MOZART COMPILER

Sonia Ferratello

Patrizia Stefani

Mateusz Jakub Kubica

IDEA

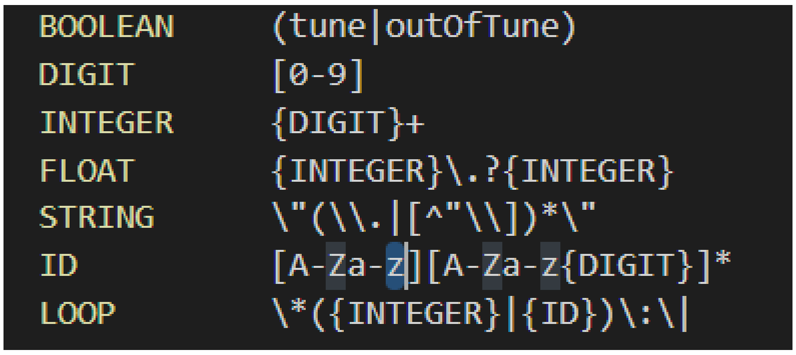
We wanted to implement a language with some music references to make it more original, therefore we decided to use some special music notations as reserved keywords for our language.

LEXICAL ANALYSIS

For the Lexical Analysis part we used the Flex tool.

Declaration

We started by writing the declaration section which includes the assignment of our names to regular expressions in order to recognize the program lexemes.



```
BOOLEAN         (tune|outOfTune)
DIGIT           [0-9]
INTEGER         {DIGIT}+
FLOAT           {INTEGER}\.?{INTEGER}
STRING          \"(\\.|[^"\\])*\"
ID              [A-Za-z][A-Za-z{DIGIT}]*
LOOP            \*({INTEGER}|{ID})\:\|
```

We have a regex that recognize the words “tune” and “outOfTune” which are our BOOLEAN values respectively for true and false.

DIGIT is an integer number from 0 to 9 which is used to build the INTEGER type , which is a composition of digits.

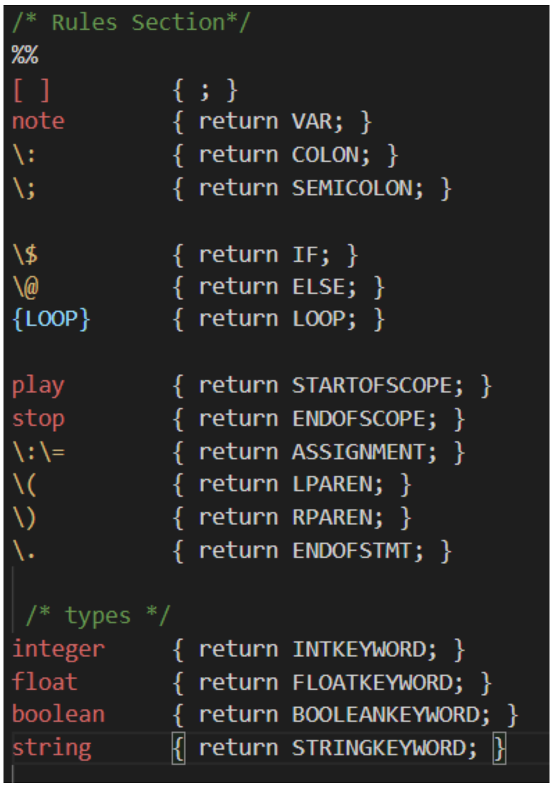
Similarly FLOAT is formed by an INTEGER followed by a "." and optionally by another INTEGER.

On the other side STRING accept all characters between quotation marks.

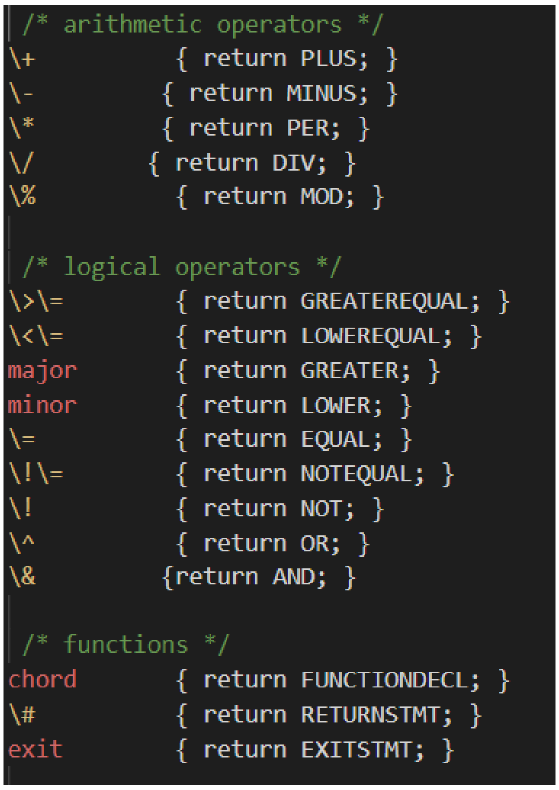
ID is used for variable names and its regex recognizes a lexeme that starts with a letter and then can be followed by other letters or DIGITs.

As last but not least we have LOOP, whose regex admits the following lexeme "*" followed by an INTEGER or by an ID and then by ":|", which is a reference to the music scores symbol . An example of a loop with an INTEGER would be "*4:|", while an example of a loop with an ID would be "*i:|", where 'i' is a previous declared variable.

Rules



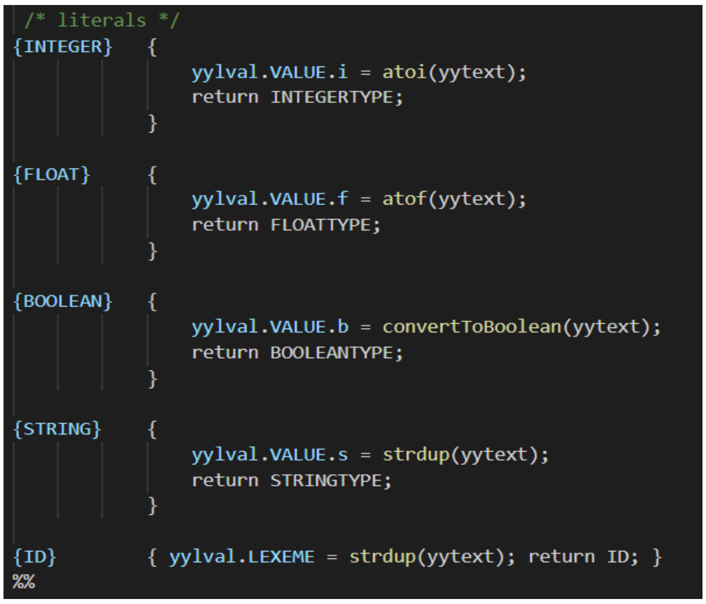
```
/* Rules Section*/
%%
[ ]         { ; }
note        { return VAR; }
\:          { return COLON; }
\;          { return SEMICOLON; }

\$          { return IF; }
\@          { return ELSE; }
{LOOP}      { return LOOP; }

play        { return STARTOFSCOPE; }
stop        { return ENDOFSCOPE; }
\:\=        { return ASSIGNMENT; }
\(          { return LPAREN; }
\)          { return RPAREN; }
\.          { return ENDOFSTMT; }

 /* types */
integer     { return INTKEYWORD; }
float       { return FLOATKEYWORD; }
boolean     { return BOOLEANKEYWORD; }
string      { return STRINGKEYWORD; }
```



```
 /* arithmetic operators */
\+          { return PLUS; }
\-         { return MINUS; }
\*         { return PER; }
\/        { return DIV; }
\%          { return MOD; }

 /* logical operators */
\>\=        { return GREATEREQUAL; }
\<\=        { return LOWEREQUAL; }
major       { return GREATER; }
minor       { return LOWER; }
\=          { return EQUAL; }
\!\=        { return NOTEQUAL; }
\!          { return NOT; }
\^          { return OR; }
\&         {return AND; }

 /* functions */
chord       { return FUNCTIONDECL; }
\#          { return RETURNSTMT; }
exit        { return EXITSTMT; }
```

