**Terminal Symbols**

* <WOOF!Program>
* <ProgramDefinition>
* <RoutineDefinition>
* <TrickDefinition>
* <CallStatement>
* <ReturnStatement>
* <PrintStatement>
* <InputStatement>
* <IfStatement>
* <DoWhileStatement>
* <ForStatement>
* <Assertion>

**Non-terminal Symbols**

* NOSE
* TAIL
* BARK
* EAT
* DO
* WHILE
* FOR
* TO
* BY
* ASSERT
* ROUTINE
* TRICK
* COMMAND
* RETURN
* IN
* OUT
* IO
* REFERENCE
* ;
* .
* ,
* :
* =
* //
* /\* \*/
* \n
* +
* -

**Pseudo-terminal Symbols**

* <statement>
* <identifier>
* <string>
* <letter>
* <digit>
* <comment>
* <ASCIICharacter>
* <datatype>
* <expression>
* <conjunction>
* <negation>
* <comparison>
* <comparator>
* <term>
* <factor>
* <secondary>
* <prefix>
* <primary>
* <variable>
* <datatype>
* <formalParameter>

**BNF for the WOOF! Programming Language**

<WOOF!Program> ::= <ProgramDefinition> EOPC

<ProgramDefinition> ::= NOSE

{ <statement> }\*

TAIL

<statement> ::= ((<PrintStatement> |

<InputStatement> |

<AssignmentStatement> |

<IfStatement> |

<DoWhileStatement> |

<ForStatement> |

<CallStatement> |

<ReturnStatement> |

<Assertion>));

<PrintStatement> ::= BARK((<string> | \n | ENDL | Expression))

{ . ((<string> | \n | ENDL | Expression)) }\*

<identifier> ::= <letter> { ((<letter> | <digit>)) }\*

<string> ::= “ { <ASCIICharacter> }\* “

<letter> ::= A | B | C | D | … | Z | a | b | c | d | … | z

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

<comment> ::= // { <ASCIICharacter> }\* EOLC

| /\* {( (<ASCIICharacter> | EOLC ) )}\* \*/

<ASCIICharacter> ::= || Every printable ASCII character in range [ ‘ ‘, ‘~’ ]

<DataDefinition> ::= <VariableDefinition> | <ConstantDefinition>

<VariableDefinition> ::= VAR <identifier> : <datatype>

{, <identifier> : <datatype> }\*

<ConstantDefinition> ::= LET <identifier> : <datatype>

{, <identifier : <datatype> }\*

<datatype> ::= ((INT | BOOL))

<InputStatement> ::= EAT([<string>] <variable>)

<AssignmentStatement> ::= <variable> { , <variable> }\* = <expression>

<expression> ::= <conjunction> { (( OR | NOR | XOR )) <conjunction> }\*

<conjunction> ::= <negation> { (( AND | NAND )) <negation> }\*

<negation> ::= [ NOT ] <comparison>

<comparison> ::= <comparator> [ (( < | <= | = | > | >= | (( <> | != )) )) <comparator> ]

<comparator> ::= <term> { (( + | - )) <term> }\*

<term> ::= <factor> { (( \* | / | % )) <factor> }\*

<factor> ::= [ (( ABS | + | - )) ] <secondary>

<secondary> ::= <prefix> [ (( ^ | \*\* )) <prefix> ]

<prefix> ::= <primary> | (( ++ | -- )) <variable>

<primary> ::= <variable> | ( <expression> ) | <literal>

<variable> ::= <identifier>

<formalParameter> ::= [ (( IN | OUT | IO | REFERENCE )) ] <identifier> : <datatype>

<literal> ::= <integer> | <boolean> | <string>

<integer> ::= <digit> { <digit> }\*

<boolean> ::= true | false

<IfStatement> ::= IF(<expression>) {

{ <statement> }\*

}

{ ELIF(<expression>){

{<statement}\* }

}\*

[ ELSE {

{ <statement> }\*

}]

