CS 315 – PROJECT 2

Group Number: 59

Group Members:

- **Süleyman Semih Demir - 21702949 - Section 3**
- **Denizhan Kemeröz - 21703471 - Section 3**
- **Ahmet Ayberk Yılmaz - 21702250 - Section 1**

Language Name: AETHER

BNF DESCRIPTION:

1. Program:

<program> ::= BEGIN <statements> END

<statements> ::= <statement> SEMICOLON | <statements> <statement> SEMICOLON | COMMENT | statement SEMICOLON COMMENT | statements statement SEMICOLON COMMENT

<statement> ::= <matched> | <unmatched>

<simple_statement> ::= <declaration> | <initialization> | <declare_function> | <call_function> | <reading> | <printing>

<matched> ::= <matched_if> | <matched_while> | <matched_for> | <simple_statement>

<unmatched> ::= <unmatched_if> | <unmatched_while> | <unmatched_for>

<matched_if> ::= IF (<comparisons>) { <matched> } ELSE { <matched> }

<unmatched_if> ::= IF (<comparisons>) { <statements> } | IF (<comparisons>) { <matched> } ELSE { <unmatched> }

2. Types:

<type> ::= BOOLEAN | INT | CHAR | STR | DOUBLE | TIME

<types> ::= BOOLEANVALUE | INTEGERVALUE | CHARVALUE | STRINGVALUE | DOUBLEVALUE | TIMEVALUE

<true> ::= “1” | “true”

<false> ::= “0” | “false”

<sign> ::= PLUS | MINUS

3. Simple Statements:

3.1. Declaration:

<declaration> ::= <type> IDENTIFIER

3.2. Initialization:

<initialization> ::= <assignment> | <type> IDENTIFIER EQUAL <arithmetic_expression> | <constant> EQUAL <arithmetic_expression>

<constant> ::= CONST <type> IDENTIFIER

<expressions> ::= <arithmetic_expression> | <expression>

<arithmetic_expression> ::= <arithmetic_expression> + <term> |
<arithmetic_expression> - <term> | <term>

<term> ::= <term> * <power> | <term> / <power> | <power>

<power> ::= <factor> ^ <power> | <factor>

<factor> ::= (<arithmetic_expression>) | INTEGERVALUE |
<call_function> | IDENTIFIER

<expression> ::= <prior_expression> | <secondary_expression>

<prior_expression> ::= IDENTIFIER <operator> IDENTIFIER |
<boolean> | <call_function>

<secondary_expression> ::= <primary_expression> <operator>
<secondary_expression> | <primary_expression>

<operator> ::= AND | OR

3.2.1. Assignment:

<assignment> ::= IDENTIFIER EQUAL <arithmetic_expression>

3.3. Comparisons:

<comparisons> ::= <comparison> | <comparison> <operator>
<comparisons>

<comparison> ::= IDENTIFIER <comparison_operator> IDENTIFIER |
IDENTIFIER <comparison_operator> <types> | <types>
<comparison_operator> <types> | <types> <comparison_operator>
IDENTIFIER | <types> <comparison_operator> <call_function> |
IDENTIFIER <comparison_operator> <call_function> |
<call_function> <comparison_operator> <call_function> |

<call_function> <comparison_operator> IDENTIFIER |
<call_function> <comparison_operator> <types>

<comparison_operator> ::= EQUALS | NOTEQUALS | BIGGER |
SMALLER | BIGGEROREQUALS | SMALLEROREQUALS

3.4. Function:

<declare_function> ::= FUNCTION <type> IDENTIFIER (<parameters>
) { <statements> <return_statement> } | FUNCTION <type>
IDENTIFIER ( ) { <statements> <return_statement> }

<parameter> ::= IDENTIFIER | <types> | <type> IDENTIFIER

<parameters> ::= <parameter> | <parameter> COMMA <parameters> |
<call_function>

<call_function> ::= CALL IDENTIFIER (<parameters>) | CALL
IDENTIFIER ( ) | <primitive_functions>

<return_statement> ::= RETURN IDENTIFIER SEMICOLON |
RETURN types SEMICOLON | RETURN SEMICOLON

