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++`, `go`, `rust`, `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.

In addition to sending me your compiler source and build environment that I can run on my linux workstation (you are responsible for ensuring that it will build on a standard linux box; if you build it on some exotic system like Haiku, we can discuss a demo on that platform), you must also turn in a one page report documenting your compiler. It should document your software, its structure, the build process, and the language features that are correctly implemented as well as those elements of the compiler that are not completed. Finally, the report should also highlight any unique features you have implemented in the system.

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 Lexical Analysis: The Scanner

2.1 Tokens

Lexical analysis involves converting strings in the input language to tokens. Here is a representative example of the object type for tokens.

```
// a globally visible enumeration (needed by both the lexer and by the parser
enum tokenType = {PLUS, MINUS, IF_RW, LOOP_RW, END_RW, L_PAREN, R_PAREN,
                  L_BRACKET, R_BRACKET, ... , NUMBER, IDENTIFIER}

class token
  tokenType: tt
  tokenMark: tm
end class
```

The token mark can be a complex data type that records secondary information about the token. For many token types (*e.g.,* `PLUS`) there is no token mark data required, the token type value fully characterizes the token. For other token types, (*e.g.,* `IDENTIFIER`) the token mark will contain additional information to characterize the token; initially this will mostly be to hold an identifier string, but later it may well contain information about complex types such as functions/procedures and their argument list signature/return type, etc. In some systems, a compiler might combine arithmetic operators, relational operators, multiplier operators, etc into a common token type and use the token mark to record the specific member of that token class that is being represented.

2.2 Supporting functions/objects

There are some key support functions that can make building the compiler much easier; especially if you encapsulate them with a strong API that permits the restructuring/extension of the underlying implementation. I will organize these support functions into 3 parts: (i) input processing, (ii) warning and error reporting, and (iii) symbol table management and reserved word setup. Below I will present suggestions for each of these components. You are not required to setup your solution this way, these are simply my recommendations to you. I will document each of these in an object-oriented basis, you are not required to setup/use an object-oriented language, this is just a convenient way for me to present the ideas.

2.2.1 Input Processing

I recommend that you create an object to manage your input file setup and location recording (what line in the file is currently being processed). This might be a bit much for this project, but it helps encapsulate stuff related to the input file being processed. It is a good plan to build it this way as if you ever move to a more complicated language where multiple files are processed while processing the designated input file, it is easy to have a stack of file points/line count variables to record where the system is in processing the various files required.

In general the lineCnt variable will be used to record which line in the input file the scanner is currently working on. In this project, this value is used primarily to help generate meaningful error messages.

```
class inFile
  private:
    file: filePtr = null  // the input file
    string: fileName
    int: lineCnt = 0      // the line count; initialized to zero
  public:
    bool: attachFile(string)  // open the named file
    char: getChar()           // get the next character
    void: ungetChar(char)     // push character back to the input file string
    void: incLineCnt()
    void: getLineCnt()
end class
```

2.2.2 Warning and Error Reporting

The main program should have an object for reporting errors and warnings. Ideally these functions will output error messages using a standard format (*e.g.,* `https://gcc.gnu.org/onlinedocs/gcc-3.3.6/gnat_ug_unx/Output-and-Error-Message-Control.html` that supporting tools such as emacs can use to, for example, automatically position your text editor/IDE to the correct file and line number corresponding to the warning/error. The API for this is fairly simple.

```
class reporting
  private:
    bool: errorStatus = False  // true if the compiler has discovered an error
  public
    void reportError(char *message)
    void reportWarning(char *message)
    bool getErrorStatus()
end class
```

While a compiler attempts 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 general the `reportError` function will set the private variable `errorStatus` to `True`.

2.2.3 Symbol Table Management/Reserved Words

Most compilers will use a hash table(s) of the symbols seen in the input file; this is called the symbol table. These symbols would be identifiers and functions. This is also a convenient place to drop entries for the reserved words in the language (such as `if`, `loop`, `end`, and so on). The symbol table will be revised and extended as you build latter parts of the compiler, so it is imperative that you keep access to it setup through and API so that its actual implementation is easily modified later. The `hashLook` function will look into the symbol table and return the token for the string argument. If this the first time an entry to the symbol table occurs for this string, then a new entry is created with a default token definition (generally `IDENTIFIER`).

```
class symbolTable
  private:
    hashTable<token>: symTab
  public:
    token: hashLook(string)  // lookup the string in the hash table
    void: setToken(string)   // change the token values for this symbol
end class
```

In general the lexer will lookup every candidate identifier string in the symbol table and return the token stored in the symbol table for that string. Thus, to make life easy, a good idea is to pre-seed the symbol table with the reserved word strings and setup the tokens for each so that instead of returning the token `IDENTIFIER`, the correct token for that reserved word is returned. Thus, before you start processing the file, you will build a simple look to iterate through the reserved word strings in order to initialize the symbol table with the reserved word tokens.

Constants

So the question sometimes comes up, how do we treat *strings* and *numerical values* in the lexical analysis phase. There is no single uniform answer to this. You can treat them: (i) directly as tokens, (ii), register them in the symbol table, or, (iii) build a separate string/numeric table(s)[1] to store the token representation of the item. If we assume that the lexer match string for the token is stored as an ascii string in the variable `tokenString`, then examples of the lexer return code for each of these options can be outlined as (showing both for a STRING token and for an INTEGER token):

```
(i)   return new token(STRING, tokenString)

      return new token(INTEGER, atoi(tokenString))

(ii)  tok = symbolTable.hashLook(tokenString)
      if (tok.tt != STRING) { tok.tt = STRING }
      return tok

      tok = symbolTable.hashLook(tokenString)
```

[1] Either a single common *constantTable* for both, or having two separate tables, one for strings and one for numerics is possible.

```
      if (tok.tt != INTEGER) { tok.tt = INTEGER }
      // optionally we might also build/add the integer value to the token
      return tok

(iii) return new token(STRING, constantTable.hashLook(tokenString))
      return new token(INTEGER, constantTable.hashLook(tokenString))
```

Of course there are many variations on this coding example. The option of using the symbol table or a separate constant table is a good way to compress the final storage map. That is, if you store them in the symbol table, then (at each scope) common constants will be represented as one item that has to be mapped into memory during the code generation phase. If instead, you store all constants in constant table(s) that transcend all scopes, you can potentially reduce the storage map size even further.