<DoWhileStatement> ::= Do { {<statement>}\* }

While(<expression>) { {<statement>}\* }

<ForStatement> ::= FOR( <variable> = <expression> TO <expression> [BY <expression> ]) {

{ <statement> }\*

}

<Assertion> ::= ASSERT( <expression> )

<CallStatement> ::= COMMAND <identifier>[( ((<expression> | <variable>))

{ , ((<expression> | <variable>)) }\* )];

<ReturnStatement> ::= DROP({<variable>}\*);

<RoutineDefinition> ::= ROUTINE <identifier> ([ ( <formalParameter> { , <formalParameter> }\* ) ]) {

{ <DataDefinition> }\*

{ <statement> }\*

}

<TrickDefinition> ::= TRICK <identifier> : <datatype> ( [<formalParameter> {, <formalParameter>}\*]) {

{ <DataDefinition> }\*

{ <statement> }\*

<ReturnStatement>

}

**Introduction to static semantics**

The current static semantics are as follows:

1. All identifiers must begin with a <letter>, but they can contain either a <digit> or a <letter> after the first <letter>.
2. The <IfStatement> and the <DoWhileStatement> must compute boolean values.
3. Values must be combined with operators in type-appropriate ways. For example, you cannot add an integer to a boolean; however, you can add an integer to an integer.

**Description of the syntax and dynamic semantics**

WOOF! Is a free format programming language and is also a case-neutral programming language.

All <statement>s must be terminated with a semicolon.

A single line <comment> may appear at the end of any source line; meaning, a single line comment extends from the // prefix to the end of the line containing the //.

A multi-line <comment> may appear any place white-space may appear. The block comment extends from the /\* prefix to the \*/ and block comments may be nested.

The <PrintStatement> can add strings, integers, and expressions to the console output. The output of “\n” causes the buffer to be output as a single line on the user’s console. ENDL also writes a new line to the output buffer. A <PrintStatement> is denoted by the keyword “BARK” which is followed by open and closed parentheses (). The value(s) included between the parentheses will be output to the console. To concatenate more than one value (i.e. two strings since this language currently only supports printing of string literals), simply add a period between the two strings as shown in the example below.

*Example <PrintStatement>*

BARK(“Hello”.” World!”);

The <InputStatement> provides the ability to read console input from the user. An <InputStatement> is denoted by the keyword “EAT” which is followed by open and closed parentheses (). In between the parentheses, there should be a string to prompt a user for input. The string should be followed by an <identifier> where the input will be stored. An example is shown below

*Example<InputStatement>*

EAT(“x? ” x);

WOOF! Programming Language also allows for the declaration of variables. A modifiable variable is denoted by the keyword “VAR” and constants are denoted by the keyword “LET.” An identifier must follow the keyword which is then followed by a colon and the type. An example of variable and constant definitions can be found below.

*Example Variable and Constant Definition*

//Variable declaration

VAR x : INT;

//Constant declaration

LET y : INT;

The <IfStatement> allows for code execution based on a conditional statement. The <IfStatement> should start with the keyword “IF,” followed by an expression enclosed in parentheses. Next there should be an “open curly brace ({),” followed by the statements to be executed. The first part of the <IfStatement> is closed by a “close curly brace (}).” The <IfStatement> can also include other (optional) conditional blocks which are denoted by the keyword “ELIF.” The “ELIF” blocks follow the same structure as the “IF” block. The <IfStatement> can also end with an optional clause denoted by the keyword “ELSE.” The “ELSE” keyword should then be followed by a series of statements enclosed in {}. An example <IfStatement> is provided below.

