BILKENT UNIVERSITY

CS315 PROJECT-1

Gogit Language

Furkan Kazım Akkurt 21702900 Section-02

Eren Şenoğlu 21702079 Section-02

Cemal Gündüz 21703004 Section-01

Language Name: Gogit

The Complete BNF Description

<comment> ::= <hashtag><sentence><endline>

<program> ::= <begin><stmts><end>

<stmts> ::= <stmt> | <stmts><stmt>

<stmt> ::= <if_stmt><end_stmt> | <nonif_stmt><end_stmt>

<nonif_stmt> ::= <loop> | <function_call> | <assignment> | <set_functions> |
<input_stmt> | <output_stmt> | <int_dec>

<input_stmt> ::= input <variable>

<output_stmt> ::= output <sentence> | output <expression>

<loop> ::= <while_stmt> | <dowhile_stmt>

<while_stmt> ::= while (<logical_exp>) then <stmts> endwhile

<dowhile_stmt> ::= do (<stmts>) while <logical_exp>

<end_stmt> ::= <semicolon>

<assignment> ::= <variable><assignment_op><expression> |

<identifier><assignment_op><arithmetic_exp>
|<identifier><assignment_op><number> |
<identifier><assignment_op><logical_exp>

<if_stmt> ::= <matched_if> | <unmatched_if>

<matched_if> ::= if(<logical_exp>) then <matched_if> else <matched_if>

| <nonif_stmt>

<unmatched_if> ::= if(<logical_exp>) then <if_stmt>

| if(<logical_exp>) then <matched_if> else <unmatched_if>

<expression> ::= <logical_exp> | <arithmetic_exp> | <set_exp> | <variable>

<arithmetic_exp> ::= <number><arithmetic_op><number> |
<number><arithmetic_op><identifier> | <identifier><arithmetic_op><identifier>

| <arithmetic_exp><arithmetic_op><number> |
<arithmetic_exp><arithmetic_op><identifier>

<logical_exp> ::= <expression><logic_op><expression> | <set><logic_op><set>

<function_dec> ::= funcdef <func_name> (<function_parameters>) startf <stmts>
return <stmt> endf

<function_call> ::= <func_name> (<function_parameters>)

<function_parameters> ::= <function_parameter> |
<function_parameters><function_parameter>

<function_parameter> ::= <set> | <number> | <name> | ""

<func_name> ::= <identifier>

<variable> ::= <set>

<identifier> ::= <letter> | <identifier><letter> | <identifier><digit>

<arithmetic_op> ::= <add_op> | <div_op> | <mul_op> | <sub_op> | <remainder_op>

<logic_op> ::= <or_op> | <and_op> | <xor_op> | <equal_op> | <lt_op> | <gt_op> |
<lte_op> | <gte_op> | <not_op> | <subset_op> | <supset_op>

<set> ::= <dollar_sign> <identifier>

<set_exp> ::= <LCB> <element_list> <RCB> | <set_union> | <set_intersect> |
<set_diff>

<set_union> ::= <set> union <set> | <set> union <set_union>

<set_intersect> ::= <set> <intersect> < set> | <set> <intersect> <set_intersect>

<set_diff> ::= <set> <diff_op> <set> | <set> <diff_op> <set_diff>

<element_list> ::= <set_element> | <set_element> <comma> <element_list>

<set_element> ::= <set> | <number> | <name>

<number> ::= <digit> | <digit><number>

<name> ::= <identifier> | <number><identifier>

<sentence> ::= empty | <name> | <symbol> | <sentence><name> | <sentence><symbol><sentence>

<set_functions> ::= <represent_set> | <is_empty_set> | <set_addition> | <set_deletion>

<set_deletion> ::= <set> delete <expression> | <set> delete <set> | <set> delete <element_list>

<set_clear> ::= <set> clear

<represent_set> ::= <set> show

<is_empty_set> ::= <set> isempty

<set_addition> ::= <set> add <expression> | <set> add <set> | <set> add <element_list>

<letter> ::= = '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'| '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'

