Team 6- Zeal

Anuradha Shilimkar

Dileep Pandey

Tharun Chintham

Varun Srivastava

Language Design Features

Imperative Language

The design of the language Zeal is imperative. The language design inspired from Java.

Arithmetic operations in zeal:-

a. + -> Addition operator

b. - -> Subtraction Operator

c. * -> Multiplication Operator

d. / -> Division Operator

e. % -> Modulus Operator

Language Design Features

Data Types in Zeal :-

a. Num - Data type to store numerical values like 1,2,3, etc.
b. Bool - Data type to store conditional values like True or False

Relational Operators in Zeal :-

a. <=
b. >=
c. >
d. <
e. =>
f. =<
g. ==
h. !=

Language Design Features

Loop Constructs in Zeal:

1. While Loop

Conditional Constructs in Zeal:

1. If – else
2. Nested if – else

Scope Handling:

a. Scope of variables are valid till the end of the block where they were declared

b. No two variables in a single scope can have a single Identifier

Syntax

Variable Declaration :-

```
num a;
a = 5;
```

Function Declaration :-

```
Function num add(num a, numb)
{
  num result;
 ……
…….
Return result;
};
```

Syntax

Variable Declaration :-

```
num a;
    a = 5;
```

Function Declaration :-

```
Function num add(num a, numb)
{
   num result;
 ……….
…….
Return result;
};
```

IF- else Statement :-

```
if( z < 11 ) {
print("z is lesser than 11");
}else
{
Print ("z is greater than 11");
};
```

While Statement :-

```
while( z < 13 ) {
print(z);
z = z + 2;
};
```

Intermediate Code

- ANTLR
- Used framework for constructing recognizers, interpreters, and compilers.
- Write grammar in a .g4 file.
- Antlr can generate the Parser and Laxer.
- Can also generate visitor class.
- Used BaseVisitor classes to parse the abstract syntax tree created.
- One can visit a parse tree and return low level language of code.
- Abstract syntax tree also support visitChild feature.

Grammar

```
grammar zeal;
//entry point for grammer
program: main_command_list+ ;

main_command_list: command_list #commands
                 | function #functions;

command_list: command
            | command command_list ;

function_command_list: command
            | command command_list ;

label_command_list_if: (command)+;

label_command_list_else: (command)+;

label_command_list_while: (command)+;
```

Grammar (contd.)

```
//assignment, if, while, function calling
command: varName=IDENTIFIER '=' expr ';' #VarAssign
       | declarations ';'  #VarInit
       | 'if' '(' bool_expr ')' '{' label_command_list_if '}' ('else' '{' label_command_list_else '}')? ';' #IfElseBlock
       | 'while' '(' bool_expr ')' '{' label_command_list_while '}' ';' #WhileBlock
       | (data_types)? varName=IDENTIFIER '=' function_call ';' #FunctionToVarAssign
       | function_call ';' #FunctionCall
       | print_statement ';' #PrintExpression
       ;
//num, bool initialisations and datatype declarations
declarations: varName='num' initialization_int
            | varName='bool'  initialization_bool
            | data_types varName=IDENTIFIER
            ;
//num datatype initialisation with multiple identifier
initialization_int: varName=IDENTIFIER '=' INT_VAL
                  | varName=IDENTIFIER '=' initialization_int ;

//bool datatype initialisation with multiple identifier
initialization_bool: varName=IDENTIFIER '=' bool_expr
                   | varName=IDENTIFIER '=' initialization_bool ;
```

