Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Teoría de Lenguajes

Segundo Cuatrimestre del 2011

Trabajo Práctico

Integrante	LU	Correo electrónico
Ezequiel Castellano	161/08	ezequielcastellano@gmail.com
Mariano Semelman	143/08	marianosemelman@gmail.com

Índice

1. Gramática

Optamos por describir la gramática de manera tal que no tenga conflictos de ningún tipo (reduce/reduce o shift/reduce). A su vez no perdimos ni agregamos poder de expresividad, ya que de esta manera podemos representar las mismas cadenas que antes.

La gramática utilizada es la siguiente:

G = < {Expression},{caracter , |, *, +, ?, ., (,)},P,E >

P:

```
Expression --> Term
    | Term Expression
    | Term | Expression


Term --> Term +
    | Term ?
    | Term *
    | Operand

Operand --> Parenthesis
    | Character

Character --> caracter
    | .

Parenthesis --> (Expression)
```

2. Implementación

La implementación fue realizada en el lenguaje C++, utilizando la herramienta Bison.

Para la implementación utilizamos la estructura Autómata, que internamente es un grafo dirigido y etiquetado, por eso su nombre LDGraph (Labeled Derived Graph).

En la clase Autómata se implementaron todos los métodos necesarios para poder concatenar autómatas, determinizarlos, mostrarlos, aplicar un operador o matchear un string.

Por otra parte la clase LDGrafo consta de un estado de inicial, un conjunto de estados finales y un vector de transiciones como estructura.

Donde las transiciones son un mapa donde la clave es la etiqueta de la transicion y el significado es el conjunto de estados que se pueden alcanzar por esa transción.

El grafo permite ver si se puede mover a un estado consumiendo un caracter, moverse por una transicion, determinizarse y ver si se encuentra en un estado de aceptación, entre sus operaciones más destacables.

Para ver más detalle sobre las estructuras en el apéndice se encuentra detallada la implementación.

3. Información y requerimientos de software

En esta sección se indicarán versiones, herramientas, compiladores y todo lo necesario para la realización del trabajo práctico.

3.1. Requerimientos

El trabajo práctico fue implementado sobre Ubuntu 11.05. Las siguientes instrucciones son válidas en cualquier sistema operativo Linux. De utilizar otro sistema operativo realizar los pasos análogos.

Es necesario tener instaladas las librerias para compilar en C++ y Bison.

Para instalar las librerias de Build Essencial (incluye librerias para compilar C++): sudo apt-get install build-essential manpages-dev.

Para instalar las librerias de Bison: sudo apt-get install bison bison-doc.

(Opcional): Para poder renderizar los archivos .dot es necesario tener instalado el ejecutable dot.

Para instalarlo: sudo apt-get install graphviz.

3.2. Compilar

Utilizar el MakeFile provisto con el código, el cual se encuentra detallado en el apéndice.

- Para compilar solamente: make.
- Para compilar los tests: make build-test.
- Para compilar los tests y ejecutarlos: make test.
- Para remover los archivos compilados: make clean.
- Para generar el gráfico de la gramática: make bison (La imagen se encuentra en ./src/graph/grammar.png).

3.3. Ejecutar

Para ejecutar el programa se deberá estar posicionado en el directorio donde se encuentre el ejecutable, es decir el directorio donde se compiló.

El archivo ejecutable se llama grep-line y para utilizarlo deberá escribirse la siguiente sentencia: ./grep-line regexp [file] .

El primer parámetro es la expresion regular, mientras que el segundo parámetro opcional es un archivo con las cadenas a matchear, en caso de no utilizarse matchea por entrada standard.

Para ver gráficos del autómata generado se deberá agregar el flag -i.

//TODO: ¿Completar como armamos las imágenes?

4. Casos de Prueba

//TODO: Estos fueron los que usamos al comienzo para ver que la gramática sea válida, falta completar en que casos la debería aceptar y en cuales no.

Las casos de prueba utilizados para validar la gramática fueron los siguientes:

```
casa
(casa)
a?
(a)?
(a?)
casa|perro
(casa)|(perro)
(casa)|perro
casa|(perro)
casa?|perro?
(casa?)|(perro)?
(casa)?|perro?
casa?|(perro)?
a+?
(asd?)?
asd|asd
```

5. Resultados

//TODO: Completar cuales fueron los resultados obtenidos.

6. Conclusiones

//TODO: Completar las conclusiones.

Nos fue muy útil para chequear que la gramática no tenga conflictos el tener una herramienta como Bison.

