NIHAO

SER 502 PROGRAMMING LANGUAGES

TEAM 28

VIBHUTI TRIPATHI

PRASHANTH RADHAKRISHNAN

JIUXU CHEN

SATISH BHAMBRI

Overview and Key Features

- Operators both Arithmetic , Relational Operators and Assignment Operator
  Arithmetic Operators - + , - , /, *
  Relational Operators - >, >=, < , <= , ==
- Decision Control Statement – If and Else
- Looping Construct – While
- Numeric Data Type – Int
- Stack Machine Model

Nihao Design Flow

Tools and Environment

- ANTLR 4 for Lexical and Syntactic Analysis
- Intermediate Code Generation in Java
- Runtime Environment designed using Python 2.7

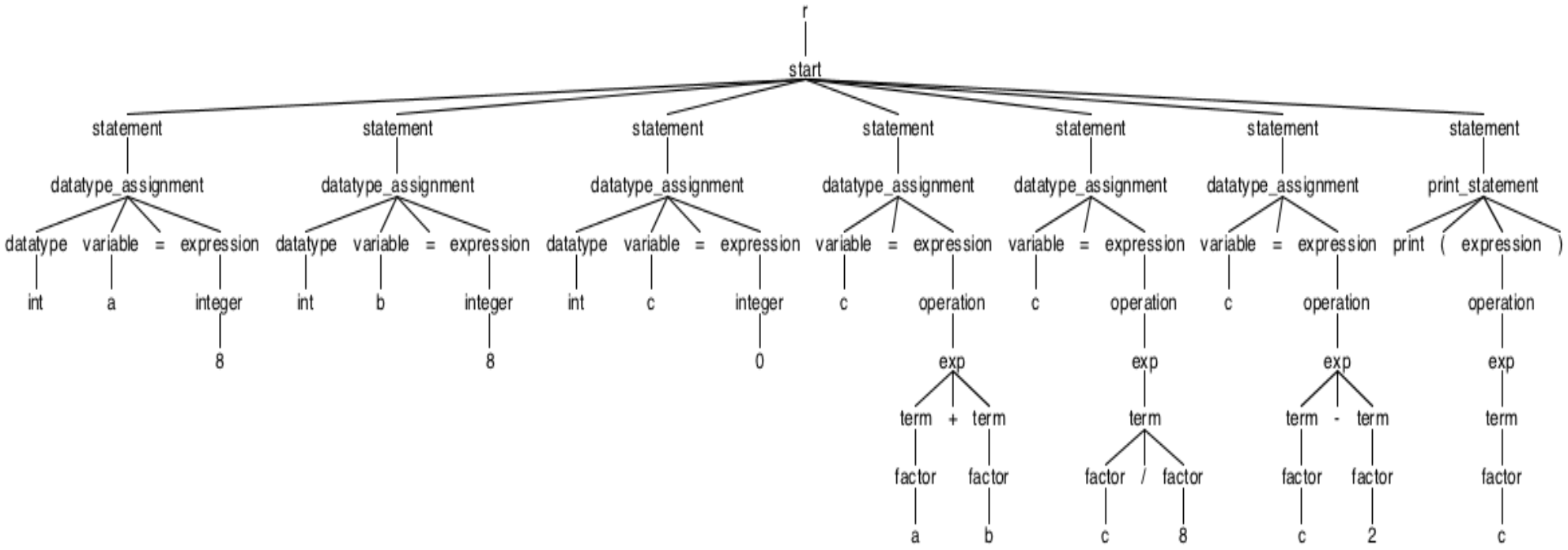
Build the Nihao Compiler

- Build the context-free grammar for our language.
- Generate the tokens using the lexical analyzer
- Use the parser to generate the parse-tree.
- Traverse the nodes in the parse-tree and generated the Intermediate Code which is in correspondence with the grammar rules and output of code.
- Build a runtime environment which processes Intermediate Code and gives the output for the program.

