SER 502 – Languages and Programming Paradigm
Spring 2020
Compiler and Virtual Machine for a Programming Language
DESIGN DOCUMENT
Team – 17

Team Members:
Chandan Yadav Nagaraj
Nachiappan Lakshmanan
Nayan Jain
Guru Preetam Kadiri
Milestone: 2
Github Link: https://github.com/ngjain/SER502-Spring2020-Team17
Name: deepC

Program file extension: .dpc

Design goal:
Simple - The language must avoid all unnecessary complexity and improve developer's efficiency of solving the problem.
Clean - The language must provide clean, reliable, consistent applications which can be easily read and understood. It is very important that when another person reads the code, user must be able to understand the synaptic structure of the code and visualise the semantics of the code.
Good performance - The language must provide good performance in terms of time and space.
Modularity - The language must help break the programs to distinct components, which can later combine to solve other problems.

Design:
The program should start with a key word "begin" and end with a key word "end" followed by a "." (dot).

Everything inside this will be considered as a Block.

Block can have series of declarations (i.e. a list of declarations) followed by commands (i.e. a list of commands).

Every declaration ends with a ";" semicolon. Consecutive commands are separated by ";".

In the declaration section the language supports three datatypes:
Integer - a data type that stores integer (numeric type)

booleanValue - a data type that stores boolean values (true, false)

string - a data type that stores sequence of characters (includes lower and upper case). (string value assignments to variables)

Command list can be any of the following constructs:
if then else construct:
Begins with a an “if" followed by a booleanValueList, “then", commandList, "else", commandList,”end_if”.

booleanValueList will accept a sequence of “and”, “or”, “not" operations between relational operators. The core functionality of the booleanValueList is to return true / false for the condition followed by “if" keyword. Validates the condition to true/false.

If the condition is true, executes the commandList in "then" part or executes the commandList in “else" part.

ternary construct:
begins with “if”, booleanValueList,”?”, (booleanValue “:" booleanValue or initialise “:" initialise)

based upon the validation condition, it it returns true then command (initialise) or booleanValue to the left of the “:” will execute or to the right will execute.

Looping constructs:
For Loop:
Traditional for loop*:*
begins with a “for”,”(“, initialisation, booleanValueList, endCondition ,”)”, “do”, commandList, “end_for”.

Initialisation is a part of command List operation. Initialise a variable with numeric type. (This is a way to associate a value to the identifier).

booleanValueList to validate the for condition each time, and end condition which tells the order in which the initialised variable should increment.

End condition can be either a “++” operator or incrementing the variable by a particular value “x:=x+1”.

Python like For Loop:
begins with a “for" identifier” ""in" "range" between two numbers. it will do a list of commands.

While Loop:
Begins with a 'while' while_condition 'do' commandList ‘end_while' While-condition is the booleanValueList.

Print:
It is a construct for printing all the identifier values. It supports all the datatypes supported by our language.

Relational operators:
The language accepts relational operators in the form - expression1 Relational Operator expression2, and its results can directly be stored in booleanValue. The relational operators compatible are:

‘==’ - returns true if both expressions are equal.
‘!=’ - returns true if both expressions are not equal.
‘<’ - returns true if expression1 is less than expression2.
‘>’ - returns true if expression1 is greater than expression2.
‘<=’ - returns true if expression1 is less than or equal to expression2.
‘>=’ - returns true if expression1 is greater than or equal to expression2.
‘and’ - returns true if expression1 and expression2 are true, or both are false else returns false.
‘or’ - returns true if either expression1 or expression2 is true else returns false.

Unary operators:
‘not’ - Not
‘++’ - increment
‘--’ - decrement

Arithmetic operators:
‘+’ - addition
‘-’ - subtraction
‘*’ - multiplication
‘/’ - division

Implementation:

Source code written in deepC

```
begin
integer x:=10;
integer z:=0;
integer i;
integer y;
integer a;
for(i:=0;i<100;i++)
do
y:=20;
y:=y+1;
while x<1000
do
z:=z+1;
x:=x+y;
if x>10 and z<10
then
print(a)
else
a:=y+x
end_if
end_while
end_for;
print(x);
print(z);
end.
```

STEP ONE:

LEXICAL ANALYSIS is the very first phase our compiler design. A lexer takes the modified source code which is written in the form of sentences. In other words, it helps you to convert a sequence of characters into a sequence of tokens. The lexical analyser breaks this syntax into a series of tokens.

ANTLRFileStream inputfile = new ANTLRFileStream(program); #program is the source code written in deepC which will be available in the file test.smp.

simpleLexer lexer = new simpleLexer(inputfile);
CommonTokenStream tokens = new CommonTokenStream(lexer);

STEP TWO:
After generating the token, the parser gets these token and check whether the source code can be the grammar to the programming language we built. It will generate a parse tree to verify the source code written in deepC.

```
simpleParser parser = new simpleParser(tokens);
ProgramContext tree = parser.program();
```

We make use of ANTLR for bot the above steps.

STEP THREE:
We will generate the intermediate code by giving semantics to the parse tree in java (ANTLR gives override functions for the grammar written).

