach całkowitych. Program maszyny składa się z ciągu rozkazów, który niejawnie numerujemy od zera. W kolejnych krokach wykonujemy zawsze rozkaz o numerze k aż napotkamy instrukcję `HALT`. Początkowa zawartość komórek pamięci jest nieokreślona, a licznik rozkazów k ma wartość 0.

W Tablicy 2 jest podana lista rozkazów wraz z ich interpretacją i kosztem wykonania. W programie można zamieszczać komentarze w postaci: `#komentarz`. Białe znaki w kodzie są pomijane. Przejście do nieistniejącego rozkazu jest traktowane jako błąd.

Rozkaz	Interpretacja	Czas
`GET` i	pobraną liczbę zapisuje w komórce pamięci p_i oraz $k \leftarrow k+1$	100
`PUT` i	wyświetla zawartość komórki pamięci p_i oraz $k \leftarrow k+1$	100
`LOAD` i	$p_0 \leftarrow p_i$ oraz $k \leftarrow k+1$	10
`STORE` i	$p_i \leftarrow p_0$ oraz $k \leftarrow k+1$	10
`LOADI` i	$p_0 \leftarrow p_{p_i}$ oraz $k \leftarrow k+1$	20
`STOREI` i	$p_{p_i} \leftarrow p_0$ oraz $k \leftarrow k+1$	20
`ADD` i	$p_0 \leftarrow p_0 + p_i$ oraz $k \leftarrow k+1$	10
`SUB` i	$p_0 \leftarrow p_0 - p_i$ oraz $k \leftarrow k+1$	10
`ADDI` i	$p_0 \leftarrow p_0 + p_{p_i}$ oraz $k \leftarrow k+1$	20
`SUBI` i	$p_0 \leftarrow p_0 - p_{p_i}$ oraz $k \leftarrow k+1$	20
`SET` x	$p_0 \leftarrow x$ oraz $k \leftarrow k+1$	50
`HALF`	$p_0 \leftarrow \lfloor \frac{p_0}{2} \rfloor$ oraz $k \leftarrow k+1$	5
`JUMP` j	$k \leftarrow k+j$	1
`JPOS` j	jeśli $p_0 > 0$ to $k \leftarrow k+j$, w p.p. $k \leftarrow k+1$	1
`JZERO` j	jeśli $p_0 = 0$ to $k \leftarrow k+j$, w p.p. $k \leftarrow k+1$	1
`JNEG` j	jeśli $p_0 < 0$ to $k \leftarrow k+j$, w p.p. $k \leftarrow k+1$	1
`RTRN` i	$k \leftarrow p_i$	10
`HALT`	zatrzymaj program	0

Tabela 2: Rozkazy maszyny wirtualnej (x, i i j są ograniczone do liczb całkowitych 64-bitowych)

Wszystkie przykłady oraz kod maszyny wirtualnej napisany w C+ zostały zamieszczone w pliku `labor4.zip` (kod maszyny jest w dwóch wersjach: podstawowej na liczbach typu `long long` oraz w wersji `cln` na dowolnych liczbach całkowitych, która jest jednak wolniejsza w działaniu ze względu na użycie biblioteki dużych liczb).

Przykładowe kody programów

Przykład 1 – Binarny zapis liczby.

```
1  # Binarna postać liczby
2  PROGRAM IS
3          n, p
4  BEGIN
5      READ n;
6      REPEAT
7          p:=n/2;
8          p:=2*p;
9          IF n>p THEN
10             WRITE 1;
11         ELSE
12             WRITE 0;
13         ENDIF
14         n:=n/2;
15     UNTIL n=0;
16 END
```

```
-2 # prosta translacja
-1 # n -> P1, p -> P2
 0 GET 1    # READ n
 1 LOAD 1   # p:=n/2
 2 HALF
 3 STORE 2
 4 LOAD 2   # p:=2*p
 5 ADD 2
 6 STORE 2
 7 LOAD 1   #  n>p
 8 SUB 2
 9 JPOS 4
10 SET 0    # WRITE 0
11 PUT 0
12 JUMP 3
13 SET 1    # WRITE 1
14 PUT 0
15 LOAD 1   # n:=n/2
16 HALF
17 STORE 1
18 LOAD 1   # n=0
19 JZERO 2
20 JUMP -19# REPEAT
21 HALT
```

```
-1 # kod zoptymalizowany
 0 GET 1
 1 LOAD 1
 2 HALF
 3 STORE 2
 4 LOAD 1
 5 SUB 2
 6 SUB 2
 7 PUT 0
 8 LOAD 2
 9 STORE 1
10 JPOS -8
11 HALT
```