7. Apéndice

7.1. Makefile

```
CXXFLAGS = -g -Wall -Wextra -std=c++0x
LDFLAGS = $(CXXFLAGS)
ABSOBJ = automata grep-line.tab main matcher graph graph_utils tests/test-automata tests/te
SOURCES= $(addsuffix .cpp, $(ABSOBJ))
OBJ = $(addsuffix .o, $(ABSOBJ))
BISON = grep-line.ypp
MAIN_OBJ = automata.o grep-line.tab.o main.o matcher.o graph.o graph_utils.o debugging.o
GRAPH_TEST_OBJ = graph.o graph_utils.o tests/test-graph.o debugging.o
AUTOMATA_TEST_OBJ = automata.o graph.o graph_utils.o tests/test-automata.o debugging.o


grep-line: bison-build $(MAIN_OBJ)
	$(CXX) $(CXXFLAGS) $(MAIN_OBJ) -o $@

clean:
	rm -f *.o tests/*.o
	rm -f graph/*.png graph/dot/*.dot
	rm -f grep-line.tab.*
	rm -f grep-line test-automata test-graph
	rm -f deps

bison-build: $(BISON)
	bison -d $(BISON)

bison:$(BISON)
	bison -d -ggraph/dots/grammar.dot $(BISON)
	dot graph/dots/grammar.dot -Tpng -o graph/grammar.png

grep-line.tab.cpp: bison-build

deps:
	$(CXX) $(CXXFLAGS) -MM -MP $(SOURCES) > $@

-include deps



test-graph: $(GRAPH_TEST_OBJ)
	$(CXX) $(CXXFLAGS) $^ -o $@

test-automata: $(AUTOMATA_TEST_OBJ)
	$(CXX) $(CXXFLAGS) $^ -o $@

build-test: grep-line test-automata test-graph
```

```
test: build-test
	./tests/run-test-regexp
	./test-automata
	./test-graph
```

7.2. Autómata

7.2.1. Headers

```
#ifndef __AUTOMATA__
#define __AUTOMATA__

#include <string>
#include <map>
#include <set>
#include "graph.hpp"

#define DOT 1
#define LAMBDA 2
#define CHAR 3

using namespace std;

class Automata {
	public:
		Automata();

		Automata(char);

		void operator|=(Automata & other);
		//Concatenate two automata
		void operator+=(Automata & other);
		//Apply unary operator
		Automata & apply_op(const char cs);

		//Builds an anychar automata
		void determinize();
		//Match a string
		bool match(string);

		Automata & operator = (const Automata & other);

		void mostrar(ostream&) const;


		int id;
```

```
        protected:
            static int getId() { return global_id++; }

        private:
            static int global_id;
            LDGraph graph;

};

//~ int global_id = 0;

ostream& operator <<(ostream& o, const Automata & a);
#endif // __AUTOMATA__
```

7.2.2. Impementación

```
#include "automata.hpp"
#include <iostream>
#include <cassert>
#include <map>
#include <string>

using namespace std;

int Automata::global_id = 0;


Automata::Automata() : graph('/') {
    id = getId();
}


//Builds an alfanum automata
Automata::Automata(char c) : graph(c) {
    id = getId();
}

Automata & Automata::operator =(const Automata & other) {
    if(this != &other) {
        graph = other.graph;
        id = other.id;
    }
    return *this;
}

//determinize the automata
void Automata::determinize(){
    graph.determinize();
}
```

```
// ——————————  operators ——————————————————————————————————————

void Automata::operator|=(Automata & other){
    if(this == &other)
        return;
    graph |= other.graph;
    //~ determinize();
    return;
}
//Concatenate two automata
void Automata::operator+=(Automata & other){
    graph += other.graph;
    return;
}

//Apply unary operator
Automata & Automata::apply_op(const char c){
    if(c == '?' || c == '*'){
        // Enable not to match this automata.
        graph.add_jump();
    }

    if (c == '+' || c == '*'){
        // Enable to match  this automata many times.(self loop)
        graph.add_inverse_jump();
    }
    //~ determinize();
    return *this;
}

//Match a string
bool Automata::match(string s){
    State actual = graph.init();
    string::iterator c = s.begin();
    while( c != s.end() and graph.can_move(actual, *c) ) {
        Dstate t;
        graph.move_state(actual, *c, t);
        //MUST BE A DETERMINISTIC FINITE AUTOMATA
        assert(t.size() == 1);
        actual = *(t.begin());
        c++;
    }
    // Estado final y cadena consumida.
    return graph.is_accepted(actual) and c == s.end();
}



//——————————— DISPLAY ——————————————————————————————————
```

```
void Automata::mostrar(ostream & o) const {
    o << graph;
}
ostream& operator <<(ostream& o, const Automata & a) {
    a.mostrar(o);
    return o;
}
```

