Team 18 - Vyass Language

Amit Noel Thokala
Sri Venkata Vivek Dhulipala
Sowmya Veldandi
Suvarshitha Kalvakuntla
Yashwanth Reddy Kikkuri

Outline

- Vyass Language characteristics
- Grammar rules Definition
- Vyass Language Design components
- Vyass Language demonstration
- Future Work

Vyass Language Characteristics

Our Language supports the following,

Primitive data types	number, bool, string
Logical operators	&, |, !
Arithmetic operators	addition(+), subtraction(-), multiplication (*), division(/), modulus(%)
Comparison operators	equal to (==), not equal to (!=), less than(<), less than or equal to (<=), greater than (>), greater than or equal to (>=)
delimiter	comma(,), semi-colon(;), colon(:)
Assignment operator, ternary operator	=, :?

Vyass Language Characteristics

Our language supports the following

- conditional statements - if, if-else
- iteration statements - while, for, for in
- begin and end statements
- break and continue statements
- print statements
- parentheses - (),{}

Grammar rules

```
grammar Vyass;

parse
  :  (x=.
        {System.out.printf("text: %-7s  type: %s \n",
         $x.text, tokenNames[$x.type]);}
    )*
    EOF
  ;
Space : [ \r\t\n\u000C] -> skip;

IF : 'if';
ELIF : 'elif';
ELSE : 'else';
RETURN : 'return';
BREAK : 'break';
CONTINUE: 'continue';
PRINT : 'print';
BEGIN : 'begin';
END : 'end';
WHILE : 'while';
FOR : 'for';
IN : 'in';
RANGE : 'range';
MAIN: 'main';
```

Grammar rules

```
ADDITION_BINARY : '+';
SUBSTRACTION_BINARY : '-';
MULTIPLICATION_BINARY : '*';
DIVISION_BINARY : '/';
MODULUS_BINARY : '%';
TERNARY : '?';
NOT_BINARY : '!';
LESSTHAN_BINARY  : '<';
GREATERTHAN_BINARY : '>';
LESSTHANEQUALS_BINARY : '<=';
GREATERTHANEQUALS_BINARY : '>=';
EQUALS_BINARY : '==';
NOTEQUALS_BINARY : '!=';
AND_BINARY : '&';
OR_BINARY : '|';
ARITHMETIC_OP : ADDITION_BINARY | SUBSTRACTION_BINARY | MULTIPLICATION_BINARY | DIVISION_BINARY | MODULUS_BINARY;
COMPARISION_OP : GREATERTHANEQUALS_BINARY | LESSTHANEQUALS_BINARY | LESSTHAN_BINARY | GREATERTHAN_BINARY | EQUALS_BINARY | NOTEQUALS_BINARY;

INTEGER_DTYPE : 'number';
BOOL_DTYPE : 'bool';
STR_DTYPE : 'string';

COMMA_SEP : ',';
SEMICOLON_SEP : ';';
COLON_SEP : ':' ;
```

Grammar rules

```
START_BLOCK : '{';
END_BLOCK : '}' ;
LEFT_PAREN : '(' ;
RIGHT_PAREN : ')';

STRING_LITERAL : ["] (~["\r\n])* ["];
BOOLEAN_LITERAL: 'true' | 'false';
INTEGER_LITERAL: [1-9] [0-9]* | [0];
IDENTIFIER : [a-zA-Z_] [a-zA-Z_0-9]*;

literalConst : INTEGER_LITERAL | BOOLEAN_LITERAL | STRING_LITERAL;
dType : INTEGER_DTYPE | BOOL_DTYPE | STR_DTYPE;

program : BEGIN functionDeclarations* mainFunctionBlock? END;
mainFunctionBlock : MAIN exprBlock ;

exprBlock: START_BLOCK variableDeclarations* statements* END_BLOCK;

variableDeclarations : dType variableList SEMICOLON_SEP;
variableList : variableInitialization variableListMulti?;
variableListMulti : COMMA_SEP variableInitialization variableListMulti?;
variableInitialization : IDENTIFIER ASSIGNMENT_BINARY literalConst | IDENTIFIER;

functionDeclarations : dType IDENTIFIER LEFT_PAREN parameters? RIGHT_PAREN functionBlock;
parameters : parameter (multiParameter)*;
```

Grammar rules

```
multiParameter : COMMA_SEP parameter;
parameter : dType IDENTIFIER;
functionBlock : exprBlock;

statements : assignmentStatement
| printStatement
| returnStatement
| breakStatement
| continueStatement
| exprBlock
| conditionalBlock
| iterativeBlock
;

assignmentStatement : assignmentList SEMICOLON_SEP;
assignmentList : IDENTIFIER ASSIGNMENT_BINARY (assignmentList | expression_expr);

printStatement : PRINT LEFT_PAREN expression_expr RIGHT_PAREN SEMICOLON_SEP;
returnStatement : RETURN SEMICOLON_SEP | RETURN expression_expr SEMICOLON_SEP;
continueStatement : CONTINUE SEMICOLON_SEP;
breakStatement : BREAK SEMICOLON_SEP;


conditionalBlock : IF LEFT_PAREN ifCondition RIGHT_PAREN exprBlock elifList;
elifList : ELIF LEFT_PAREN ifCondition RIGHT_PAREN exprBlock elifList | elseBlock?;
```

