مستند پروژه کامپایلر لولو

- مستند فاز اول پروژه طراحی کامپایلر زبان LULU
- ترم اول سال تحصیلی ۹۸-۱۳۹۷

اعضای گروه:

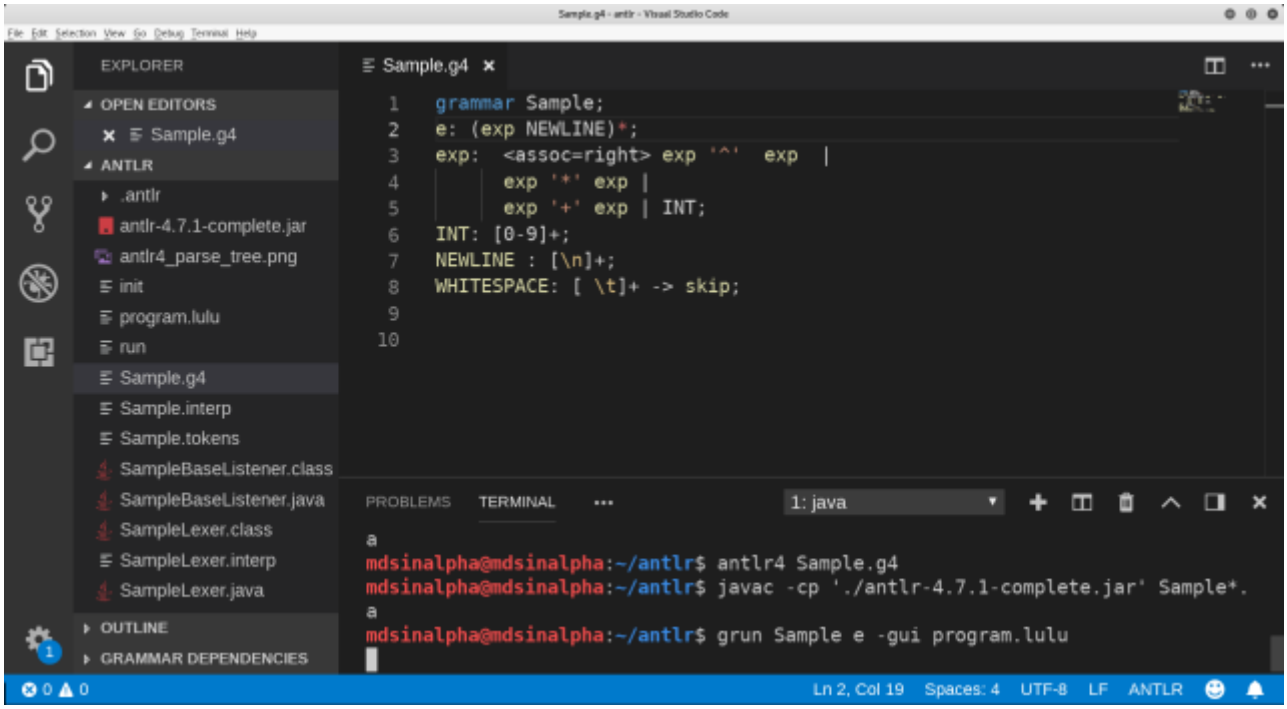
- محسن دهباشی ۹۵۳۶۱۱۱۳۳۰۲۸
- سید محمد هاشمی ۹۵۳۶۱۱۱۳۳۰۸۴
- پوریا زمانی نژاد ۹۵۳۶۱۱۱۳۳۰۳۶

مقدمه

در این فاز از پروژه از ابزار $antlr$ برای تجزیه و تحلیل لغوی و ساختار دستوری زبان برنامه نویسی لولو استفاده شد که مراحل استفاده از این ابزار برای رسیدن به این هدف به ترتیب تیتر های درج شده در مستند می باشد.

مرحله اول: بارگیری و نصب

- در این مرحله از $Extension$ تحلیل ساختاری گرامر $antlr$ برای نرم افزار توسعه یکپارچه $VSCode$ جهت سهولت و استفاده از گرامر این ابزار و نوشتن گرامر زبان لولو تحت یک فایل g4 استفاده شد.
- ضمنا در ابتدا برای آزمون درستی نصب و راه اندازی های اولیه یک گرامر ساده و ابتدایی نوشته شد که قواعد آن بصورت زیر بود:



- سپس جهت اجرای پیاده سازی گرامر که بصورت خودکار توسط ابزار $antlr$ انجام می شود فایل کتابخانه jar مربوط به این ابزار در $CLASSPATH$ سیستم عامل قرار داده شد و $alias$ بندی های مربوطه انجام شد.

```
export                    CLASSPATH=".:/usr/local/lib/java1.8/antlr-4.7.1-
complete.jar:$CLASSPATH"
alias antlr4='java -Xmx500M -cp "$CLASSPATH" org.antlr.v4.Tool'
alias grun='java org.antlr.v4.gui.TestRig'
```

