Compilers

Report Phase III

Ahmed Tarek ElGamal 9

Mohamed Ahmed AbdElTawab Heeba 52

Mohamed Samir Shabaan 55

Data Structure:

1. map<string, pair<int, int > > sym : to save the identifier as a key and make pair from it’s type(int , float) and it’s index(entry number) as a value.
2. stack<int > labeling : to use it in if condition and while (gen(go to label)).
3. stack<int > expectedDataType : to use it in (assignment and conditions).

Phase III Workflow:
In this phase we worked along with both tools bison and flex.
Flex enabled us to write some lexical functions to match the stream of tokens arrived to our parser and report the appearance of these lexical analyzed tokens to our parser and provide it with all the needed info about the matched token this is done by the yacc and lex files compiled together in our final stage of that phase of the project.
Flex structure:

```
%{
c declarations phase
%}
flex declarations phase
%%
matching phase + reporting to bison phase
%%
c definitions phase
```

Bison structure:

```
%{
c declarations phase
%}
bison declarations phase
%%
rule compiling phase + semantic actions phase
%%
c definitions phase
```

So what happens is in the bison declaration phase we define some %tokens which represent terminals for us that the lex will use as return value in case of matched these terminals in the token stream once this report take place the parser (yacc) takes action according to the rule compiling phase which in it there exist some rules that direct the yacc to which place it should go and as a result it can use the corresponding semantic action for it, about the c phases it' s compiled of some functions that help us to do the work in more systematic way, finally we can go around that simple

communication between the two programs using the union defined in the bison declaration phase that could contain another methods of communication between the two programs such as using structures, integer, pointer and float.
This is a brief introduction that showed what we should do in this phase we should make the most use of these utilities to generate a byte code generator for JAVA.
N ow we will discuss each phase of the code in detail and we will start with the lex part.
This is the declaration phase that we included our directives and declared some functions to be implemented in the declaration phase.

```
%{
#include "y.tab.h"
#include <stdlib.h>
#include <cstring>
void yyerror(char *);
%}
```

this is the part of declaring the matching regular expressions and assign them to different identifiers.

```
letter
[a-zA-Z]
number
[0-9]+(\L|[eE][-+]?[0-9]+)?
float
[0-9]*\.[0-9]+(\L|[eE][-+]?[0-9]+)?
assign
"="
plus
"+"
minus
"-"
multiply
"*"
divide
"/"
mod
"%"
semicolon
";"
open_round "("
close_round ")"
open_curly
"{"
close_curly "}"
greater_than ">"
less_than "<"
greater_equal ">="
less_equal "<="
equal_equal "=="
not_equal "!="
%%
```

use the previously defined identifiers along with defined %tokens in the bison declaration part to feed
the bison with all the needed information it needs about the matched operation just happened in the lex
part.

```
[ \t]+
";" { return SEMICOLON; }
"(" { return OPEN_ROUND; }
")" { return CLOSE_ROUND; }
"{" { return OPEN_CURLY; }
"}" { return CLOSE_CURLY; }
("else") { return ELSE; }
```

```
("if") { return IF; }
("while")
{ return WHILE; }
("int") { yylval.type_t = INTEGER; return INTEGER; }
("float") { yylval.type_t = FLOAT; return FLOAT; }
("boolean") { return BOOLEAN; }
{letter}({letter}|{number})* { char* ret = (char *)malloc(strlen(yytext)+1);
memcpy(ret,yytext,strlen(yytext)); ret[strlen(yytext)] = '\0'; yylval.str_t = ret; return
IDENTIFIER; }
{assign} { return ASSIGN; }
{float} { yylval.dou_t = atof(yytext); return FLOAT_LITERAL; }
{number}+ { yylval.int_t = atoi(yytext); return INTEGER_LITERAL; }
{plus} { return PLUS; }
{minus} { return MINUS; }
{divide} { return DIVIDE; }
{multiply} { return MULTIPLY; }
{mod} { return MOD; }
{greater_than} { return GREATER_THAN; }
{less_than} { return LESS_THAN; }
{greater_equal} { return GREATER_EQUAL; }
{less_equal} { return LESS_EQUAL; }
{equal_equal} { return EQUAL_EQUAL; }
{not_equal} { return NOT_EQUAL; }
\n        {++lineCounter;}
"System.out.println"   {return SYSTEM_OUT;}
%%
```