3.5. Reading / Printing:

<reading> ::= READ (IDENTIFIER) SEMICOLON

<printing> ::= PRINT (IDENTIFIER) SEMICOLON | PRINT (<type>)
SEMICOLON | PRINT (<call_function>) SEMICOLON

3.6. Loop:

<matched_for> ::= FOR (<initialization> SEMICOLON <comparisons>
SEMICOLON <assignment>) { <matched> }

<unmatched_for> ::= FOR (<initialization> SEMICOLON
<comparisons> SEMICOLON <assignment>) { <unmatched> }

<matched_while> ::= WHILE (<comparisons>) { <matched> }

<unmatched_while> ::= WHILE (<comparisons>) { <unmatched> }

4. Primitive Functions:

<primitive_functions> ::= <call_read_inclination> | <call_read_altitude> |
<call_read_temperature> | <call_read_acceleration> |
<call_toggle_camera> | <call_take_picture> | <call_read_time> |
<call_connect_computer> | <call_up> | <call_down> | <call_turn_right> |
<call_turn_left> | <call_forward> | <call_back> | <call_left_flip> |
<call_right_flip> | <call_front_flip> | <call_back_flip>

<call_read_inclination> ::= CALL READINCLINATION ( )

<call_read_altitude> ::= CALL readAltitude ( )

<call_read_temperature> ::= CALL READTEMPERATURE ( )

<call_read_acceleration> ::= CALL READACCELERATION ( )

<call_toggle_camera> ::= CALL TOGGLECAMERA ( )

<call_take_picture> ::= CALL TAKEPICTURE ( )

<call_read_time> ::= CALL READTIME ( )

<call_connect_computer> ::= CALL CONNECTCOMPUTER (STRINGVALUE
COMMA STRINGVALUE) | CALL CONNECTCOMPUTER (IDENTIFIER
COMMA STRINGVALUE)

<call_up> ::= CALL UP (INTEGERVALUE) | CALL UP (IDENTIFIER)

<call_down> ::= CALL DOWN (INTEGERVALUE) | CALL DOWN(
IDENTIFIER)

<call_turn_right> ::= CALL TURNRIGHT (INTEGERVALUE) | CALL
TURNRIGHT (IDENTIFIER)

<call_turn_left> ::= CALL TURNLEFT (INTEGERVALUE) | CALL TURNLEFT (IDENTIFIER)

<call_forward> ::= CALL VOID FORWARD (INTEGERVALUE) | CALL VOID FORWARD (IDENTIFIER)

<call_back> ::= CALL BACK (INTEGERVALUE) | CALL BACK (IDENTIFIER)

<call_left_flip> ::= CALL LEFTFLIP ( )

<call_right_flip> ::= CALL RIGHTFLIP ( )

<call_front_flip> ::= CALL FRONTFLIP ( )

<call_back_flip> ::= CALL BACKFLIP ( )

DETAILED EXPLANATION OF ALL CONSTRUCTS OF AETHER LANGUAGE:

1. <program>

<program> ::= BEGIN <statements> END

Think about a sandwich. It has a slice of bread on the top and another one at the bottom. Between these two slices of bread, there are various vegetables, different types of meat or even various fruits. Aether also has a very similar structure to this.

Here, from the BNF description of non-terminal “<program>”, it can be seen that for any program (written in Aether) to run, the BEGIN (bread on the top) and the END (bread at the bottom) commands should be placed before and after all the statements (vegetables, meat, fruits).

2. <statements>

<statements> ::= <statement> SEMICOLON | <statements> <statement> SEMICOLON | COMMENT | statement SEMICOLON COMMENT | statements statement SEMICOLON COMMENT

Non-terminal “<statements>” refers to a group of many single “<statement>”. In Aether, the actual part to be executed is the “<statements>” part.

Here, the most important rule is putting a semicolon after each statement. With this rule, language tells the computer that it should separate the code before the semicolon from the code after the semicolon. For a program which includes more than one statement, “<statements>” has the possibility of “<statements> <statement> SEMICOLON”. This possibility is there for recursion.