Przykład 2 – GCD.

```
1  PROCEDURE gcd(a,b,c) IS
2    x,y
3  BEGIN
4    x:=a;
5    y:=b;
6    WHILE y>0 DO
7      IF x>=y THEN
8        x:=x-y;
9      ELSE
10       x:=x+y;
11       y:=x-y;
12       x:=x-y;
13     ENDIF
14   ENDWHILE
15   c:=x;
16 END
17
18 PROGRAM IS
19   a,b,c,d,x,y,z
20 BEGIN
21   READ a;
22   READ b;
23   READ c;
24   READ d;
25   gcd(a,b,x);
26   gcd(c,d,y);
27   gcd(x,y,z);
28   WRITE z;
29 END
```

```
0  JUMP 28
1  LOADI 2 # x:=a  # gcd
2  STORE 5
3  LOADI 3 # y:=b
4  STORE 6
5  LOAD 6  # WHILE y>0
6  JPOS 2
7  JUMP 18
8  LOAD 5  # x>=y
9  SUB 6
10 JNEG 5
11 LOAD 5  # x:=x-y
12 SUB 6
13 STORE 5
14 JUMP 10
15 LOAD 5  # x:=x+y
16 ADD 6
17 STORE 5
18 LOAD 5  # y:=x-y
19 SUB 6
20 STORE 6
21 LOAD 5  # x:=x-y
22 SUB 6
23 STORE 5
24 JUMP -19        # ENDWHILE
25 LOAD 5  # c:=x
26 STOREI 4
27 RTRN 1  # return gcd
28 GET 7   # READ a
29 GET 8   # READ b
30 GET 9   # READ c
31 GET 10  # READ d
32 SET 7   # gcd(a,b,x)
33 STORE 2
34 SET 8
35 STORE 3
36 SET 11
37 STORE 4
38 SET 41  # set return
39 STORE 1
40 JUMP -39        # call gcd
41 SET 9   # gcd(c,d,y)
42 STORE 2
43 SET 10
44 STORE 3
45 SET 12
46 STORE 4
47 SET 50  # set return
48 STORE 1
49 JUMP -48        # call gcd
50 SET 11  # gcd(x,y,z)
51 STORE 2
52 SET 12
53 STORE 3
54 SET 13
55 STORE 4
56 SET 59  # set return
57 STORE 1
58 JUMP -57        # call gcd
59 PUT 13
60 HALT
```