In the rules section we specified what to do when a regex is matched.

In particular we return a specific keyword for each pattern, that will be then processed by the Syntax Analyzer.



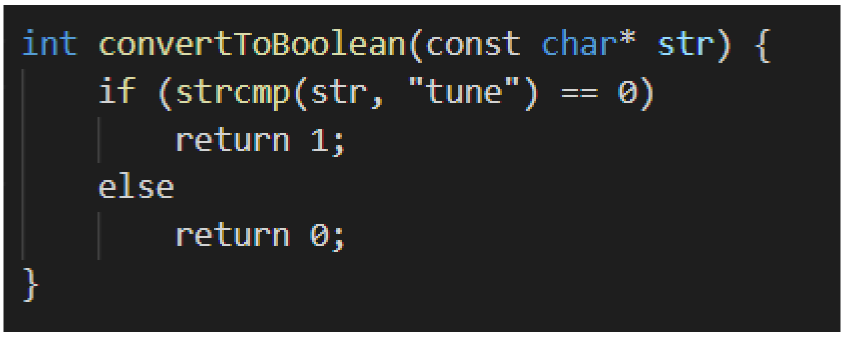
```
/* literals */
{INTEGER}   {
                yylval.VALUE.i = atoi(yytext);
                return INTEGERTYPE;
            }

{FLOAT}     {
                yylval.VALUE.f = atof(yytext);
                return FLOATTYPE;
            }

{BOOLEAN}   {
                yylval.VALUE.b = convertToBoolean(yytext);
                return BOOLEANTYPE;
            }

{STRING}    {
                yylval.VALUE.s = strdup(yytext);
                return STRINGTYPE;
            }

{ID}        { yylval.LEXEME = strdup(yytext); return ID; }
%%
```

Here we take care of the previous declarations.

To explain this part of the lex file, we shall also check out the %union declared in the Parser file. The Lexer passes to the attribute VALUE of global variable yylval the lexeme converted to respectively integer, float, Boolean, string. The VALUE union (defined in the header file types.h) can assume different types (int i, float f, bool b, char* s); in this way we can distinguish between the different types of lexemes.

The ID just pass its string value to yylval.LEXEME and returns the keyword ID.

AUXILIARY FUNCTIONS



```
int convertToBoolean(const char* str) {
    if (strcmp(str, "tune") == 0)
        return 1;
    else
        return 0;
}
```

We decided to write this function so that Boolean values are easier to manage.

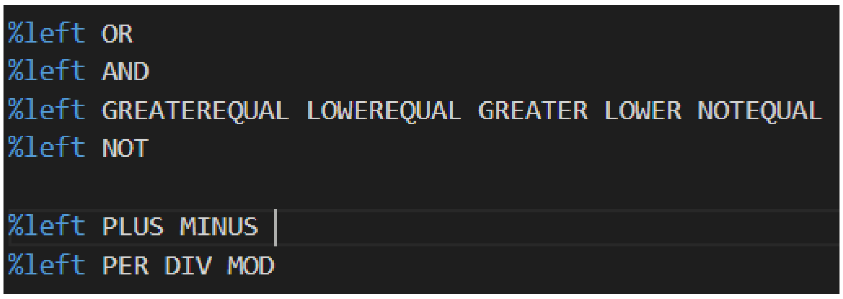
“tune” will be stored as 1, while “outOfTune” will be 0.

SYNTAX ANALYSIS

To create the Parser we used the Bison tool.

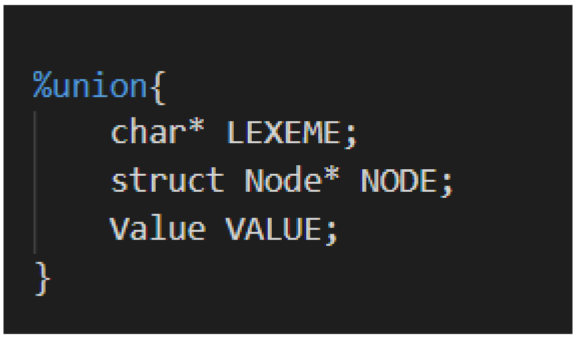
DEFINITIONS

Here we define tokens which are returned from the Lexer.



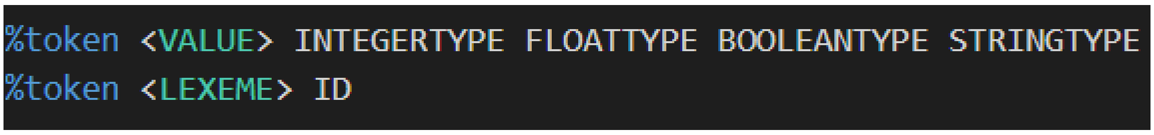
```
%left OR
%left AND
%left GREATEREQUAL LOWEREQUAL GREATER LOWER NOTEQUAL
%left NOT

%left PLUS MINUS
%left PER DIV MOD
```

We defined also the precedence of operators: the below ones have higher precedence than the above ones, while operators declared on the same %left line have the same level of precedence. For instance in the expressions multiplication, division and modulo have higher precedence on addition and subtraction; concerning logical expressions instead, on the lower level we have the OR operator.