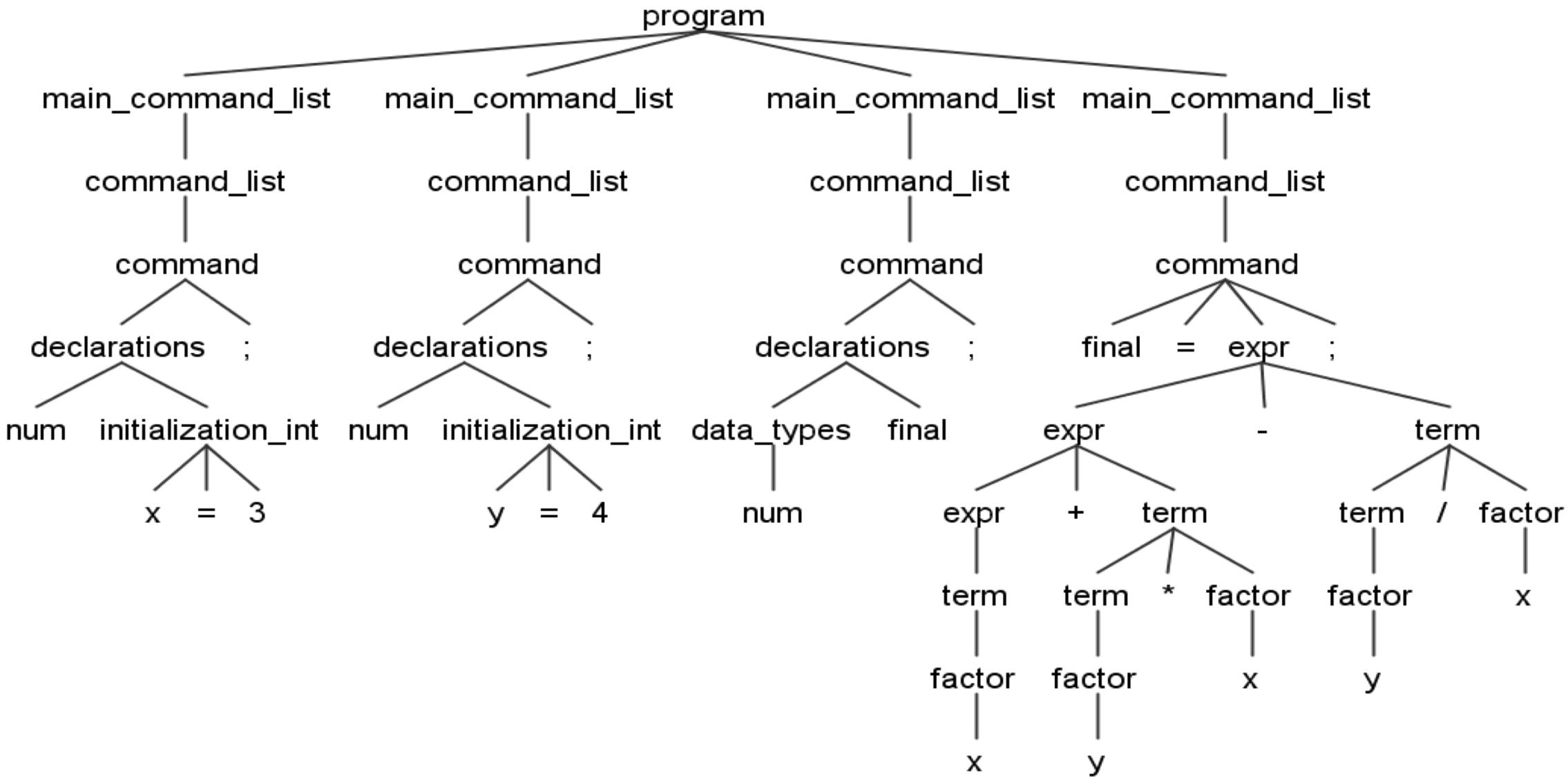
Grammar (contd.)

```
//boolean evaluations
bool_expr: 'true' #TrueExpression
         | 'false' #FalseExpression
         | left=expr '==' right=expr #Equality
         | left=expr '!=' right=expr #NotEqual
         | left=expr '=<' right=expr #EqualLessThan
         | left=expr '<=' right=expr #LessThanEqual
         | left=expr '=>' right=expr #EqualGreaterThan
         | left=expr '>=' right=expr #GreaterThanEqual
         | left=expr '>' right=expr #GreaterThan
         | left=expr '<' right=expr #LessThan
         | '!' '(' bool_expr ')' #NotEqual
         | '(' left=bool_expr ')' '&&' '(' right=bool_expr ')' #AndOperator
         | '(' left=bool_expr ')' '||' '(' right=bool_expr ')' #OrOperator
         ;
//precedence expression evaluation
expr: left=expr '+' right=term #Add
    | left=expr '-' right=term #Sub
    | term #TermExpression
    ;
//modulus multiplication and
term: left=term '*' right=factor #Multiply
    | left=term '/' right=factor #Divide
    | left=term '%' right=factor #Mod
    | factor #FactorExpression
```