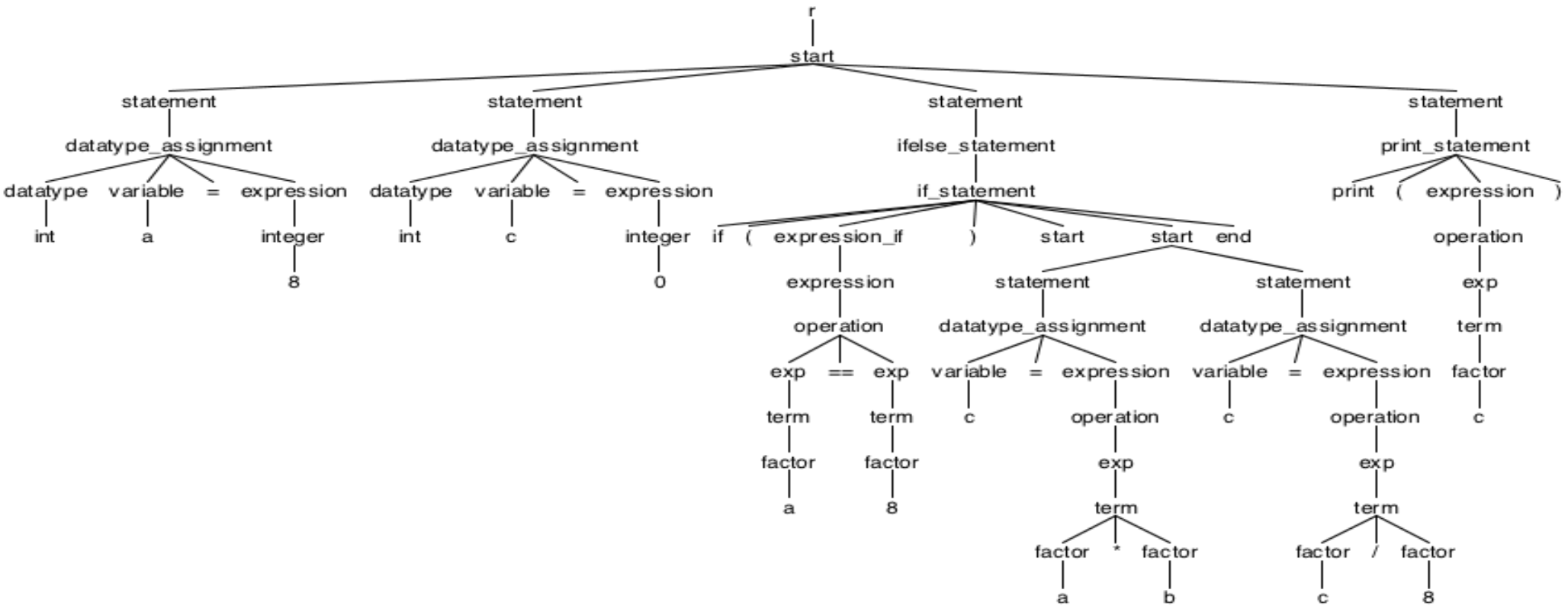
Nihao Grammar

- **Extended Backus–Naur form** (**EBNF**) is a family of meta syntax notations, any of which can be used to express a context-free grammar.
- An EBNF consists of terminal symbols and non-terminal production rules which are the restrictions governing how terminal symbols can be combined into a legal sequence.
- The EBNF defines production rules where sequences of symbols are respectively assigned to a nonterminal:
- **digit excluding zero** = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;

Parse Tree Example For Arithmetic Operations



Parse Tree Example For Decision Control Statement



Antlr

- Antlr is a tool for Java user to generate parse-tree and intermediate code by given .g4 grammar.
- It can do the lexical and syntax analysis automatically.
- Parse-tree is a good method to check whether or not our grammar is defined properly.
- As the tree walker traverses nodes on the parse-tree, we follow this action and implement the exposed inferfaces in order to generate intermediate code.