- همچنین برای پیاده سازی تست گرامر نوشته شده تحت ابزار $antlr$ مراحلی باید طی شود که نیازمند استفاده از دستورات زیر در ترمینال سیستم عامل است.

```
antlr4 Sample.g4
```

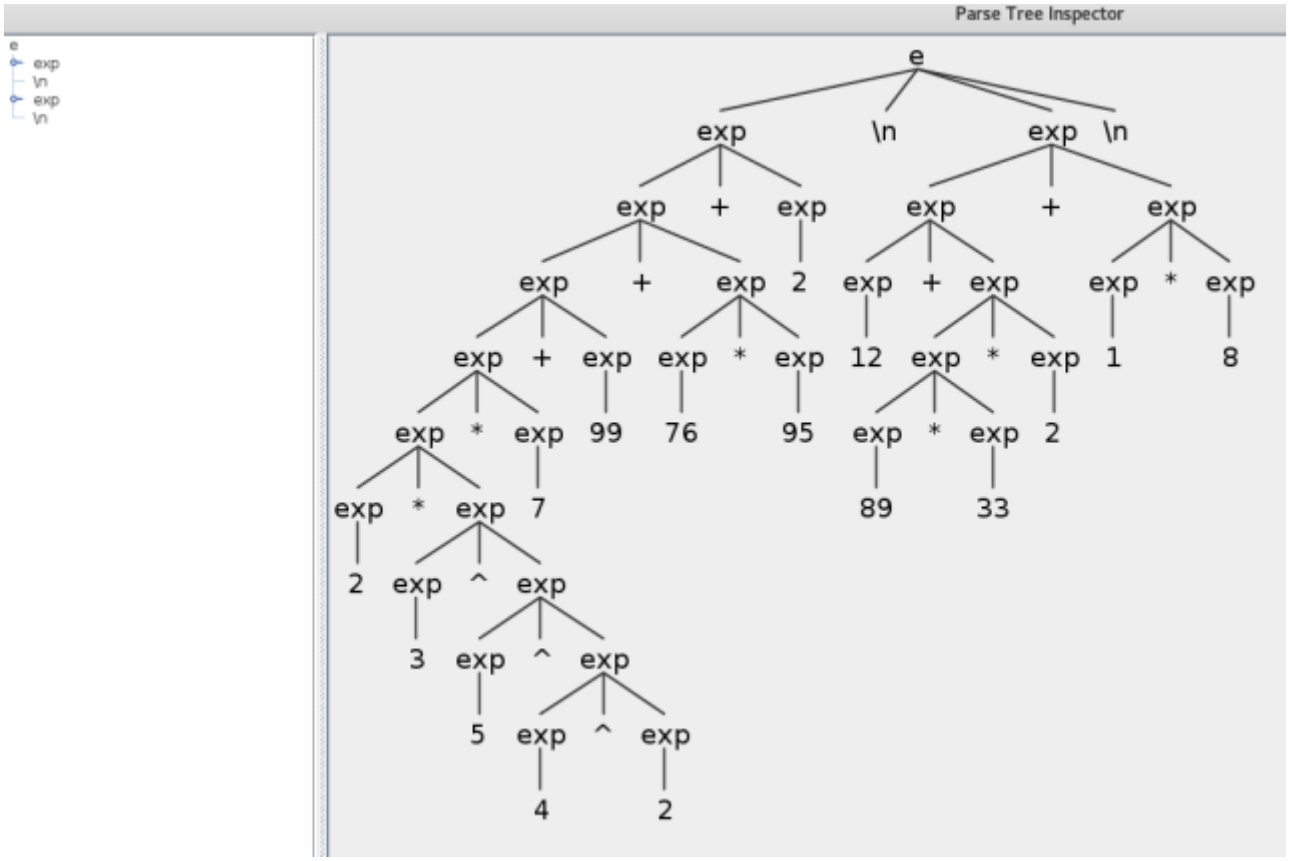
این دستور از روی گرامر نوشته شده در فایل ورودی، سورس کدهای جاوای ماشین این زبان را تولید می کند.

```
javac Sample*.java
```

دستور کامپایل کردن ماشین.

```
grun Sample e -gui program.lulu
```

دستور شبیه سازی ماشین گرامر $Sample$ برای نشانه غیرپایانی e به عنوان نشانه شروع گرامر بر روی رشته ورودی ذخیره شده در فایل $program.lulu$.



