Generatore automa caratteristico per grammatiche LALR(1)

Compilazione

Per compilare il programma è sufficiente invocare gcc passando tutti i file "*.c" e "*.h" in "src" sottocartelle incluse; comoda alternativa è utilizzare il Makefile fornito.

```
# Working directory: src

# Manual
gcc main.c alg/lalr.c ..... -o main

# Makefile
make
```

Verrà quindi generato il file eseguibile "main".

Input

Per la generazione dell'automa è necessario fornire una grammatica.

Per semplicità di implementazione ho effettuato le seguenti scelte:

- $V = {[A-Za-z]} \cup \text{VALID_SYMBOLS}$ (lib/utils.h);
- $T = {[a-z]} \cup \text{VALID_SYMBOLS}$ (lib/utils.h);
- S: lo START SYMBOL è identificato dal driver della prima produzione passata in input;
- P: identificato nell'input.

Nota: Qualora si volessero modificare tali assunzioni è necessario rettificare le funzioni e le costanti in "lib/utils.c" e "lib/utils.h", per avere simboli composti da più lettere è indispensabile rivedere buona parte del programma modificando le strutture dati e creando apposite funzioni di confronto e riconoscimento: nulla di che ma ho preferito la semplicità su questo aspetto nella mia implementazione.

File di input

Il programma attende su stdin uno stream di caratteri che identifica un insieme di produzioni.

Esempio:

```
S -> L=R | R
L -> *R | i
R -> L
```

Note e vincoli

- Il programma si preoccupa di eliminare gli spazi nell'input;
- Il separatore tra driver e body delle production è identificato da INPUT_DRIVER_BODY_SEP in "lib/inputParser.h" (default "->");
- Tutti i simboli del vocabolario sono caratteri ASCII stampabili (char standard di C), i caratteri UTF-16/32 potrebbero dare problemi;
- Il simbolo S1 in "lib/utils.h" (default "~") è utilizzato per identificare S' nella produzione canonica S' -> S, non può essere quindi utilizzato come simbolo del vocabolario;
- Il simbolo EPSILON in "lib/utils.h" (default "#") è utilizzato per identificare la epsilon,non può essere quindi utilizzato come simbolo del vocabolario;
- Il simbolo EOW in "lib/utils.h" (default "$") è utilizzato per identificare "fine parola".

Utilizzo

Come già detto in precedenza, il programma si aspetta su stdin una lista di produzioni che si attengano ai vincoli espressi poco fa.

In caso di chiamata errata al programma, riporto l'output fornito:

```
Usage: ./<executable> <flags> < <input> OR ./<executable> <flags>
Where:
    Executable: executable filename (default main);
    Flags: if empty all flags ("rtc") are added otherwise a string composed
by one or more of the following chars:
            'r' for generating raw output in output.txt file;
            't' for generating graph.tex file;
            'c' for console debug output.
    Input: input stream containing the definition of the grammar (usually a
filename).
i.e. ./main tc < inputs/5 OR ./main r < inputs/1
```

Output

I flag 'r' e 't' permettono di generare rispettivamente i file OUT_RAW_FILENAME (default "output.txt") OUT_TEX_FILENAME (default "graph.tex") in "lib/output.h".

L'automa in output sarà del tipo LR(1) merged.

Con il flag 'c' sarà possibile visualizzare a console anche l'automa LR(1) da cui deriva.

Raw output

Formato del file (# = numero):

```
<tot # of states>\n
foreach state
<state #> <NON_FINAL_MARKER|FINAL_MARKER>
    foreach transition
    <transition symbol>EDGE_SYMBOL_STATE_SEP<target state>
```

```
    \n
foreach final state
<state #>
    foreach reducing item (if more than one separated by R_ITEM_SEP)
    <item production driver>R_ITEM_ARROW<item production body> <item
lookahead set (no need for separator since symbols are supposed to besingle
char)>
    \n
\n
<start symbol> <start state #>
```

Le costanti in maiuscolo sono modificabili in "lib/output.h".

Esempio di output.txt:

```
10
0 N S>1 a>2
1 F
2 N A>3 b>4
3 N B>5 b>6 d>7
4 F
5 N e>8
6 N c>9
7 F
8 F
9 F
1 ~->S $
4 A->b bd
7 B->d e
8 S->aABe $
9 A->Abc bd

S 0
```

LaTEX output

Sarà generato un file "graph.tex" composto da:

- Il contenuto del file PRE_FILENAME in "lib/output.h" (default "src/files/pre.tex");
- L'output generato dal programma;
- Il contenuto del file POST_FILENAME in "lib/output.h" (default "src/files/post.tex").

Talvolta la visualizzazione del grafo potrebbe risultare poco comprensibile, in tal caso è opportuno rieseguire il programma, il nuovo grafo sarà leggermente diverso e, forse di più facile comprensione.