*Example <IfStatement>*

IF( x == 1 ) {

BARK(“Hello”);

} ELIF( x == 2) {  
 BARK(“WORLD”);

} ELSE {

BARK(“ELSE”);

}

WPL also includes a <DoWhileStatment> which begins with the keyword “DO” followed by statements enclosed in {}. After the }, should be the keyword “WHILE” and an expression enclosed in (). Next, should be a series of statements that are enclosed in { }. The code in the “DO” block will execute once and then keep executing each time the expression in the () evaluates to true. The code in the “WHILE” block will only execute each time the expression in the () evaluates to true, meaning it may not execute at all. An example <DoWhileStatement> can be found below.

*Example <DoWhileStatement>*

DO {

EAT(“x? ” x);

} WHILE(x <> 0) {

BARK(“Hello!”);

}

WPL includes a counted loop called a *for loop*. A For loop (<ForStatement>) starts with the keyword “FOR” which is followed by some parameters that determine how long the loop will run. The parameters are enclosed in parenthesis. There is a total of three possible parameters for the for loop (one of the parameters is optional). The first parameter is a variable to serve as a counter followed by the keyword “TO” and an expression (the second parameter). The expression after the keyword “TO” is where the loop will stop. The last parameter is optional; this parameter specifies what the counter will increment by and is denoted by the keyword “BY” followed by an integer. After the parameters should follow a series of statements enclosed in {}. The <ForStatement> ends after the }. An example of the <ForStatement> can be found below.

*Example <ForStatement>*

LET x: INT = 10;

VAR i: INT;

FOR(i = 1 TO x BY 1){

BARK(“i = ”.i.ENDL);

}

<Assertion> statements are also included. Assertions are denoted by the keyword “ASSERT” followed by an expression enclosed in parenthesis. Should the expression evaluate to false, a run-time error will occur, else the program will continue running as normal. An example can be found below.

*Example <Assertion>*

LET x: INT = 10;

ASSERT(x < 20);

A <RoutineDefinition> is how WPL handles subprograms. subprograms in WPL are globally scoped and include (1) a globally unique identifier, (2) an optional list of <formalParameter>s, and (3) the subprogram body. The formal paramaters can either be IN, OUT, INOUT, or REFERENCE parameters. The body of the subprogram can contain as many or as few <statement>s as the developer wishes. A subprogram is denoted by the keyword “ROUTINE” followed by the identifier and the formal parameters enclosed in (). The body of the subprogram contains a series of <statement>s enclosed in {}. The subprogram ends with the optional <ReturnStatement> or at the } if a <ReturnStatement> is not present. The formal parameterAn example of a <RoutineDefinition> is produced below.

*Example <RoutineDefinition>*

ROUTINE(IN x1: INT, OUT x2: BOOL, IO x3: INT, REFERENCE x4: BOOL) {

LET x5 : INT = 10;

x1 = x3 + x5;

x2 = NOT x4;

x3 = 24;

x4 = x2;

RETURN

}

A <CallStatement> is what SPL uses to run the subprograms mentioned above. Denoted by the keyword “COMMAND,” a <CallStatement> will run the subprogram specified after the keyword with the parameters passed. The call statement is structured as follows: COMMAND identifier(parameters);. The identifier is the identifier of the subprogram and the parameters passed must match the types of the formal parameters in the <RoutineDefinition>. An example <CallStatement> is produced below.

*Example <CallStatement>*

COMMAND subprogram1(x1, x2, x3, x4);

A <TrickDefinition> is a function in the Woof! Programming Language. Similar to the <RoutineDefinition> a <TrickDefinition> also takes in a parameters and can return values. All functions must have return types specified at the functions definition. Functions can return any of the data types supported in WPL. Another note on functions in WPL is they can be defined to be recursive. Unlike the <RoutineDefinition>, a <TrickDefinition> is not called using the <CallStatement>. Instead, the function is called by using the function’s identifier followed by the values to be passed enclosed in parenthesis. An example <TrickDefinition> is produced below

*Example <TrickDefinition>*

TRICK DoAThing : INT(IN x) {

VAR y: INT;

y = 5;

DROP(x+y);

}

//This is how to call a function;

BARK(“DoAThing Result = ”. DoAThing(5));