

Lista zadań nr 6

Zadanie 1.

Skompiluj kod z wykładu. Sprawdź na przykładzie, że interpreter działa poprawnie.

Zadanie 2.

Zaproponuj gramatykę bezkontekstową języka palindromów nad alfabetem $\Sigma = \{a, b\}$. Na przykład: ϵ , **bab**, **abaaba**, **babbbbaabbbab**.

Zadanie 3. (2 pkt)

Gramatykę w postaci BNF można reprezentować w OCamlu przy użyciu następującego typu:

```
type 'a symbol =
  | T of string (* symbol terminalny *)
  | N of 'a      (* symbol nieterminalny *)

type 'a grammar = ('a * ('a symbol list) list) list
```

Jest to lista asocjacyjna (czyli lista par klucz–wartość), w której kluczami są symbole nieterminalne, a wartościami lista prawych stron produkcji dla danego symbolu nieterminalnego. Każda prawa strona produkcji to lista, która może zawierać symbole terminalne i nieterminalne. Proszę zwrócić uwagę, że całość parametryzowana jest typem 'a symboli nieterminalnych: każda gramatyka może używać innego typu. Przykładowo, omawiana na wykładzie niejednoznaczna gramatyka dla wyrażeń arytmetycznych z jednym symbolem nieterminalnym

$$\langle E \rangle ::= \langle E \rangle + \langle E \rangle \mid \langle E \rangle * \langle E \rangle \mid (\langle E \rangle) \mid 1 \mid 2$$

może być reprezentowana jako:

```
let expr : unit grammar =
  [(), [[N (); T "+"; N ()];
        [N (); T "*"; N ()];
```

```
[T "("; N (); T ")"];
[T "1"];
[T "2"]]
```

Zaimplementuj funkcję

```
generate : 'a grammar -> 'a -> string
```

która generuje słowo należące do języka poprzez rozwijanie symbolu nieterminalnego przy użyciu produkcji wybranej w sposób pseudolosowy. Na przykład:

```
# generate expr ();
- : string = "(((1*1*2)))"
# generate expr ();
- : string = "2"
# generate expr ();
- : string = "1+1*1"
# generate expr ();
- : string =
"(2+2+2+1*1*2+1+2+2*1+2*2+2+((1+2+2+(2)*1+(2*1)+2*2+(2)+2*1+2*1+2+(1*1)
+2)*2*1+1+(1+2*(2)+(2)+1*1*((1))+2*(2)+(1*1)*(2)*((1+1*1*((1))*2*2+1*2*
1*2+1+(1)+1+(1)+2+2*1))+1*2*2+2*2*2)*((2)*2+2))*1)"
```

Kolejny przykład: dla gramatyki

```
let pol : string grammar =
[ "zdanie", [[N "grupa-podmiotu"; N "grupa-orzeczenia"]]
; "grupa-podmiotu", [[N "przydawka"; N "podmiot"]]
; "grupa-orzeczenia", [[N "orzeczenie"; N "dopelnienie"]]
; "przydawka", [[T "Piekny "];
[T "Bogaty "];
[T "Wesoly "]]
; "podmiot", [[T "policjant "];
[T "student "];
[T "piekarz "]]
; "orzeczenie", [[T "zjadl "];
[T "pokochal "];
[T "zobaczyl "]]
; "dopelnienie", [[T "zupe."];
[T "studentke."];
[T "sam siebie."];
[T "instytut informatyki."]]
]
```

chcemy generować napisy takie jak:

```
# generate pol "zdanie";
- : string = "Wesoly piekarz zobaczyl zupe."
# generate pol "zdanie";
- : string = "Piekny student zjadl sam siebie."
# generate pol "zdanie";
- : string = "Bogaty student pokochal instytut informatyki."
```

Do rozwiązania mogą przydać się następujące funkcje:

```
Random.int : int -> int    - losowa liczba z zakresu 0..n-1
List.length : 'a list -> int - długość listy
List.assoc : 'a -> ('a * 'b) list -> 'b    - wyszukanie elementu na liście asocjacyjnej
String.concat : string -> string list -> string - konkatencja listy stringów
z separatorem
```

Zadanie 4. (2 pkt)

Zaimplementuj funkcję

```
parens_ok : string -> bool
```

Sprawdza ona, czy argument jest napisem zawierającym tylko symbole '(' oraz ')', które tworzą poprawne nawiasowanie. Na przykład:

```
# parens_ok "(()())";;
- : bool = true
# parens_ok "()()";;
- : bool = false
# parens_ok "((()))";;
- : bool = false
# parens_ok "(x)";;
- : bool = false
```

Do rozwiązania zadania **nie** używaj generatora parserów. Napis (string) str można zmienić na listę znaków (char list) używając:

```
List.of_seq (String.to_seq str)
```

Zadanie 5. (2 pkt)

Rozbuduj funkcję parens_ok z poprzedniego zadania tak, by sprawdzała poprawne nawiasowania składające się z symboli '(', ')', '[', ']', '{' oraz '}'. Na przykład:

```
# parens_ok "()[]";;
- : bool = true
# parens_ok "[()[]{}]";;
- : bool = true
# parens_ok "{[]}";;
- : bool = false
# parens_ok "({})";;
- : bool = false
```

Zadanie 5.

Zmodyfikuj ewaluator z wykładu tak, by operacje arytmetyczne były interpretowane w arytmetyce modulo 5.

Zadanie 6.

Rozbuduj język programowania z wykładu o operatory $<$, $<=$, $>$, $>=$, $<>$. Rozszerz odpowiednio lexer, parser i ewaluator.

Zadanie 7. (2 pkt)

Rozbuduj język z wykładu o leniwe operatory działające na wartościach boolowskich $\&\&$ (koniunkcja) oraz $\|\|$ (alternatywa). Leniwość polega na tym, że nie ewaluujemy drugiego argumentu, jeśli znamy wartość całego wyrażenia po ewaluacji pierwszego argumentu. Przykładowo, ewaluacja programu

```
true || if 2 / 0 = 2 then false else false
```

nie powinna skończyć się błędem. Czemu nie możemy zrobić tego podobnie, jak w poprzednim zadaniu?