CC473: Programming Languages Translation - Spring 2020

Project Report
Lexical Analyzer Generator

Youssef Sherif ID: 4319

Effat Hassan ID: 4149

Mohamed Khamis ID: 3823

This report contains a high level overview of the architecture and major decisions taken throughout all phases.

For more details about the algorithms used, please check the code itself as it is heavily commented.

Phase 1

In this phase we discuss the decisions taken by us in the project as well as the basic architecture of the project. We also discuss all the algorithms implemented.

This project is split into 5 main components.

1) Lexical Grammar Rules file Parser.
2) RegEx parser RegEx toNFA converter.
3) NFA Thompson's construction.

4) DFA subset construction.
5) DFA input matcher.

Lexical Grammar Rules Parser

In this part we read the input file lexical_rules.txt line by line and create 4 data structures:

Map<String, String> **regularExpressions**;

- Key is the value before the colon symbol and the regEx string is the value after the colon symbol.

Map<String, String> **regularDefinitions**;

- Key is the value before the equal symbol and the regEx string is the value after the equal symbol.

Set<String> **keyWords**;

- Set of keywords between curly braces { }.

Set<String> **punctuation**;

- Set of punctuation between square brackets [].

Regular Expression Parser and RegEx to NFA converter

This component is responsible for parsing the input RegEx string and to call the NFA Thompson Construction functions to create the NFA.

This component depends:

Part Class:

public Part(String type, String expression) :- constructor

Part Types:-

public static final String ***GROUP*** = **"GROUP"**;

public static final String ***DEF*** = **"DEF"**;

public static final String ***NOOP*** = **"NOOP"**;

Part methods:-

public String getExpression() :- returns the expression as it is

public char getNFACharacter() :- we call this function when the part becomes the smallest part we can possibly have for example ‘a’ or ‘*’ and if it encounters an escaped character it returns charAt(1(else it returns charAt(0)

public Boolean isAsterisk() :- returns true if part is asterisk

public Boolean isPlus() :- returns true if part is plus

public boolean isDefinition() :- returns true if part is definition

public void setAsterisk() :- sets part as an asterisk part

public void setPlus() :- sets part as plus part.

PartFactory Class:

This class creates instances of Part objects.

It has the following Part creation methods:

public PartFactory(Map<String, String> regularDefinitionsNames,

Set<String> keyWords,

Set<String> punctuation) :- constructor

public Part createPart(String expression) :- Checks if it ends in ‘*’ or ‘+’ and also if it is a definition and returns the correct part accordingly.

public Part createGroupPart(String expression, **char** parenthesisPostfix) :- checks the postfix for '*' or '+' and creates a group part and returns it.

public Part createNoOpPart(String expression) :- we still can't decide what type this part is so we create it with the NOOP tag so we can work on it later.

RegEx Functions:

public String preProcess(String regExString)

- We preprocess the string by replacing all '\' characters with ' \' (space followed by backwards slash). The reason we do this is to handle concatenations like '\=\=' and '>\=' etc.
- Also we append '|' at the beginning. This does not affect the regEx but is added to make it work for regEx made of a single character.

private List<Part> findOredParts(String regExString)

- This function takes an input regEx and returns a List of Part objects that will be Ored together. The function iterates character by character and if it finds a '|' it creates a Part with value of previous characters. This loop ignores '|' inside groups since they will be handled later.

private List<Part> findGroupedParts(String regExString)

- This function iterates on the regEx character by character and combines all characters within '(' and ')' into a single part and appends any characters before into its own Part and any characters after to its own Part as well. It also has a parenthesis counter to take the correct group. Any groups within other groups will be handled recursively during conversion to NFA later on.

private List<List<Part>> tokenizeParts(List<Part> tokens)

- This function takes the List of Parts that we obtained from tokenizeOrs and tokenizeGroups and combines them into a List of Lists. It adds groups separately but in the correct sequence to deal with them as a whole Part.