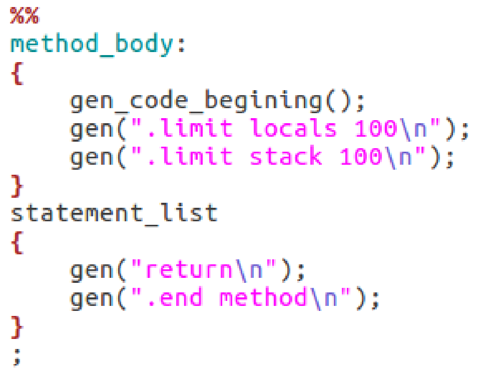
this part for declaring the functions that already was defined in the declaration part of the file.l

```
int yywrap(void) {
return 1;
}
```

So as most of the code is self-explanatory now we are done with the lex
file part.
N ow we will explain the components of our yacc file and the intent of
each part of the code.

Rules and semantic:

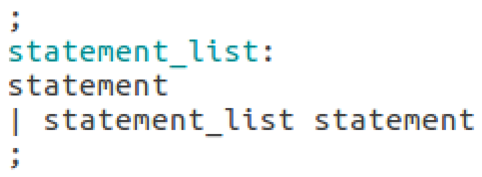
<u>method_body:</u>



```
%%
method_body:
{
    gen_code_begining();
    gen(".limit locals 100\n");
    gen(".limit stack 100\n");
}
statement_list
{
    gen("return\n");
    gen(".end method\n");
}
;
```

Here we emitted the start of the code of the static class definition alongside with some basic operations such as limiting the stack of the compiler.

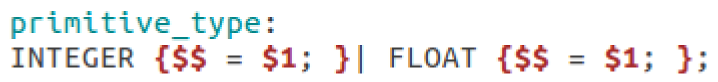
<u>statement_list:</u>



```
;
statement_list:
statement
| statement_list statement
;
```

As we don’t use any return type for the statement list so we didn’t need any semantic action for this specific nonterminal.

<u>primitive_type:</u>

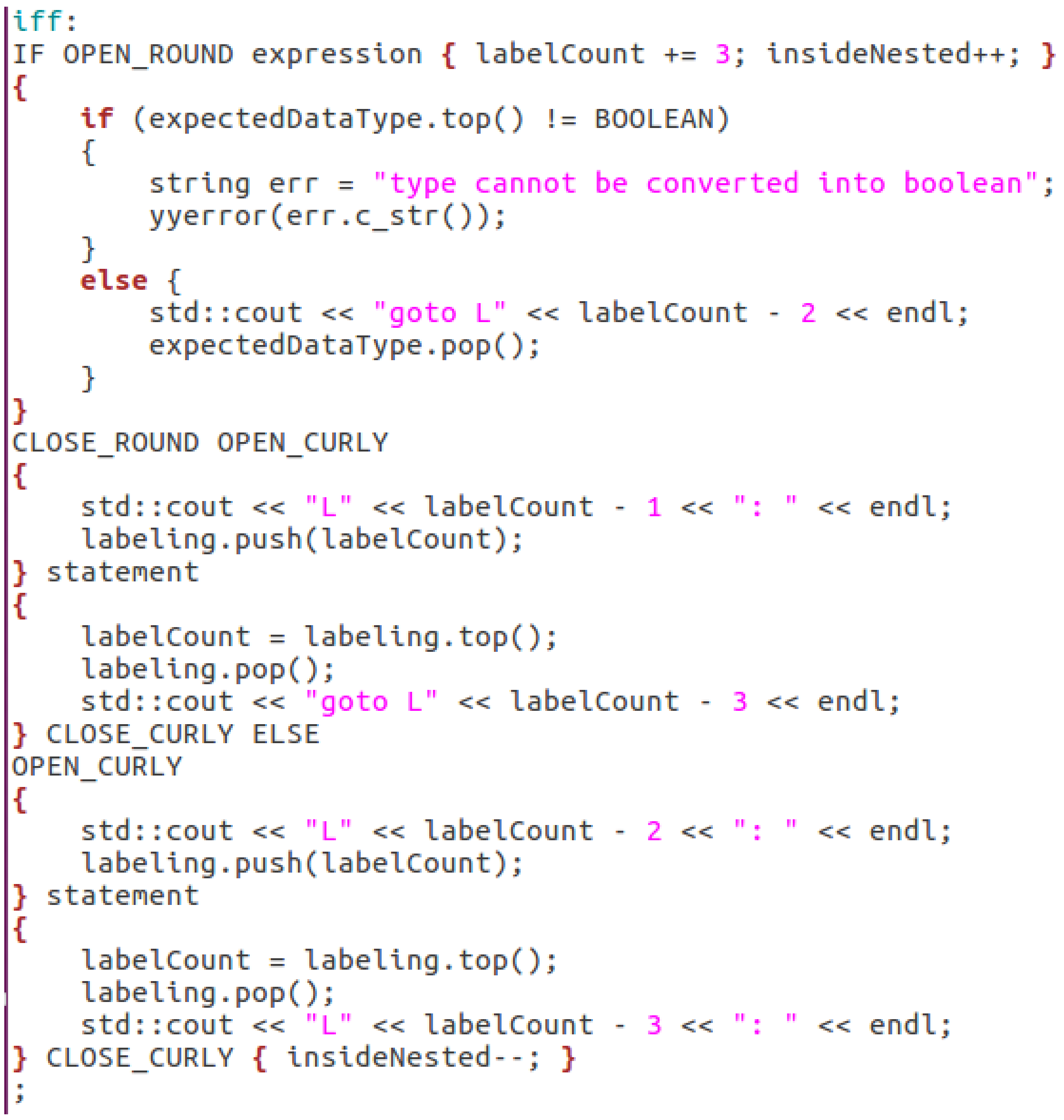


```
primitive_type:
INTEGER {$$ = $1; }| FLOAT {$$ = $1; };
```