<digit> ::= 0|1|2|3|4|5|6|7|8|9

<symbol> :: = <LCB> | <RCB> | <LSB> | <RSB> | <comma> | <semicolon> | <underscore> | <equal> | <dot> | <char_identifier> | <space>

<LCB> ::= {

<RCB> ::= }

<LSB> ::= [

<RSB> ::=]

<comma> ::= ,

<add_op> ::= +

<sub_op> ::= -

<mul_op> ::= *

<div_op> ::= \/

<remainder_op> ::= %

<or_op> ::= ||

<and_op> ::= &&

<xor_op> ::= xor

<equal_op> ::= ==

<lt_op> ::= <

<gt_op> ::= >

<lte_op> ::= <=

<gte_op> ::= >=

<not_op> ::= !=

<subset_op> ::= subset

<supset_op> ::= supset

<space> ::= “ ”

<sign> ::= +|-

<dot> ::= .

<assignment_op> ::= =

<semicolon> ::= ;

<hashtag> ::= #

<endline> ::= \n

<diff_op> ::= \\

<dollar_sign> ::= $

<begin> ::= begin

<end> ::= end

<union> ::= [Uu]

<intersect> ::= [Nn]

EXPLANATIONS OF CONSTRUCTS

1- <program> ::= <begin><stmts><end>

Program starts with begin reserved word, statements between and end reserved words are considered proper statements and program consist of these three elements.

2- <comment> ::= <hashtag><sentence><endline>

Our comments starts with hashtag followed by sentences until the endline.

3- <stmts> ::= <stmt> | <stmts><stmt>

Statements are a statement or a collection of statements.

4- <stmt> ::= <if_stmt><end_stmt> | <nonif_stmt><end_stmt>

Statement can be an if statement or a non-if statement. For each case statements end with end statement.

5- <nonif_stmt> ::= <loop> | <function_call> | <assignment> | <set_functions>

| <input_stmt> | <output_stmt

Non-if statement can be a loop, a call to function, an assignment statement, a call for set functions, input statement to take input or output statement to give output.

6- <input_stmt> ::= input <variable>

To take input in our program, we use input statements. It consist of input reserved word and a variable, everything that follows these two elements will be added to the specified set.

7- <output_stmt> ::= output <sentence> | output <expression>

To show output, we use “output” reserved word. We can output a sentence or an expression.

8- <loop> ::= <while_stmt> | <dowhile_stmt>

We have two kinds of loops in our language. There are while and do-while loops.

9- <while_stmt> ::= while (<logical_exp>) then <stmts> endwhile

While loop is used with parentheses that consist of a logical expression, in our program. In the parentheses there is logical expression to check if program enters the loop or not. We used “then” reserved word to indicate the where the loop starts to execute. We use “endwhile” reserved word to indicate the end of the loop.

10- <dowhile_stmt> ::= do <stmts> while (<logical_exp>)

Do followed by statements in our program. Loop starts with “do” and ends with “while” reserved words. After while, in the parentheses there is logical expression to check if program enters the loop in second iteration or not because do while loops always perform at least one iteration.

11- <end_stmt> ::= <semicolon>

In our program at the end of every single statement there is a semicolon to indicate that statement is ended.

12- <assignment> ::= <variable><assignment_op><expression>

| <identifier><assignment_op><arithmetic_exp>
|<identifier><assignment_op><number>

|<identifier><assignment_op><logical_exp>

Assignment is a type of non-if statement. In this statement type there is an assignment operator in the middle. Left hand side of assignment operator will be assigned to expression on the right hand side. Our variable is just sets, so all expression can be the left hand side of a variable. Arithmetic and logical expressions and numbers are the left hand side of “number” type. All numeric results as well as boolean results will be hold with that type. Boolean results will stored as C type, 0 will be false and the other values will be considered as true.

13- <if_stmt> ::= <matched_if> | <unmatched_if>

There is two kinds of if statement. Matched if is used for if there is an if statement for every else in program, unmatched_if used for if there is discrepancy about number of if's and else's.

