Ira A. Fulton Schools of Engineering
ARIZONA STATE UNIVERSITY

STARK Programming Language

By

Debarati Bhattacharyya (ASU ID : 1213439446)
Janani Thiagarajan(ASU ID : 1213296368)
Shilpa Bhat (ASU ID: 1212901948)
Vivek Faldu (ASU ID : 1214392296)

CONTENTS

INTRODUCTION

STARK is a static, strongly typed imperative programming language. The word "Stark" means simple and powerful which are the main design goals. It is easy to learn and provides all the major features of a high level language.

Tools:

- *ANTLR 4*

Lexical analysis – tokenization

Parser generation – parse tree

- Java 8

Compiler is written in Java

Interpréter and Runtime environment use *JRE*

Data Structures used : Stack and Hashmap

GETTING STARTED

We will first take a look at the processing of a *STARK* program.

- The source code will be given to lexical analyzer to generate the tokens
- The generated tokens are then given to parser to build a parse tree
- The parse tree would be given semantics to generate an intermediate code
- The intermediate code is then interpreted by the runtime environment to generate an output

The source code in *STARK* will be compiled and executed in two steps :

- Compile the source code to generate the intermediate code
- Interpret the intermediate code to produce the final output

STEPS TO RUN *STARK*

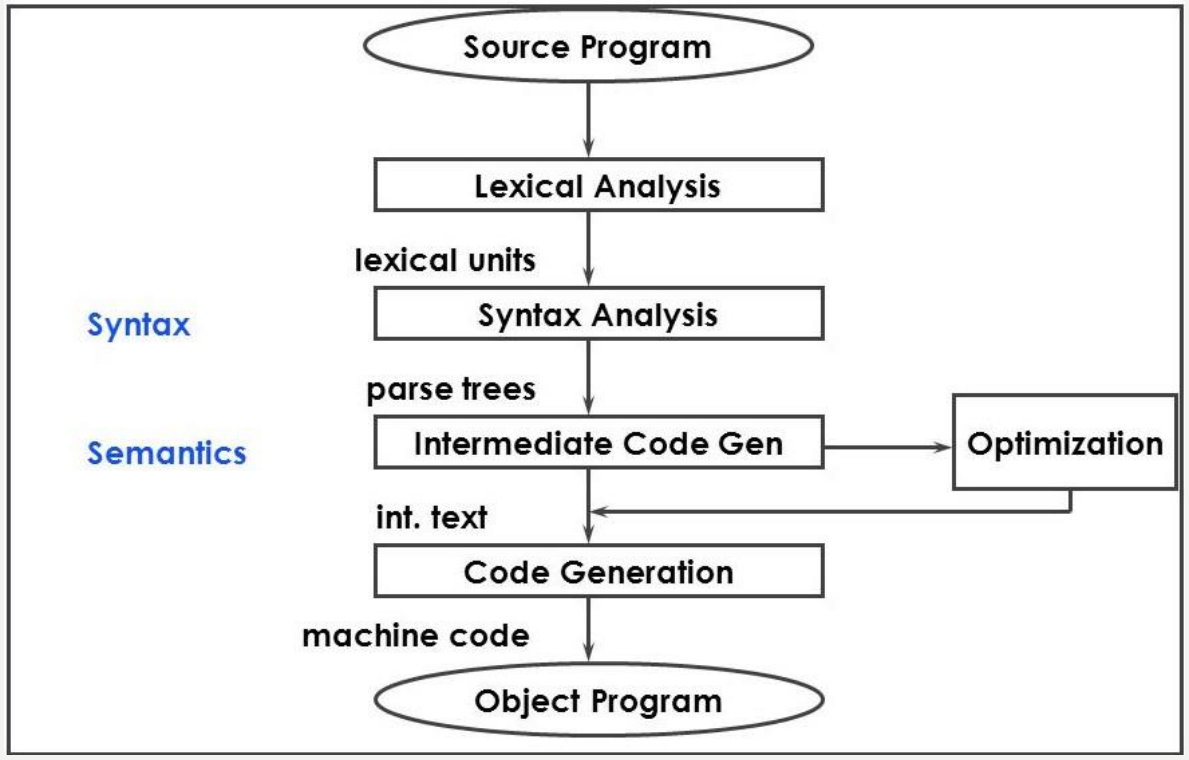
- STARK programs can be written and saved with any standard editor such as Notepad etc.
- Download STARK.jar from the git repository
- Follow the below steps to compile and execute your *STARK* programs:
- Windows Users :
  - Use Windows Command Prompt from the location of stark.jar
  - Use the command for compiling your *STARK* source code:
  - ***java stark.jar -c sourcePath intermediateCodePath***
  - This will generate and display the intermediate code
  - Use the command for executing your *STARK* intermediate code:
  - ***java stark.jar -e intermediateCodePath***

