1 Compiler Project

The class project is to build a simple recursive decent (LL(1)) compiler by hand (not using compiler construction tools such as `flex` or `antlr`). You can use any imperative block structured programming language that supports recursion and for which I can install a standard debian package to test your solution on my computer. Examples of languages that students have used for this class include: `c`, `c++`, `java`, and `python`. If you are not certain that your desired programming language is ok, please check with me. While you can use a wide selection of languages, you cannot use any language features for constructing compiler subsystems (regular expression parsers, etc). That said, I encourage you to use some of the more complex builtin data structures of these languages such as hash tables. Again, if you have questions about what you can and cannot do, please ask.

I have organized the compiler project into 5 development phases with deadlines scattered throughout the course semester period. While these deadline are soft, I will use your history of early/late to assign plus/minus graduations to your final grade. I encourage you to attempt to complete these phases early.

2 The Scanner

I recommend that you build a main program for the scanner that has processes command line arguments to obtain the filename for the input program. The main program should have a static variable to hold the current line count, error status, and you should also have some set of error reporting functions; at most likely something like:

```
void reportError(char *message)
void reportWarning(char *message)
```

While a compiler will attempt to continue in the presence of both errors and warnings, an error condition will generally cause the compiler to proceed only with the parse and type checking phases; code optimization and generation should not occur when errors are encountered in the parsing of the input program.

In addition to providing error reporting functions, the main program should call the scanner initialization functions (to attach the input file) and then it should simply invoke the scanner for tokens until EOF is reached. Debug statements of the tokens reported would be a good idea.

The scanner should have an initialize method to setup some initial values in the system (e.g., zero out the line counter, clear the error flags, etc) and attach the scanner to a file:

```
bool scanner->init(char *filename),
```

and another method to scan for the next token:

```
token scanner->getToken().
```

Of course the first call could be a more general purpose initialization method that attaches the a file, initializes some tables (e.g., reserved words) and so on.

The scanner must skip whitespace, newlines, tabs, and comments; comments are start with the string "//" and continue to the next newline character. It should count newlines to aid the error reporting functions.

Illegal characters should be treated as whitespace separators and reported as errors. These errors should not stop the parser or semantic analysis phases, but they should prevent code generation from occurring.

Your scanner should recognize the following tokens. Identifiers defined as:

```
[a-zA-Z][a-zA-Z0-9_]*
```

strings defined as:

```
"[a-zA-Z0-9 _,;:.']*"
```

numbers defined as:

```
[0-9][0-9_]*[.[0-9_]*]?
```

the following:

```
: ; , + - * / ( ) < <= > >= != = := { }
```

the following reserved words:

```
string      case
int         for
bool        and
float       or
global      not
in          program
out         procedure
if          begin
then        return
else        end
```

and of course `EOF` (end-of-file).

While the scanner can be constructed to recognize reserved words and identifiers separately, I *strongly* recommend that you fold them together as a common case in your scanner and seed the symbol table with the reserved words and their corresponding token type. More precisely, I recommend that you incorporate a rudimentary symbol table into your initial scanner implementation. While the data types of the symbol table entries are likely to expand as you build additional capabilities into your compiler, initially you can have the symbol table entries record the token type and have a pointer to the string for the identifier/reserved word. For example, each element in your symbol table could have the following structure:

```
sym_table_entry : record
  token_type : TOKEN_TYPES;
  token_string : *char;
end record
```

where `TOKEN_TYPES` is the enumeration type of all your token types.

Operationally, I would build the symbol table so that new entries are created with the `token_type` field initialized to IDENTIFIER. You can then seed the symbol table with reserved words in the scanner's initialize method. The easiest way to do this is to setup an array of reserved word and their token type. Then walk through the array to do a hash look up with each reserved word string and change the `token_type` field to the specified token type. We will go over this in class.