We defined %type of primitive type as type_t which hold the type of the following identifier whether it was integer or float.

IF:

Here we used the stack data structure to store the label count so that in case of nested while or if statements can be handled easily alongside with the insideNested global variable which allow us to define the depth of our nested statements.

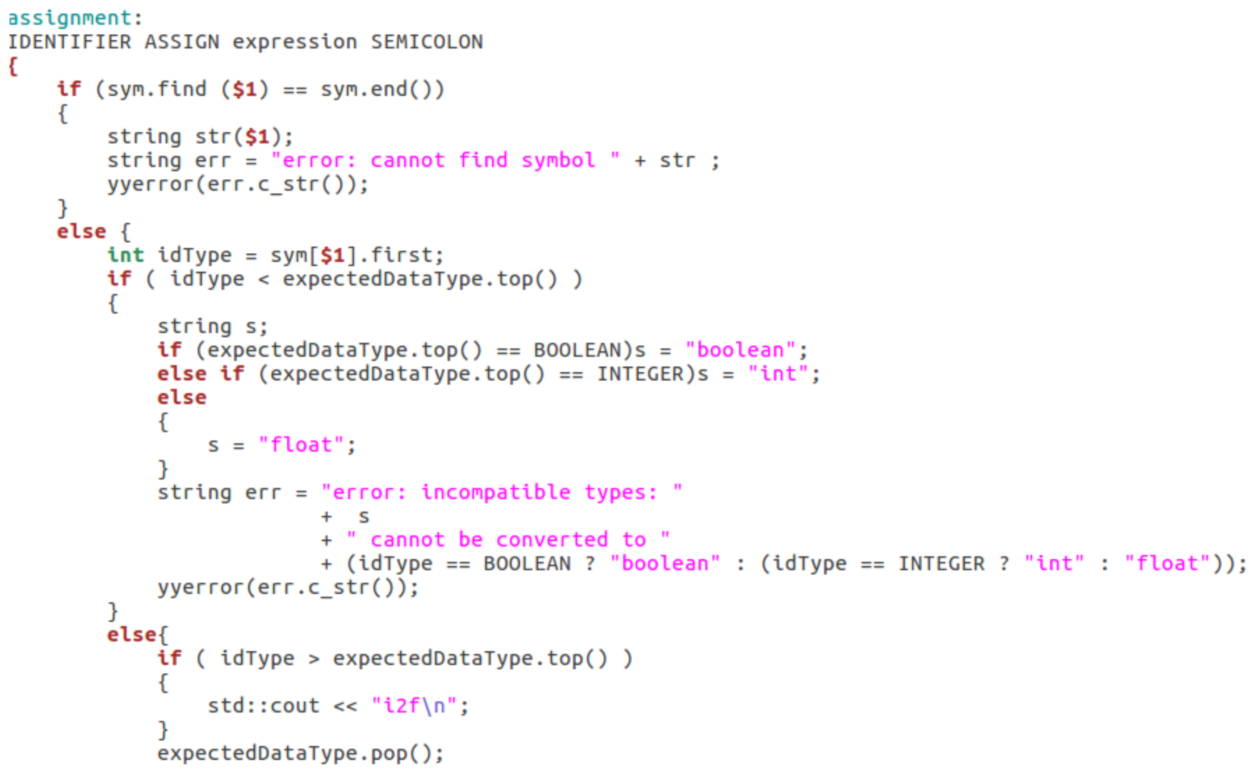


```
iff:
IF OPEN_ROUND expression { labelCount += 3; insideNested++; }
{
    if (expectedDataType.top() != BOOLEAN)
    {
        string err = "type cannot be converted into boolean";
        yyerror(err.c_str());
    }
    else {
        std::cout << "goto L" << labelCount - 2 << endl;
        expectedDataType.pop();
    }
}
CLOSE_ROUND OPEN_CURLY
{
    std::cout << "L" << labelCount - 1 << ": " << endl;
    labeling.push(labelCount);
} statement
{
    labelCount = labeling.top();
    labeling.pop();
    std::cout << "goto L" << labelCount - 3 << endl;
} CLOSE_CURLY ELSE
OPEN_CURLY
{
    std::cout << "L" << labelCount - 2 << ": " << endl;
    labeling.push(labelCount);
} statement
{
    labelCount = labeling.top();
    labeling.pop();
    std::cout << "L" << labelCount - 3 << ": " << endl;
} CLOSE_CURLY { insideNested--; }
;
```

