Programowanie – czerwiec 2004
Wersja A

Za cały egzamin będzie można dostać 100 punktów (nie licząc punktów bonusowych). Progi są następujące: 34 punkty daje ocenę dostateczną, 47 dostateczną z plusem, 60 dobrą, 73 dobrą z plusem, 86 bardzo dobrą.

W jednym zadaniu pojawia się sformułowanie **dowolny język programowania**. *Oznacza ono jeden z języków: Pascal, C, C++, Python, Haskell, SML lub Prolog.*

Zadanie 1. (**18p**) Gramatyka G_1 nad alfabetem $\{a, b\}$, określona jest przez przez zbiór produkcji P_1

$$P_1 = \{S \to aSb,\ S \to bSa,\ S \to \varepsilon, S \to SS\}$$

a) Co to znaczy, że gramatyka jest jednoznaczna (**2p**). Pokaż, że G_1 nie jest jednoznaczna (**3p**).
b) Zdefiniuj jednoznaczną gramatykę G_2, taką że $L(G_1) = L(G_2)$, a G_2 ma co najwyżej dwa symbole nieterminalne (**5p**). Uzasadnij, że G_2 istotnie jest jednoznaczna.(**5p**)
c) Rozważmy język $L_3 = L(G_1) \cap L((a^+b)^*)$. Czy L_3 jest regularny? (**1p**) Jeżeli tak, to podaj wyrażenie regularne, które go opisuje, jeżeli nie, to podaj gramatykę, która go generuje (**4p**).

Zadanie 2. (**15p**) Zamień poniższe programy na ich odpowiedniki, w których nie ma instrukcji `goto`, a jedynymi strukturami sterującymi są pętla `while` i instrukcja `if`. Możesz wprowadzać nowe zmienne (również tablicowe), przypisywać im wartości i sprawdzać je.

Występujące w programach warunki nie wywołują efektów ubocznych.

a) (**5p**)
```
L1: C1;
L2: C2
    if (b1) goto L1;
    if (b2) goto L2;
```

b) (**5p**)
```
    if (b1) goto L1;
    C2;
    do {
        C3;
L1:     if (b2) break;
        C4;
    } while (b3)
```

c) (**5p**)
```
int p(int x) {
   if (x<=0) return 1;
   return p(x-1)+p(x/2);
}
```

Zadanie 3. (**15p**) Poniższe fragmenty programów w języku C nie robią tego, co mają robić. Dla każdego z nich powiedz, gdzie jest problem i napisz program poprawnie (zwróć również uwagę na styl, w każdym podpunkcie można zdobyć 2 punkty za wskazanie błędu i 2 za przedstawienie poprawnego programu):

a) Program usuwa z pamięci listę (**5p**)
```
for (p=Lista; p != 0; p = p->next)
   free(p);
```

b) Program oblicza sumę liczb całkowitych zawartych w pliku. Separatorem jest ciąg niecyfr, zakładamy że zarówno liczby, jak i ich suma zmieszczą się w typie `long`. (**5p**)
```
long n = 0;
long suma = 0;
int c;

while ( (c = getc(Plik) != EOF) {
    if (is_digit(c)) n = 10*n + c; // is_digit -- czy znak jest cyfrą
    else suma += n;
}
```

c) Odwracanie napisu `s`, tak aby `trudne` stało się `endurt`. (**5p**)

```
int i=0;
int j;
for (; s[i++]; ) ;
for (j=0; j< i/2; j++) {
       s[j] = s[i-j];
       s[i-j] = s[j];
}
```

Zadanie 4. (**18p**) W zadaniu tym używamy składni Haskella. Rozwiązanie możesz przedstawić w Haskelu albo w SML-u. Zdefiniujemy typ drzewo:

```
data Tree a = Lf a | Br (Tree a) (Tree a)
```

(jak widać wartości są przechowywane w liściach, a najmniejsze drzewo ma 1 element). Wyrażenie arytmetyczne zbudowane z liczb oraz znaku `+` możemy reprezentować za pomocą takiego drzewa, przykładowo `1+(2+3)`, w tej reprezentacji ma postać `Br (Lf 1) (Br (Lf 2) (Lf 3))`.

a) Napisz ogonową funkcję `sum`, która bierze listę i zwraca sumę jej elementów. W funkcji tej nie wolno wywoływać innych funkcji. (**3p**)

b) Napisz funkcję `val`, która bierze drzewo, traktuje je jak wyrażenie w sposób opisany powyżej i zwraca jego wartość. (**3p**)

c) Funkcja `tolist` zdefiniowana jest następująco:

```
tolist (Lf x) = [x]
tolist (Br t1 t2) = (tolist t1) ++ (tolist t2)
```

Typ `Tree` rózni się nieco od używanego na wykładzie. Sformuuj dla tego typu zasadę indukcji (**2p**). Następnie z jej pomocą udowodnij (**10p**), że dla każdego drzewa `T` przechowującego liczby, mamy

```
sum (tolist T) = val T
```

