Formal Languages and Compiler Design - Lab7

Requirement

Implement a parser algorithm

1. One of the following parsing methods will be chosen (assigned by teaching staff):
    a. recursive descendent
    b. ll(1)
    c. lr(0)
2. The representation of the parsing tree (output) will be (decided by the team):
    d. productions string (max grade = 8.5)
    e. derivations string (max grade = 9)
    f. table (using father and sibling relation) (max grade = 10)

PART 1: Deliverables

1. *Class Grammar* (required operations: read a grammar from file, print set of nonterminals, set of terminals, set of productions, productions for a given nonterminal, CFG check)
2. Input files: *g1.txt* (simple grammar from course/seminar), *g2.txt* (grammar of the minilanguage - syntax rules from Lab 1b)

PART 2: **Deliverables**

Functions corresponding to the assigned parsing strategy + appropriate tests, as detailed below:
Recursive Descendent - functions corresponding to moves (*expand*, *advance*, *momentary insuccess*, *back*, *another try*, *success*)
LL(1) - functions *FIRST*, *FOLLOW*
LR(0) - functions *Closure*, *GoTo*, *CanonicalCollection*

PART 3: Deliverables

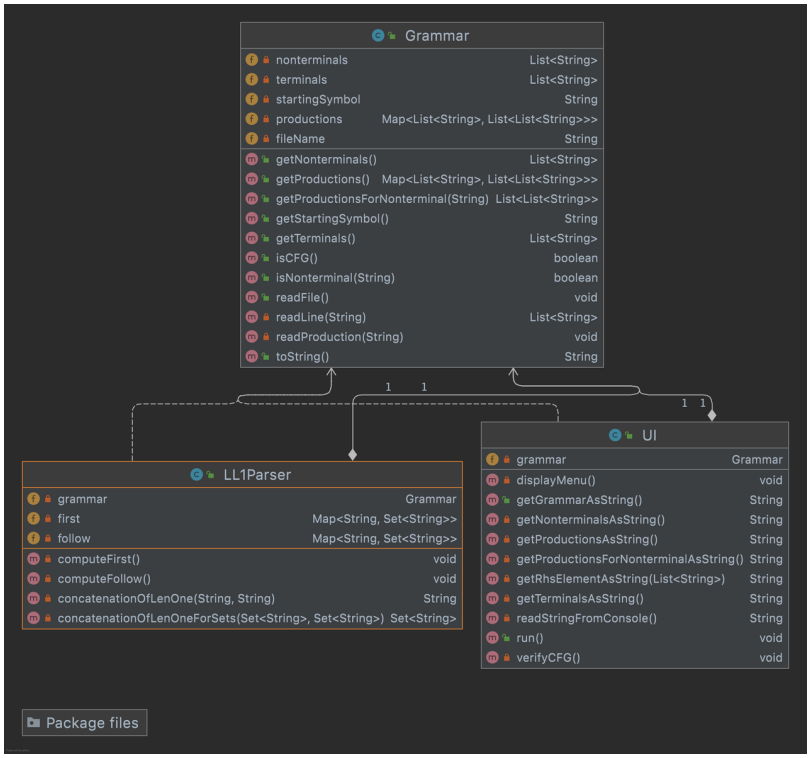
1. Algorithms corresponding to *parsing table* (if needed) and *parsing strategy*

2. Class *ParserOutput* - DS and operations corresponding to choice 2.a/2.b/2.c (Lab 5) (required operations: transform parsing tree into representation; print DS to screen and to file)

Remark: If the table contains conflicts, you will be helped to solve them. It is important to print a message containing row (symbol in LL(1), respectively state in LR(0)) and column (symbol) where the conflict appears. For LL(1), values (α,i) might also help.

Design

Class Diagram



Grammar

```
public class Grammar {
    private List<String> nonterminals;
    private List<String> terminals;
    private String startingSymbol;
    private Map<List<String>, List<List<String>>> productions;
}
```

The Grammar class has a list of strings for nonterminals and terminals. The startingSymbol and the fileName where the grammar is defined are of type String. The productions are kept as a Map which has the key the left hand side of the production, a List<String>, and the value a List<List<String>>.