Intermediate Code Generation

```
@Override
public void enterIf_statement(NihaoParser.If_statementContext ctx) {
    sb.append("IFCONDITION"+'\n');
}
@Override
public void exitIf_statement(NihaoParser.If_statementContext ctx) {
    sb.append("END"+'\n');
}
@Override public void enterElse_statement(NihaoParser.Else_statementContext ctx) {
    sb.append("ELSECONDITION"+'\n');
    sb.append("START"+'\n');
}
@Override public void exitElse_statement(NihaoParser.Else_statementContext ctx) {
    sb.append("END"+'\n');
}
@Override
public void enterWhile_statement(NihaoParser.While_statementContext ctx) {
    sb.append("WHILE"+'\n');
}
@Override
public void exitWhile_statement(NihaoParser.While_statementContext ctx) {
    sb.append("OD"+'\n');
}
```

Intermediate Code Definition

- First we need to define the format of primitive data type called int. For example:

  PUSH 0

  PUSH a

  ASSIGN

  In Nihao, it means either int a=0 or a=0(where data type is optional in our language)

- In addition to these basic operators, we also define Boolean data type, If-then else statement, While-statement, Print and Relational operators.

Sample Program

```
int a=8
int b=8
int c=0
c=a*b
c=c/8
print(c)
```

Intermediate Code for Sample Program

- PUSH 8
  PUSH a
  ASSIGN
  PUSH 8
  PUSH b
  ASSIGN
  PUSH 0
  PUSH c
  ASSIGN
  PUSH b
  PUSH a
  *
  PUSH c
  ASSIGN
  PUSH 8
  PUSH c
  /
  PUSH c
  ASSIGN
  PUSH c
  PRINT

Code Snippets for Runtime Environment and Semantic Analysis

```
token_tags = [
      (r'[ \n\t]+',                    None),
      (r'//.*',                        None),
      (r'\+',                          OPERATOR),
      (r'-',                           OPERATOR),
      (r'\*',                          OPERATOR),
      (r'/',                           OPERATOR),
      (r'<=',                          OPERATOR),
      (r'<',                           OPERATOR),
      (r'>=',                          OPERATOR),
      (r'>',                           OPERATOR),
      (r'!=',                          OPERATOR),
      (r'==',                          OPERATOR),
      (r'TRUE',                        BOOLEAN),
      (r'FALSE',                       BOOLEAN),
      (r'ENDELSE',                     ENDELSE),
      (r'ELSECONDITION',           ELSECONDITION),
      (r'WHILE',                           WHILE),
      (r'DO',                                DO),
      (r'OD',                                OD),
      (r'START',                         START),
      (r'ENDIF',                           ENDIF),
      (r'END',                               END),
      (r'IFCONDITION',                 IFCONDITION),
      (r'POP',                             POP),
      (r'PUSH',                            PUSH),
      (r'PRINT',                           PRINT),
      (r'ASSIGNSTACK',                 ASSIGNSTACK),
      (r'ASSIGN',                          ASSIGN),
      (r'CONDITION.+',                 CONDITION),
      (r'RESSTACK',                        STACK),
      (r'[0-9]+',                          CONSTANT),
      (r'[A-Za-z_][A-Za-z0-9_]*',          VARIABLE),
      (r'\"(\\.|[^\"])*\"',                STRING),
]
```

Code Snippet for Lexer

```
def lexer(token_stream, token_tags):
    position = 0
    tokens = []
    while position < len(token_stream):
        match = None
        for token_tag in token_tags:
            character, tag = token_tag
            regex = re.compile(character)
            match = regex.match(token_stream, position)
            if match:
                expression = match.group(0)
                if tag:
                    token = (expression, tag)
                    tokens.append(token)
                break
        if not match:
            sys.stderr.write('Illegal character: %s\\n' % token_stream[position])
            sys.exit(1)
        else:
            position = match.end(0)
    return tokens
```

Contribution of Team Members

- Satish Bhambri and Vibhuti Tripathi: Designing EBNF Grammar, Semantic Analysis and Runtime Environment in Python
- Prashanth Radhakrishnan and Jiuxu Chen : Lexical and Syntactic Analysis and Intermediate Code Generation

THANKYOU