Grammar (contd.)

```
function: 'function' returnType=return_types functionName=IDENTIFIER  '(' (argumentList+=arguments)* ')' '{' function_command_list
(return_stmt ';')*  '}';
arguments: data_types varName=IDENTIFIER | data_types varName=IDENTIFIER ',' arguments;

function_call: functionName=IDENTIFIER '(' (params)* ')' ;
params: varName=IDENTIFIER | varName=IDENTIFIER ',' params;

data_types: 'num' #NumericalDataType
          | 'bool' #BooleanDataType;

return_types: data_types #ReturnTypeDataType
            | 'void' #ReturnTypeVoid;

return_stmt: 'return' varName=IDENTIFIER #ReturnVariable
           | 'return' expr #ReturnExpression
           ;
IDENTIFIER: [a-zA-Z]+[0-9]* ;
INT_VAL: [-]? [0-9]+ ;
TEXT: '"' (~["])* '"'
;
WHITESPACE: [ \t\n\r]+ -> skip ;
BLOCK_COMMENT : '/*' .*? '*/' -> skip;
LINE_COMMENT:   '//' ~[\r\n]* -> skip;
```

Abstract Syntax Tree



Intermediate code

ADD,SUB,MUL,DIV for arithmetic operations.

Pop out two constants from stack, perform operation and push back the result

Precedence is properly maintained

Eg. 3 + 5

- push 3
- Push 5
- Add

LOAD and STORE command to move variables between Symbol table and stack and vice-versa

```
num a = b + c;
        load b;
        load c;
        add
        store a
```

Intermediate code (contd.)

- if-else and while statement
- Used labels to indicate statements to be executed if conditions are true or not.
- If_true: label_1_
- If _not_true:lable_1_else

Zeal Code

- num a = 3;
- num b = 2;
- num c;
- if(a < b) {
- c = b;
- }else{
- c = a;
- };

Intermediate code

- LOAD 2
- STORE b
- num c
- LOAD a
- LOAD b
- BGE a, b, label_1_else
- LOAD b
- STORE c
- label_1_else:
- LOAD a
- STORE c
- END

Intermediate Code(contd.)

Definition of function is done between function name and End function.

Number of arguments required by the function is printed. This enables to initialize the arguments that are passed that are passed to it.

Number of arguments is also used when the function is called to indicate variables from stack that need to be pushed as argument to the function.

Now its working for the void return type function

- NUM x
- LOAD 3
- STORE x
- NUM y
- LOAD 4
- STORE y

Intermediate Code (contd.)

- call_function addOperation
- param1:x
- param2:y
- end_function
- FUNCTION addOpration:<void>:<2>
- ARGUMENT num m
- ARGUMENT num n
- num final
- ADD
- LOAD m

- We worked on recursive call of function with void return data type

Intermediate Code (contd.)

```
function void fib(num x)
{
 if(((x == 1)||(x == 2))||(x == 0)){
  print(x);
  } else
  {
  num a;
  a = x - 1;
  num b;
  b = x - 2;
  num c;
  c = fib(a);
  num d;
  d = fib(b);
  num result;
  result = c + d;
  print(result);
  };
```

Intermediate Code(contd.)

```
LOAD 1
STORE y
call_function fib
param1:y
end_function
FUNCTION fib:<void>:<1>
ARGUMENT num x
OR OR LOAD x
LOAD 1
BNE x, 1, LOAD x
LOAD 2
BNE x, 2, LOAD x
LOAD 0
BNE x, 0, label_1_else
WRITE x
label_1_else:
num a
SUB
LOAD x
```

Intermediate Code(contd.)

```
LOAD 1
STORE a
num b
SUB
LOAD x
LOAD 2
STORE b
num c
param2:a
num d
param3:b
num result
ADD
LOAD c
LOAD d
STORE result
WRITE result
END_FUNCTION
END
```

Generating Intermediate Code

- Write the program in the file with .zl extension.
- Automatic compilation will create .zlclass file with the same filename
- This .zlclass can now be fed to the runtime and automatic compile the runtime.