Przykład 3 – Sito Eratostenesa.

```
1  PROCEDURE licz(T s, n) IS
2    j
3  BEGIN
4    FOR i FROM 2 TO n DO
5      s[i]:=1;
6    ENDFOR
7    FOR i FROM 2 TO n DO
8      IF s[i]>0 THEN
9        j:=i+i;
10       WHILE j<=n DO
11         s[j]:=0;
12         j:=j+i;
13       ENDWHILE
14     ENDIF
15   ENDFOR
16 END
17 PROCEDURE wypisz(T s, n) IS
18 BEGIN
19   FOR i FROM n DOWNTO 2 DO
20     IF s[i]>0 THEN
21       WRITE i;
22     ENDIF
23   ENDFOR
24 END
25 PROGRAM IS
26   n, sito[2:100]
27 BEGIN
28   n:=100;
29   licz(sito,n);
30   wypisz(sito,n);
31 END
```

```
 0 JUMP 74
 1 SET 2   # FOR i # licz
 2 STORE 5
 3 SET -2  # i'
 4 ADDI 3
 5 STORE 6
 6 JNEG 13 # FOR
 7 LOADI 2 # addr s[i]
 8 ADD 5
 9 STORE 7
10 SET 1   # s[i] := 1
11 STOREI 7
12 SET 1   # i++
13 ADD 5
14 STORE 5
15 SET -1  # i'--
16 ADD 6
17 STORE 6
18 JUMP -12         # ENDFOR
19 SET 2   # FOR i
20 STORE 5
21 SET -2  # i'
22 ADDI 3
23 STORE 6
24 JNEG 28 # FOR
25 LOADI 2 # addr s[i]
26 ADD 5
27 STORE 7
28 LOADI 7 # s[i]>0
29 JZERO 16
30 LOAD 5  # j:=i+i
31 ADD 5
32 STORE 4
33 LOADI 3 # WHILE
34 SUB 4
35 JNEG 10
36 LOADI 2 # s[j]:=0
37 ADD 4
38 STORE 7
39 SET 0
40 STOREI 7
41 LOAD 4
42 ADD 5
43 STORE 4
44 JUMP -11         # ENDWHILE
45 SET 1   # i++
46 ADD 5
47 STORE 5
48 SET -1  # i'--
49 ADD 6
50 STORE 6
51 JUMP -27         # ENDFOR
52 RTRN 1  # return licz
53 LOADI 10         # FOR i # wypisz
54 STORE 11
55 SET -2  # i'
56 ADDI 10
57 STORE 12
58 JNEG 15
59 LOADI 9 # addr s[i]
60 ADD 11
61 STORE 13
62 LOADI 13
63 JPOS 2
64 JUMP 2
65 PUT 11
66 SET -1  # i--
67 ADD 11
68 STORE 11
69 SET -1  # i'--
70 ADD 12
71 STORE 12
72 JUMP -14         # ENDFOR
73 RTRN 8  # return wypisz
74 SET 13  # s[0] 2-15:100-113
75 STORE 14
76 SET 100 # n:=100
77 STORE 114
78 SET 14  # licz(s,n)
79 STORE 2
80 SET 114
81 STORE 3
82 SET 85  # set return
83 STORE 1
84 JUMP -83         # call licz
85 SET 14  # wypisz(s,n)
86 STORE 9
87 SET 114
88 STORE 10
89 SET 92  # set return
90 STORE 8
91 JUMP -38         #call wypisz
92 HALT
```

Optymalność wykonywania mnożenia i dzielenia

```
1  # Rozkład na czynniki pierwsze
2  PROCEDURE check(n,d,p) IS
3    r
4  BEGIN
5      p:=0;
6      r:=n%d;
7      WHILE r=0 DO
8          n:=n/d;
9          p:=p+1;
10         r:=n%d;
11     ENDWHILE
12 END
13
14 PROGRAM IS
15   n,m,potega,dzielnik
16 BEGIN
17   READ n;
18   dzielnik:=2;
19   m:=dzielnik*dzielnik;
20   WHILE n>=m DO
21     check(n,dzielnik,potega);
22     IF potega>0 THEN # jest podzielna przez dzielnik
23       WRITE dzielnik;
24       WRITE potega;
25     ENDIF
26     dzielnik:=dzielnik+1;
27     m:=dzielnik*dzielnik;
28   ENDWHILE
29   IF n!=1 THEN # ostatni dzielnik różny od 1
30     WRITE n;
31     WRITE 1;
32   ENDIF
33 END
```

Dla powyższego programu koszt działania kodu wynikowego na załączonej maszynie powinien być porównywalny do poniższych wyników (mniej więcej tego samego rzędu wielkości - liczba cyfr oznaczonych przez $*$):

```
...
Uruchamianie programu.
? 1234567890
> 2
> 1
> 3
> 2
> 5
> 1
> 3607
> 1
> 3803
> 1
Skończono program (koszt: ** *** ***; w tym i/o: 1 100).
```

```
...
Uruchamianie programu.
? 12345678901
> 857
> 1
> 14405693
> 1
Skończono program (koszt: ** *** ***; w tym i/o: 500).
...
Uruchamianie programu.
? 12345678903
> 3
> 1
> 4115226301
> 1
Skończono program (koszt: *** *** ***; w tym i/o: 500).
```