BLOCK DIAGRAM FOR SOURCE CODE PROCESSING

1. Lexical Analysis:
Input : source code
Output : lexical units or tokens

2. Parser:
Input : token string
Output : parse tree

3. Bytecode:
Input : parsing tree
Output : generate intermediate code



GRAMMAR FOR *STARK*

- *STARK* supports a wide range of arithmetic and relational operators. Besides this, it also supports logical operators.
- *STARK* has two primary datatypes :
  - int – used for positive numeric integers such as 0,1,2 and so on
  - bool – used for boolean values such as 'true' and 'false'
- Identifiers are symbols used to uniquely identify a program element in the code. *STARK* identifiers are alphanumeric elements but they must begin with a character
- STARK also incorporates conditional statements like if, if-else and iterative constructs like while loop
- With STARK, user can define and use their own functions to perform certain tasks
- Each valid *STARK* statement must end in a ';'
- Comments can be added with '//' and '/*……*/' for multi line comments

```
grammar stark;

program :  statementList functionDefn;
statementList : statement  statementList |
                statement;
statement : declarationStmt ';' |
            initializationStmt ';'|
            assignmentStmt ';'|
            ifStatement |
            whileStatement |
            displayStatement ';' |
            functionCall ';';
initializationStmt: 'int' IDENTIFIER '=' expression|
                    'bool' IDENTIFIER '=' boolExpression;

declarationStmt : 'int' IDENTIFIER|
                  'bool' IDENTIFIER;
assignmentStmt : IDENTIFIER '=' boolExpression|
                 IDENTIFIER '=' expression ;

ifStatement : 'if' '(' boolExpression ')' '{' statementList '}' |
              'if' '(' boolExpression ')' '{' statementList '}' 'else' '{' statementList '}' ;

whileStatement : 'while' '(' boolExpression ')' '{' statementList '}' ;

displayStatement : 'display' expression;

relationalExpression : expression '==' expression |
                       expression '!=' expression|
                       expression '<' expression |
                       expression '<=' expression|
                       expression '>' expression |
                       expression '>=' expression|
                       expression '==' BOOLVALUES|
                       expression '!=' BOOLVALUES;
```

```
logicalExpression : relationalExpression '&&' relationalExpression |
                    relationalExpression '||' relationalExpression |
                    relationalExpression '&&' logicalExpression |
                    relationalExpression '||' logicalExpression |
                    '!'relationalExpression |
                    '!'logicalExpression;
boolExpression : relationalExpression |
                 logicalExpression |
                 BOOLVALUES;

expression : term '+' expression|
             term '-' expression|
             term;
term : factor '*' term|
       factor '/' term|
       factor '%' term|
       factor;
factor : '(' expression ')' | IDENTIFIER | NUMBER ;

functionDefn : 'func' functionName '(' parameters ')' '{' statementList  returnStatement '}'|
               'func' functionName '(' parameters ')' '{' returnStatement '}'|  ;
functionName : IDENTIFIER;
parameters : declarationStmt ',' parameters |
             declarationStmt |  ;
returnStatement : 'return' expression ';'|
                  'return' boolExpression';'|  ;
functionCall : IDENTIFIER '=' functionName '(' arguments ')'|functionName '(' arguments ')' ;
arguments : IDENTIFIER ',' arguments | NUMBER ',' arguments | NUMBER | IDENTIFIER |  ;

//Terminals
BOOLVALUES : 'true' | 'false' ;
IDENTIFIER : [a-zA-Z][a-zA-Z0-9]* ;
NUMBER :[0-9]+;
WS : [ \t\r\n]+ -> skip ;
MULTICOMMENT : '/*'.*?'*/' -> skip;
SINGLECOMMENT : '//' ~[\r\n]* -> skip;
```

DATATYPES SUPPORTED

- Stark supports **integer**(int) and **boolean**(bool) datatypes.
- Operations related to these datatypes are supported as well
  - Operations supported for int
    - Addition, Subtraction, Multiplication, Division and Modulus operator (**+ - * / %)**
    - Stark also supports relational operator like **> , <, >=, <=, ==, !=** to compare two integer values
  - Operations supported for bool
    - Logical operators like **&&** (logical AND), || (Logical OR), **!** (Logical NOT)

OPERATORS IN *STARK*: OVERVIEW

Arithmetic :