مرحله دوم: تحلیل لغوی

- در این مرحله گرامر زبان لولو از لحاظ لغوی بررسی شد و همه عبارت های منظم مربوط به $token$ های این زبان نوشته شد.
- در مرحله باز بینی $regex$ های نوشته شده، اولویت توکن ها مشخص شد و برخی توکن های جدیدی جهت رفع ابهام گرامر تولید شد. در واقع برای رعایت **اولویت عملگر ها** در گرامر و همچنین رفع ابهام ناشی از کاراکتر های مشترک در عبارت های منظم توکن ها(مثل '-' که هم بعنوان علامت نفی استفاده می شود و هم علامت تفریق)، بایستی توکن بندی گسترده تر می شد.
- در نهایت مجموعا ۱۸ توکن و ۱۲ $fragment$ برای گرامر تولید شد:

```
//Tokens:

BOOL_CONST:     'true' | 'false';

PRIM_TYPE:      'int' | 'bool' | 'float' | 'string';

ACCSSMOD:       'private' | 'public' | 'protected';

ID :            LETTER (LETTER | DIGIT) * ;

UNARY_OP:       '!' | '~';

MINUS:          '-';

ARIT_P1:        '*' | '/' | '%';

ARIT_P2:        '+';

BITWISE_AND:    '&';

BITWISE_OR:     '|';

BITWISE_XOR:    '^';

LOGICAL_AND:    '&&';

LOGICAL_OR:     '||';

REL:            '<=' | '>=' | '<' | '>';

REL_EQ:         '==' | '!=';

STRING_CONST:   '\'' (ESCAPE | ~[\\])*? '\'';

REAL_CONST:     REAL_DECIMAL | REAL_HEX;

INT_CONST:      '0'[xX]HEX+ | DIGIT+;
```

- عبارت منظم مربوط به توکن $REALCONST$ در مبنای ۱۶ بصورتی کار می کند که کل ارقام(قسمت صحیح و اعشاری) را در مبنای ۱۶ تشخیص می دهد بطوریکه اگر نشانه غیر پایانی e به منظور توان عدد ظاهر شود حتما بعد از آن باید علامت + یا - نیز ظاهر شود.(تفاوت بین رقم و نشانه توان وقتی قابل تشخیص است که بتوان + یا - را قبل از توان عدد مشاهده کرد)

```
//Fragments:

fragment LETTER:         [a-zA-Z_#];

fragment DIGIT:          [0-9];

fragment HEX:            [a-fA-F0-9];

fragment ESCAPE:          '\\n' | '\\r' | '\\t' | '\\\\' | '\\0' | '\\"' |
'\\' [xX] HEX HEX;

fragment    REAL_DECIMAL:           (REAL_DECIMAL_L    |    REAL_DECIMAL_R)
REAL_DECIMAL_EXP?;

fragment REAL_DECIMAL_L: (DIGIT+) '.' (DIGIT+)?;

fragment REAL_DECIMAL_R: (DIGIT+)? '.' (DIGIT+);

fragment REAL_DECIMAL_EXP:([eE]([+-])? DIGIT+);

fragment REAL_HEX:       '0'[xX] (REAL_HEX_L | REAL_HEX_R) REAL_HEX_EXP?;

fragment REAL_HEX_L:     (HEX+) '.' (HEX+)?;

fragment REAL_HEX_R:     (HEX+)? '.' (HEX+);

fragment REAL_HEX_EXP:   ([eE]([+-]) HEX+);
```

- همچنین در این مرحله قواعد دور ریختن فاصله های خالی(از یک توکن شناخته شده تا توکن بعدی) و کامنت ها نیز به گرامر اضافه شدند:

```
//Skip:

WHITESPACE:   [ \n\t\r] -> skip;

LINE_COMMENT: '%%' .*? ('\n'|EOF) -> skip;

COMMENT:      '%~' .*? '~%' -> skip;
```