While :

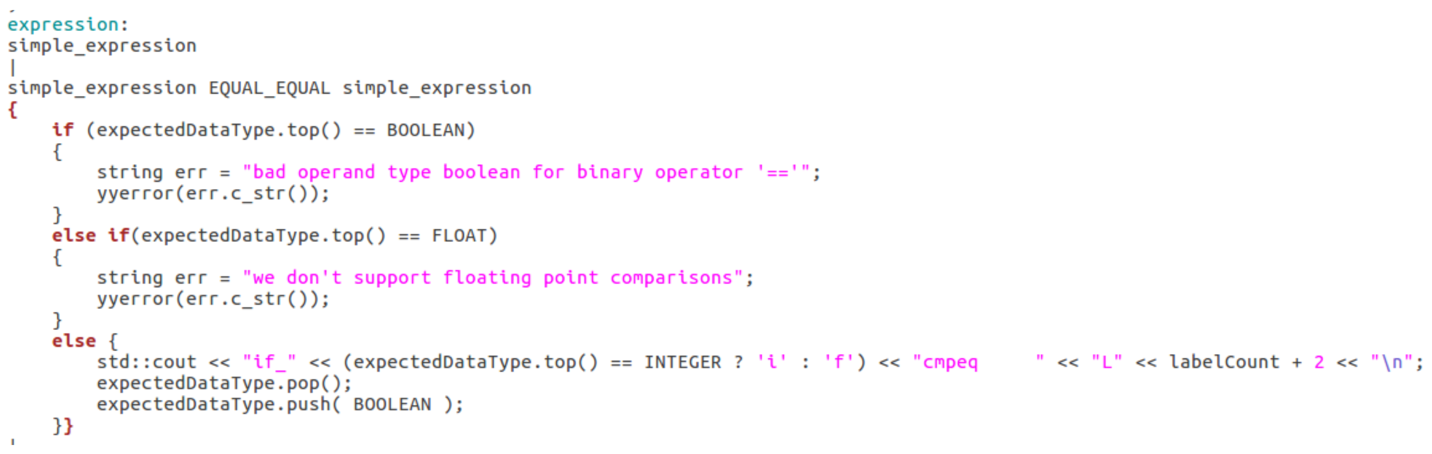
Same Logic of IF but going to condition label to check again.