This creates a new rule to determine the statements and identify them by adding SEMICOLON after them. Also there can be comments after the statements.

3. <statement>

<statement> ::= <matched> | <unmatched>

A statement can either be a matched or unmatched. Conditional statement is basically an “if else” statement in many languages like Java or C++.

To let the programmer to write a code which includes intertwined “if” loops, Aether language has a recursive structure of “<matched>” non-terminal.

<matched> ::= <matched_if> | <matched_while> | <matched_for> | <simple_statement>

<matched_if> ::= IF (<comparisons>) { <matched> } ELSE { <matched> }

- If the result of the “<comparisons>” is true, then the first “<matched>” non-terminal (after “)”) is regarded as the statement written inside the current if loop. If this statement is conditional, then the first option of “<matched>” non-terminal is chosen again in the second iteration and now the “<comparisons>” non-terminal is regarded as the comparisons statement in the next if loop. It goes in this way up to the iteration.
- If the result of the “<comparisons>” is false, then the entire scenario told above is the same for the second “<matched>” non-terminal (after “)”).

The second option for “<conditional_statement>” is the “<unmatched>” non-terminal.

<unmatched> ::= <unmatched_if> | <unmatched_while> | <unmatched_for>

<unmatched_if> ::= IF (<comparisons>) { <statements> } | IF (<comparisons>) { <matched> } ELSE { <unmatched> }

If the result of the “<comparison>” is true, then the first “<matched>” non-terminal (after “)”) is regarded as the statement written inside the current if loop. Unmatched shows that the else belongs to the <unmatched>’s if. It can also create an if statement without else.

3.1. <simple_statement>

<simple_statement> ::= <declaration> | <initialization> | <declare_function> | <call_function> | <reading> | <printing>

A simple statement is all statements except the “if else” statement. Most of the job is done by this type of statement. Creating any variable and giving a value to it, making some iterative calculations with it; creating some code parts called “functions” to do a job which is needed again and again, reading some value from the user and printing them etc.

3.1.1. <declaration>

<declaration> ::= <type> IDENTIFIER SEMICOLON

With “<declaration>” non-terminal, the programmer can create an empty variable, without assigning a value into it. The only rule is before the variable name, the type should be written and these two different words should be separated with a space between them.

3.1.2. <initialization>

<initialization> ::= <assignment> | <type> IDENTIFIER EQUAL <arithmetic_expression> | <constant> EQUAL <arithmetic_expression>

If the variable to be initialized is declared before, then the first option (explained in part 3.2.3) of non-terminal “<initialization>” would be selected. Otherwise, since there is a rule saying that a variable cannot be initialized if it is not declared before initialization; the second option will be selected.

“<arithmetic_expression>” non-terminal is explained in part 3.1.11

3.1.3. <assignment>

<assignment> ::= IDENTIFIER EQUAL <arithmetic_expression>

This non-terminal is valid for the variables which have already been declared. It is used to assign an integer, character, boolean or a whole string into the variables.

“<arithmetic_expression>” non-terminal is explained in part …….

3.1.4. <matched_for>, <unmatched_for>, <matched_while>, <unmatched_while>

In Aether language, there are only two types of loops namely FOR and WHILE. The syntax for each of them is different from each other.

<matched_for> ::= FOR (<initialization> SEMICOLON <comparisons> SEMICOLON <assignment>) { <matched> }

<unmatched_for> ::= FOR (<initialization> SEMICOLON <comparisons> SEMICOLON <assignment>) { <unmatched> }

At the beginning of the loop, a variable should be initialized with a value, then comparisons should be defined in the signature of for loop. Programmer can do the next step’s calculations for the variable which will be sentinel to the loop. The other parts are controlled by the loop itself. If the comparisons return true, then it means loop will run the statements between brackets one more time. Otherwise, the loop will stop iterating.

Regarding the second type of loop;

<matched_while> ::= WHILE (<comparisons>) { <matched> }

<unmatched_while> ::= WHILE (<comparisons>) { <unmatched> }

Between parentheses, there are only comparisons, so it seems a bit easier than for loop. It also iterates when the until the comparisons return false.