I would also recommend defining character classes to streamline your scanner definition. In short, what this means is you should define an array indexed by the input character that maps an ASCII character into a character class. For example mapping all the digits [0-9] into the `digit` character class, letters [a-zA-Z] into the `letter` character class, and so on (of course you have to define the character classes in some enumeration type. I will go over this more in class for you.

3 The Parser

Build a recursive decent parser that looks only at the immediate next token to control the parse. That is, build an LL(1) parser from the project programming language specification given elsewhere in these webpages.

The parser should have at least one resync point to try to recover from a parsing error.

4 Type Checking

Incorporate type checking into the parser and perform type checking while the statements are parsed. Your principle concern is with scoping and type matching. At least for expressions and statements, your parsing rules will now have to be expanded to return the type result for the construct just parsed. The upper rules will use that type information to assert type checks at its level.

A full symbol table complete with scoping data must be constructed. You must be able to define a scope and remove a scope as the parse is made. You can achieve scoping by having nested symbol tables or by chaining together the entries in the symbol table and placing scope entry points that can be used to control how symbols are removed when your parser leaves a scope.

5 Code Generation

Basically the generated file should have declarations for your memory space, register space and a flat C (no subroutines) with goto's used to branch around the generated C file.

Your generated C must follow the style of a load/store architecture. You may assume a register file sized to your largest need and a generic 2-address instruction format. You do not have to worry about register allocation and you should not carryover register/variable use from expression to expression. Thus a program with two expressions:

```
c := a + b;
a : = c - b;
```

would generate something like:

```
;c := a + b;
R[1] = MM[44]; assumes variable a is at location 44
R[2] = MM[56];
R[1] = R[1] + R[2];
MM[32] = R[1];
;d : = a + c + b;
R[1] = MM[44];
R[2] = MM[68];
R[1] = R[1] + R[2];
MM[44] = R[1];
```

You can also use indirection off the registers to define memory locations to load into registers. For example your code generator can generate something like this:

```
R[1] = MM[R[0]+4];
```

You can statically allocate/assign some of the registers for specific stack operation (pointers). The stack must be built in your memory space.

For conditional branching (goto) you can use an if statement with a then clause but not with an else clause. Furthermore the condition must be evaluated to true/false (0/1) prior to the if statement so that the condition in the if statement is limited to a simple comparison to true/false. Thus for conditional branching only this form of an if statement is permitted:

```
if (R[2] = true) then goto label;
```

The code generator is to output a restricted form of C that looks much like a 3-address load/store architecture. You can assume an unbounded set of registers, a 32M memory space containing space for static memory and stack memory. Your machine code should look something like (I forget C syntax, so you may have to translate this to real C):

```
 Reg[3] = MM[Reg[SP]];
 Reg[SP] = Reg[SP]  2;
 Reg[4] = MM[12]; // assume a static
// variable at
// location 12
 Reg[5] = Reg[3] + Reg[4]
 MM[12] = Reg[5];
```

You must use simple C: assignment statements, goto statements, and if statements. No procedures, switch statements, etc.

You must evaluate the conditional expressions in if statements and simply reference the result (stored in a register) in the if statement of your generated C code.

Basically you should generate C code that looks like a simple 3-address assembly language.

6 Runtime

For the runtime environment, you should enter the runtime function names and type signatures into your symbol table prior to starting the parse of the input files. To code generate for these functions, you can either special case them and use C function calls or you can have a static (handwritten) C program with predefined labels (on the hand written code C code that calls your library functions) that you generated code can goto. This second option sounds more difficult but is probably much easier to implement as it's not a special case in your code generator.

There are several (globally visible) predefined procedures provided by the runtime support environment, namely:

- bool getBool()
- integer getInteger()
- string getFloat()
- string getString()
- integer putBool(bool)
- integer putInteger(integer)
- integer putFloat(float)
- integer putString(string)

The put operations always returns the value 0. These functions read/write input/output to standard in and standard out. If you prefer, these routines can read/write to named files such as input and output.