Programming Project 1 –Type Checking and Storage Allocation

Due date: March 15, 2013 at midnight PST

Description: In this first compiler project, you will be asked to use and understand an existing parser and perform two program analysis and code generation intermediate tasks, namely type checking and storage allocation for a subset of the C programming language. We now describe each of these two tasks and provide a series of illustrative examples that will help you understand the goal of the project as well as develop a work plan.

Type Checking: As a strongly typed language a C compiler must ensure that it understands how to evaluate and store the binary representation of every high-level program abstraction the program manipulates. For example, if a variable is declared in the source program as an `int` in C the compiler will use 32-bits to store the representation of that variable's value. If there is an assignment statement in the program of the form `a = b;` the compiler must ensure before code generation that both variables have compatible types.

A type checker is thus an intermediate compiler analysis pass that checks that the types of the expressions involved in assignment and any copying operation (such as passing an argument to a function in a function call) 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 type mismatch is, either output a warning or an error message (this is specified below).

In order to keep the project's scope reasonable, you may make the following assumptions. First, there are only four base types, namely: `void`, `char`, `int`, and `double`. In addition there are user-defined types using `struct` and `union` as well as pointer types to the base and user-defined types. To keep the implementation simple, you can assume that a single level of pointer type is allowed (although the grammar we have provided does not strictly enforces it).

Types of basic constant expressions originate from constants (like `5`, which has the type `int`, and `6.31` which has the type `double`), formal parameters (like `x` in `void f(int x){…}`), and variable declarations in the beginning of blocks (like `{int x; double 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`). There may be global variable declarations, which will correspond to global storage allocation, as describe in the second component of the project.

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 to handle sequences of statements, `block`-, `if`-, `while`-, `assignment`-, and `return`-statements; expressions containing constants, variables, `(`, `)` , `~`, `-`, `+`, `*`, `/`, `<`, `>`, `++`, `--`, `==`,`!=`, `<=`, `>=`, `&&` , `||`. The boolean operators expect the operands to be of the same numeric type, either `int` or `double`, but never any user-defined type. Pointer types for the purpose

of Boolean operations, such as a nil comparison are to be treated as integer. In this simple language there are no type cast operators. The return type of a function can be `void`, which means that nothing should be returned and cannot be used as part of any expression.

A type violation or 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 `double` is expected but an `int` is given. In this case a warning message shall be printed. For all other kinds of type errors, an error message shall be printed instead. When a `double` is expected but an `int` is given, the type checker shall (internally and only for the purposes of the subsequent analysis) perform an automatic type cast from `int` to `double`. 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. This approach allows for a warning to generate a single message and not create a cascading sequence warning messages for all the occurrences of mild type violations in the same expression tree (see the examples at the end of this document for further clarification).

Storage Allocation: After type checking the compiler is ready to carry out the first phase of an internal pass of storage allocation. In this phase the compiler needs to determine for each of the functions it has to generate code for, the amount of static storage and/or stack storage is required to support the corresponding function/procedure execution.

For instance, if a function has declared as local variables two integer variables and one double variable, it requires two slots of 32 bits and one slot of 64 bits as the base storage space in the function's Activation Record (AR) in addition to other run-time storage space requirements. Storage space for the arguments and return value of the function is also required. This tasks of computing the storage requirements is complicated by the presence of nested scopes in languages such as C. As discussed in class, you will need to understand the life-times of the various scopes to produce a reduced storage requirement for a given function as life-times that are disjoint in time (function's execution) can share the same activation record storage.

As part of this phase of your project you will determine for each function the number of bytes required to hold the local variables each function manipulates. This space requirement is to include the space required for the function arguments. In addition you need to determine for each of the user-defined type the offset of each `struct` and `union` field in preparation for code generation. Do not forget that global file variables are to be analyzed as well and reported as part of a global function as one of the examples at the end of this document illustrates.

Input to your Project: As part of the evaluation effort, we are going to be using a series of small C program files to automatically test you code development. We will be using a total of 20 such tests 10 of which will be made available to you for your own use as aids in development and interim code testing. The remainder 10 tests (which will be of the same size and code complexity) will be used to augment the automated testing and evaluation procedure.