Intermediate code for the above source program:

```
ASSIGNINT x 10
STORE x
ASSIGNINT z 0
STORE z
INTEGER i
INTEGER y
INTEGER a
FORBEGIN 1
PUSH 0
STORE i
FOR_EVALUATE 1
LOAD i
PUSH 100
LESSTHAN
JUMP FOREND 1
INCREMENT i
PUSH 20
STORE y
LOAD y
PUSH 1
ADD
STORE y
```

```
WHILE_BEGIN 1
LOAD x
PUSH 1000
LESSTHAN
JUMP WHILE_END 1
LOAD z
PUSH 1
ADD
STORE z
LOAD x
LOAD y
ADD
STORE x
IFBEGIN
LOAD x
PUSH 10
GREATERTHAN
LOAD z
PUSH 10
LESSTHAN
AND
JUMP ELSE_BEGIN
PRINT a
ELSE_BEGIN
LOAD y
LOAD x
ADD
STORE a
ELSE_END
IFEND
LOOP WHILE_BEGIN 1
WHILE_END 1
LOOP FOR_EVALUATE 1
FOREND 1
PRINT x
PRINT z
```

STEP FOUR:

Runtime environment will interpret the intermediate code and produce the output. This is also written in java.

Data Structures Used:

STACK:

We used stack for expression evaluation in the runtime. It is a recursive data structure with which you can push, pop variables at any instant and evaluate expressions with arithmetic operators.

MAP:

We used HashMap<String,Integer> to keep track of the variables state each time.

Grammar:

```
grammar deepC;

DIGIT
: [0-9]+ ('.'[0-9]+)?
| [0-9]+ '.' [0-9]+ ('.'[0-9]+)?
;

string : (lowerCharacter | upperCharacter)+;
lowerCharacter : 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' |'n' | 'o' | 'p' | 'q' | 'r'
| 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z' ;
upperCharacter : 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' |'M' | 'N' | 'O' | 'P'
| 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' ;

identifier : lowerCharacter;

program: block '.' ;
block: 'begin' declarationList ';' commandList ';' 'end' ;

declarationList : declaration| declaration ';' declarationList;

declaration
: datatype='integer' name=identifier #intDeclaration
| datatype='booleanValue' name=identifier #BooleanDeclaration
| datatype='string' name=identifier #StringDeclaration
| 'integer' name=identifier ':=' val=primary #intAssignment
| 'booleanValue' name=identifier ':=' val=booleanValue
#booleanAssignment
| 'string' name=identifier ':=' val=string #stringAssignment
```

```
| #nodec
;

commandList: command| command ';' commandList;

command: 'if' if_condition 'then' commandList elsepart 'end_if'
#ifElseStatement
| booleanValueList ternarycondition ternary #boolTernary
| 'while' while_condition 'do' commandList 'end_while'#whileStatement
| 'for' '('initialize ';'for_condition';' endCondition ')' 'do' commandList 'end_for'
#forLoopStatement
| 'for' for_range commandList 'end_for' #forLoopPython
| initialize #initializeDummy1
| print #printdummy
//|endCondition #dummyEnd
;

ternarycondition: '?';
for_range: name=identifier 'in' 'range' '(' begin=DIGIT ',' end=DIGIT')';
elsepart: 'else' commandList;
if_condition: booleanValueList;
while_condition:booleanValueList;
for_condition: booleanValueList;

initialize: name = identifier ':=' val=string #variableStringAssignment
| name = identifier ':=' val=booleanValue #variableBooleanAssignment
| name = identifier ':=' val=expression #variableExpressionAssignment
;

ternary: cond1 ':' cond2 #ternaryInitialize
| booleanValue ':' booleanValue #ternarbooleanValue
;

cond1 : initialize;
cond2 : initialize;

endCondition : initialize #initializeDummy2
| name=identifier '+' '+' #incremental
| name=identifier '-' '-' #decremental
;
```

```
booleanValue: 'true' #trueStatement
|'false' #falseStatement
| expression '==' expression #equalExpression
| expression '!=' expression #notequalExpression
| 'not' '('booleanValue')' #negation
| expression '<' expression #lessThan
| expression '>' expression #greaterThan
| expression '>=' expression #greaterThanOrEqual
| expression '<=' expression #lesserThanOrEqual
;

booleanValueList: booleanValue #booleanDummy
| booleanValue 'and' booleanValueList #logicalAndRelation
| booleanValue 'or' booleanValueList #logicalOrRelation
;

expression : expression '+' term1#addition
| expression '-' term1#subtraction
| term1 #term1dummy;

term1: term1 '*' term2 #multiplication
| term1 '/' term2 #division
| term2 #dummyterm2
;

term2:
'(' expression ')' #bracketExpression
| name=identifier #identifierVariable
| primary #number;
print: 'print' '(' name=identifier ')' #printidentifier
|'print' '(' name=string ')' #printstring
;

primary: DIGIT
;
```

Security
Status
Help
Contact GitHub
Pricing
API
Training
Blog
About