مرحله سوم: تحلیل دستوری

- در این مرحله ساختار دستوری زبان و گرامر مستقل از متن آن پیاده سازی شد، این گرامر مجموعا ۲۶ نشانه غیرپایانی و ۶۲ قاعده تولید دارد که نشانه غیرپایانی شروع کننده آن $program$ می باشد.

```
//Rules:

program:        ft_dcl? ft_def+;

ft_dcl:         'declare' '{' (func_dcl | type_dcl | var_def)+ '}';

func_dcl:       ('(' args ')' '=')? ID '('(args | args_var)? ')' ';';

args:           type ('['']')* |

                args ',' type ('['']')*;

args_var:       type ('['']')* ID |

                args_var ',' type ('['']')* ID;


type_dcl:       ID ';';

var_def:        ('const')? type var_val (','var_val)* ';';

var_val:        ref ('='expr)?;

ft_def:         type_def | func_def;

type_def:       'type' ID(':' ID)? '{' component+ '}';
```

```
component:      ACCSSMOD? (var_def | func_def);

func_def:         ('(' args_var ')' '=')? 'function' ID '(' args_var? ')'
block;

block:          '{' (var_def | stmt)* '}';

stmt:           assign ';' |

                func_call ';' |

                cond_stmt |

                loop_stmt |

                          'return'  ';'|  'break'  ';'  |  'continue'  ';'  |
'destruct'('['']')* ID ';';

assign:         (<assoc=right> (var | '(' var (',' var)* ')') '=' expr);

var:            (('this' | 'super')'.')? ref ('.' ref)*;

ref:            ID ('[' expr ']')*;

expr:           '(' expr ')' |

                UNARY_OP expr | MINUS expr |

                expr ARIT_P1 expr |

                expr ARIT_P2 expr | expr MINUS expr |

                expr BITWISE_AND expr |

                expr BITWISE_XOR expr |

                expr BITWISE_OR expr |

                expr REL expr |

                expr REL_EQ expr |
```

```
                expr REL_EQ expr |

                expr LOGICAL_AND expr |

                expr LOGICAL_OR expr |

                'allocate' handle_call |

                func_call | var | list | 'nil' | const_val;

func_call:      (var'.')? handle_call |

                'read' '(' var ')' | 'write' '(' var ')';

list:           '[' (expr|list) ( ',' (expr | list))* ']';

handle_call:    ID '(' params? ')';

params:         expr | expr ',' params;

cond_stmt:      'if' expr block ('else' block)? |

                  'switch' var '{' ('case' INT_CONST ':' block)* 'default'
':' block '}';

loop_stmt:      'for' ( type? assign)? ';' expr ';' assign? block |

                'while' expr block;

const_val:      INT_CONST | REAL_CONST | BOOL_CONST | STRING_CONST;


type:           PRIM_TYPE | ID;
```

- لازم بذکر است که گرامر بالا در واقع رفع ابهام شده گرامر اولیه ی نوشته شده است.
- از این رفع ابهام ها می توان به رعایت اولویت عملگرها در نشانه غیرپایانی $expr$، تغییر شرکت پذیری قاعده تولید نشانه غیر پایانی $assign$ به شرکت پذیری از راست و دادن اولویت به $Primitive$ تایپ ها نسبت به آی دی در نشانه غیرپایانی $type$ اشاره کرد.
- همچنین برای رفع ابهام کامل گرامر آزمون های زیر غیرموثر نبودند:

```
grun Lulu expr -gui
2 * 4 -6 | 8
```

-

```
grun Lulu expr -gui
-a + 6 + b * 2
```

-

```
grun Lulu assign -gui
(a, b, c) = 2 * 4 - 6
```

-

```
grun Lulu assign -gui
(a, b, c) = -a + b * c -2 / 7 | 5 * ddd#
```

-

```
grun Lulu loop_stmt -gui
for int a = 6; a == b ; a = a + 1 {
if c == C#6 { (A,b) = a * 7 / 6; } }
```

-

```
grun Lulu program -gui
%% A sample program
declare{
    int a;
    mytype;
    (int, float) = f1(float b);
}
(int r) = function start() {
    return;
}
```

مرحله چهارم: آزمون درستی

- برای تست کامپایلر لولو و اطمینان از صحت درستی پیاده سازی آن، گرامر کامپایلر بررسی شد و ۱۰ برنامه مختلف به این زبان نوشته شد. این برنامه ها و درخت تجزیه آنها در فولد $Tests$ موجود می باشد.
- ضمنا چندین اشکال کوچک مثل عدم تشخیص کامنت خط آخر حین تست برنامه ها در پیاده سازی پیدا و رفع شد.

روند کار گروهی

- تقسیم کار در کل بصورتی بود که همه اعضای گروه در روند پیاده سازی پروژه حضور فعال داشتند و نوشتن قواعد گرامر و رفع ابهام آنها و همچنین تست، بصورت دسته جمعی انجام شد.
- علاوه بر این تمرکز هر یک از افردا تیم بر روی کارهای زیر نیز بوده است:
  - **محسن دهباشی:** راه اندازی و کار با ابزار و نوشتن مستند پروژه
  - **سید محمد هاشمی:** نوشتن برنامه هایی جهت آزمون درستی
  - **پوریا زمانی نژاد:** نوشتن عبارت های منظم مربوط به توکن های گرامر

تعهدنامه اخلاقی

ما(محسن دهباشی، سید محمد هاشمی، پوریا زمانی نژاد) تعهد می نماییم که پروژه تحویل داده شده نتیجه کار گروهی ما بوده و در هیچ یک از بخش های انجام شده از کار دیگران کپی برداری نشده است. در صورتی که مشخص شود که پروژه تحویل داده شده کار این گروه نبوده است، طبق ظوابط آموزشی با ما برخورد شده و حق اعتراض نخواهیم داشت.