LL1Parser

```
public class LL1Parser {
    private Grammar grammar;
    private Map<String, Set<String>> first;
    private Map<String, Set<String>> follow;
}
```

- The LL1Parser has a map, in which every terminal and nonterminal is a key and its corresponding FIRST set.
- `private String concatenationOfLenOne(String a, String b)`
  - returns the concatenation of length 1 between String a and String b
- `private Set<String> concatenationOfLenOneForSets(Set<String> a, Set<String> b)`
  - returns the set of results after every String from a was concatenated with every String from b

- `private void computeFirst()`
    - computes FIRST for every terminal and stores the result in the first Map

Implementation

Parser
Tests

Tests

Grammar

g1.txt - CFG

- Input

```
S A B C
a b c
S
S -> A B
A -> a S a | c C
B -> b S | E
C -> a A | E
```

- Output

```
Grammar{N=[S, A, B, C], E=[a, b, c], S='S',
P={[A]=[[a, S, a], [c, C]], [B]=[[b, S], [E]], [S]=[[A, B]], [C]=[[a, A], [E]]}}

Is CFG: true
```

g1-notCFG.txt - not CFG

- Input

```
S A B C
a b c
S
S A -> A B
A -> a S a | c C
B -> b S | E
C -> a A | E
```

- Output

```
Grammar{N=[S, A, B, C], E=[a, b, c], S='S',
P={[A]=[[a, S, a], [c, C]], [B]=[[b, S], [E]], [C]=[[a, A], [E]], [S, A]=[[A, B]]}}

Is CFG: false
```

g2.txt

- Input

```
program decllist compstmt decllist declaration type simple_type array_type stmtlist st
START ENDPRG identifier const INT BOOLEAN CHAR STRING ARRAY ; : [ ] BEGIN END := ( ) +
program
program -> START decllist compstmt ENDPRG
decllist -> declaration ; | declaration ; decllist
declaration -> identifier : type
simple_type -> INT | BOOLEAN | CHAR | STRING
array_type -> ARRAY [ INT ] simple_type
type -> simple_type | array_type
compstmt -> BEGIN stmtlist END
stmtlist -> stmt ; | stmt ; stmtlist
stmt -> simple_stmt | struct_stmt
simple_stmt -> assign_stmt | io_stmt
assign_stmt -> identifier := expression
expression -> arith1 arith2
arith1 -> arith1 + arith2 | arith1 - arith2 | E
arith2 -> multiply1 multiply2
multiply1 -> multiply1 * multiply2 | multiply1 / multiply2 | E
multiply2 -> ( expression ) | identifier | const
io_stmt -> READ ( identifier ) | WRITE ( identifier )
struct_stmt -> compstmt | ifstmt | whilestmt
ifstmt -> IF condition THEN stmt | IF condition THEN stmt ELSE stmt
whilestmt -> WHILE condition DO stmt
condition -> expression RELATION expression
RELATION -> < | <= | = | > | >= | != | AND | OR
```

- Output

```
Grammar{N=[program, decllist, compstmt, decllist, declaration, type, simple_type,
array_type, stmtlist, stmt, simple_stmt, assign_stmt, expression, arith1, arith2,
multiply1, multiply2, io_stmt, struct_stmt, ifstmt, whilestmt, condition, RELATION],

E=[START, ENDPRG, identifier, const, INT, BOOLEAN, CHAR, STRING, ARRAY, ;, :, [, ],
BEGIN, END, :=, (, ), +, *, -, /, %, READ, WRITE, IF, THEN, ELSE, WHILE, DO, <, <=,
=, >, >=, !=, AND, OR],

S='program',

P={[RELATION]=[[<], [<=], [=], [>], [>=], [!=], [AND], [OR]],
[struct_stmt]=[[compstmt], [ifstmt], [whilestmt]], [array_type]=[[ARRAY, [, INT, ],
simple_type]], [ifstmt]=[[IF, condition, THEN, stmt], [IF, condition, THEN, stmt,
ELSE, stmt]], [simple_type]=[[INT], [BOOLEAN], [CHAR], [STRING]],
[io_stmt]=[[READ, (, identifier, )], [WRITE, (, identifier, )]],
[stmtlist]=[[stmt, ;], [stmt, ;, stmtlist]],
[program]=[[START, decllist, compstmt, ENDPRG]],
[expression]=[[arith1, arith2]], [declaration]=[[identifier, :, type]],
[type]=[[simple_type], [array_type]], [multiply1]=[[multiply1, *, multiply2],
[multiply1, /, multiply2], [E]], [whilestmt]=[[WHILE, condition, DO, stmt]],
[stmt]=[[simple_stmt], [struct_stmt]], [assign_stmt]=[[identifier, :=, expression]],
```

```
[multiply2]=[[(, expression, )], [identifier], [const]],
[condition]=[[expression, RELATION, expression]], [decllist]=[[declaration, ;],
[declaration, ;, decllist]], [simple_stmt]=[[assign_stmt], [io_stmt]],
[arith1]=[[arith1, +, arith2], [arith1, -, arith2], [E]],
[arith2]=[[multiply1, multiply2]], [compstmt]=[[BEGIN, stmtlist, END]]}}

Is CFG: true
```

