Parsarea si executia unei masini Turing
Termen de predare: 13.01.2013

Cerinta

Se cere sa scrieti un program care parseaza un fisier de intrare (cu extensia .mt) in care sunt descrise mai multe masini Turing, apoi executa una din masinile Turing din fisier pentru banda de intrare primita ca parametru si intoarce continutul benzii dupa terminarea executiei masinii Turing.

Puteti alege intre urmatoarele variante pentru implementarea temei:

- **flex** (nu Adobe Flex, ci The Fast Lexical Analyzer) pentru partea de parsare si **C/C++** pentru partea de executie a masinii Turing; pentru sisteme Ubuntu flex se poate instala folosind apt-get; de asemenea, puteti compila sursele ultimei versiuni, care sunt disponibile la http://flex.sourceforge.net; tot acolo gasiti si manualul de utilizare pentru flex
- **JFlex** pentru partea de parsare si **Java** pentru partea de executie a masinii Turing; puteti instala pachetul jflex folosind apt-get sau puteti descarca sursele de la http://jflex.de; tot acolo gasiti si documentatie pentru utilizarea jflex
- **Alex** pentru partea de parsare si **Haskell** pentru partea de executie a masinii Turing; puteti instala pachetul alex folosind apt-get; variante alternative de instalare, precum si documentatie, gasiti la http://www.haskell.org/alex

Este important sa folositi cat mai mult facilitatile de analiza lexicala disponibile in flex / Jflex / Alex pentru parsare si sa nu faceti parsarea de mana in C/C++ / Java / Haskell.

Limbajul pentru descrierea masinilor Turing

Limbajul este descris printr-o gramatica BNF si foloseste aceeasi conventie de culori ca si articolul Wikipedia despre BNF:

- albastru = neterminali
- verde = operatori ai limbajului BNF si paranteze ajutatoare
- rosu = terminali (elemente care fac parte efectiv din limbajul descris)

Cu violet au fost colorate exemplele. Cele care nu sunt bold reprezinta variante extreme de folosire a spatiilor. Daca va e mai usor in implementare, puteti presupune ca astfel de cazuri extreme nu apar in fisierele de test.

De asemenea au fost marcate cu galben elementele mai avansate ale limbajului, elemente care nu sunt strict necesare pentru descrierea unei masini Turing, dar care pot face o astfel de descriere mai usor de citit. Pentru un maxim de 8 puncte din 10 puteti ignora aceste elemente si sa parsati doar limbajul de baza. Vor exista si fisiere de test speciale pentru limbajul de baza.

```
<source> ::= <alphabet-decl> (<comment> | <code>)*
// un fisier sursa contine declararea alfabetului, urmata de comentarii si/sau cod

<alphabet-decl> ::= alphabet :: (<symbol> )+ ;
```

// cuvantul cheie “alfabet”, separat prin “::” de lista de simboluri, care se termina cu “;”
// se presupune implicit ca “;” nu poate face parte din alfabet
// limbajul nu impune, dar in general declararea alfabetului se va face pe o singura linie, iar simbolurile vor fi separate prin spatii
^alfabet :: a b c # ;
^alfabet::abc#;
^alfabet:: ab
c #   ;

<comment> ::= ; <text> <end-of-line>
// un comentariu incepe cu “;” la inceput de linie si se termina la sfarsitul liniei (pentru simplitate, nu se pot pune comentarii pe aceeasi linie cu codul)
^; acesta este un comentariu

<code> ::= <symbol-decl> | <set-decl> | <mt-decl>
// codul dintr-un fisier sursa consta in declararea de simboluri, multimi de simboluri sau masini Turing

<symbol-decl> ::= <name> = <symbol> ;
// se da nume unui simbol
// simbolul trebuie sa faca parte din alfabet (in caz contrar, comportamentul programului este nedefinit = nu trebuie sa dati mesaje de eroare, puteti presupune ca fisierele de intrare sunt corecte)
^void = # ;
^void=#;
^ void   =
#   ;

<set-decl> ::= <name> := { <symbol> (, <symbol>)* } ;
// se da un nume unei multimi de simboluri, specificata intre acolade
// simbolurile sunt separate prin virgule
// simbolurile trebuie sa faca parte din alfabet
^<any> = {a, b, c} ;
^<any>={a,b,c};
^ <any> = {
     a   ,   b ,
     c } ;

<symbol> ::= <letter> | <digit> | <other>

// simbolurile pot fi litere, cifre sau caractere speciale
// <other> trebuie sa includa cel putin #, $, * si @, dar puteti adauga si alte simboluri

<name> ::= (<letter> | <digit> | _)*
// numele poate fi orice combinatie de litere, cifre si "_"

<mt-decl> ::= <name> ::= (<mt>)+ ;;
// se defineste o noua masina Turing ca o concatenare de 1 sau mai multe masini Turing

<mt> ::= [<name>@](<mt-call> | <mt-trans>) | &<name>
// operatorul @ se foloseste pentru a da un nume temporar unei masini Turing; numele este valabil din momentul definirii pana la sfarsitul declaratiei curente de masina Turing
// numele dat cu @ se foloseste cu operatorul & pentru scrierea schemelor de masini Turing care contin cicluri
// nu se pun spatii inainte si dupa @, respectiv dupa &
^R_infinit ::= start@[R] #start ;; // masina care merge mereu la dreapta

<mt-call> ::= [<name>] | [<elementary>]
// se pot apela masini elementare sau masini definite anterior in fisierul sursa
// exemple de apeluri: [L], [R(!#)], [Copy]

<elementary> ::= L | R | <elem> | L([!]<elem>) | R([!]<elem>)
// masinile elementare sunt cele discutate si la seminar
// notatia pentru masinile elementare este compacta (nu contine spatii)
// pentru alfabetul {a, b} masinile elementare sunt L, R, a, b, L(a), L(b), R(a), R(b), L(!a), L(!b), R(!a), R(!b).