Una modalità di visualizzazione del grafo consiste nel convertire il file .tex in .pdf con il comando:

```
pdflatex graph.tex
```

Riguardo il funzionamento del programma

Struct e strutture dati

Strutture dati d'appoggio

La seguenti struct sono strutture dati necessarie all'implementazione dei vari algoritmi e sono:

- Array dinamico che una volta finito lo spazio di allocamento, raddoppia la sua dimensione;
- Hash set di tipo char*, gentilmente preso in prestito da https://github.com/barrust/set;
- Linked list ad-hoc per il programma necessara per la creazione di Graph.

```
typedef struct {
    void** data;
    int used;
    int size;
} List;
void initList(List*, int);
void insertList(List*, void*);
void freeList(List*);

typedef struct  {
    char* _key;
    uint64_t _hash;
} SimpleSetNode, simple_set_node;
typedef struct  {
    simple_set_node **nodes;
    uint64_t number_nodes;
    uint64_t used_nodes;
    set_hash_function hash_function;
} SimpleSet, simple_set;
int set_add(SimpleSet *set, const char *key);
int set_remove(SimpleSet *set, const char *key);
int set_contains(SimpleSet *set, const char *key);
//...etc

typedef struct node {
    int dest;
    char symbol;
    int isFinal;
    struct node* next;
} Node;
typedef struct {
    Node *head;
} LinkedList;
```

Grammar

La seguente struct contiene tutte le production della grammatica fornita in input.

```
typedef List Grammar;
```

Automa

La seguente struct contiene la definizione di un automa, ovvero i suoi stati e le sue transizioni.

```
typedef struct {
    List* nodes;                    //<State>
    List* transitions;              //<Transition*>
    SimpleSet* transitions_keys;    //<Transition*>
                                    /**
                                     * transitions_keys contain only
τ(from,symbol) for OPTIMIZATION
                                     */
} Automa;

void initAutoma(Automa*, int initLen); //initLen: initial length of nodes
and transitions
```

Graph

La seguente struct è un'alternativa per la memorizzazione di un automa, comoda per fornire l'output e capire come sono collegati gli stati dell'automa stesso.

```
typedef struct {
    int nodesCount;
    int* finals;
    LinkedList* nodes;
} Graph;

Node* createNode(int n, char symbol, Node* next); //Notice: the link is
added at the beginning of the LinkedList
Graph* createGraph(int n, int* finals);
void addEdge(Graph* g, int src, int dest, char symbol);
void printGraph(Graph* g);
```

State

La seguente struct identifica uno stato dell'automa: una raccolta di item LR(1), suddivisi in kernel e la sua espansione. Tale struttura è fondamentale per il funzionamento dell'algoritmo.

```
typedef struct {
    List* items;   //<LR1item>
    int kernelSize;
} State;
```

```
void printState(State*, int);
State* createState(int initSize);
void destroyState(State*);
int sameKernel(State*, State*, int onlyLR0);
/**
 * returns: TRUE if i is equals to one of the kernel expansion items
 */
int kernelExpansionContains(State* s, LR1item* i, int onlyLR0);
/**
 * returns: TRUE if exists an item has the marker at the end of the body of
the production
 */
int isFinal(State*);
```

LR(1) item

La seguente struct identifica un item di tipo LR(1).

```
typedef struct {
    Production* p;
    int marker;
    SimpleSet* ls; //Lookahead Set
} LR1item;

void printItem(LR1item*);
LR1item* createItem(Production*,int);
char getMarkedSymbol(LR1item*);
int itemsEqual(LR1item*, LR1item*, int onlyLR0);
int markerAtTheEnd(LR1item*); //returns: TRUE if the marker is at the end
of the body of the production
void destroyItem(LR1item*);
```

Transition

La seguente struct rappresenta un collegamento fra stati dell'automa.

```
typedef struct {
    int from;
    int to;
    char symbol;
} Transition;

Transition* createTransition(int from, int to, char symbol);
/**
 * returns: if(onlyKey) "<from> <symbol> <to>"
 *          else        "<from> <symbol>"
 */
char* serializeTransition(Transition*, int onlyKey);
```

```
/**
 * Parses a string like "<from> <symbol> <to>"
 */
Transition* parseTransition(char*);
void printTransition(Transition*);
```

Algoritmi

Grazie a tutte le struct e strutture dati appena descritte è ora possibile implementare gli algoritmi che generano gli automi LR(1) e LR(1)m per la grammatica di input, per maggiori dettagli sull'implementazione fare riferimento al file "alg/lalr.c".

```
/**
 * Computes ONLY the closure0 for the state s
 */
void closure0(Grammar* g, State* s);
/**
 * Computes the closure0 for the state s and updates the lookahead sets
 */
void closure1(Grammar* g, State* s);
/**
 * returns: a LR(1) automa for the grammar g
 */
Automa* generateLR1automa(Grammar* g);
/**
 * returns: a NEW LRm(1) automa from the lr1A automa
 * Notice: items are referenced (NOT COPIED)
 */
Automa* generateLR1Mautoma(Automa* lr1A, Graph* lr1G);
```

Funzionamento

Una volta create le strutture dati e gli algoritmi, il funzionamento del programma è chiaro e semplice:

- Parsing della grammatica ricevuta come input;
- Creazione del automa LR(1) e relativo grafo;
- Creazione del automa LR(1)m (derivato dal precedente) e relativo grafo;
- Output richiesti.

```
Grammar* g = parseInput(stdin);

Automa* lr1A = generateLR1automa(g);
Graph* lr1G = createGraphFromAutoma(lr1A, flags & CONSOLE_OUTPUT, "\nLR(1)
automa:\n\n");

Automa* lr1mA = generateLR1Mautoma(lr1A, lr1G);
Graph* lr1mG = createGraphFromAutoma(lr1mA, flags & CONSOLE_OUTPUT,
"\nLR(1)m automa:\n\n");
```

Extra

Ho provato ad ottimizzare il confronto tra kernel di stati calcolando un hash basato sulle componenti variabili di ogni stato. Non sono purtroppo riuscito a creare una funzione di hash che non abbia collisioni e sia affidabile.

E' possibile utilizzare questa funzione aggiungedo il carattere 'o' ai flag.

Le funzioni di hash testate sono in "structs/LR1item: getItemHash". Qualora si riesca ad implementare una funzione affidabile il programma eseguito con il flag 'o' dovrebbe fornire output corretti e con tempi di esecuzione più rapidi.