7.3. Grafo

7.3.1. Headers

```
#ifndef __GRAPH__
#define __GRAPH__
#include <map>
#include <vector>
#include <stack>
#include <set>
#include <iostream>


using namespace std;

typedef unsigned int State;
typedef map<char, set<State> > Trans;
typedef Trans::const_iterator mit;
typedef set<State> Dstate;

//Labeled Directed Graph
class LDGraph {

    public:
        LDGraph(char c);
        void operator |=(const LDGraph &);
        void operator +=(const LDGraph &);
        LDGraph & operator =(const LDGraph &);

        void add_jump();
        void add_inverse_jump();
        // Determinize
        void determinize();

        //observers
        State init() const;
        bool is_accepted(State state) const;
        void mostrar(ostream &) const;
        bool can_move(State state, char c) const;
        void move_state(State state, char c, Dstate &) const;
```

```
    private:
        void move_dstate(const Dstate & dstate, Dstate &state, char c) const;
        void dtran(const Dstate & dstate, Dstate & res, char c) const;
        void clausura_lambda_dstate(Dstate & dstate) const;
        void clausura_lambda_state(State state, Dstate & res) const;
        void add_transition(State o, State d, char t);
        void add_transition(State o, Dstate d, char t);
        void add_transition(Dstate origins, State destiny, char t);
        void add_transition(Dstate origins, Dstate destinies, char t);
        set<char> available_trans(const Dstate & s) const;
        unsigned int  copy_states(const LDGraph &);

        //Fields
        vector< Trans > _rels;
        State _init;
        Dstate _tail;



};
ostream& operator<<(ostream & o, const LDGraph& g);
#endif //__GRAPH__
```

