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

Termin oddania: ostatnie zajęcia przed 22 stycznia 2022
Wysłanie do wykładowcy (MS TEAMS): przed 23:45 2 lutego 2022

Używając BISON-a i FLEX-a 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, niewłaściwe użycie nazwy tablicy,...), a w przypadku braku błędów zwracać kod na maszynę wirtualną. Kod wynikowy powinien wykonywać się jak najszybciej (w miarę optymalnie, mnożenie i dzielenie powinny być wykonywane w czasie logarytmicznym w stosunku do wartości argumentów).

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;
2. `PLUS MINUS TIMES DIV MOD` oznaczają odpowiednio dodawanie, odejmowanie, mnożenie, dzielenie całkowitoliczbowe i obliczanie reszty na liczbach całkowitych, reszta z dzielenia powinna mieć taki sam znak jak dzielnik;
3. `EQ NEQ LE GE LEQ GEQ` oznaczają odpowiednio relacje $=$, $\neq$, $<$, $>$, $\leqslant$ i $\geqslant$ na liczbach całkowitych;
4. `ASSIGN` oznacza przypisanie;
5. deklaracja `tab[-10:100]` oznacza zadeklarowanie tablicy `tab` o 111 elementach indeksowanych od -10 do 100, identyfikator `tab[i]` oznacza odwołanie do `i`-tego elementu tablicy `tab`, deklaracja zawierająca pierwszą liczbę większą od drugiej powinna być zgłaszana jako błąd;
6. pętla `FOR` ma iterator lokalny, przyjmujący wartości od wartości stojącej po `FROM` do wartości stojącej po `TO` kolejno w odstępie +1 lub w odstępie -1 jeśli użyto słowa `DOWNTO`;
7. 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);
8. iterator pętli `FOR` nie może być modyfikowany wewnątrz pętli (kompilator w takim przypadku powinien zgłaszać błąd);
9. pętla `REPEAT-UNTIL` kończy pracę kiedy warunek napisany za `UNTIL` jest spełniony (pętla wykona się przynajmniej raz);

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

```
program      -> VAR declarations BEGIN commands END
              | BEGIN commands END

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

commands     -> commands command
              | command

command      -> identifier ASSIGN 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
              | READ identifier;
              | WRITE value;

expression   -> value
              | value PLUS value
              | value MINUS value
              | value TIMES value
              | value DIV value
              | value MOD value

condition    -> value EQ value
              | value NEQ value
              | value LE value
              | value GE value
              | value LEQ value
              | value GEQ value

value        -> num
              | identifier

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

Tablica 1: Gramatyka języka

10. instrukcja `READ` czyta wartość z zewnątrz i podstawia pod zmienną, a `WRITE` wypisuje wartość zmiennej/liczby na zewnątrz;
11. pozostałe instrukcje są zgodne z ich znaczeniem w większości języków programowania;
12. `pidentifier` jest opisany wyrażeniem regularnym `[_a-z]+`;
13. `num` jest liczbą całkowitą w zapisie dziesiętnym (w kodzie wejściowym liczby są ograniczone do typu `long long` (64 bitowy), na maszynie wirtualnej nie ma ograniczeń na wielkość liczb, obliczenia mogą generować dowolną liczbę całkowitą);
14. małe i duże litery są rozróżniane;
15. w programie można użyć komentarzy postaci: `( komentarz )`, które nie mogą być zagnieżdżone.

Maszyna wirtualna Maszyna wirtualna składa się z 8 rejestrów (r_a, r_b, r_c, r_d, r_e, r_f, r_g, r_h), licznika rozkazów k oraz ciągu komórek pamięci p_i, dla $i = 0, 1, 2, ...$ (z przyczyn technicznych $i \leqslant 2^{62}$). 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 rejestrów i 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)`, które nie mogą być zagnieżdżone. Białe znaki w kodzie są pomijane. Przejście do nieistniejącego rozkazu lub wywołanie nieistniejącego rejestru jest traktowane jako błąd.

Rozkaz	Interpretacja	Czas
`GET`	pobraną liczbę zapisuje w rejestrze r_a oraz $k \leftarrow k+1$	100
`PUT`	wyświetla zawartość rejestru r_a oraz $k \leftarrow k+1$	100
`LOAD` x	$r_a \leftarrow p_{r_x}$ oraz $k \leftarrow k+1$	50
`STORE` x	$p_{r_x} \leftarrow r_a$ oraz $k \leftarrow k+1$	50
`ADD` x	$r_a \leftarrow r_a + r_x$ oraz $k \leftarrow k+1$	10
`SUB` x	$r_a \leftarrow r_a - r_x$ oraz $k \leftarrow k+1$	10
`SHIFT` x	$r_a \leftarrow \lfloor 2^{r_x} \cdot r_a \rfloor$ oraz $k \leftarrow k+1$	5
`SWAP` x	$r_a \leftrightarrow r_x$ oraz $k \leftarrow k+1$	1
`RESET` x	$r_x \leftarrow 0$ oraz $k \leftarrow k+1$	1
`INC` x	$r_x \leftarrow r_x + 1$ oraz $k \leftarrow k+1$	1
`DEC` x	$r_x \leftarrow r_x - 1$ oraz $k \leftarrow k+1$	1
`JUMP` j	$k \leftarrow k+j$	1
`JPOS` j	jeśli $r_a > 0$ to $k \leftarrow k+j$, w p.p. $k \leftarrow k+1$	1
`JZERO` j	jeśli $r_a = 0$ to $k \leftarrow k+j$, w p.p. $k \leftarrow k+1$	1
`JNEG` j	jeśli $r_a < 0$ to $k \leftarrow k+j$, w p.p. $k \leftarrow k+1$	1
`HALT`	zatrzymaj program	0

Tablica 2: Rozkazy maszyny wirtualnej ($x \in \{a, b, c, d, e, f, g, h\}$ i $j \in \mathbb{Z} \setminus \{0\}$)

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 naturalnych, która jest jednak wolniejsza w działaniu ze względu na użycie biblioteki dużych liczb).

Przykładowe kody programów

Przykład 1 – Binarny zapis liczby.