14- <matched_if> ::= if(<logical_exp>) then <matched_if> else <matched_if>

| <nonif_stmt>

To match if and else's there is "if" and "else" reserved words. There is logical expression to check if statements between then and else will be executed or not. Statements after then and else must be matched if or matched if must be a non-if statement to define matched if recursively.

15- <unmatched_if> ::= if(<logical_exp>) then <if_stmt> | if(<logical_exp>) then <matched_if> else <unmatched_if>

Form of unmatched if statements are similar to matched if statements. Same "if" and "else reserved word are used to decide which statement will be executed. In this form after if there is no else or after if there will be matched if and after else there will be unmatched if to continuously keep unmatched status.

16- <expression> ::= <logical_exp> | <arithmetic_exp> | <set_exp> | <variable>

In our program there are four kinds of expressions. An expression can be logical expression, arithmetic expression, set expression or it can be a variable.

17- <arithmetic_exp> ::= <number><arithmetic_op><number>

| <number><arithmetic_op><identifier>

| <identifier><arithmetic_op><identifier>

|<arithmetic_exp><arithmetic_op><number>

| <arithmetic_exp><arithmetic_op><identifier>

In our program, arithmetic expressions are conducted with arithmetic operator between numbers or arithmetic expression and a number, to make it operate on more than two numbers.

18- <logical_exp> ::= <expression><logic_op><expression>

| <set><logic_op><set>

Logical expressions are expressions which returns boolean variable(true or false). Logical expression are conducted with logical operators between expressions or sets.

19- <function_dec> ::= funcdef <func_name> (<function_parameters>) startf <stmts> return <stmt> endf

To declare function we use “funcdef” reserved word for declaring the function. After funcdef we give a name for function. Then in parenthesis there could be given argument which can be parameter list or it can be empty. “startf” reserved word used to determine where the function statements will start and “endf” reserved word determines where the function statements end. With “return” reserved word we determine what function will return if there is any.

20- <function_call> ::= <func_name> (<function_parameters>)

To call functions, function name that is followed by parentheses is the correct grammar. In the parenthesis there could be argument given which is set, or it can be empty.

21- <function_parameters> ::= <function_parameter>

| <function_parameters><function_parameter>

Function parameters can be a single parameter or a number of parameters.

22- <function_parameter> ::= <set> | <number> | <name> | ""

Function parameter indicates what we can use as a parameter for function in Gogit language. It can be a set, a number, a name or it can be left as empty.

23- <func_name> ::= <identifier>

Function names must apply the rules of identifier.

24- <variable> ::= <set>

In Gogit language variables are sets.

25- <identifier> ::= <letter> | <identifier><letter> | <identifier><digit>

Identifier’s first character must be letter and remaining part can be digit or letter.

26- **<arithmetic_op> ::= <add_op> | <div_op> | <mul_op> | <sub_op>**

| <remainder_op>

Arithmetic operations consist of add, division, multiplication, subtraction and remainder operation.

27- <logic_op> ::= <or_op> | <and_op> | <xor_op> | <equal_op> | <lt_op> | <gt_op> | <lte_op> | <gte_op> | <not_op> | <subset_op> | <supset_op>

All logic operations that are specified with a special symbol or string form the logic operations.

28- <set> ::= <dollar_sign> <identifier>

Sets can only be named as dollar sign followed by an identifier. It can be said that dollar signs specializes the sets.

29-<set_exp> ::= <LCB> <element_list> <RCB> | <set_union>

| <set_intersect> | <set_diff>

In Gogit, expressions that are special to sets are indicated by set expressions. Left curly bracket-element list- right curly bracket for set declaration or assignment, there is union operation, intersect operation and difference operation are the special functionalities for sets.

30- <set_union> ::= <set> <union> <set> |<set_exp> <union> <set>

Union operation can be done by union operator in the middle. As a result of this operation unique elements of both sets will be collected into a single set. Since this expression is left recursive, our language accepts more than one union operations in a statement.