7.3.2. Impementación

```
#include "graph.hpp"
#include "graph_utils.hpp"
#include <cassert>
#include <stack>
#include <initializer_list>
#include <algorithm>

using namespace std;



#define INIT 0
#define TAIL 1

LDGraph::LDGraph(char c) : _rels(2) {
    _init = INIT;
    _tail = Dstate({TAIL});
    if( c == '.') {
        //Graph does not support '.' transition as is
        for( char b = '0'; b < '9' + 1; b++) {
            add_transition(INIT, TAIL, b);
        }
        for( char b = 'A'; b < 'Z' + 1; b++) {
```

```
            add_transition(INIT, TAIL, b);
        }
        for( char b = 'a'; b < 'z' + 1; b++) {
            add_transition(INIT, TAIL, b);
        }
        add_transition(INIT, TAIL, '␣');
    } else {
        add_transition(INIT, TAIL, c);
    }
}

LDGraph & LDGraph::operator =(const LDGraph & other) {
    //~ cout << "Copying" << endl;
    if(this != &other) {
        _rels.clear();
        assert(copy_states(other) == 0);
        _init = other._init;
        _tail = other._tail;
    }
    return *this;
}



void LDGraph::operator |=(const LDGraph & other){
    if(this == &other)
        return;
    int offset = copy_states(other);
    add_transition(_init, other._init + offset, '/');
    add_transition(offset_dstates(other._tail, offset), _tail, '/');
}

void LDGraph::operator +=(const LDGraph & other){
    unsigned int offset = copy_states(other);
    add_transition(_tail, other._init + offset,  '/');
    _tail = offset_dstates(other._tail, offset);
}

/** Given a LDGraph, it appends all its states to this
 * renaming them with new ids.
 * returns the beggining of the new states*/
unsigned int LDGraph::copy_states(const LDGraph & other) {
    unsigned int offset = _rels.size();
    unsigned int osize = other._rels.size();
    _rels.resize(osize + offset);
    for(unsigned int i = 0; i < osize; i++) {
        Trans actual = other._rels[i];
        for(mit it = actual.begin(); it != actual.end(); it++) {
            char k = it->first;
            for(Dstate::iterator sit = it->second.begin();
```

```
                sit != it->second.end();
                sit++) {
                int v = *sit + offset;
                add_transition(i + offset, v, k);
            }
        }
    }
    return offset;
}

void LDGraph::add_jump(){
    add_transition(_init, _tail, '/');
}

void LDGraph::add_inverse_jump(){
    add_transition(_tail, _init, '/');
}

void LDGraph::add_transition(State origin, State destiny, char t) {
    _rels[origin][t].insert(destiny);
}

void LDGraph::add_transition(State origin, Dstate destinies, char t) {
    for(Dstate::iterator it = destinies.begin(); it != destinies.end(); it++)
        add_transition(origin, *it, t);
}

void LDGraph::add_transition(Dstate origins, State destiny, char t) {
    for(Dstate::iterator it = origins.begin(); it != origins.end(); it++)
        add_transition(*it, destiny, t);
}

void LDGraph::add_transition(Dstate origins, Dstate destinies, char t) {
    for(Dstate::iterator it = origins.begin(); it != origins.end(); it++)
        add_transition(*it, destinies, t);
}

void LDGraph::dtran(const Dstate & dstate, Dstate & res, char c) const {
    move_dstate(dstate, res, c);
    clausura_lambda_dstate(res);
}

void LDGraph::clausura_lambda_dstate(Dstate & dstate) const {
    for(set< State >::iterator it = dstate.begin();
        it != dstate.end();
        it++) {
        clausura_lambda_state(*it, dstate);
    }
}
void LDGraph::clausura_lambda_state(State state, Dstate & res) const {
```

```
    res.insert(state);
    for(mit it = _rels[state].begin(); it != _rels[state].end(); it++){
        if(it->first == '/' )
            for( set<State>::const_iterator sit = it->second.begin();
                 sit != it->second.end();
                 sit++) {
                if(not res.count(*sit))
                    clausura_lambda_state(*sit, res);
            }
    }
}
void LDGraph::move_state(State state, char c, Dstate & res) const {
    //assert(can_move(state, c));
    Dstate follow = _rels[state].find(c)->second;
    for( Dstate::iterator dit = follow.begin();
        dit != follow.end();
        dit++) {
        res.insert(*dit);
    }

    return ;
}

bool LDGraph::can_move(State state, char c) const {
    return _rels[state].count(c);
}
void LDGraph::move_dstate(const Dstate & dstate, Dstate & res, char c) const {
    assert(c != '/');
    for(Dstate::const_iterator it = dstate.begin();
            it != dstate.end();
            it++) {
        if(can_move(*it, c))
            move_state(*it, c, res);
    }
}

set< char> LDGraph::available_trans(const Dstate & s) const {
    set<char> res;
    for(Dstate::const_iterator st = s.begin(); st != s.end(); st++) {
        Trans m = _rels[*st];
        for(mit it = m.begin(); it != m.end(); it++) {
            if(it->first != '/')
                res.insert(it->first);
        }
    }
    return res;
}

State LDGraph::init() const {
    return _init;
```

```
}

bool LDGraph::is_accepted(State t) const {
    return _tail.count(t);
}



void LDGraph::determinize() {
    Dstate initial({_init});
    clausura_lambda_dstate(initial);
    vector< Dstate > dstates({initial});
    vector< Trans > relations(1);
    stack< State > pending_dstates( {0} );
    Dstate new_tail;
    while( not pending_dstates.empty() ) {
        // pop pending set of states to determinize.
        State d = pending_dstates.top();
        pending_dstates.pop();
        //For each available transition
        set<char> chars = available_trans(dstates[d]);
        for(set<char>::iterator chs = chars.begin(); chs != chars.end(); chs++) {
            Dstate s;
            //Apply Dtran(A, c) A = set of states; c = transition;
            dtran(dstates[d], s, *chs);
            // Save the translation of this new state.
            relations[d][*chs] = find_or_create(dstates, relations, pending_dstates, s);
        }
        //IF this state contains a end state, i'll add it to the end.
        add_to_tail(dstates[d], new_tail, _tail, d);
    }
    _rels = relations;
    _tail = new_tail;
}


void LDGraph::mostrar(ostream & o) const {
    o << "digraph  {" << endl;
    for(unsigned int i = 0; i < _rels.size(); i++) {
        o << i;
        o << "[ ";
        o << ((i == _init)? "label=\"Init\"" : " ");
        o << ((_tail.count(i))? "style=\"filled\"" : " ");
        o << " ]";
        o << endl;
        mit it;
        for(it = _rels[i].begin();
            it != _rels[i].end();
            it++) {
            for( Dstate::iterator sit = it->second.begin();
```

```
                sit != it->second.end();
                sit++) {
                o << i;
                o << " -> ";
                o << *sit;
                o << "[ label=\"";
                o << it->first;
                o << "\" ]" << endl;
            }
        }
    }
    o << '}' << endl;
}


ostream& operator<<(ostream & o, const LDGraph& g) {
    g.mostrar(o);
    return o;
}
```

7.4. Gramática

```
/* Grep line.  */


%{
#include "automata.hpp"
#include "define.hpp"
#include "debugging.hpp"
%}

%token ALFANUM


%%/* Grammar rules and actions follow.
    * Bison supplies a default action for each rule: $$ = $1. */

input: expression       { automata = $1; guardar($1, "5Final");}

expression: term            {guardar($1, "3term"); $$ = $1; cout << "Paso b" << endl;}
    | term expression       { $$ = $1; $$ += $2;  }
    | term '|' expression   { $$ = $1; $$ |= $3; guardar($1, "3pri"); guardar($3, "4seg");
;

term: term '+'          { $$ = $1.apply_op('+');}
    | term '?'          { $$ = $1.apply_op('?');}
    | term '*'          { $$ = $1.apply_op('*'); }
    | operand           { $$ = $1; guardar($$, "2Operand");}
```

```
;

operand: parenthesis
    | character    { guardar($1, "1Char"); $$ = $1;}
;

character: ALFANUM
    | '.'                  { $$ = Automata('.'); }
;

parenthesis: '(' expression ')'
;



%%
```