LL1Parser

g1.txt

- Input Grammar

```
S A B C
a b c
S
S -> A B
A -> a S a | c C
B -> b S | E
C -> a A | E
```

- Output

```
FIRST:
A [a, c]
B [b, E]
S [a, c]
C [a, E]

FOLLOW
A [a, b, E]
B [a, E]
S [a, E]
C [a, b, E]
```

g2.txt

- Input Grammar

```
program decllist compstmt decllist declaration type simple_type array_type stmtlist st
START ENDPRG identifier const INT BOOLEAN CHAR STRING ARRAY ; : [ ] BEGIN END := ( ) +
program
program -> START decllist compstmt ENDPRG
decllist -> declaration ; | declaration ; decllist
declaration -> identifier : type
simple_type -> INT | BOOLEAN | CHAR | STRING
array_type -> ARRAY [ INT ] simple_type
type -> simple_type | array_type
compstmt -> BEGIN stmtlist END
stmtlist -> stmt ; | stmt ; stmtlist
```

```
stmt -> simple_stmt | struct_stmt
simple_stmt -> assign_stmt | io_stmt
assign_stmt -> identifier := expression
expression -> arith1 arith2
arith1 -> arith1 + arith2 | arith1 - arith2 | E
arith2 -> multiply1 multiply2
multiply1 -> multiply1 * multiply2 | multiply1 / multiply2 | E
multiply2 -> ( expression ) | identifier | const
io_stmt -> READ ( identifier ) | WRITE ( identifier )
struct_stmt -> compstmt | ifstmt | whilestmt
ifstmt -> IF condition THEN stmt | IF condition THEN stmt ELSE stmt
whilestmt -> WHILE condition DO stmt
condition -> expression RELATION expression
RELATION -> < | <= | = | > | >= | != | AND | OR
```

- Output

```
FIRST
io_stmt [READ, WRITE]
program [START]
type [ARRAY, CHAR, STRING, INT, BOOLEAN]
multiply1 [E, *, /]
multiply2 [identifier, const, (]
decllist [identifier]
arith2 [identifier, const, (, *, /]
arith1 [E, +, -]
assign_stmt [identifier]
compstmt [BEGIN]
struct_stmt [WHILE, BEGIN, IF]
whilestmt [WHILE]
condition [<=, OR, AND, <, !=, =, >, >=]
simple_type [CHAR, STRING, INT, BOOLEAN]
simple_stmt [READ, identifier, WRITE]
array_type [ARRAY]
expression [E]
ifstmt [IF]
RELATION [<=, OR, AND, <, !=, =, >, >=]
stmtlist [;]
declaration [identifier]
stmt [WHILE, BEGIN, IF]


FOLLOW
struct_stmt [ELSE, ;]
expression [<=, OR, ), DO, AND, ELSE, THEN, ;, <, !=, =, >, >=]
ifstmt [ELSE, ;]
RELATION [THEN, DO]
simple_type [;]
stmtlist [END]
io_stmt [ELSE, ;]
program [E]
type [;]
declaration [;]
multiply1 [identifier, const, (, *, /]
```

```
whilestmt [ELSE, ;]
multiply2 [identifier, <=, OR, const, (, ), *, +, DO, -, /, AND, ELSE, THEN, ;, <, !=,
condition [THEN, DO]
decllist [BEGIN]
simple_stmt [ELSE, ;]
arith2 [<=, identifier, OR, const, (, ), *, +, DO, -, /, AND, ELSE, THEN, ;, <, !=, =,
array_type [;]
arith1 [identifier, const, (, *, +, -, /]
stmt [ELSE, ;]
assign_stmt [ELSE, ;]
compstmt [ENDPRG, ELSE, ;]
```

g3.txt

- Input Grammar

```
S A B C
( ) int + *
S
S -> A B
A -> ( S ) | int C
B -> + S | E
C -> * A | E
```

- Output

```
FIRST
A [(, int]
B [E, +]
S [(, int]
C [E, *]


FOLLOW
A [E, ), +]
B [E, )]
S [E, )]
C [E, ), +]
```