31- <set_intersect> ::= <set><intersect> < set>

| <set_exp> <intersect > <set>

Set intersection operation can be done by intersection operator in the middle. As a result of this operation elements that both set have will be collected into a single set. This expression can be consist of different number set expressions as well. Like union, also this expression is left recursive and accepts more than one intersect operations.

32- <set_diff> ::= <set> <diff_op> <set> | <set_diff> <diff_op> <set>

Set diff expression used for finding difference of sets. This expression used with difference operator and can be used with combining different number of set expressions as well. Like the previous operators, this also supports more than one operation in a statement.

33- <element_list> ::= <set_element> |<element_list><comma><set_element>

Our element list consist of one or more numbers of set elements and each must be separated by a comma symbol.

34- <set_element> ::= <set> | <number> | <name>

Set elements in Gogit can be a set, a number or a name which might also have numbers in it.

35- <number> ::= <digit> | <digit><number>

Numbers consist of one or more digits.

36- <name> ::= <identifier> | <number><identifier>

We planned names as strings which consist of digits and letters. There is no rules in their order, however names can not include symbols, operators or other special characters.

37- <sentence> ::= empty | <name> | <symbol> | <sentence><name>

| <sentence><symbol><sentence>

Sentences are symbol involved version of the names.

38- <set_functions> ::= <represent_set> | <is_empty_set> | <set_addition>

| <set_deletion> | <set_clear>

Set functions are pre-written set functions that support basic functionalities to ease working experience with sets. Functionalities involve printing set, checking if empty, addition or deletion of an element or deleting all elements of a set which called as clear.

39- <set_clear> ::= <set> clear

Clear function deletes all elements that are present in the set.

40- <represent_set> ::= <set> show

Show function prints out all the elements that are present in the set.

41- <is_empty_set> ::= <set> isempty

This function used to check if set is empty or not, returns boolean depending on the condition.

42- <set_addition> ::= <set> add <expression> | <set> add <set>

| <set> add <element_list>

User can add elements with this function. Function allows user to add an expression, a set or an element list. If user performs “3+5” as expression “8” will be added to set.

43- <set_deletion> ::= <set> delete <expression> | <set> delete <set>

| <set> delete <element_list>

This function used to delete one or more set elements from the set. Can be also used to delete the result of specific expression. The example above also holds for the deletion operation.

44- <letter> ::= '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'

| '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'

It is the list of all usable letters in our language.

45- <digit> ::= 0|1|2|3|4|5|6|7|8|9

It is the list of all usable digits in our language.

46- <symbol> :: = <LCB> | <RCB> | <LSB> | <RSB> | <comma> | <semicolon>

| <underscore> | <equal> | <dot> | <char_identifier> | <space>

It is the list of all usable symbols in our language.

47- <LCB> ::= {

Left curly bracket used to indicate starting point of set expression which shows the set elements.

48- <RCB> ::= }

Right curly bracket used to indicate ending point of set expression which shows the set elements.

49 -<LSB> ::= [

Left square bracket can be used in sentences which forms comments.

50- <RSB> ::=]

Right square bracket can be used in sentences which forms comments.

51- <comma> ::= ,

Comma is used to separate set elements. If a set consists of multiple elements, comma is placed between each of them.

52- <or_op> ::= ||

Used for the logical or operation.

53- <and_op> ::= &&

Used for logical and operation.

54- <xor_op> ::= xor

Reserved word “xor” performs the xor operation.

55- <equal_op> ::= ==

Equality operator used to determine two variables are equal or not.

56- <lt_op> ::= <

This operator is used to determine a variable is less than another variable.

57- <gt_op> ::= >

This operator is used to determine a variable is greater than another variable.

58- <lte_op> ::= <=

This operator is used to determine a variable is less than or equal to another variable.

59- <gte_op> ::= >=

This operator is used to determine a variable is greater than or equal to another variable.

60- <not_op> ::= !=

This operator is used as a conditional operator to compare to variables. It checks inequality of the operands.