The main difference between two loops is that in the while loop, the new form of the control value is defined within brackets, not parentheses.

“<comparisons>” and “<expression>” non-terminal is explained in part 3.2.15 and 3.2.11.2.

3.1.5. <declare_function>

<declare_function> ::= FUNCTION <type> IDENTIFIER (<parameters>) { <statements> <return_statement> } | FUNCTION <type> IDENTIFIER ( ) { <statements> <return_statement> }

This non-terminal is used to declare a new function with proper signature. According to this signature’s rules, in order to declare a function FUNCTION word is needed as a starter. It is followed by the return type of the function, then the name of the function. Between the parenthesis the parameters are placed (explained in part 3.2.7).

After closing the parenthesis, between the brackets statements should be written. With the proper call by using “<call_function>” (explained in part 3.2.6) the “<statements>” and “<return_statement>”between brackets are executed.

<return_statement> ::= RETURN IDENTIFIER SEMICOLON | RETURN types SEMICOLON | RETURN SEMICOLON

Return statement can be written by using RETURN, IDENTIFIER, and SEMICOLON on order. It can also be written as RETURN types SEMICOLON, or RETURN SEMICOLON.

3.1.6. <call_function>

<call_function> ::= CALL IDENTIFIER (<parameters>) | CALL IDENTIFIER ( ) | <primitive_functions>

This non-terminal is used to declare a new function with proper signature. According to this signature's rules, in order to declare a function "call" word is needed as a starter. It is followed by the name of the function. Between the parenthesis the parameters are placed (explained in part 3.2.7). There could be no parameters or primitive functions (explained in part 4).

3.1.7. <parameters>

<parameters> ::= <parameter> | <parameter> COMMA <parameters> | <call_function>

The parameters include the type and variable or directly the variable, type, or function call. Also, parameters can be more than one by having a comma between.

<parameter> ::= IDENTIFIER | <types> | <type> IDENTIFIER

A parameter can be an IDENTIFIER, <types>, or <type> IDENTIFIER

3.1.8. <comment>

<comment> ::= COMMENT STRINGVALUE NEWLINE

This non-terminal is used to create a comment with proper signature. According to this signature's rules, in order to start commenting COMMENT is needed as a starter. It is followed by the STRINGVALUE of the function.

After the STRINGVALUE, comment finishes with NEWLINE.

3.1.9. <reading>

<reading> ::= READ (IDENTIFIER) SEMICOLON

This non-terminal is used to get input with proper signature. According to this signature's rules, in order to read from the user the word "read" is needed as a starter. After "read", between the parentheses the identifier of the input is needed. Reading finishes with SEMICOLON.

3.1.10. <printing>

<printing> ::= PRINT (IDENTIFIER) SEMICOLON | PRINT (<type>) SEMICOLON | PRINT (<call_function>) SEMICOLON

This non-terminal is used to send output with proper signature. According to this signature's rules, in order to print the word "print" is needed as a starter. After "print", between the parentheses the identifier of the output is needed. Reading finishes with SEMICOLON.

3.1.11. <arithmetic expression>

<arithmetic_expression> ::= <arithmetic_expression> + <term> | <arithmetic_expression> - <term> | <term>

Arithmetic expression uses recursion by adding and subtracting <term>.

<term> ::= <term> * <power> | <term> / <power> | <power>

Term is used recursively by multiplication and dividing <power>, so that these gain priority over adding and subtracting

<power> ::= <factor> ^ <power> | <factor>

Power is used to prioritize the power recursively.

<factor> ::= (<arithmetic_expression>) | INTEGERVALUE | <call_function> | IDENTIFIER

Factor calls <arithmetic_expression> to prioritize the expressions inside the parentheses. Factor also can be INTEGERVALUE or <call_function>.

3.1.12. <expression>

<expression> ::= <prior_expression> | <secondary_expression>

<expression> is used to identify logical expressions specifically by dividing itself to <prior_expression> and <secondary_expression>.