Wskazówka: zdefiniuj naiwną wersję `sum` *(na przykład* `nsum`*), udowodnij jej równoważność z wersją ogonową i następnie w głównym dowodzie używaj wersji naiwnej.*

Zadanie 5. (**24p**) W zadaniu tym będziemy rozważać uproszczoną wersję zadania z autkami. Zasady są następujące:

- Autko jezdzi po macierzy o wymiarach $N \times N$. Elementy tej macierzy to znaki `'.'`, `'x'`, `'!'` oznaczające odpowiednio asfalt, przeszkodę oraz metę.
- Współrzędne auta są liczbami całkowitymi (z przedziału od 1 do N)
- Prędkość jest liczbą całkowitą z przedziału od 0 do 4.
- Kierunek zadany jest parą (dx, dy), przy czym $dx, dy \in \{-1, 0, 1\}$, a $(0, 0)$ nie jest dozwolonym kierunkiem.
- Ponieważ punkt 1,1 znajduje się w lewym górnym rogu, to jednostkowy skręt w lewo zmienia przykładowo kierunek (+1,0) na (+1,-1), a jednostkowy skręt w prawo zmienia kierunek (1,1) na (0,1). Inaczej mówiąc, skręt jest zawsze o 45 stopni.

Stan autka zadany jest przez cztery elementy: współrzędną x, współrzędną y, prędkość v oraz kierunek (dx, dy). Następny stan określony jest za pomocą poniższego pseudokodu:

```
if "przyspieszamy" v++;
if "hamujemy" v--;
if (v<0) v=0;
if (v>4) v=4;
if "skret w prawo" zmien-odpowiednio (dx,dy)
if "skret w lewo" zmien-odpowiednio (dx,dy)
x += dx * v
y += dy * v
```

Poniżej opisane programy powinieneś implementować w *jednym* wybranym języku. Typy poniższych funkcji[1] są „abstrakcyjne”, musisz opisać jakie konkretne konstrukcje z języka im odpowiadają. Liczy się również elegancja rozwiązania.

a) Napisz funkcję `next: (Stan,Decyzja) -> Stan`, która dla danego stanu autka znajduje stan w następnym kroku symulacji, przy założeniu, że decyzja należy do zbioru `gaz,hamulec,lewo,prawo,jazda`. Jak widać, nie można jednocześnie przyśpieszać i skręcać. `jazda` oznacza niezmienianie prędkości oraz kierunku jazdy. (**3p**)

[1] Jeżeli używasz Prologa, to musisz pisać nie funkcje, a predykaty o odpowiedniej arności

b) Napisz funkcję `evaluate: (Plansza,Stan) -> {ok,kraksa,finisz}`, która sprawdza czy stan przy danej planszy umożliwia dalszą jazdę (stoimy na asfalcie), kończy jazdę w wyniku najechania na przeszkodę, ewentualnie kończy jazdę w wyniku najechania na pole docelowe. (**3p**)

c) Napisz funkcję `nextstates: (Plansza,Lista-Stanow) -> Lista-Stanow`, który dla stanów znajdujących się w argumencie obliczy wszystkie „niekraksowe" stany, które mogą być otrzymane w wyniku jednego kroku symulacji. (**4p**)

d) Napisz funkcję `toset: Lista-Stanow -> Lista-Stanow`, która usuwa powtarzające się w argumencie stany. (**4p**)

e) Napisz funkcję `race: (Plansza,Stan) -> Integer`, która zwraca najmniejszą liczbę kroków symulacji, w wyniku której auto przy zadanym stanie początkowym dotrze do jednego z pól oznaczonych przez `'!'`. Jeżeli do mety nie da się dojechać, powinno się zwrócić wartość `-1`. (**10p**)

Zadanie 6. Odpowiedz na poniższe pytania. Proszę o odpowiedź na każde z pytań, przy czym odpowiedź należy wybrać ze zbioru: `T,N,?`. Odpowiedź `?` warta jest zawsze 0 punktów, odpowiedzi `T` oraz `N` są warte -2 lub 2, w zależności od tego, czy są poprawne.

a) Odcięcie na końcu klauzuli nigdy nie zmienia semantyki programu w Prologu

b) Poprawne w Pythonie wyrażenie opisujące listę, zbudowane z liczb oraz znaków `[ , ]` (przecinek też występuje) zawsze opisuje poprawną listę w Haskelu.

c) Jeżeli w Prologowym programie nie występują zmienne, to dla każdego zapytanie drzewo poszukiwań jest skończone.

d) W poprawnym programie w C++ znajduje się instrukcja `x = a[y]`. Oznacza to, że wyrażenie `y+y` jest poprawne.

e) Gdyby w języku D^* zamienić pętle `while` na pętlę `for (i=0;i<N;i++) ...` oraz zabronić modyfikacji zmiennej sterującej w ciele pętli, wówczas pytanie: „czy program wypisze liczbę 13" stałoby się rozstrzygalne.