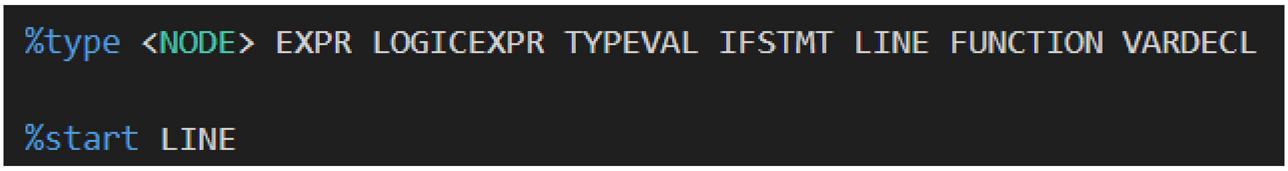


```
%union{
    char* LEXEME;
    struct Node* NODE;
    Value VALUE;
}
```

We defined a %union, which can assume the following types: LEXEME, NODE, VALUE.



```
%token <VALUE> INTEGERTYPE FLOATTYPE BOOLEANTYPE STRINGTYPE
%token <LEXEME> ID
```



```
%type <NODE> EXPR LOGICEXPR TYPEVAL IFSTMT LINE FUNCTION VARDECL

%start LINE
```

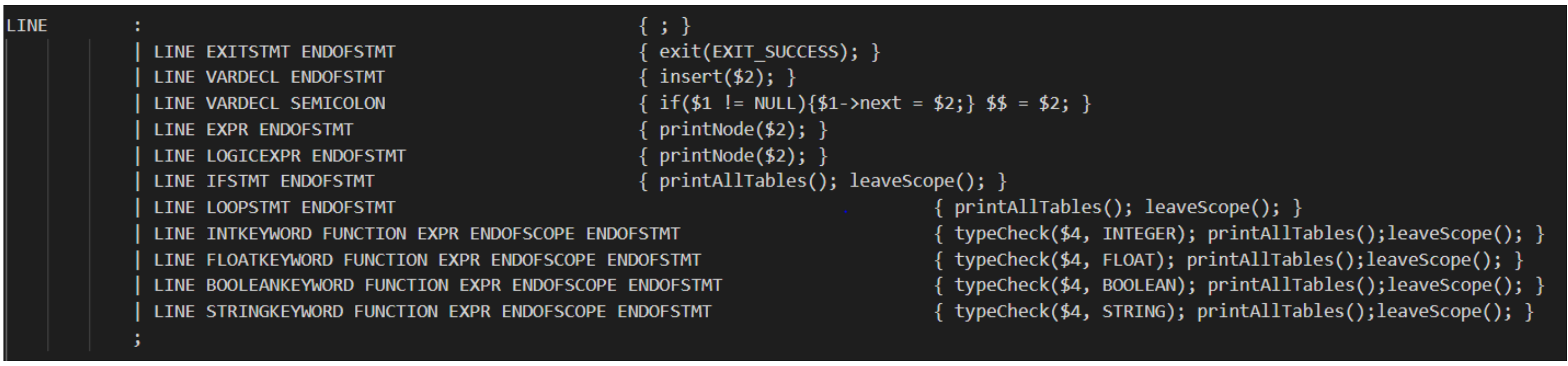
To these particular tokens we assign a specific type, which are used to give the possibility to those tokens and non terminal symbols to assume a value of a particular type.

For example when the Lexical Analyzer recognizes an integer (lets say 6), it passes it to yylval.VALUE.i and returns INTEGERTYPE, so the parser knows that what production rule to apply.

We define also the scope of our grammar, which is LINE.

GRAMMAR RULES WITH ASSOCIATED SEMANTICS

LINE



```
LINE        :                                               { ; }
            | LINE EXITSTMT ENDOFSTMT                       { exit(EXIT_SUCCESS); }
            | LINE VARDECL ENDOFSTMT                        { insert($2); }
            | LINE VARDECL SEMICOLON                        { if($1 != NULL){$1->next = $2;} $$ = $2; }
            | LINE EXPR ENDOFSTMT                           { printNode($2); }
            | LINE LOGICEXPR ENDOFSTMT                      { printNode($2); }
            | LINE IFSTMT ENDOFSTMT                         { printAllTables(); leaveScope(); }
            | LINE LOOPSTMT ENDOFSTMT                                           { printAllTables(); leaveScope(); }
            | LINE INTKEYWORD FUNCTION EXPR ENDOFSCOPE ENDOFSTMT                { typeCheck($4, INTEGER); printAllTables();leaveScope(); }
            | LINE FLOATKEYWORD FUNCTION EXPR ENDOFSCOPE ENDOFSTMT              { typeCheck($4, FLOAT); printAllTables();leaveScope(); }
            | LINE BOOLEANKEYWORD FUNCTION EXPR ENDOFSCOPE ENDOFSTMT            { typeCheck($4, BOOLEAN); printAllTables();leaveScope(); }
            | LINE STRINGKEYWORD FUNCTION EXPR ENDOFSCOPE ENDOFSTMT             { typeCheck($4, STRING); printAllTables();leaveScope(); }
            ;
```

The scope LINE has an empty production and other recursive productions which recognize some specific statements, in particular we can write "exit." to exit the program, we can declare a variable and a function, execute (logical) expressions, write if statements and loop statement.

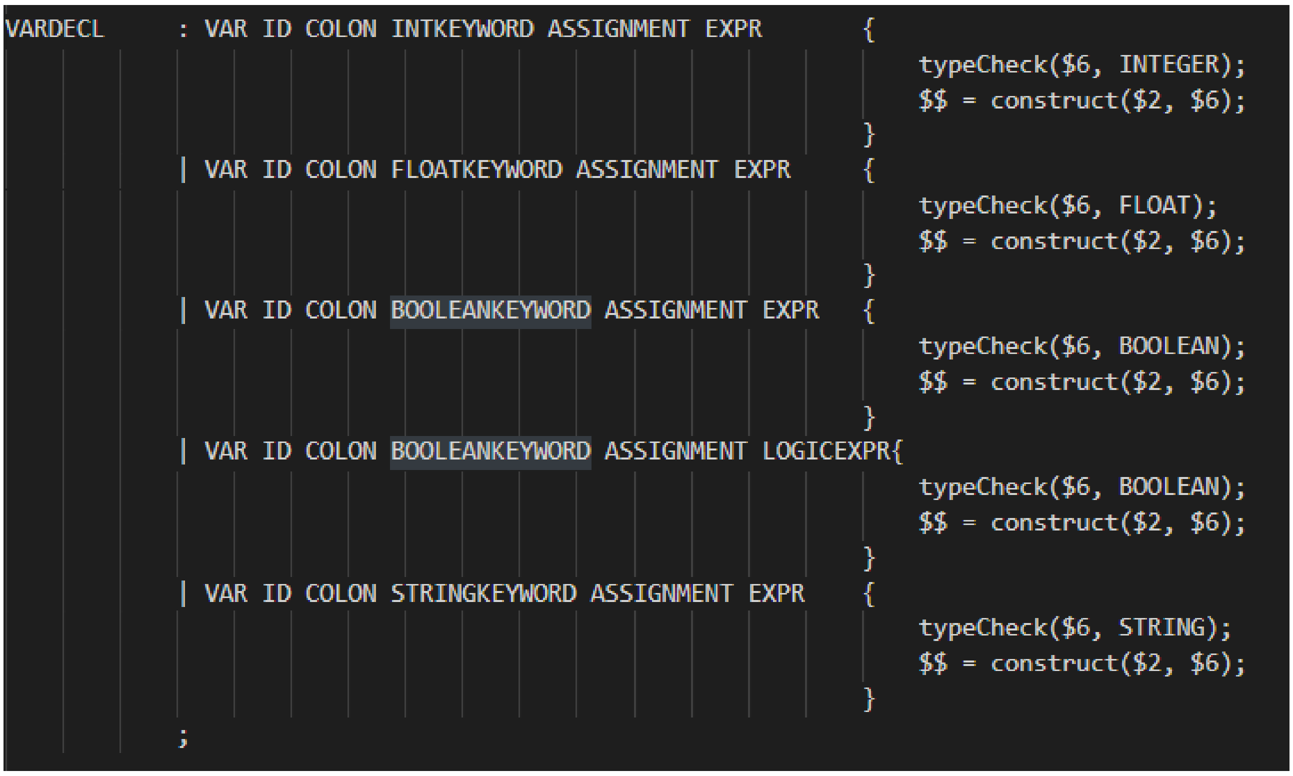
All of this must end with ENDOFSTMT (".").