You can assume for the purposes of this project that all input files we will be using to test your programming project are syntactically correct with respect to the grammar provided. At the end of this document there is a sample set of input files and compiler outputs expected from your project.

Output of your Project: The main compiler project file includes the main function. This main function includes a section of the code of the form shown below where the parsing occurs.

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

After this parsing pass, you should invoke a function you have developed that traverses the abstract syntax tree (AST) generated during the parsing (by the `yyparse` function and as discussed in class) and carry out the type checking and storage allocation phases of this project.

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 examples 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.

```
1:   void f(int a){          warning: type cast int to double, line 5
2:     int x;                type error: int expression expected, line 6
3:     double y;
4:     x = 12;
5:     y = 23;
6:     x = 3.78;
7:   }
```

Example 2.

```
1:  void f(int a){           type error: int expression expected, line 4
2:    int x;                 type error: int expression expected, line 6
3:    x = x + a * 3;
4:    while(3.5){
5:      x=3.78;
6:    };
7:  }
```

Example 3.

```
1:  int A[10];               type error: int expression expected, line 4
2:  double x;
3:  int f(){
4:    return 1.1;
5:  }
```

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

Output from the Storage Allocator: We now present several examples 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.

```
1:   void f(int a){           global: 0 bytes
2:     int x;                 f: 16 bytes
3:     double y;
4:     x = 12;
5:     y = 23;
6:     x = 3.78;
7:   }
```

Example 2.

```
1:  void f(int a){            global: 0 bytes
2:    int x;                  f: 8 bytes
3:    x = x + a * 3;
4:    while(3.5){
5:      x=3.78;
6:    };
7:  }
```

Example 3.

```
1:  int A[10];                global: 48 bytes
2:  double x;                 f: 4 bytes
3:  int f(){
4:    return 1.1;
5:  }
```

In order to separate the output of the two passes you will have to include an output separator as illustrated below for the code example 3 above.

```
--- Type Checker Pass ---
type error: int expression expected, line 4
--- Storage Allocation Pass ---
global: 48 bytes
f: 4 bytes
```

Base Code and Project Development: To make this project accessible in the allotted time and expected student effort, you will be given a scanner (a Lex `scanner.l` file) and skeleton parser (a YACC `parser.y` file). The compiler code is anchored in the main function in the `main.c` file.

The code basis for this project is developed in C and you are suggested to continue your work in that same development environment. The grammar productions already have code that builds an abstract syntax tree (AST) using very simple library functions that manipulate generic data structures. These data structures can be augmented for the needs of type checking and storage allocation derivation. You need to: 1) understand and complete the grammar and 2) fill in the rules (C code) to perform the type checking possibly with the help of an auxiliary symbol table.

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 o a set of electronic automated tests.

How to Generate a Compiler Executable: There is a `makefile` with the distribution files which includes the lexer, parser and the auxiliary library function. Still you can manually generate the compiler executable by manually issuing the command sequence below:

```
flex scanner.l
yacc -d -v parser.y
```

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

```
gcc -c lex.yy.c
gcc –c y.tab.c
gcc -g -o a.out main.o y.tab.o lex.yy.o lib/node.c lib/tabid.c lib/util.c -lfl -g
```

Some test cases with the correct results can be found at the class website. Test and make sure that everything works as your projected will be automatically graded using the UNIX `diff` utility.

Turn-in Instructions: You are expected to turn in your project electronically by e-mailing the instructor and including the following string as your message subject: "`CSCI565-Spring2013 Project1 number`" where `number` stands for your USC student id. Please included as an attachment to the e-mail message a tared and zipped file with all your files and `makefile` named `number.proj1.tar.zip`. Before submitting your project please make sure that there are no external folder dependences and that the commands below generate an `a.out` executable and results in the creation of a folder `number.proj1` with executable file `a.out` in it. Please make sure you do understand these commands. 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.

```
unzip number.proj1.tar.zip
tar -xvf number.proj1.tar
cd number.proj1
make
```