<prior_expression> ::= IDENTIFIER <operator> IDENTIFIER | <boolean> | <call_function>

Prior expression can be <operator>between identifiers or between identifier and constant, or directly can be either function call or boolean.

<secondary_expression> ::= <primary_expression> <operator> < secondary_expression> | <primary_expression>

Secondary expression can be used recursively with <operator> (explained in part 3.2.12) between primary expression and secondary expression, or it

can be primary expression directly. It ensures the precedence in the logical expression part.

3.1.13. <operator>

<operator> ::= AND | OR

This non-terminal is used for identifying the operators with proper signature. According to this signature's rules, <operator> can either be AND, OR, or <comparison_operator>(explained in part 3.2.13)

3.1.14. <comparison_operator>

<comparison_operator> ::= EQUALS | NOTEQUALS | BIGGER | SMALLER | BIGGEROREQUALS | SMALLEROREQUALS

This non-terminal is used for identifying the comparison operators with proper signature. According to this signature's rules, <comparison_operator> can either be EQUALS, NOTEQUALS , BIGGER , SMALLER, BIGGEROREQUALS, or SMALLEROREQUALS.

3.1.15. <comparisons>

<comparisons> ::= <comparison> | <comparison> <operator> <comparisons>

Comparisons create lots of comparisons with an operator recursively.

<comparison> ::= IDENTIFIER <comparison_operator> IDENTIFIER | IDENTIFIER <comparison_operator> <types> | <types> <comparison_operator> <types> | <types> <comparison_operator> IDENTIFIER | <types> <comparison_operator> <call_function> | IDENTIFIER

<comparison_operator> <call_function> | <call_function>
<comparison_operator> <call_function> | <call_function>
<comparison_operator> IDENTIFIER | <call_function>
<comparison_operator> <types>

This non-terminal is used for identifying the comparison with proper signature. According to this signature's rules, <comparison> can be a comparison operator between identifiers, types, function calls or their combinations.

4. Primitive Functions:

<primitive_functions> ::= <call_read_inclination> | <call_read_altitude> | <call_read_temperature> | <call_read_acceleration> | <call_toggle_camera> | <call_take_picture> | <call_read_time> | <call_connect_computer> | <call_up> | <call_down> | <call_turn_right> | <call_turn_left> | <call_forward> | <call_back> | <call_left_flip> | <call_right_flip> | <call_front_flip> | <call_back_flip>

Primitive functions can be one of the decided functions that are explained next.

4.1. <call_read_inclination>

<<call_read_inclination> ::= CALL READINCLINATION ( )

This non-terminal is used to read the inclination with proper signature. According to this signature's rules, in order to read the inclination CALL is needed as a starter. After CALL, the name of the function READINCLINATION is needed. After the name parenthesis open and close.

4.2. <call_read_altitude>

<call_read_altitude> ::= CALL readAltitude ( )

This non-terminal is used to read the altitude with proper signature. According to this signature's rules, in order to read the inclination the word CALLis needed as a starter. After CALL, the name of the function "readAltitude" is needed. After the name parenthesis open and close.

4.3. <read_temperature>

<call_read_temperature> ::= CALL READTEMPERATURE ( )

This non-terminal is used to read the temperature with proper signature. According to this signature's rules, in order to read the temperature CALLis needed as a starter. After CALL, the name of the function READTEMPERATURE is needed. After the name parenthesis open and close.

4.4. <call_read_acceleration>

<call_read_acceleration> ::= CALL READACCELERATION ( )

This non-terminal is used to read the acceleration with proper signature. According to this signature's rules, in order to read the acceleration the word CALLis needed as a starter. After CALL, the function READACCELERATION is needed. After the name parenthesis open and close.

4.5. <call_toggle_camera>

<call_toggle_camera> ::= CALL TOGGLECAMERA ( )

This non-terminal is used to toggle the camera with proper signature. According to this signature's rules, in order to toggle the camera CALLis needed as a starter. After CALL, the name of the function TOGGLECAMERA is needed. After the name parenthesis open and close.

