

Project 2 – Parser and Type checker using the UNIX YACC Tool

Due date: March 12, 2010 at midnight PST

Description: In this second part of the compiler project, you will be asked to develop a parser and a type checker for the mini-C programming language described in the first part of the project. You will be given a scanner (a Lex file) and skeleton parser (a YACC file). You will need to make modest modifications to the parser specification provided so that type checking is performed alongside with and 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 a UNIX environment. 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 UNIX. See the instruction on how to turn-in your projects. The evaluations of the solutions are based on a set of electronic automated tests.

The language mini-C was briefly described in the first part (Lexical Analyzer) of the project. Here some details are given by means of a skeleton of one its possible context free grammars (CFG).

Parsing. The parser shall be developed using the parser generator tool YACC.

In this second part of the project, there are mainly two things to do: First, based on the skeleton CFG (described here), develop a complete grammar for mini-C in YACC. Second, make sure that the parser explicitly creates a parse tree when a program is being parsed and performs type checking. The type checking also implies you need to construct a symbol table in the process of parsing the input file. Recall you will be using the solution to this project in the last programming assignment.

The grammar for mini-C described in this document is ambiguous. Moreover it is not complete: productions for some of the non-terminals such as `expression` and `unary_expression` are missing. It is your task to develop a non-ambiguous grammar in YACC, i.e. there must not be any warnings about conflicts when you generate the parser using YACC. Conflicts can be fixed by adding new non-terminals or by assigning precedence. But, it is important that you understand why there is a conflict before you try to fix it. By running YACC with the `-v` (verbose) option, YACC also generates a file `y.output`. This file contains a description of the state machine that YACC generates, and it also describes the conflicts.

Parse Tree Construction: To construct a parse tree, you shall use the semantic actions associated to the productions of the grammar and use the node functions as provided by the library of simple node functions provided to you at the class web page. This library has a variety of AST manipulation functions you can use. The example below highlights the simple use of the functions in this library to construct a parse tree.

```
E -> E + T      { $$ = node($2, $1, $3); }
E -> T          { $$ = $1 }
```

The identifiers $\$$, $\$1$, $\$2$, ... are *synthesized attributes* and their values are computed bottom up. The identifier $\$$ refers to the left hand side of the production and $\$1$, $\$2$, ... refer to the right hand side symbols ($\$2$ refers to the character '+'). So when the first rule above is reduced, a new node in the parse tree is created. The second rule above has only symbol on the right hand side, and in many cases we do not need to create nodes for those productions.

The parse tree will be used in the third part of the project so it is important that you build it although you could perform type checking without it.

Type Checking: 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 with some of the grammar productions but without any rules. You need to: 1) understand and complete the grammar and 2) fill in the rules (C code) to perform the type checking with the help of an auxiliary symbol table.

Tokens

The tokens (IDENTIFIER, CONSTANT, SIZEOF, PTR_OP, EQ_OP, AND_OP, OR_OP, TYPEDEF, INT, VOID, STRUCT, IF, ELSE, WHILE and RETURN) that the parser expects to receive from the lexer is defined in the file yacc2.y that the parser is generated from. By running YACC with the option `-d`, YACC generates a header file `y.tab.h` with definitions of the tokens. The description of the lexer `lex2.l` can include this file.

Output of your project

The parser will repeatedly call the lexer, which returns the next token. So now you shall comment out the main function in the source code of the lexer `lex2.l`. Let the file `yacc2.y` contain the source code to the parser. In `yacc2.y` add the code

```
main() {
    do{yyparse();}
    while(!feof(yyin));
}
```

This will make the parser call the lexer for the next token until there are no more tokens. Now make sure that, if the input to the parser is grammatically correct then there is no output from the parser, and if it the input is not grammatically correct then the parser says on which line the first error is, and then terminate. In case there are errors, and the first error is on line number 11, then the output from your parser should be

```
syntax error, line 11
```

to `stderr`.

If there were no errors then the parser shall output some information about the parse tree. It shall how declarations there are on top level, and for each function (procedure) how many declarations, statements and binary operations (`*`, `/`, `+`, `-`, `<`, `>`, `==`, `&&`, `||`) there are inside the function. For instance parsing the program

How to Generate a Parser Executable:

Generate the lexer and parser using `flex` and `YACC`, respectively:

```
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.

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 example of the expected output for your 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;
  void y;
  x=a * 3.4;
  if(y==y){
    x=3.78;
  };
}
```

warning: type cast int to float, line 4
type error: int expression expected, line 4

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. 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.