The only cases in which you do not have to use ENDOFSTMT are in VARDECL inside function declarations, if statements and loop statements, in which you have to use SEMICOLON (“;”); this is to accomplish scoped symbol tables.

For the Grammar we have also an associated semantics:

- when we have EXITSTMT ENDOFSTMT (‘exit.’) the program will exit successfully;
- in case of VARDECL ENDOFSTMT (VARDECL is defined below and is of the following form: “note ID : type := value”) the program will insert the declared variable in the global symbol table;
- in case of VARDECL SEMICOLON (this construct must be used only inside function declarations and if statements), the program checks whether LINE is different from NULL (so if it is a not NULL NODE), and if the condition holds, then assigns to LINE Node->next the new variable declared, so that we can insert more than one variable in the scoped symbol table. Then NODE of VARDECL is assigned to the head of the rule LINE;
- for EXPR ENDOFSTMT and LOGICEXPR ENDOFSTMT we just print the result Node;
- when we encounter a LINE of the form IFSTMT ENDOFSTM the program will print all the symbol tables to show that nested tables are present; then it will change the table pointer to the parent one with ‘leaveScope()’;
- in case of a type keyword (INTKEYWORD, FLOATKEYWORD, BOOLEANKEYWORD, STRINGKEYWORD) FUNCTION EXPR ENDOFSTMT (for example ‘integer chord cat () play note g : string := “some-string”; stop.’) the program will check if the return type matches the declared return type, then, like above will print all the symbol tables and then leave the current scope to reach the parent one.

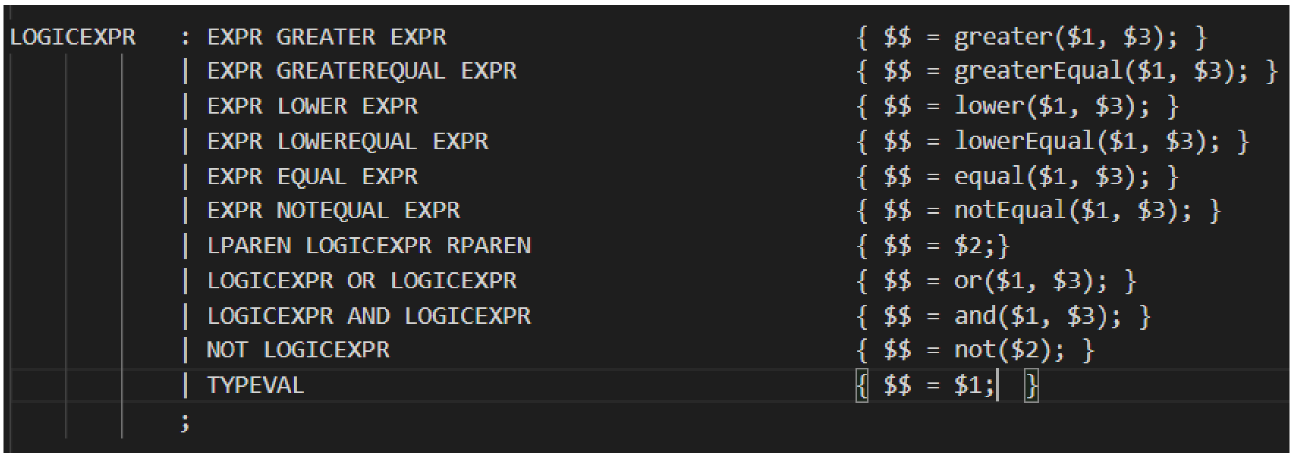
VARDECL



```
VARDECL     : VAR ID COLON INTKEYWORD ASSIGNMENT EXPR        {
                                                                typeCheck($6, INTEGER);
                                                                $$ = construct($2, $6);
                                                             }
            | VAR ID COLON FLOATKEYWORD ASSIGNMENT EXPR      {
                                                                typeCheck($6, FLOAT);
                                                                $$ = construct($2, $6);
                                                             }
            | VAR ID COLON BOOLEANKEYWORD ASSIGNMENT EXPR    {
                                                                typeCheck($6, BOOLEAN);
                                                                $$ = construct($2, $6);
                                                             }
            | VAR ID COLON BOOLEANKEYWORD ASSIGNMENT LOGICEXPR{
                                                                typeCheck($6, BOOLEAN);
                                                                $$ = construct($2, $6);
                                                             }
            | VAR ID COLON STRINGKEYWORD ASSIGNMENT EXPR     {
                                                                typeCheck($6, STRING);
                                                                $$ = construct($2, $6);
                                                             }
            ;
```

For all the possible types of our declared variables , we check whether the declared type is coherent with the type of the EXPR, if not we throw an error; then we modify the EXPR node, assigning the ID string in its field (that until now was NULL) and pass it to the head of the rule.