public NFA toNFA(String regExString)

- This is the main toNFA function which takes a String of regEx and tokenizeOrs on the String. It loops on the resulting List<Part> and calls the internal private toNFA function. It then Ors the resulting NFA into one large NFA and returns it.

private List<NFA> getConcatenatedNFAsList(String expression)

- This is the internal toNFA function.
- It splits the expression on white spaces.
- Checks if the part is a definition and if it is calls the replaceRange function and converts the String result into NFA recursively
- If the part is asterisk or plus it calls the NFA functions to update the NFA.
- Adds the results to a list of NFAs to concatenate

private String replaceRange(String string)

- This function iterates character by character on strings like 0 - 9 and a - z and converts them into 0 | 1 | 2 | 3 ... | 9 and a | b | c | .. | z using ascii values.

NFA Thompson's Construction

This algorithm depends on NFAState.

NFAState class:

private int numStates;

boolean finalState;

ArrayList<NFAState> **next**;

ArrayList<Character> **edges**;

public NFAState(**boolean** finalState, **int** numStates) :- constructor

public int getStateNo() :- returns NFA state number.

NFA class:

public static char *EPSILON* = **'Ɛ'**;

private static NFA *nfa*;

private NFAState **startt**;

private NFAState **finall**;

private Set<Character> **alphabet**;

private int numStates;

public static NFA getInstance() :- returns a singleton NFA object.

public NFA edge(Character attr):- creates a new edge with 2 new unique states by incrementing the singleton NFA state number. It creates a temporary NFA with the 2 states.

public static NFA combineNFAsOr(List<NFA> nfalist) :- ORs the nfalist and returns the resulting NFA.

- Creates 2 new states (start and final) and loops on the list with index i.
    - Add EPSILON edge to start
    - Add nfalist.get(i).**startt** to start .next
    - Add final state to nfalist.get(i).**finall**.**next**

public static NFA combineNFAsConcatenate(List<NFA> nfalist) :- Concatenates the nfalist and returns the resulting NFA.

- Concatenate function does not create any new states.
- It loops on the nfalist with index i to nfalist.size()-1:
    - nfalist.get(i).**finall**.**edges**=nfalist.get(i+1).**startt**.**edges**;
    - nfalist.get(i).**finall**.**next**=nfalist.get(i+1).**startt**.**next**;
    - nfalist.get(i).**finall**.**finalState** = **false**;
- NFA.getInstance().**finall** = nfalist.get(size-1).**finall**;
- **return** NFA.getInstance()

public NFA plus(NFA nfa) :- plus is a combination of concatenation and asterisk operation.

- Create a temporary list of NFAs
- Append nfa.asterisk(nfa)
- Return nfa.concatenate(nfa)

public NFA asterisk(NFA inputnfa)

- Create 2 new states startState and finalState
- startState.**next**.add(inputnfa.**startt**);
- startState.**edges**.add(*EPSILON*)
- startState.**edges**.add(*EPSILON*);
- startState.**next**.add(finalState);
- inputnfa.**finall**.**edges**.add(*EPSILON*);
- inputnfa.**finall**.**next**.add(inputnfa.**startt**);
- inputnfa.**finall**.**edges**.add(*EPSILON*);
- inputnfa.**finall**.**next**.add(finalState);
- *nfa*.**startt** = startState;
- *nfa*.**finall** = finalState;
- **return** *nfa*;

DFA: Subset Construction Algorithm:

The algorithm converts NFA to DFA depending mainly on 2 main algorithms, the epsilon closure and move algorithms.

DFA State has the following charateristics:

- private boolean finalstate;
- private ArrayList<NFAState> collectionStates;
- private Integer id;
- private boolean marked;
- private HashMap<DFAState, Character> TransTable;

Functions supported for DFA state:

- public void addCollectionState(NFAState inputState) *[adds NFA state to collection of states forming that DFA state]*
- public ArrayList<NFAState> getCollectionStates()
- public boolean isCollectionState(NFAState inputState)
- public void mark()
- public boolean isMarked()
- public void assignNextState(DFAState nextState, Character inputSymbol) *[Set transition for DFA state]*
- public Integer getID()

As for the DFA, it consists of DFA states and a transition table describing transitions between these states.