2.3 The Scanner

I recommend that you build a scanner object with the principle API call `scan()` that returns the next token in the input file. Normally your parser will call the `scan()` function to get the next token to determine the next course of action for the parser. When the file has advanced to the end of the file, you should have an end-of-file (`EOF`) token that can be returned to the parser.

For purposes of this step of your compiler, I recommend that you build a main program that initializes the symbol table and iteratively calls `scan()` until the end-of-file is reached.

The scanner must skip white-space, 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 white-space separators and reported as errors. These errors should not stop the parser or semantic analysis phases, but they should prevent code generation from occurring.

The tokens your scanner should recognize are the tokens found in the project language specification.

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.

I am leaving the remainder of this section in place; it is from an earlier version of this document that may or may not be helpful to you.

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 (during the scanner initialization step described above) 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.

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. If you really would prefer to build a LALR parser that is possible, but please discuss it with me first.

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

You have two options for code generation. The first (and recommended) option is to use the LLVM backend optimizer and code generator. In this case your code generation phase would really be a translator to the LLVM intermediate form (either the memory resident IR or the llvm assembly). The second option is to generate a file containing a restricted C program space as documented below.

5.1 Generating C

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

would generate something like:

```
// c := a + b;
R[1] = MM[44]; // assumes variable a is at MM location 44
R[2] = MM[56]; // assumes variable b is at MM location 56
R[3] = R[1] + R[2];
MM[32] = R[3]; // assumes variable c is at MM location 32
// d := a + c + b;
R[1] = MM[44];
R[2] = MM[32];
R[3] = R[1] + R[2];
R[4] = MM[56];
R[5] = R[3] + R[4];
MM[144] = R[5]; // assumes variable d is at MM location 144
```

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 64M bytes of 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.

5.2 Generating LLVM Assembly

See other lecture notes on LLVM.

5.3 Activation Records

See other lecture notes on Code Generation and Figure 1.

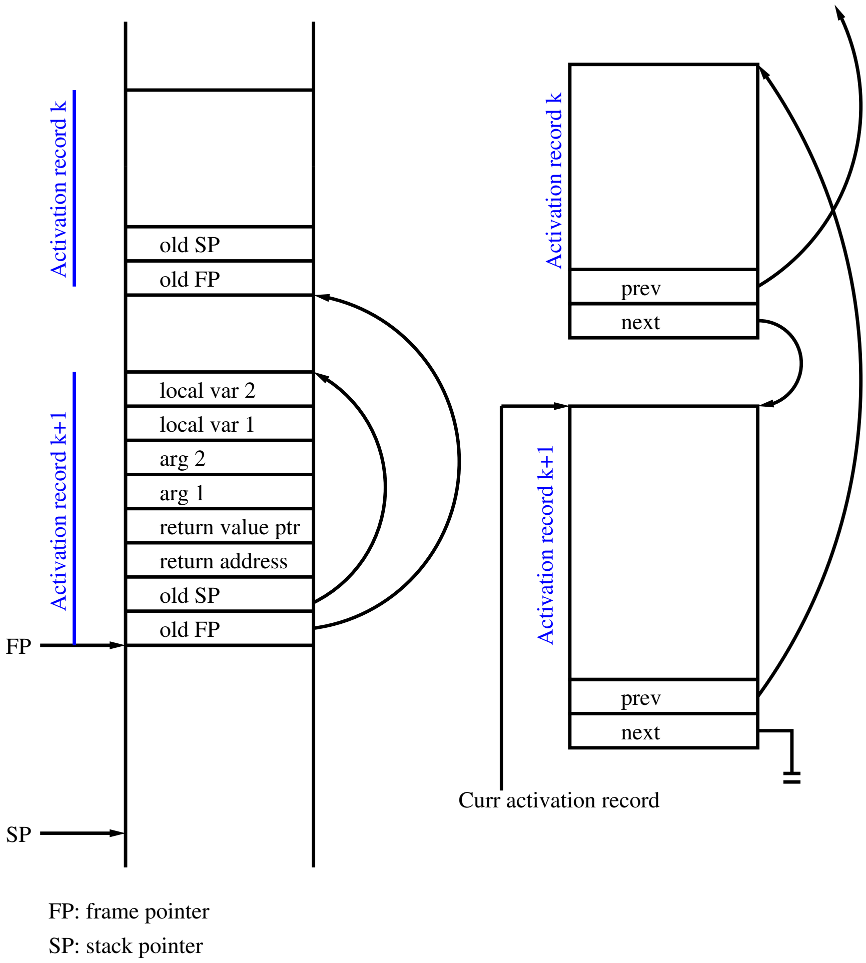

Figure 1: The call chain of activiation records; stack model on left and heap model on right

6 Runtime

If you are using the LLVM infrastructure, you can use the libc supported gets, puts, atoi, etc functions as your runtime system. This means that you will not end up writing the runtime library other than adapting the code generator to interface to the libc standard.

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 the functions described in the project language description.