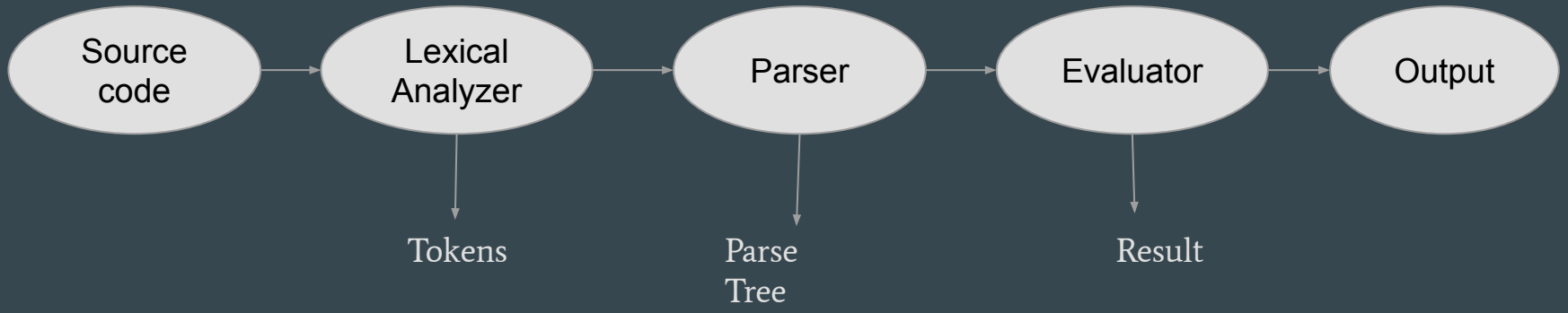
Grammar rules

```
elseBlock : ELSE exprBlock;
ifCondition : expression_expr;

iterativeBlock : whileTraditionalBlock | forBlock;
whileTraditionalBlock : WHILE LEFT_PAREN whileCondition RIGHT_PAREN exprBlock;
whileCondition : expression_expr;
forBlock : FOR (forTraditionalBlock | forInRangeBlock);
forTraditionalBlock : LEFT_PAREN forInit? SEMICOLON_SEP forCondition SEMICOLON_SEP forUpdate? RIGHT_PAREN exprBlock;
forInRangeBlock : IDENTIFIER IN RANGE LEFT_PAREN forInRangeLowerLimit COMMA_SEP forInRangeUpperLimit RIGHT_PAREN exprBlock;
forInit : forInitStatement (COMMA_SEP forInitStatement)*;
forInitStatement : forAssign;
forCondition : expression_expr;
forUpdate : forUpdateStatement (COMMA_SEP forUpdateStatement)*;
forUpdateStatement : forAssign;
forAssign : assignmentList;
forInRangeLowerLimit : expression_expr;
forInRangeUpperLimit : expression_expr;

functionCall : IDENTIFIER LEFT_PAREN values? RIGHT_PAREN;
values : functionValue (COMMA_SEP functionValue)*;
functionValue : expression_expr;

expression_expr : express;
rExpress : express;
```

Grammar rules

```
express:
LEFT_PAREN express RIGHT_PAREN      #parametersExpression
| '-' express       #unaryNegationExpression
| '!' express       #unaryNotExpression
| express MULTIPLICATION_BINARY rExpress        #binaryMultiplicationExpression
| express DIVISION_BINARY rExpress      #binaryDivisionExpression
| express MODULUS_BINARY rExpress       #binaryModulusExpression
| express ADDITION_BINARY rExpress      #binaryAdditionExpression
| express SUBSTRACTION_BINARY rExpress      #binarySubstractionExpression
| express GREATERTHANEQUALS_BINARY rExpress     #binaryGreaterThanEqualsExpression
| express LESSTHANEQUALS_BINARY rExpress        #binaryLessThanEqualsExpression
| express GREATERTHAN_BINARY rExpress       #binaryGreaterThanExpression
| express LESSTHAN_BINARY rExpress      #binaryLessThanExpression
| express EQUALS_BINARY rExpress        #binaryEqualsExpression
| express NOTEQUALS_BINARY rExpress     #binaryNotEqualsExpression
| express '&&' rExpress     #logicalAndExpression
| express '||' rExpress     #logicalOrExpression
| express '?' ternaryTrue ':' ternaryFalse    #ternaryCondnExpression
| INTEGER_LITERAL       #integerLiteralExpression
| BOOLEAN_LITERAL       #booleanLiteralExpression
| STRING_LITERAL        #stringLiteralExpression
| IDENTIFIER        #identifierExpression
| functionCall      #functionCallExpression;
ternaryTrue : express;
ternaryFalse : express;
```

Vyass Language design components



Source Code

- The source code refers to a program file that can be executed using the Vyass Language.
- The file has an extension “.vyass”. For example the file could be input.vyass.
- The lexer then reads this file as the input.

Lexical Analyser

- The lexical analyser access the “input.vyass” file that holds the source code and scans it character by character.
- It then translates these characters into tokens which are meaningful and only Vyass language can understand.
- The tokens are then stored as a list.
- These tokens are now provided as an input to the parser.

Parser

- The parser checks to see if the source code adheres with the Vyass Language's syntax requirements.
- From the list of tokens produced by the lexical analyzer, it constructs a parse tree.
- If any tokens cannot be parsed, the source code does not adhere to the syntax requirements of the language, then the parser outputs an error message.
- Vyass Language employs a top-down parsing approach.

Evaluator and Intermediate Code

- The parser generates a intermediate code file.
- It is a ..int extension file.
- The intermediate code file is comprised of the parse tree that is generated by the parser.
- The evaluator is accountable for interpreting the intermediate code and running the program according to syntax-based semantics.

Vyass Language Demonstration

Sample run

```
begin
    number fact(number num, bool t) {
            if(num == 0) {
                return 1;
            } else {
                return num*fact(num-1, t);
            }
        }
    main {
        number num = 6;
        print(fact(num, true));
    }
```

Future Work

- To implement lambda expressions.
- To accommodate more data types such as float, double, long, etc.
- To support function overloading/polymorphism.
- To support object oriented programming.