The DFA has the following characteristics:

- private DFAState DFAStart;
- private Set<Character> alphabet;
- public static char EPSILON = 'ε';
- private ArrayList<DFAState> DFAStates;
- private int stateCounter = 0;
- private Table<Integer, Character, Integer> DFATransitions;
- private NFA Nfa;

The code for converting the NFA to DFA follows the following:

```
initially, ϵ-closure(s0) is the only state in Dstates and it is unmarked;
while there is an unmarked state T in Dstates do begin
        mark T;
        for each input symbol a do begin
                U := ϵ-closure(move(T, a));
                if U is not in Dstates then
                        add U as an unmarked state to Dstates;
                Dtran[T, a] := U
        end
end
```

The epsilon closure follows this:

```
push all states in T onto stack;
initialize ϵ-closure(T) to T;
while stack is not empty do begin
        pop t, the top element, off of stack;
        for each state u with an edge from t to u labeled ϵ do
                if u is not in ϵ-closure(T) do begin
                        add u to ϵ-closure(T);
                        push u onto stack
                end
end
```

The move operation iterates through all the NFA states forming a DFA state and for each NFA state it iterates through all possible edges for that state and checks if that edges matches a certain input if so then it adds the NFA state pointed to by that edge into a collection of NFA states forming a new DFA state. This is repeated for all the input symbols in an alphabet in an effort to map out the whole NFA, reduce redundancies and form the required DFA in a format that is acceptable. As this operation goes on a table representing all possible transitions where each DFA state is matched against all possible inputs in the alphabet of the language and when there is a valid transition it is indicated.

Phase 2

In this phase we implement the parser generator.

The parser generator consists of the following parts.

CFG class:

This class contains the following constructors

```
public CFG(CFGAmbiguity ambiguity)

public CFG(CFGRulesFile cfgRulesFile)
```

And the following attributes

```
private Map<String, Set<String>> first;

private Map<String, Set<String>> follow;

private Table<String, String, List<String>> parsingTable;
```

And the following functions

```
public void createLL1Table(Map<String, Set<String>> first,
                           Map<String, Set<String>> follow)

public void createFirstAndFollowSets()

public List<List<String>> getRuleByKey(String key)

public boolean containsKey(String key)
```

It can either take a CFGRulesFile or a CFGAmbiguity.

CFGRulesFile class:

public CFGRulesFile(String fileName)

Reads the file path and creates

private final Map<String, String> cfgRules

which is a hashmap of the rule and its production

CFGAmbiguity interface:

An interface with the following method.

List<CFGEntry> solve();

We create separate classes for each ambiguity for example:

public class LeftFactoring implements CFGAmbiguity

public class LeftRecursion implements CFGAmbiguity

Each class solves a given problem by calling the solve method.

Constants class:

Shared constants across all parser functions

```
public static final String EPSILON = "ε";

public static final String DOLLAR_SIGN = "$";
```

LL1 class:

This class contains the following functions

```
public Set<String> first(List<List<String>> rule)

public Set<String> follow(String nonTerminal,

                Map<String, Set<String>> follow)

public List<List<String>> parse(List<String> tokens,

            Table<String, String, List<String>> parsingTable,

            List<CFGEntry> productions,

            Set<String> terminals) throws Exception
```

Phase 3

We wrote the lexical rules and the grammar rules using flex and bison.