61- <subset_op> ::= subset

This reserved is used to indicate a subset of a set. It is used to replace the subset symbol in math.

62- <supset_op> ::= supset

This reserved word is used to function in superset operation which checks superset condition of two set. It is used to replace the super set symbol in math.

63- <space> ::= “ “

Space character is used widely in programs. Between words or non-terminals space is used to separate them.

64- <sign> ::= -

Plus and minus signs are used in several places. One use of minus sign is in representing negative integers and floating point numbers. Also they are used in addition and subtraction.

65- <dot> ::= .

Used in floating point numbers and sentences.

66- <assignment_op> ::= =

Used to assign an identifier or a variable to another identifier or variable.

67- <semicolon> ::= ;

Semicolon is used to indicate the end of a statement.

68- <hashtag> ::= #

Hashtag is used to define comments in the program.

69- <endline> ::= \n

Used to indicate the end of the line.

70- <diff_op> ::= \/

This operator is used to perform subtraction for sets.

71- <dollar_sign> ::= $

This symbol indicates that the given variable is a set.

72- <begin> ::= begin

Reserved word that indicates the beginning of the program.

73- <end> ::= end

Reserved word that indicates the end of the program.

74- <union> ::= [Uu]

This is operator for union expression of sets.

75- <intersect> ::= [Nn]

This is operator for intersection expression of sets.

LEX DESCRIPTION

BOOLEAN true|false

BEGIN begin

END end

DIGIT [0-9]

LETTER [A-Za-z]

UNION [Uu]

INTERSECT [Nn]

PLUS_OP \+

MINUS_OP \-

MUL_OP *

DIV_OP \/

ASSIGNMENT \=

REMAINDER_OP \%

DIFF_OP \\

NEW_LINE \n

EQUALITY_OP \=\=

HASHTAG #

IF if

THEN then

ELSE else

DO do

WHILE while

INPUT input

OUTPUT output

FUNCDEC funcdec

STARTF startf

ENDF endf

RETURN return

OR_OP \|\|

AND_OP \&\&

XOR_OP xor

LT_OP \<

GT_OP \>

LTE_OP \<\=

GTE_OP \>\=

NOT_OP !=

SUBSET_OP subset

SUPSET_OP supset

SEMICOLON \;

SIGN [-]

DOLLAR_SIGN \$

LCB \{

RCB \}

LSB \[

RSB \]

LP \(

RP \)