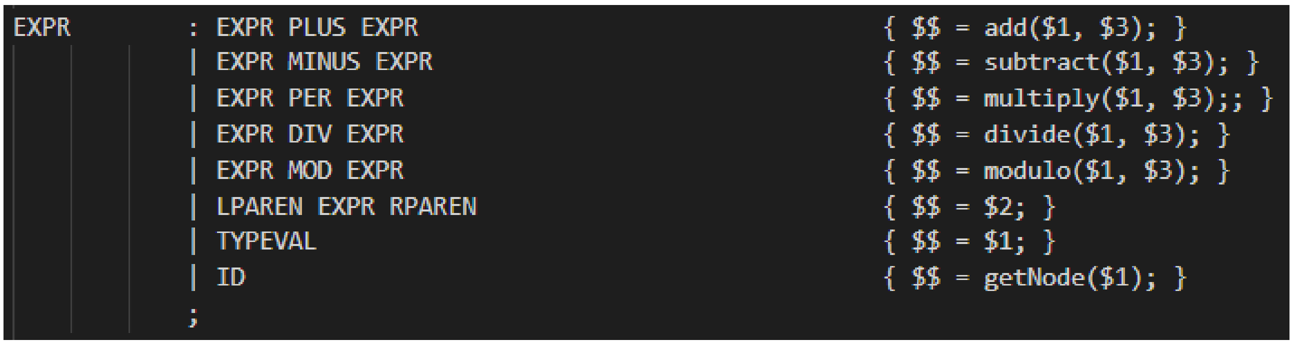
LOGICEXPR



```
LOGICEXPR   : EXPR GREATER EXPR                          { $$ = greater($1, $3); }
            | EXPR GREATEREQUAL EXPR                     { $$ = greaterEqual($1, $3); }
            | EXPR LOWER EXPR                            { $$ = lower($1, $3); }
            | EXPR LOWEREQUAL EXPR                       { $$ = lowerEqual($1, $3); }
            | EXPR EQUAL EXPR                            { $$ = equal($1, $3); }
            | EXPR NOTEQUAL EXPR                         { $$ = notEqual($1, $3); }
            | LPAREN LOGICEXPR RPAREN                    { $$ = $2;}
            | LOGICEXPR OR LOGICEXPR                     { $$ = or($1, $3); }
            | LOGICEXPR AND LOGICEXPR                    { $$ = and($1, $3); }
            | NOT LOGICEXPR                              { $$ = not($2); }
            | TYPEVAL                                    { $$ = $1;  }
            ;
```

The LOGICEXPR rule and its associated semantics give the possibility to deal with logic expressions starting from two arithmetic EXPR. LOGICEXPRS are all binary operators , except the NOT operator, which is unary. All the methods called in the semantics part are defined in the logicexpr.h file. The resulting nodes are passed to the head of the rule.

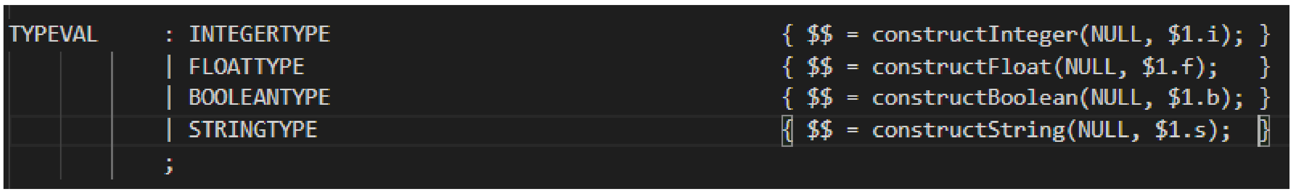
EXPR



```
EXPR        : EXPR PLUS EXPR                             { $$ = add($1, $3); }
            | EXPR MINUS EXPR                            { $$ = subtract($1, $3); }
            | EXPR PER EXPR                              { $$ = multiply($1, $3);; }
            | EXPR DIV EXPR                              { $$ = divide($1, $3); }
            | EXPR MOD EXPR                              { $$ = modulo($1, $3); }
            | LPAREN EXPR RPAREN                         { $$ = $2; }
            | TYPEVAL                                    { $$ = $1; }
            | ID                                         { $$ = getNode($1); }
            ;
```

EXPR rule and its correlated semantics have been created to deal with arithmetic expressions. TYPEVAL just passes the node created before to the head of the rule; ID, instead is retrieving the Node corresponding to the found ID and passes it to EXPR. The functions in the semantics for the other productions are defined in the expr.h file.

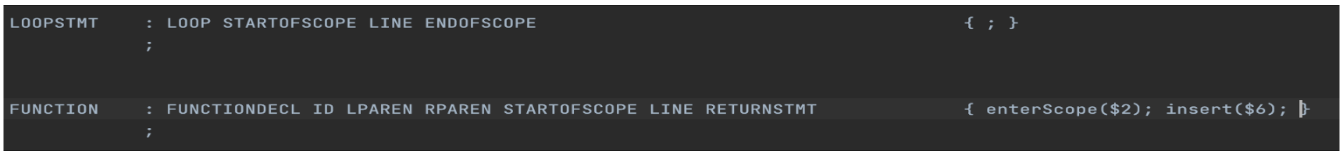
TYPEVAL



```
TYPEVAL     : INTEGERTYPE                         { $$ = constructInteger(NULL, $1.i); }
            | FLOATTYPE                           { $$ = constructFloat(NULL, $1.f);   }
            | BOOLEANTYPE                         { $$ = constructBoolean(NULL, $1.b); }
            | STRINGTYPE                          { $$ = constructString(NULL, $1.s);  }
            ;
```

Given the different type values recognized by the Lexer, the program will construct a new Node with NULL id (anonymous node) and the right value type and passes it to the head of the rule.

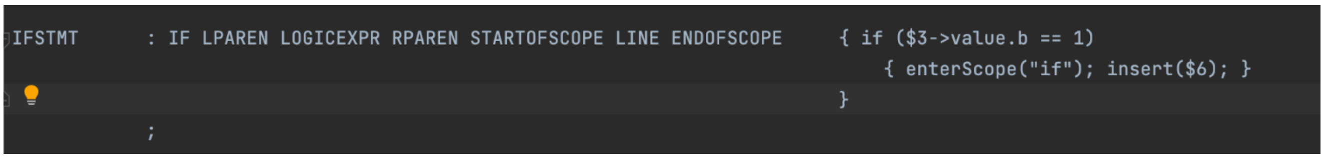
LOOPSTMT & FUNCTION



```
LOOPSTMT    : LOOP STARTOFSCOPE LINE ENDOFSCOPE                              { ; }
            ;

FUNCTION    : FUNCTIONDECL ID LPAREN RPAREN STARTOFSCOPE LINE RETURNSTMT     { enterScope($2); insert($6); }
            ;
```

When the program encounters FUNCTION, it creates a new scope, which is nested inside the parent scope and ensures that the parent scope does not have access to the new scope. The variable declared inside these code blocks are then inserted in the new scope. In case of LOOPSTATEMENT nothing is being executed, since it’s impossible execute loop at the parsing time.

