MP25 @ II UWr 27 marca 2025 r.

Lista zadań nr 5

Zadanie 1. (2 pkt)

Rozważmy gramatykę nad alfabetem (zbiorem symboli terminalnych) złożonym z nawiasów otwierających i zamykających ($\Sigma = \{(,)\}$) oraz ze zbiorem produkcji:

$$P = \{S \to (S)S, S \to (, S \to \epsilon\}$$

Jak można opisać język generowany przez tą gramatykę? Pokaż, że gramatyka ta jest niejednoznaczna. Następnie zaproponuj jednoznaczną gramatykę generującą ten sam język.

Kontekst: gramatyka ta jest związana z problemem parsowania instrukcji w językach programowania, które dopuszczają konstrukcje postaci `if (E) S` oraz `if (E) S else S`.

Zadanie 2. (2 pkt)

Zaproponuj wyrażenia regularne definiujące następujące języki nad alfabetem $\Sigma = \{0, 1\}$:

- Wszystkie słowa, w których po wystąpieniu jedynki nie występują żadne zera.
- Wszystkie słowa postaci jak w poprzednim podpunkcie, ale zawierające co najmniej jedną jedynkę oraz co najmniej jedno zero.
- Dopełnienie języka z poprzedniego podpunktu, tzn. wszystkie słowa nie będące postaci jak w poprzednim podpunkcie.
- Wszystkie słowa zawierajace parzystą liczbę zer i dowolną liczbę jedynek.
- Wszystkie słowa w których nie występują dwie następujące po sobie jedynki, tzn. podciągi postaci 11.

Zadanie 3.

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`) można zmienić na listę znaków (`char list`) używając funkcji:

```
let list_of_string s = String.to_seq s |> List.of_seq
```

Wskazówka: Napis jest poprawnym nawiasowaniem, gdy nawiasów otwierających jest tyle samo, co zamykających, ale także każdy nawias zamykający zamyka jakiś nawias otwierający (rozważ napis `")("`). Dla tego warunku wystarczy, by w każdym prefiksie było przynajmniej tyle nawiasów otwierających, co zamykających.

Zadanie 4. (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
```

Wskazówka: W tym wypadku nawias zamykający może zamknąć tylko nawias otwierający tego samego typu. Nie wystarczy więc pamiętać, ile było nawiasów, ale trzeba także wiedzieć, jakiego były rodzaju. Dobrą strukturą do tego jest stos, na który wrzucamy nawiasy otwierające, które zdejmujemy, gdy zobaczymy pasujący nawias zamykający. Stos można zaimplementować przy użyciu listy.

Zadanie 5.

Zmodyfikuj interpreter z wykładu tak, aby wyrażenia były obliczane przy użyciu liczb zmiennopozycyjnych `float` zamiast całkowitych `int`. Należy też zmodyfikować lekser tak, aby akceptował stałe liczbowe z kropką dziesiętną (np. `6.25`).

Zadanie 6. (2 pkt)

Zmodyfikuj interpreter z zadania 5, dodając do niego:

- Operator potęgowania `**` o priorytecie wyższym niż mnożenie, wiążący w prawo.
- Operator prefiksowy logarytmu naturalnego `log`, o priorytecie wyższym niż wszystkie operatory binarne. Np. napis `"log log 3 + 5"` powinien być parsowany tak samo, jak `"(log (log 3)) + 5"`.
- Stałą matematyczną e. *Wskazówka:* Do implementacji tego podpunktu nie musisz rozszerzać składni abstrakcyjnej.

Zadanie 7. (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

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

może być reprezentowana jako:

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

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

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 "sam siebie."];
                    [T "instytut informatyki."]]
  ]
```

chcemy generowć 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`
– konkatenacja listy stringów z separatorem