COMMA \,

ADD add

DELETE delete

CLEAR clear

SHOW show

EMPTY isempty

SPACE [ ]

```
NUMBER ({DIGIT})+

FLOATING_POINT {DIGIT}*(\.{DIGIT}+)

IDENTIFIER {LETTER}({LETTER}|{DIGIT})*

NAME {IDENTIFIER}|({NUMBER}{IDENTIFIER})

COMMENT {HASHTAG}([^\n])*\n



%%

{ADD} {printf("ADD ");};

{DELETE} {printf("DELETE ");};

{CLEAR} {printf("CLEAR ");};

{SHOW} {printf("SHOW_CONTENT ");};

{UNION} {printf("UNION ");};

{EMPTY} {printf("IS_EMPTY ");};

{INTERSECT} {printf("INTERSECT ");};

{FUNCDEC} {printf("FUNCDEC ");};

{STARTF} {printf("STARTF ");};

{INPUT} {printf("INPUT ");};

{OUTPUT} {printf("OUTPUT ");};

{END} {printf("END ");}

{ENDF} {printf("ENDF ");};

{DIFF_OP} {printf("DIFF_OP ");};

{LETTER} {printf("IDENTIFIER ");};

{BEGIN} {printf("BEGIN ");}
```

```
{BOOLEAN} {printf("BOOLEAN ");}

{PLUS_OP} {printf("PLUS_OP ");}

{MINUS_OP} {printf("MINUS_OP ");}

{MUL_OP} {printf("MUL_OP ");}

{DIV_OP} {printf("DIV_OP ");}

{REMAINDER_OP} {printf("REMAINDER_OP ");}

{ASSIGNMENT} {printf("ASSIGNMENT ");}

{NEW_LINE} {printf("NEW_LINE ");}

{EQUALITY_OP} {printf("EQUALITY_OP ");}

{HASHTAG} {printf("HASHTAG ");}

{IF} {printf("IF ");}

{THEN} {printf("THEN ");}

{ELSE} {printf("ELSE ");}

{DO} {printf("DO ");}

{WHILE} {printf("WHILE ");}

{RETURN} {printf("RETURN ");}

{OR_OP} {printf("OR_OP ");}

{AND_OP} {printf("AND_OP ");}

{XOR_OP} {printf("XOR_OP ");}

{LT_OP} {printf("LT_OP ");}

{GT_OP} {printf("GT_OP ");}

{LTE_OP} {printf("LTE_OP ");}

{GTE_OP} {printf("GTE_OP ");}

{NOT_OP} {printf("NOT_OP ");}
```

```
{SUBSET_OP} {printf("SUBSET_OP ");}

{SUPSET_OP} {printf("SUPSET_OP ");}

{SEMICOLON} {printf("SEMICOLON ");}

{DOLLAR_SIGN} {printf("DOLLAR_SIGN ");}

{LCB} {printf("LCB ");}

{RCB} {printf("RCB ");}

{LSB} {printf("LSB ");}

{RSB} {printf("RSB ");}

{LP} {printf("LP ");}

{RP} {printf("RP ");}

{COMMA} {printf("COMMA ");}

"$"{IDENTIFIER} {printf("SET ");}

{NUMBER} {printf("NUMBER ");}

{FLOATING_POINT} {printf("FLOATING_POINT ");}

{IDENTIFIER} {printf("IDENTIFIER ");}

{NAME} {printf("NAME ");}

{COMMENT} {printf("COMMENT ");}



. {printf("%s", yytext);}


%%

int yywrap(void) {

    return 1;
```

```
}
int main(void){
    yylex();
    return 0;
}
```

PROGRAM EXAMPLES

Program Example1

```
begin
#set declarations first
$cemo = {11,22,33,araba,elma1};
$eren = {ufp361,panky361};
$akkurt = {korkusuz};
#trying set operations
$akkurt show;
$akkurt add "99";
$eren = $akkurt u $cemo;
$eren delete;
$cemo isempty;
$eren = {11,22,33};
```

```
$akkurt = $cemo / $eren;

#trying if statement

if ($eren subset $cemo) then delete $cemo ;

else $akkurt = $cemo n $eren ;

funcdec examplefunc() #declare the function

startf

$cemo = {1,2,3};

return 0 ;

endf

examplefunc(); #call the function

$cemo show;

end
```

Program Example 2

```
begin

$a = {1,2,3,hello};

$b = {3,4,world};

$c = {try1,{helloWorld,1,2},try2};

$d = $c U $b n $a ;

$d delete {helloWorld,1,2} ;

$a clear ;

a = 8 + -3 ;

b = 1 ;

do
```

```
$c add b;
a = a - 1 ;
while (a > 0 )
end
```

Program Example 3

```
begin
$set1 = {1,2,3,4};
$set2 = {23};
funcdef foo()
startf
if (3 == 3) then
$uniSet = $set1 U $set2;
else
$set1 clear;
return 1;
endf
#infinite loop
while (true) then
$set1 add 321;
endwhile
end
```

EVALUATION OF COGIT

While creating this language we have given our most of attention to make this language easy to use for who will work with sets.

Gogit's readability is its most significant feature. We chose reserved words very simple like "delete", "add" so that anyone can read and understand how program works easily.

Writability is also considered a lot. Operators are chosen from their own mathematical structure(+ plus, - minus) so that author remembers how to implement things easily. Also there is not only a single way to implement something, for example author can use parameters while writing his function or he can leave it empty and create his function without any parameter.

Finally to make reliability maximum we tried our language with different example programs includes edge cases of language and see how language reacts these cases. By doing that we corrected our problems with reliability.