4.6. <call_take_picture>

<call_take_picture> ::= CALL TAKEPICTURE ( )

This non-terminal is used to take a picture with proper signature. According to this signature's rules, in order to take a picture CALL is needed as a starter. After CALL, the name of the function TAKEPICTURE is needed. After the name parenthesis open and close.

4.7. <call_read_time>

<call_read_time> ::= CALL READTIME ( )

This non-terminal is used to read the time with proper signature. According to this signature's rules, in order to read the time, CALL is needed as a starter. After CALL , the name of the function READTIME is needed. After the name parenthesis open and close.

4.8. <call_connect_computer>

<call_connect_computer> ::= CALL CONNECTCOMPUTER (STRINGVALUE COMMA STRINGVALUE) | CALL CONNECTCOMPUTER (IDENTIFIER COMMA STRINGVALUE)

This non-terminal is used to connect to the computer with proper signature. According to this signature's rules, in order to connect to the computer, CALL is needed as a starter. After CALL, the name of the function CONNECTCOMPUTER is needed. After the name between the parenthesis, there are two strings where there is a comma between or IDENTIFIER.

4.9. <call_up>

<call_up> ::= CALL UP (INTEGERVALUE) | CALL UP (IDENTIFIER)

This non-terminal is used to move up with proper signature. According to this signature's rules, in order to move up, CALL is needed as a starter.

After CALL , the name of the function “up” is needed. Then closing parenthesis.

4.10. <down>

<call_down> ::= CALL DOWN (INTEGERVALUE) | CALL DOWN (IDENTIFIER)

This non-terminal is used to move down with proper signature. According to this signature’s rules, in order to move down, CALL is needed as a starter. After CALL, the name of the function “down” is needed. After the name between the parenthesis, the function’s type INTEGERVALUE or IDENTIFIER is written.

4.11. <call_turn_right>

<call_turn_right> ::= CALL TURNRIGHT (INTEGERVALUE) | CALL TURNRIGHT (IDENTIFIER)

This non-terminal is used to turn right with proper signature. According to this signature’s rules, in order to turn right, CALL is needed as a starter. After CALL , the name of the function “turnRight” is needed. After the name between the parenthesis, the function’s type INTEGERVALUE or IDENTIFIER is written.

4.12. <call_turn_left>

<call_turn_left> ::= CALL TURNLEFT (INTEGERVALUE) | CALL TURNLEFT (IDENTIFIER)

This non-terminal is used to turn left with proper signature. According to this signature’s rules, in order to turn left, CALL is needed as a starter. After CALL , the name of the function “turnLeft” is needed. After the name between the parenthesis, the function’s type INTEGERVALUE or IDENTIFIER is written.

4.13. <call_forward>

<call_forward> ::= CALL VOID FORWARD (INTEGERVALUE) | CALL VOID FORWARD (IDENTIFIER)

This non-terminal is used to move forward with proper signature. According to this signature's rules, in order to move forward, CALL is needed as a starter. After CALL , the name of the function "forward" is needed. After the name between the parenthesis, the function's type INTEGERVALUE or IDENTIFIER is written.

4.14. <call_back>

<call_back> ::= CALL BACK (INTEGERVALUE) | CALL BACK (IDENTIFIER)

This non-terminal is used to move back with proper signature. According to this signature's rules, in order to move back, CALL is needed as a starter. After CALL , the name of the function "back" is needed. After the name between the parenthesis, the function's type INTEGERVALUE or IDENTIFIER is written.

4.15. <call_left_flip>

<call_left_flip> ::= CALL LEFTFLIP ( )

This non-terminal is used to flip left with proper signature. According to this signature's rules, in order to flip left, CALL is needed as a starter. After CALL , the name of the function "leftFlip" is needed. After the name parenthesis open and close.

4.16. <call_right_flip>

<call_right_flip> ::= CALL RIGHTFLIP ( )

This non-terminal is used to flip right with proper signature. According to this signature's rules, in order to flip right, CALL is needed

as a starter. After CALL , the name of the function “rightFlip” is needed. After the name parenthesis open and close.