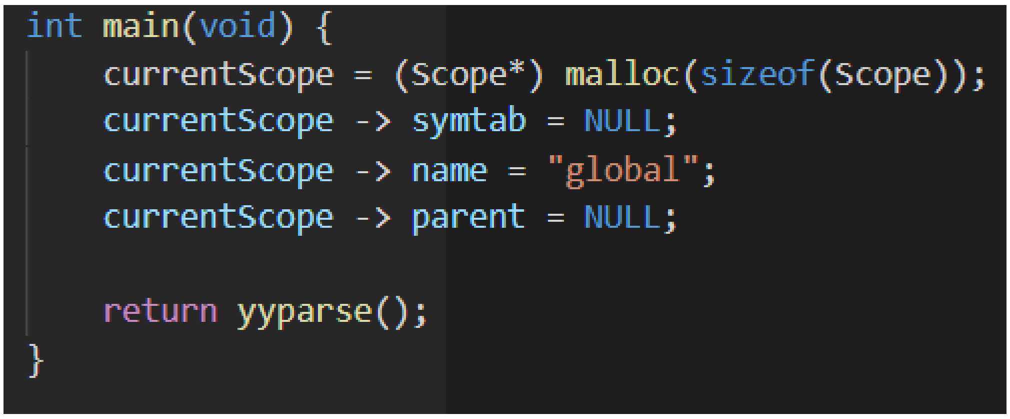
IFSTMT



```
IFSTMT      : IF LPAREN LOGICEXPR RPAREN STARTOFSCOPE LINE ENDOFSCOPE     { if ($3->value.b == 1)
                                                                              { enterScope("if"); insert($6); }
                                                                          }
            ;
```

In case of encountering if statement, the program finds a LOGICEXPR which if ($3 -> value.b == 1), then it will create a new scope for the if in which it will insert the variables declared in the LINE $6.

AUXILIARY ROUTINES



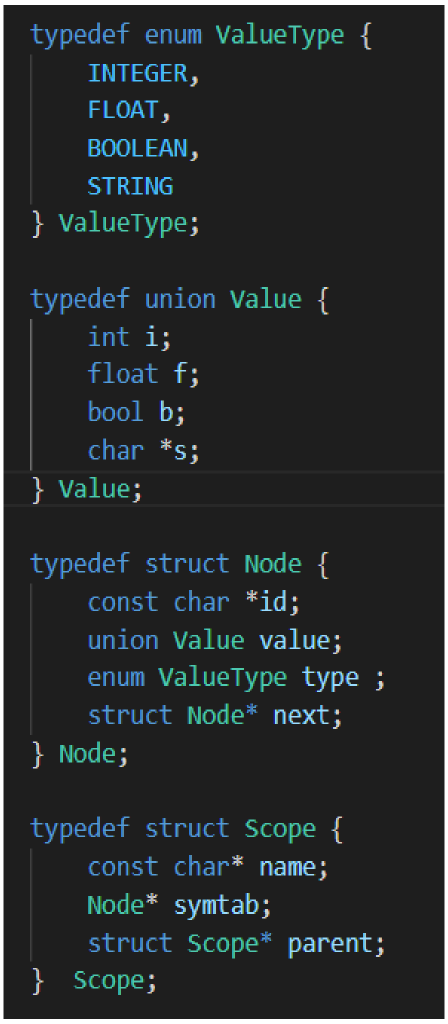
```
int main(void) {
    currentScope = (Scope*) malloc(sizeof(Scope));
    currentScope -> symtab = NULL;
    currentScope -> name = "global";
    currentScope -> parent = NULL;

    return yyparse();
}
```

In the last part we call the main function, allocate memory for the global scope, initialize its head node (symtab) to NULL, set its name to “global” and its field ‘parent’ to NULL, because it’s the root scope.

FUNCTIONS & STRUCTURES FOR SEMANTIC ANALYSIS

To be more organized we decided to have some header c files which help during the Semantic Analysis Phase of our compiler.

types.h

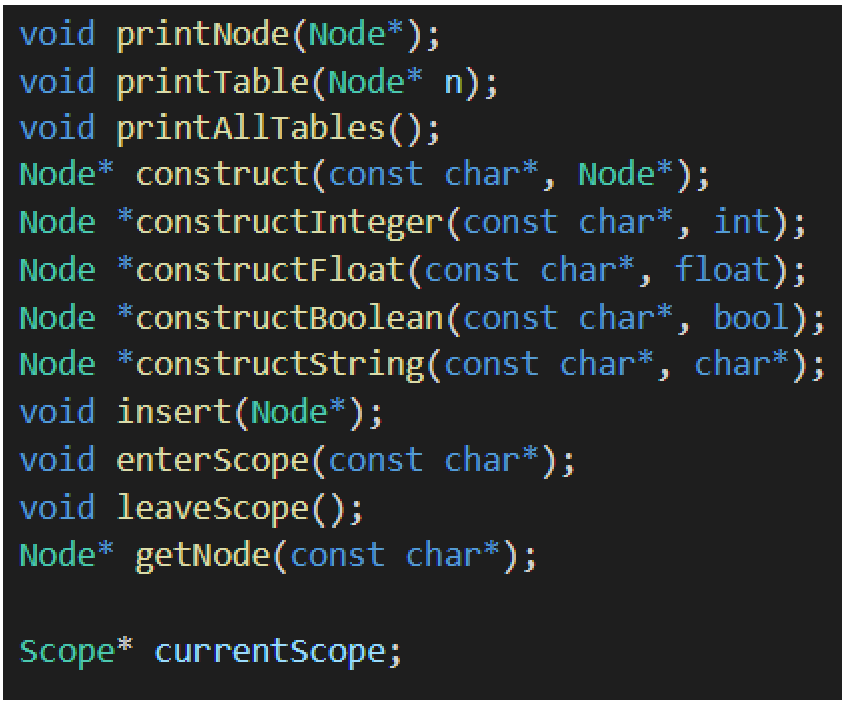


```
typedef enum ValueType {
    INTEGER,
    FLOAT,
    BOOLEAN,
    STRING
} ValueType;

typedef union Value {
    int i;
    float f;
    bool b;
    char *s;
} Value;

typedef struct Node {
    const char *id;
    union Value value;
    enum ValueType type ;
    struct Node* next;
} Node;

typedef struct Scope {
    const char* name;
    Node* symtab;
    struct Scope* parent;
}  Scope;
```

- **ValueType** : an enum that represents the possible types which a Node can possess;
- **Value**: an union which can take different types of value (int i, float f, bool b, char *s);
- **Node**: a structure to construct a Linked List to create a Symbol Table; it contains an *id*, a *value* (which is of type *Value*), a *type* for the value (which recall the enum *ValueType*), and a recursive call to *next* which is always of type *Node* and represent the next Node in the Linked List.
- **Scope**: a structure to keep track of the different scopes (Symbol Tables); it contains a *name*, a pointer to the first Node of the Linked List to which it is connected, and a pointer to the *parent* scope (for

example, if, at the beginning of the program we declare a function, a new scope is created, and it points to the parent *Scope*, which would be the global scope).