Assignment:
In the assignment operation we needed just to check for the boolean and float assignment to integer or the float assigned to float which are forbidden in our code.



```
assignment:
IDENTIFIER ASSIGN expression SEMICOLON
{
    if (sym.find ($1) == sym.end())
    {
        string str($1);
        string err = "error: cannot find symbol " + str ;
        yyerror(err.c_str());
    }
    else {
        int idType = sym[$1].first;
        if ( idType < expectedDataType.top() )
        {
            string s;
            if (expectedDataType.top() == BOOLEAN)s = "boolean";
            else if (expectedDataType.top() == INTEGER)s = "int";
            else
            {
                s = "float";
            }
            string err = "error: incompatible types: "
                         +  s
                         + " cannot be converted to "
                         + (idType == BOOLEAN ? "boolean" : (idType == INTEGER ? "int" : "float"));
            yyerror(err.c_str());
        }
        else{
            if ( idType > expectedDataType.top() )
            {
                std::cout << "i2f\n";
            }
            expectedDataType.pop();
```

Expression:
For the expression it’s sufficient to understand one expression and apply the same logic on the other relative operations we fix the jump to label for the cases of if and while so that we can always jump to the same point in case of the condition being true.



```
expression:
simple_expression
|
simple_expression EQUAL_EQUAL simple_expression
{
    if (expectedDataType.top() == BOOLEAN)
    {
        string err = "bad operand type boolean for binary operator '=='";
        yyerror(err.c_str());
    }
    else if(expectedDataType.top() == FLOAT)
    {
        string err = "we don't support floating point comparisons";
        yyerror(err.c_str());
    }
    else {
        std::cout << "if_" << (expectedDataType.top() == INTEGER ? 'i' : 'f') << "cmpeq      " << "L" << labelCount + 2 << "\n";
        expectedDataType.pop();
        expectedDataType.push( BOOLEAN );
    }}
```

simple expression:

For the simple expression we defined the %type for it to be a type_t so can store the data type generated by the operation which was hold in the production then we try to see if the data types that generates the data type of the simple expression is compatible or not.

```
simple_expression:
term
|
PLUS term
{
    if (expectedDataType.top() == BOOLEAN)
    {
        string err = "bad operand type boolean for unary operator '+'\n";
        yyerror(err.c_str());
    }
}
|
MINUS term
{
    if (expectedDataType.top() == BOOLEAN)
    {
        string err = "bad operand type boolean for unary operator '-'\n";
        yyerror(err.c_str());
    }
    else {
        char typeChar = (expectedDataType.top() == INTEGER ? 'i' : 'f');
        std::cout << typeChar << "neg\n";
    }       }
|
simple_expression PLUS
{
    if (expectedDataType.top() == BOOLEAN)
    {
        string err = "bad operand type boolean for binary operator '+'" ;
        yyerror(err.c_str());
    }
}
term
{
    if (expectedDataType.top() == BOOLEAN)
```

Factor:

Term was given also the data type as its %type so that we can check for its type compatibility.

```
factor:
IDENTIFIER
{
    //assert ($2.type != BOOLEAN && "Error in factor logic!!");

    if (sym.find ($1) == sym.end())
    {
        string str($1);
        string err = "error: cannot find symbol " + str;
        yyerror(err.c_str());
    }
    else {
        int idType = sym[$1].first;
        int type = max (idType, expectedDataType.size() ? ( expectedDataType.top() == BOOLEAN ? INTEGER : expectedDataType.top() ) : INTEGER );

        if (expectedDataType.size() && ( expectedDataType.top() == BOOLEAN ? FLOAT : expectedDataType.top() ) < type)
        {
            std::cout << "i2f\n";
        }

        pushIndex (type, sym[$1].second);

        if (idType < type)
        {
            std::cout << "i2f\n";
        }
        if(expectedDataType.size() && expectedDataType.top() != BOOLEAN) expectedDataType.pop();
        expectedDataType.push( type );
    }
}
```

Assumptions:

1) No Boolean casting to any type is allowed.
2) The relative operations are defined only for the integer operands.
3) No casting from float to integer is allowed (int x; x = 5.0;) .