Operator	Name	Operand	Function
+	Addition	a + b	Adds two numbers
-	Subtraction	a – b	Subtracts two numbers
/	Division	a / b	Divides two numbers
*	Multiplication	a * b	Multiplies two numbers
%	Modulus	a % b	Returns the remainder when a is divided by b

OPERATORS IN *STARK*: OVERVIEW

Relational and Logical :

Operator	Name	Operand	Returns True If
<	Less than	a < b	a is less than b
>	Greater than	a > b	a is greater than b
<=	Less than or equal to	a <= b	a is less than or equal to b
>=	Greater than or equal to	a >= b	a is greater than or equal to b
==	Is equal	a == b	a is equal to b
!=	Not equal to	a != b	a is not equal to b
&&	Logical AND	a && b	Both a and b are true
||	Logical OR	a || b	Either a or b is true

Besides this, we also support '!' which is the logical NOT operator. For eg: !a will compliment the value stored in a.

IDENTIFIERS

Identifiers are a sequence of one or more characters. Consecutive characters in the identifiers are optional and could comprise of **alphanumeric characters** but should **begin with an alphabet.**

- IDENTIFIER : [a-zA-Z][a-zA-Z0-9]*
- Eg : a, sum, num2

VARIABLES

- Variables are used to store data values of a certain type- **int or bool values**
- The stored value of variables can change during program execution.
- Stark supports variable **declarations , initialization and assignments operations**
- Variables should be declared or initialized before they can be used. Else Stark gives **compiler error**
  - Sample usage
    - int a; a=10;
    - int a=10;
    - bool b = true;
    - bool b;

EXPRESSIONS

- Stark supports the following types of expression
    - **Numeric expressions**
    - **Relational expressions**
    - **Logical expressions**

    Relational and logical expressions together form boolean expressions i.e. they return a boolean value based on the operation
- Numeric expressions handle **precedence** in the following order
    - ***, /, %**
    - **+, -**

    They are **left associative** as well.

STATEMENTS

- Stark statements could be any of the following :-
    - Declaration statement
    - Assignment statement
    - Initialization statement
    - If Statement (Also **Nested IF**)
    - If – Else Statement (Also **Nested IF-ELSE**)
    - While Statement (Also **Nested WHILE**)
    - Display Statement
    - **Function call**
- **Single line and multi-line comments** are also supported

DECLARATION STATEMENT

- Declaration should be in the following format :
  - **Datatype <varName>;**

  Example –

  int count;

  bool isZero;

INITIALIZATION STATEMENT

- To initialize a variable , the following format has to be followed
  - **dataType variableName = Value ;**
- Example –

```
int count = 0;
bool isZero = false;
```

ASSIGNMENT STATEMENT

- Assignment statement helps assign a value to a variable. It can be used in the following format
    - **variableName = Value ;**
- Example –

  count = 0;

  isZero = false;

DISPLAY STATEMENT

- Value of an expression or a variable can be displayed using the display statement
  - **display expression**;

- Example –

  display sum;

  display a+b;

IF STATEMENT

- Stark supports if condition in the following format

if(condition) {

Statement list;

}

Example : if(count>10) {

count = count – 1;

}

- Nested If is supported.
- Example – if(count>10) {

count = count -1;

if(count>5) {

display count;

}

}

IF ELSE STATEMENT

- If else can be used in the following format.

```
if(condition) {
    statementlist;
} else {
 statementlist;
}
```

Example

```
if(count>10) {
    count = count – 1;
} else {
    display count;
}
```

Nested if-else is also supported.

WHILE STATEMENT

- While statement can be used in the following format.

```
while(condition) {
    statementlist;
}
```

Example

```
while(a>5) {
    display a;
    a = a-1;
}
```

Nested while is also supported.

FUNCTION CALL

- Function call can be used in the following format

  identifier = functionName(arguments) *arguments are optional

  or

  functionName(arguments) *arguments are optional

- Functions may or may not return a value. Function returning a value should be assigned to a variable of appropriate data type
- Example:
  - int sum; sum=add(a,b);
  - add(a,b);
  - add(5,10);
- Arguments can be of same or different data types

FUNCTION DEFINITION

- Function definition can be used in the following format
  - func functionName(parameters) { statementList; returnStatement }

    or

    func functionName(parameters) { returnStatement } *parameters are optional
- Example:
  - func add(int a,int b){

    int c = a+b;

    return c;

    }
- All the variables in the functions are **locally scoped**
- Parameters can be of same or different data types

RETURN STATEMENT

- Return statement can be used in the following format to return value from a function
  - **return expression**

  Example –

  return count;

  return a+b;