Flex Rules

```
%x ML_COMMENT


letter   [a-zA-Z]
digit    [0-9]
digits   {digit}+
ID       {letter}({letter}|{digit})*
NUM              {digit}+|{digit}+"."{digits}(\L|"E"{digits})

%%


"char"           { return CHAR; }
"int"            { return INT; }
"float"          { return FLOAT; }
"double"         { return DOUBLE; }
"if"             { return IF; }
"else"           { return ELSE; }
```

```
"while"          { return WHILE; }

"for"            { return FOR; }



"+"|"-"                  { return ADDOP; }

"*"|"/"                  { return MULOP; }

"++"|"--"                { return INCR; }

"||"                     { return OROP; }

"&&"                     { return ANDOP; }

"=="|"!=""">"|">="|"<"|"<="   { return RELOP; }



"("              { return LPAREN; }

")"              { return RPAREN; }

"{"              { return LBRACE; }

"}"              { return RBRACE; }

";"              { return SEMI; }

"."              { return DOT; }

","              { return COMMA; }

"="              { return ASSIGN; }
```

```
{ID}            {
                // insert identifier into symbol table
                insert(yytext, strlen(yytext), UNDEF, lineno);
                return ID;
        }
{NUM}           { return NUM; }



"\n"            { lineno += 1; }
[ \t\r\f]+      /* eat up whitespace */

.       { yyerror("Unrecognized character"); }

%%
```

Bison Rules

Define Tokens

```
%token <cval>   CHAR
%token <ival>   INT
%token <fval>   FLOAT
```

```
%token <dval>   DOUBLE

%token <idval>  IDENTIFIER

%token <infixval> OROP

%token <infixval> ANDOP

%token <infixval> RELOP

%token <infixval> ADDOP

%token <infixval> MULOP


%token IF

%token ELSE

%token WHILE

%token FOR

%token CONTINUE

%token BREAK

%token VOID

%token RETURN

%token INCR

%token LPAREN

%token RPAREN

%token LBRACE

%token RBRACE
```

```
%token SEMI

%token DOT

%token COMMA

%token ASSIGN

%token ICONST

%token FCONST

%token CCONST

%token STRING

%token NUMBER


%type <s_type>          METHOD_BODY

%type <p_type>          PRIMITIVE_TYPE

%type <decl_type> DECLARATION

%type <expr_type> EXPRESSION

%type <asgn_type> ASSIGNMENT

%type <stmt_type> STATEMENT

%type <stmt_type> STATEMENT_LIST

%type <stmt_type> IF_STATEMENT

%type <stmt_type> WHILE_STATEMENT

%type <infix_type>  INFIX_OPERATOR
```

Define Start

%start METHOD_BODY

Define Rules

METHOD_BODY:

METHOD_BODY:

STATEMENT_LIST {$$;}

;

STATEMENT_LIST:

STATEMENT {$$;}

| STATEMENT_LIST STATEMENT {$$;}

;

STATEMENT:

DECLARATION {$$;}

| IF_STATEMENT {$$;}

| WHILE_STATEMENT {$$;}

| ASSIGNMENT {$$;}

;

DECLARATION: PRIMITIVE_TYPE IDENTIFIER SEMI {$$;};

PRIMITIVE_TYPE:

INT {$$;}

| FLOAT {$$;}

;

IF_STATEMENT: IF LPAREN EXPRESSION RPAREN LBRACE STATEMENT RBRACE ELSE LBRACE STATEMENT RBRACE {$$;} ;

WHILE_STATEMENT: WHILE LPAREN EXPRESSION RPAREN LBRACE STATEMENT RBRACE {$$;} ;

ASSIGNMENT: IDENTIFIER ASSIGN EXPRESSION SEMI {$$;} ;

EXPRESSION:

NUMBER {$$;}

| EXPRESSION INFIX_OPERATOR EXPRESSION {$$;}

| IDENTIFIER {$$;}

| LPAREN EXPRESSION RPAREN {$$;}

;

INFIX_OPERATOR:

INCR {$$;}

| MULOP {$$;}

| RELOP {$$;}

| OROP {$$;}

| ANDOP {$$;}

;