4.17. <call_front_flip>

<call_front_flip> ::= CALL FRONTFLIP ( )

This non-terminal is used to flip front with proper signature. According to this signature’s rules, in order to flip front, CALL is needed as a starter. After CALL , the name of the function “frontFlip” is needed. After the name parenthesis open and close.

4.18. <call_back_flip>

<call_back_flip> ::= CALL BACKFLIP ( )

This non-terminal is used to flip back with proper signature. According to this signature’s rules, in order to flip back, CALL is needed as a starter. After CALL, the name of the function “backFlip” is needed. After the name parenthesis open and close.

EVALUATION OF AETHER LANGUAGE

1. **Readability**

The terms in Aether language mostly depend on English language. This makes them easy to read to people who don’t know this language because most of the people are familiar with English. As an example, in real life people use the terms “up, down, right, left” as the direction indicators. In Aether also, these terms are used to determine the direction of the drone.

Another feature of Aether is, it has many terms which are also used in many different known programming languages like Java, C++ etc. This situation gives the notion that Aether has existed in the market for decades.

2. **Writability**

The terms in Aether language are easy to write because of their similarity with English literature and their short names. Also, there are easier expressions for people's choices like for and while for loop declarations. The small number of the primitive constructs make the language more writible. However, there can be too much orthogonality, so errors can go undetected. The best thing is that in Aether, the most used terms like primitive types are shortened in such a way that when a programmer sees the shortened version of primitive type, this person most probably understands what type it is. Some examples are "int" for integer type, "str" for string types etc.

3. **Reliability**

Aether language has no ambiguous terms, even though the language is short and uses lots of recursion. There is not much error and exception handling but when it is written in proper languages it is a reliable language. To increase the reliability lots of tests are used. One of the most unreliable things is pointers in C++. In the design of the Aether language, for example, pointers are not included consciously for this reason.

ALL CONFLICTS SOLVED

After the first successful compilation, there were 1 shift/reduce and 3 reduce/reduce conflicts.

After printing "y.output" file with command "$cat y.output", we get the the information belov;

"State 150 contains 1 shift/reduce conflict.

State 108 contains 2 reduce/reduce conflicts.

State 158 contains 1 reduce/reduce conflict."

1) In the y.output file , state 150, which has the shift/reduce conflict, was shown as;

"150: shift/reduce conflict (shift 128, reduce 85) on COMMA

state 150

parameters : parameters . COMMA parameters (85)

parameters : parameters COMMA parameters . (85)

COMMA shift 128

CLOSE_P reduce 85”

We solved this conflict by replacing “parameters COMMA parameters” with “parameter COMMA parameters”.

2) State 108 was shown as;

“108: reduce/reduce conflict (reduce 33, reduce 60) on SEMICOLON

108: reduce/reduce conflict (reduce 33, reduce 60) on CLOSE_B

state 108

initialization : IDENTIFIER EQUAL arithmetic_expression SEMICOLON . (33)

assignment : IDENTIFIER EQUAL arithmetic_expression SEMICOLON . (60)

. reduce 33”

This reduce/reduce conflict was related to SEMICOLON. The problem was that we were enforcing all statements to have semicolon at the end and in yacc file, the explanation of “statements” was enforcing statements to have semicolon at the end, again. After we removed SEMICOLON from statements in yacc file, it’s solved.

3) State 158 was shown as;

“158: reduce/reduce conflict (reduce 50, reduce 56) on CLOSE_P

state 158

expression : prior_expression . (50)

secondary_expression : prior_expression . operator secondary_expression (55)

secondary_expression : prior_expression . (56)

AND shift 164

OR shift 165

EQUALS shift 80

NOTEQUALS shift 81

BIGGER shift 82

SMALLER shift 83

BIGGEROREQUALS shift 84

SMALLEROREQUALS shift 85

CLOSE_P reduce 50

operator goto 169

comparison_operator goto 167”

This conflict was caused by the situation that in yacc file, “secondary_expression” was declared as “prior_expression operator secondary_expression | prior_expression”. In other words, because the first possibility could be “prior_expression operator prior_expression”, there was a conflict. We changed the structure and the related other structures and the conflict was gone.