Project 2 – Type checker for C using the UNIX YACC Tool

Due date: May 13, 2009

Description: In this second part of the compiler project, you will be asked to develop a type checker for the programming language C. You will be given a lexer (a Lex file) and parser (a YACC file). These two files you shall change so that, besides lexical analysis and parsing, type checking is performed during parsing.

The project shall be developed individually. You are encouraged to discuss with other students, ask teachers, search for ideas on the Internet, etc. But, do not copy code. The solutions will be tested in the Linux-i386 environment. The evaluations of the solutions are based on these tests. The compiler can be developed in many environments. But, when the solutions are delivered, it must be possible to generate executables from their source codes, and compile the results, etc., in Linux-i386. See the instruction on how to turn-in your projects.

The type checker shall, given a program, check that the types of the expressions are the expected ones. If the type of an expression is not the expected one then the type checker shall, depending on how serious the error is, either output a warning or an error message (this is specified below).

In order to keep the project a reasonable size, you may make the following assumptions: In the programs to be type checked, there are three types: `void`, `int`, and `float`. Types of expressions originate from constants (like `5`, which has the type `int`, and `6.31` which has the type `float`), formal parameters (like `x` in `void f(int x){…}`), and variable declarations in the beginning of blocks (like `{int x; float y;…}`). Types of simpler expressions are then propagated to more complex expressions (if the type of `x` and `y` is `int` then the type of `x + y` is also `int`). Variable declarations only occur in the top-most block, i.e., there are no global variable declarations and there are no declarations in nested blocks. You may also assume that a program consist of a single function definition.

The expected type of an expression can be determined by (1) operator and the other sub-expressions (for example, in `x + y`, `x` and `y` are expected to have the same type), (2) the type of the left-hand side in an assignment (in `x=y`, `y` is expected to have the same type as `x`), (3) the control statement where the expression occurs (the boolean guards in control statements is expected to have the type `int`), and (4) return types of functions (in `int f(…){…return x;}`, `x` is expected to have type `int`).

The type checker should be able of handling sequences of statements, block-, `if`-, `while`-, assignment-, and `return`-statements; expressions containing constants, variables, `(`, `)` , `~`, `-`, `+`, `*`, `/`, `<`, `>`, `++`, `--`, `==`,`!=`, `<=`, `>=`, `&&` , `||`, and the type casts `(int)` and `(float)`. The boolean operators expect the operands to be of type `int`; the numerical operators expect the operands to be of the same type, `int` or `float`, except that `++` and `--` only expect `int`; the relations are relations between elements of same type, `int` or `float`. The type casts `(int)` and `(float)` change type of an expression from `float` to `int`, and from `int` to `float`. The return type of a function can be `void`, which means that nothing should be returned. Variables can also be of type `void`, but they cannot be used for anything.

A type error occurs when the type of an expression is not the same as the expected type. A particular mild kind of type error occurs when a `float` is expected but an `int` is given. In this case a warning shall be printed. For all other kinds of type errors, an error message shall be printed. When a `float` is expected but an `int` is given, the type checker shall do an automatic type cast from `int` to `float`. When an error for which a warning is to be printed, after the warning is printed, the type checker continues, but for each expression, only the "innermost" and first detected warning shall be printed. For the more serious type errors, the type checker halts after the error message is printed.

What to do?: You will be given a lexer and a parser grammar without any rules. You need to: 1) understand the grammar and 2) fill in the rules (C code) to perform the type checking with the help of an auxiliary symbol table.

Output from the type checker: The error messages shall be printed to `stdout`, and they shall contain information on what kind of error it is and at what line it occurs. We now present several exmple of the expected output for your type type checker. For each example we indicate the program on the left and the corresponding output on the right:

Example 1.

```
float f(int a){
  int x;
  float y;
  x=12;
  y=23;
  x=3.78;
  return 0;
}
```

```
warning: type cast int to float, line 5
type error: int expression expected, line 6
```

Example 2.

```
float f(int a){
  int x;
  x=x + a * (int)3.4;
  while(3.5){
    x=3.78;
  };
}
```

```
type error: int expression expected, line 4
```

Example 3.

```
void main(int a){
  int x;
  float z;
  void y;
  z= x + 1 + x;
  if(y==y){
    x=3.78;
  };
}
```

```
warning: type cast int to float, line 5
type error: numerical expression expected, line 6
```

Example 4.

```
void main(int a){
    int x;                          warning: type cast int to float, line 4
    void y;                         type error: int expression expected, line 4
    x=a * 3.4;
    if(y==y){
      x=3.78;
    };
  }
```

Example 5.

```
int f(){
    return 1.1;                     type error: int expression expected, line 2
  }
```

Clues: The easiest way to implement this type checker is probably to use a symbol table and an attribute grammar, and use a table where you, given an operation and types of operands, can lookup type of the resulting expressions.

How to Generate a Parser Executable: Generate the lexer and parser using flex and yacc, respectively (you need to include these commands on your makefile):

```
flex lex2.l
yacc -d -v yacc2.y
```

The results consist of the files lex.yy.c, y.tab.c and y.tab.h. Compile and link:

```
gcc -c lex.yy.c
gcc –c y.tab.c
gcc lex.yy.o y.tab.o -ll
```

Some test cases with the correct results can be found at the class website. Test and make sure that everything works on the Linux machine (raquel.tagus.ist.utl.pt). A lex file, a yacc file and some test cases with the correct results can be found at the class website. Test and make sure that everything works. Your project will be graded automatically using `diff`:

```
a.out < test.in > temp
diff temp test.out
```

Turn-in Instructions: As with the first programming assignment you will turn in your project on Fénix. Please make sure you have a group and can submit your files. If you are student number `XYZ`, you need to create a file named `XYZ.proj2.tar.zip` created using the `zip` and `tar` utilities, and make sure that the commands

```
unzip XYZ.proj2.tar.zip
tar -xvf XYZ.proj2.tar
cd XYZ.proj2
make
```

results in the creation of a folder `XYZ.proj2` with executable file `a.out` in it. Please make sure you do understand this. Also, inside this file there cannot be any input and/or output files and your makefile will have to create the lexer and parser files using the flex/lex and yacc/bison commands.