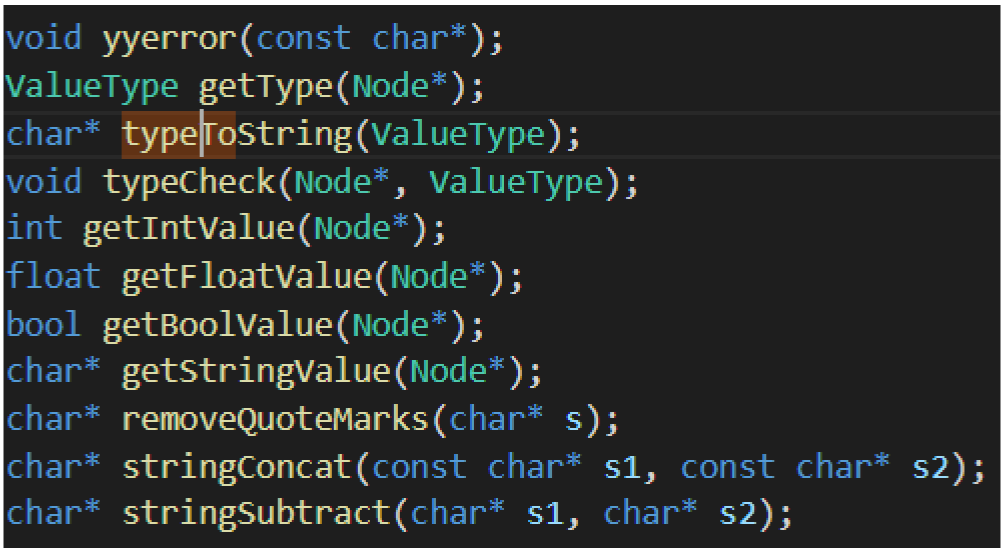
symtab.h



```
void printNode(Node*);
void printTable(Node* n);
void printAllTables();
Node* construct(const char*, Node*);
Node *constructInteger(const char*, int);
Node *constructFloat(const char*, float);
Node *constructBoolean(const char*, bool);
Node *constructString(const char*, char*);
void insert(Node*);
void enterScope(const char*);
void leaveScope();
Node* getNode(const char*);

Scope* currentScope;
```

- The first three functions are used to print the nodes constructed and the variables stored in the Symbol Tables;
- **construct(const char*, Node*)** assigns an *id* to the Node passed as second parameter;
- **construct[Integer|Float|Boolean|String](const char*, type)** allocates space for a *Node* and sets a specific type to that *Node;*
- **insert(Node*)** insert a particular Node to the current Symbol Table;
- **enterScope(const char*)** creates a new scope assigning the name passed as parameter;
- **leaveScope()** changes the pointer of the current scope to its parent;
- **getNode(const char*)** returns a *Node* given its name;
- **Scope* currentScope** is a global pointer to the scope you are inside currently.

helpers.h



```
void yyerror(const char*);
ValueType getType(Node*);
char* typeToString(ValueType);
void typeCheck(Node*, ValueType);
int getIntValue(Node*);
float getFloatValue(Node*);
bool getBoolValue(Node*);
char* getStringValue(Node*);
char* removeQuoteMarks(char* s);
char* stringConcat(const char* s1, const char* s2);
char* stringSubtract(char* s1, char* s2);
```

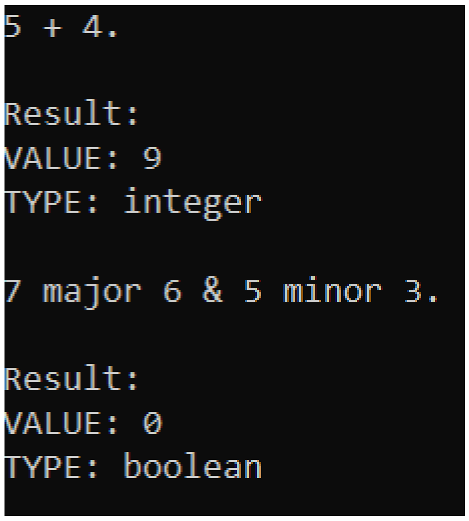
- ❖ **yyerror(const char*)** is used to print errors;
- ❖ **getType(Node*)** is used to retrieve the *type* of a *Node;*
- ❖ **typeToString(ValueType)** converts the enum *ValueType* values to type strings;
- ❖ **typeCheck(Node*, ValueType)** checks if the declared type matches the type of the *Node*.
- ❖ **get(Int/Float/Bool/String)Value(Node*)** retrieves value from *Node.*
- ❖ **removeQuoteMarks(char* s)** removes quote marks from string variable, to allow more comfortable string concatenation and subtraction;
- ❖ **stringConcat(const char* s1, const char* s2)** concatenates 2 strings;
- ❖ **stringSubtract(char* s1, char* s2)** removes all occurrences of s2 characters from s1.

expr.h & logicexpr.h

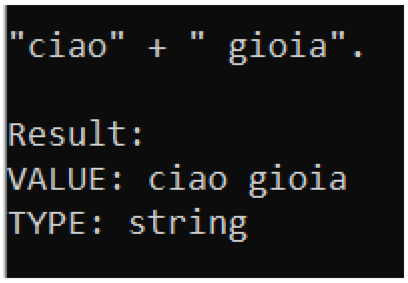
This two files contain all the functions necessary to compute expressions and logical expressions.

EXAMPLES

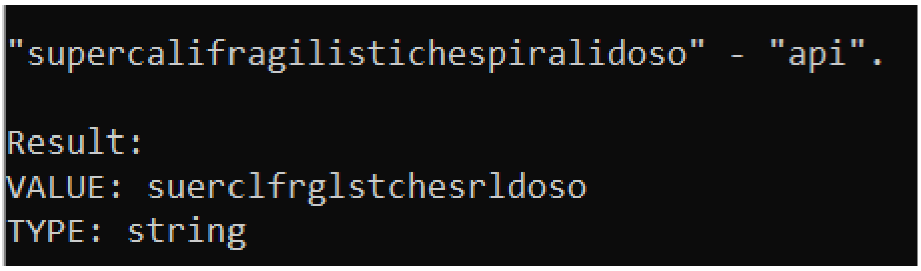
Expressions and Logical Expressions without variables



```
5 + 4.

Result:
VALUE: 9
TYPE: integer

7 major 6 & 5 minor 3.

Result:
VALUE: 0
TYPE: boolean
```

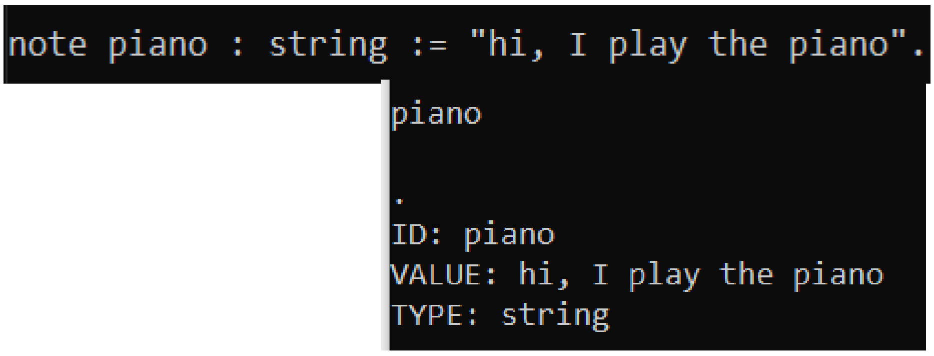


```
"ciao" + " gioia".

Result:
VALUE: ciao gioia
TYPE: string
```