```
1  VAR
2      n, p
3  BEGIN
4      READ n;
5      IF n GEQ 0 THEN
6          REPEAT
7              p ASSIGN n DIV 2;
8              p ASSIGN 2 TIMES p;
9              IF n NEQ p THEN
10                 WRITE 1;
11             ELSE
12                 WRITE 0;
13             ENDIF
14             n ASSIGN n DIV 2;
15         UNTIL n EQ 0;
16     ENDIF
17 END
```

```
-1 (zapis binarny liczby)
 0 RESET d
 1 INC d
 2 RESET e
 3 DEC e
 4 GET
 5 JNEG 21
 6 SWAP b
 7 RESET a
 8 ADD b
 9 SHIFT e
10 SHIFT d
11 SWAP c
12 RESET a
13 ADD b
14 SUB c
15 JZERO 5
16 RESET a
17 INC a
18 PUT
19 JUMP 3
20 RESET a
21 PUT
22 SWAP b
23 SHIFT e
24 JZERO 2
25 JUMP -19
26 HALT
```

```
-1 (zapis binarny liczby - zoptymalizowany)
 0 RESET c
 1 DEC c
 2 GET
 3 JNEG 12
 4 SWAP b
 5 RESET a
 6 ADD b
 7 SHIFT c
 8 SWAP b
 9 SUB b
10 SUB b
11 PUT
12 SWAP b
13 JPOS -9
14 HALT
```

Przykład 2 – Sito Eratostenesa.

```
1  VAR ( sito Eratostenesa)
2      n, j, sito[2:100]
3  BEGIN
4      n ASSIGN 100;
5      FOR i FROM n DOWNTO 2 DO
6          sito[i] ASSIGN 1;
7      ENDFOR
8      FOR i FROM 2 TO n DO
9          IF sito[i] NEQ 0 THEN
10             j ASSIGN i PLUS i;
11             WHILE j LEQ n DO
12                 sito[j] ASSIGN 0;
13                 j ASSIGN j PLUS i;
14             ENDWHILE
15             WRITE i;
16         ENDIF
17     ENDFOR
18 END
```

```
0   RESET h
1   INC h
2   RESET a          (generowanie 100)
3   INC a
4   SHIFT h
5   INC a
6   INC h
7   INC h
8   SHIFT h
9   INC a
10  DEC h
11  SHIFT h
12  DEC h
13  SWAP b           (r[b]<-n)
14  RESET a
15  ADD b
16  SWAP d           (r[d]<-i=n)
17  RESET a
18  ADD b
19  DEC a
20  DEC a            (licznik for r[e])
21  JNEG 14          (wyjście for)
22  SWAP e
23  RESET a
24  ADD d
25  DEC a
26  DEC a            (numer pamięci [i-2])
27  SWAP f
28  SWAP h
29  STORE f          (sito[j]<-0)
30  SWAP h
31  DEC d
32  DEC e
33  SWAP e
34  JUMP -13         (endfor)
35  RESET a
36  INC a
37  INC a
38  SWAP d           (r[d]<-i=2)
39  RESET a
40  ADD b
41  DEC a
42  DEC a            (licznik for r[e])
43  JNEG 35          (wyjscie for)
44  SWAP e
45  RESET a
46  ADD d
47  DEC a
48  DEC a            (numer pamięci [i-2])
49  SWAP f
50  LOAD f           (r[a]<-sito[i])
51  JZERO 23         (sito[i]==0)
52  RESET a
53  ADD d
54  ADD d
55  SWAP c           (r[c]<-j=i+i)
56  RESET a
57  ADD b
58  SUB c            (j<=n)
59  JNEG 12          (wyjście WHILE)
60  RESET a
61  ADD c
62  DEC a
63  DEC a            (numer pamięci [j-2])
64  SWAP f
65  RESET a
66  STORE f          (sito[j]<-0)
67  SWAP c
68  ADD d
69  SWAP c           (r[c]<-j=j+i)
70  JUMP -14
71  SWAP d
72  PUT              (WRITE i)
73  SWAP d
74  INC d
75  DEC e
76  SWAP e
77  JUMP -34
78  HALT
```

Optymalność wykonywania mnożenia i dzielenia

```
1  ( Rozkład liczby na czynniki pierwsze )
2  VAR
3      n, m, reszta, potega, dzielnik
4  BEGIN
5      READ n;
6      dzielnik ASSIGN 2;
7      m ASSIGN dzielnik TIMES dzielnik;
8      WHILE n GEQ m DO
9          potega ASSIGN 0;
10         reszta ASSIGN n MOD dzielnik;
11         WHILE reszta EQ 0 DO
12             n ASSIGN n DIV dzielnik;
13             potega ASSIGN potega PLUS 1;
14             reszta ASSIGN n MOD dzielnik;
15         ENDWHILE
16         IF potega GE 0 THEN ( czy znaleziono dzielnik )
17             WRITE dzielnik;
18             WRITE potega;
19         ELSE
20             dzielnik ASSIGN dzielnik PLUS 1;
21             m ASSIGN dzielnik TIMES dzielnik;
22         ENDIF
23     ENDWHILE
24     IF n NEQ 1 THEN ( ostatni dzielnik )
25         WRITE n;
26         WRITE 1;
27     ENDIF
28 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: 1100).
...
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).
```