Reference Manual

PROGRAMMING LANGUAGE

TEAM 23 | SER 502 | APRIL 14, 2017

PROF. AJAY BANSAL

CONTENTS

1. INTRODUCTION

The design goal of Orange is to create a new language that is as simple as peeling an orange. Creating this new language will enable users to get an idea of how to implement the language methodically in a simple manner.

The first step in building Orange is to define the grammar of the language for which we use Flex as lexer/scanner and Bison as a parser.

Orange is a high-level language which is compiled, interpreted and also which generates intermediate code. This intermediate code is read line by line and executed in runtime environment.

2. COMPILING AND RUNNING ORANGE PROGRAMS

Orange source file should have .or as extension
Intermediate file has .pulp as extension

3. OPERATORS OF ORANGE

OPERATORS	DESCRIPTION
“+”	Addition
“-”	Subtraction
“*”	Multiplication
“/”	Division
“AND”	AND
“OR”	OR
“NOT”	NOT
“->”	Assignment operator
“~”	delimiter
“->->”	Comparison operator
“!->”	Not Equal to
“->>”	Greater than equal to
“-><“	Leasser than equal to
“>”	Greater than
“<”	Lesser than
“YES”	True
“NO”	False
“LP“	Left parentheses
“RP”	Right parentheses
“{“	Block begin
“}”	Block end

Keywords

Int	Integer
Float	Float
Boo	Boolean
Read	Input
write	output
nl	New line
is	if
isnot	else
stack	Stack
push	Push
pop	Pop
loop	while loop

4. LEXICAL ANALYSIS

Lexical analysis is the first phase of compilation.
During this phase, source code received character-by-character is transformed into a sequence of "tokens."

For example, for the following expression in our language:
write (3 + x *2) ~

the resulting stream of tokens might be:
(keyword "write")
(delim "(")
(int 3)
(operator "+")
(identifier "x")
(operator "*")
(int 2)
(delim ")")
(delim “~”)

5. SIMPLE ORANGE PROGRAMS

I/O Operation

Our language takes an input from the user using the “read” statement and displays output using the “write” statement as shown below.

```
int a~
read a~
write a~
```

Output

```
Enter value : 2
2
```

Expected intermediate code for the above is:

```
decl a
put a
get a
dsp
end
```

Conditional statements

```
a=4~

is LP a > 0 RP

{

        write a

}
```

Output:

4

Expected Intermediate code for the above is :

get 4

put a

get a

get 0

grt

bne 8

get a

dsp

end

Looping construct

Our language uses “loop” to define while loop as shown below.

```
read a~
loop LP a > 1 RP~
{~
    write a~
    a = a - 1 ;
}~
```

Output :

Enter value: 2

2

Expected intermediate code for the above is:

amc

put a

get a

get 1

grt

get a

dsp

sub

get a

get 1

put a

bne 8

end

6. GRAMMAR:

Parsing is done in accordance with BNF rules and using the following grammar in our language:

<PROGRAM> → <BLOCK>

<BLOCK> → {<read> '~'| <write> '~'| <while> '~'| <if> '~'| <statement> '~' | <var> '~'} | <assignment>

<assignment> → <identifier> '-->' <exp>

<while> → { 'loop' 'LP' <exp> 'RP' '{' <BLOCK> '}' }

<var> → {'int'|'float'|'boo'} <identifier>

<if> → { 'is' 'LP' <exp> 'RP' <BLOCK> ['isnot' <BLOCK>] }

<read> → 'read' <exp> '~'

<write> → 'write' <exp> '~'

exp → {exp1 + exp2 | exp1 - exp2 | exp1 * exp2 | exp1 / exp2| "(" exp1 ")" | identifier | int | float | boo} '~'

<identifier> → [a-zA-Z]

int → {D}{D}*

float → {D}+"."+?{D}*

boo → 'YES' | 'NO'

D → [0-9]