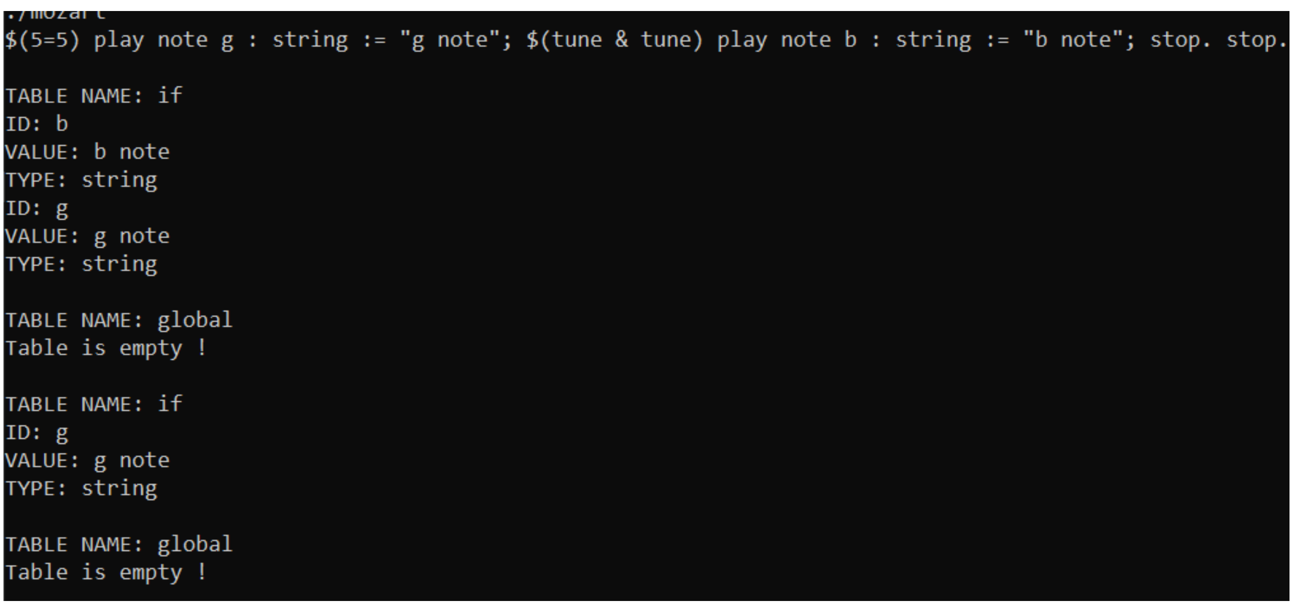


```
"supercalifragilistichespiralidoso" - "api".

Result:
VALUE: suerclfrglstchesrldoso
TYPE: string
```

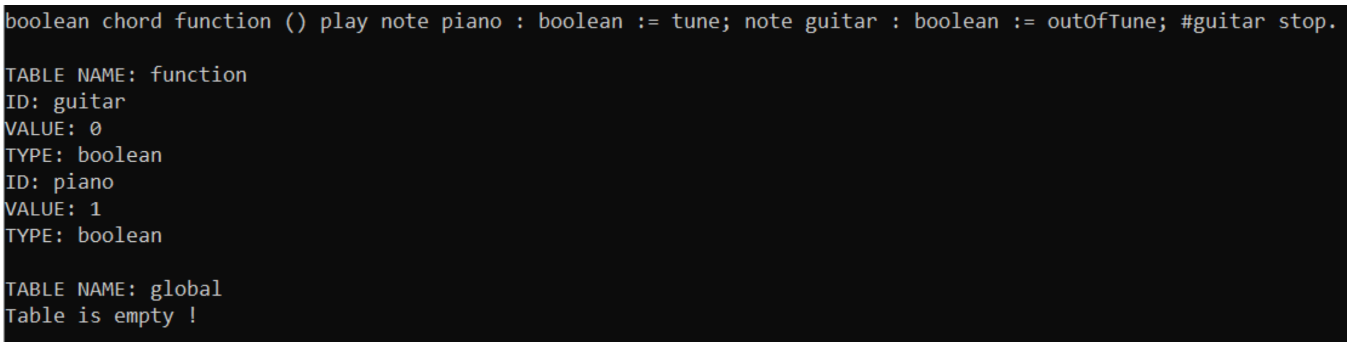
Variable Declaration and if statement



```
note piano : string := "hi, I play the piano".
piano

.
ID: piano
VALUE: hi, I play the piano
TYPE: string
```

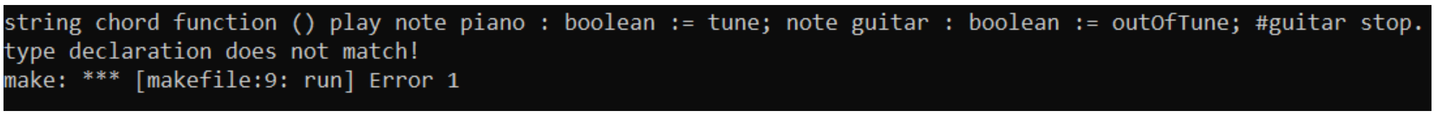


```
./mozart
$(5=5) play note g : string := "g note"; $(tune & tune) play note b : string := "b note"; stop. stop.

TABLE NAME: if
ID: b
VALUE: b note
TYPE: string
ID: g
VALUE: g note
TYPE: string

TABLE NAME: global
Table is empty !

TABLE NAME: if
ID: g
VALUE: g note
TYPE: string

TABLE NAME: global
Table is empty !
```

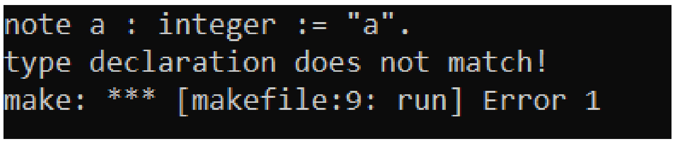


```
boolean chord function () play note piano : boolean := tune; note guitar : boolean := outOfTune; #guitar stop.

TABLE NAME: function
ID: guitar
VALUE: 0
TYPE: boolean
ID: piano
VALUE: 1
TYPE: boolean

TABLE NAME: global
Table is empty !
```

Type checking errors examples



```
string chord function () play note piano : boolean := tune; note guitar : boolean := outOfTune; #guitar stop.
type declaration does not match!
make: *** [makefile:9: run] Error 1
```



```
note a : integer := "a".
type declaration does not match!
make: *** [makefile:9: run] Error 1
```