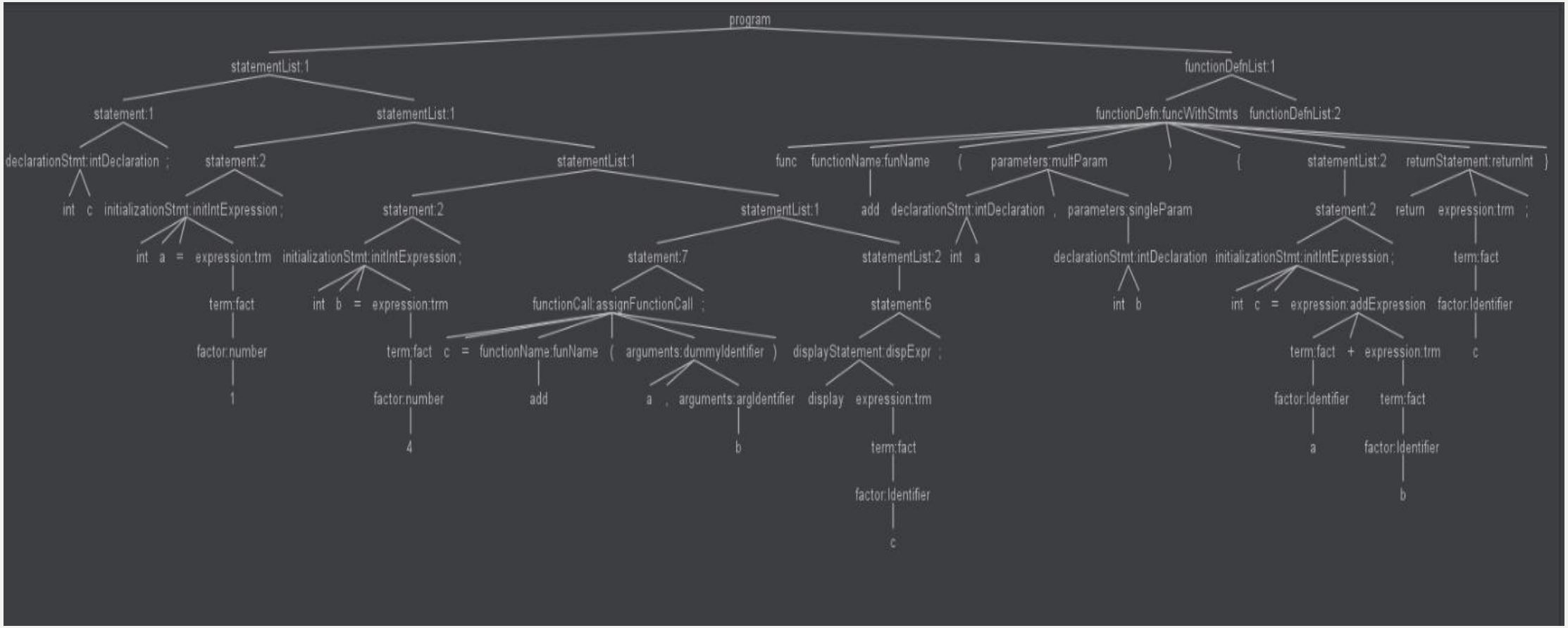
SAMPLE PROGRAM WITH INTERMEDIATE CODE

```
int c;
int a = 1;
int b = 4;
c=add(a,b);
display c;

func add(int a,int b){
int c = a+b;
return c;
}
```

```
DECINT c
DECINT a
PUSH 1
STORE a
DECINT b
PUSH 4
STORE b
LOAD b
LOAD a
CALL add 1 2
END CALL add 1
STORE c
LOAD c
DISPLAY c
HALT
BEGIN FUNC add
DECINT a
DECINT b
DECINT c
LOAD a
LOAD b
ADD
STORE c
LOAD c
RET c
END FUNC add
```

SAMPLE PARSE TREE



FUTURE IMPROVEMENTS

The popularity of a programming language depends to some extent, on its designer's willingness to extend its features. On this note, we would like to improve the following for *STARK* in future :

- Support more data types and unary operators
- Support additional looping constructs
- Support data structures like array and stack
- Support recursion for user-defined functions

REFERENCES

- Kenneth C. Louden and Kenneth A. Lambert, Programming Languages Principles and Practice, Third edition, Boston. Cengage Learning, 2011
- Alfred V. Aho and Ravi Sethi, Compilers Principles, Techniques and Tools
- Antlr4 documentations
- SER502 Spring2018 lecture slides

ACKNOWLEDGEMENT

We express our sincere thanks to our professor Mr. Ajay Bansal and his teaching assistant Xiangyu Guo for their continued support and encouragement. We appreciate the learning opportunities they provided to us.

Thank You!