<elem> ::= <symbol> | <<name>> | &<name>
// masinile elementare se pot scrie si punand in locul unui simbol numele acestuia: L(<void>)
// de asemenea se poate folosi numele unei "variabile" in care a fost memorat anterior un simbol (detalii despre memorare mai jos, la <set>): L(&x)

<mt-trans> ::= ((<transition>)*)

```
// o masina Turing care foloseste simbolul citit de pe banda este definita ca o lista de tranzitii
// daca lista nu contine nicio tranzitie, obtinem masina Turing care nu face nimic, echivalenta cu starea "halt"
// limbajul nu impune, dar tranzitiile se vor trece de obicei pe linii distincte, indentate fata de linia care contine "("
^inversare ::= (
     {a} -> [b] ;
     {b} -> [a] ;
) ;;

<transition> ::= <set> -> (<mt> )+ ;
// o tranzitie precizeaza intr-o multime toate simbolurile pentru care masina executa aceeasi tranzitie
// limbajul nu impune, dar in general se va pune spatiu inainte si dupa "->", precum si inainte de ";"
// multimea de tranzitii nu acopera in mod necesar toate simbolurile din alfabet; pentru simbolurile neacoperite, se considera ca masina se opreste (nu face nimic)

<set> ::= [<name>@][!](<<name>> | { <element> (, <element> )* })
// modul simplu de a scrie o multime de simboluri consta in enumerarea elementelor din multime: {a, b, c}
// fiecare simbol poate fi inlocuit cu numele sau, care trebuie sa fi fost definit anterior: {<void>, <separator>}
// "memorarea" simbolului citit intr-o variabila se face cu ajutorul operatorului @:
x@{a, b} // se memoreaza in x care dintre simbolurile din multime (a sau b) a fost citit de pe banda
// multimea poate fi inlocuita cu numele ei, definit anterior: <any>

// pentru exemplificarea sintaxei, consideram un exemplu complet: masina Turing care copiaza un sir w. Pornind cu w pe banda si capul de citire in stanga lui w (#w#), sa ajunga la #w#w#
// cel mai simplu (in raport cu sintaxa) mod de a scrie aceasta masina este urmatorul:
alphabet :: a b # ;
Copy ::= start@[R] (
     {a} -> [#] [R(#)] [R(#)] [a] [L(#)] [L(#)] [a] &start ;
     {b} -> [#] [R(#)] [R(#)] [b] [L(#)] [L(#)] [b] &start ;
     {#} -> [L(#)] ;
```

```
) ;;
// orice masina Turing poate fi descrisa cu un subset mai simplu al limbajului pe care l-am definit, adica fara nume de simboluri, fara nume de multimi si fara variabile (limbajul de baza)
// daca adaugam si variabile, obtinem:
alphabet :: a b # ;
Copy ::= start@[R] (
     x@{a, b} -> [#] [R(#)] [R(#)] [&x] [L(#)] [L(#)] [&x] &start ;
     {#} -> [L(#)] ;
) ;;
// putem adauga si nume pentru simboluri si obtinem o varianta mai generica, in sensul ca am putea adauga oricand elemente noi la alfabet fara a fi nevoie sa se modifice codul care descrie masina Turing si am putea de asemenea sa precizam un alt simbol pentru spatiul liber de pe banda. Descrierea seamana destul de bine cu schema corespunzatoare masinii Turing.
alphabet :: a b # ;
void = # ;
Copy ::= start@[R] (
     x@!{<void>} -> [<void] [R(<void>)] [R(<void>)] [&x]
                            [L(<void>)] [L(<void>)] [&x] &start ;
     {<void>} -> [L(<void>)] ;
) ;;
```

Reprezentarea benzii

Banda este nemarginita la stanga si la dreapta. Aceasta se va reprezenta ca un sir de caractere care cuprinde toata informatia utila de pe banda, pozitia capului de citire / scriere, precum si cate un singur # la stanga si la dreapta pentru a marca restul benzii.

Pozitia capului de citire / scriere va fi marcata prin simbolul “>” pozitionat inaintea simbolului de pe pozitia curenta.

Exemplu: daca pe banda avem doua siruri abba si baab, separate prin #, putem avea urmatoarele cazuri:
#>#abba#baab# = capul de citire la stanga primului sir
#>####abba#baab# = capul de citire cateva pozitii mai la stanga
#ab>ba#baab# = capul de citire pe al doilea b din abba
#abba#baab####>## = capul de citire undeva dupa sfarsitul sirului

Specificatii program

Executabilul se va numi **mtx**. Programul va primi 3 parametri din linia de comanda:

- numele fisierului de intrare, cu extensia .mt
- numele masinii Turing care se executa
- configuratia initiala (continutul benzii + pozitia capului)

Rezultatul consta in configuratia finala (continutul benzii + pozitia capului).

De exemplu, pentru masina Copy definita mai sus, presupunand ca fisierul in care a fost definita este input.mt, avem:
./mtx input.mt Copy “#>#abba#” => #>#abba#abba#