Ledgard Compiler Project

Once you've finished this project, you will have all the pieces of a Linux compiler for the Ledgard programming language. All you'll need to add is a "driver" that allows the user to specify on the command line the file(s) to compile and automatically invokes the assembler and linker to complete the compilation. The project will involve your making use of pretty much everything you've learned in previous CS courses, plus some new stuff you'll learn in this class.

You will complete the project in six phases. The compressed archive project.tar.gz (available on the wiki) contains the skeleton code you'll need. Once you've downloaded this file, the command

```
tar xvf project.tar.gz
```

will create and populate the following directories (one for the project as a whole, with subdirectories for each phase).

Each of the subdirectories contains the files necessary to compile and test a single phase of the project:

- class interfaces (.h; readonly)
- skeleton code for you to complete (.cpp)
- driver program for basic testing (main.cpp)
- input files for basic testing (.led)
- correct output from running the driver on the test input (.out)

Each one also contains a "makefile," which simplifies the process of compiling the code and linking the components. If phase*n* is the correct directory, the command

```
make
```

Will compile your code and the driver, linking it with the code from the earlier phases to create a phase*n*-driver program that you can use to test your code, and the command

```
make test
```

will run the driver program using the sample input files and report on the results. Note that these tests are in no way exhaustive, and you will certainly want to run your own tests as well, perhaps even modifying the makefile to do this automatically for you.

Each phase also contains a "`precompiled`" directory, which contains correct, precompiled versions of all the modules (including the one you're working on) as well as the driver program, so that you can compare the results of your code with that of (presumably) correct code. The precompiled modules from earlier phases are used in building each phase, so if you're having trouble with one phase, you can work on the next one before you've completed it (and come back to the difficult one later). The only exception to this rule is phase 3. you need to complete phase 2 successfully before you can work on phase 3.

For each phase, you will submit your appropriate .cpp file to your instruction as an *attachment* to an email message. Your instructor will test your program thoroughly as soon as he can, and return the results to you via email (along with some hints about why you failed one or more of the tests.) You may resubmit corrected versions of each as often as you like, until you pass all your instructor's tests, or until the last day of classes, whichever comes first.

You may make any changes you like to the completion/submission file (except when a comment in the file explicitly for bits such a change), but your changes must still be compatible with the class header files (which are read-only and cannot be changed). If you make incompatible changes, your code will not compile or link correctly when your instructor tries to test it.

1 Phase 0: Ledgard Programming

File to complete and submit: `phase0.led`

To get you started working with the Ledgard language, you need to write a program in that language. You must submit a file named `phase0.led` containing a Ledgard program that sorts its input. Your program must first read an integer value that specifies how many values are to follow (but no more than 100), and then read that many values into an integer array. It must then sort the values in the array into ascending order, and print out the values in order, one value per output line.

You may use any sort algorithm you wish, although recursive algorithms like quick-sort will be tricky; you'll have to do your own stack management.

2 Phase 1: Lexical Analyzer

File to complete and submit: `lexer.cpp`

For this phase you will complete the implementation of the `Lexer` class. A Lexer is an object that turns the strings in a given input stream into tokens, and operates like a queue (the Token type is defined in the `lexer.h` header file along with the Lexer class)

The `pop` and `empty` member functions have already been implemented. Your job is to implement the class constructor and the `curr` member function, which returns the current token in the stream (`ENDFILE` if there are no more tokens left). The easiest way to do this is to read and tokenize the entire stream, putting each token into the data queue and have curr simply return `data.front()` (and this is probably what you'll want to do for your first draft). A better (and slightly more difficult) implementation would be to do nothing in the constructor, but to read and tokenize (perhaps one line at a time) only if the data queue is empty when curr is called.

3 Phase 2: Parser

File to complete and submit: `parser.cpp`

This phase implements the `Parser` class (the parse member function in particular), which parses a Ledgard program using the technique of recursive descent. Each production in the grammar should have a corresponding function which returns `true` or `false`, depending on whether or not the parse is successful (some productions, of course, may be combined into a single function, for efficiency). The `parse_program` function, which calls `parse_decl` and `parse_stmt` is already implemented. The rest is up to you.

The `mustbe` function is the only one that reports errors. Use this function; do not attempt your own error reporting. The present version exits the program when the first error occurs. We will discuss in class various ways a parser can recover from errors, but for the purposes of this project, leave things as they are.

4 Phase 3: Generating a Parse Tree

File to complete and submit: `parser.cpp`

This is the only phase of the project that requires that the previous phase be completed before work can be done on it. This is also the only phase which requires the use of pointers, so you need to be careful to avoid memory leaks. Your job is to modify the functions you wrote for Phase 2 so that instead of returning a boolean value, they return a pointer to an appropriate `Parse_Tree` node. These node types are defined in `parse-tree.h`, and rely on inheritance, polymorphism, and recursion to do their jobs properly. Close study of this file will more than repay the effort.

Many of the member functions declared in `parse-tree.h` will not be implemented until later phases. Dummy versions of these functions are defined in the `unimplemented.cpp` file. Each succeeding phase will implement some of these functions, and leave some unimplemented. The last phase, of course, completes the implementation, and this file then disappears from the project.

5 Phase 4: Symbol Table

File to complete and submit: `symtab.cpp`

Usually, the symbol table work is done during the parsing phase, but in the interest of clarity, we keep it separate here. Our symbol table is a `struct` containing three parts: a `map` of names to types, a `set` of names that are used but never defined, and a set of names that have been defined more than once.

Note that the symbol table variable, `symtab`, is defined as a *global* variable. This is one of the very few situations in which a global variable is proper programming practice. See if you can understand why.

You will implement the member functions `build_symtab` (for declarations) and `check_symbols` (for everything else). The `build_symtab` function adds every identifier in a declaration (and its type) to the symbol table, adding identifiers to the multiply-defined set whenever an attempt is made to define an identifier that has already been defined. The `check_symbols` function adds any identifier not defined in the symbol table to the undefined set.

6 Phase 5: Type Checking

File to complete and submit: `typecheck.cpp`

Type checking, too, is usually done during parsing, as the symbol table is built, but again, we do it in a separate phase. This is probably the most difficult of the phases to get right, as we want to catch every type error we can, but not to report any error more than once. The `get_type` and `match_types` member functions are the ones you'll need to implement. Only the match types functions report any errors, and must use the `report_type_error` function (defined in `typecheck.h`). Source lines with multiple type errors should only be reported once.

The implementation of the `get_type` functions is mostly straightforward, except for variables. To determine the type of a subscripted variable, it is necessary to find the base type of the variable. To do this, you will need to use the `dynamic_cast` operation. If the variable `t` is a pointer to a `Type_Node`, the construction

```
Array_Node* a = dynamic_cast<Array_Node*>(t);
```

will determine if `t` actually points to an `Array_Node`. If it does, the assignment succeeds, and `a->get_base_type()` gives you the array's base type. If not, `nullptr` will be assigned to a. If a variable that is not actually an array is subscripted, `get_type` should return `Type_Node::VOID` to indicate that there is no such variable.

The implementation of the `match_types` functions is somewhat trickier, as the types of all subexpressions have to be checked for type compatibility as well. These functions must use calls to `get_type` to enforce the Ledgard type matching rules. A few extra `match_types` member functions are already implemented in order to give you a better idea of what's involved. Again, variables with subscripts are likely to cause the most problems.

7 Phase 6: Code Generation

File to complete and submit: `codegen.cpp`

Code generation is pretty much a case of determining the proper code pattern for each construct and generating code that matches that pattern. The code patterns for each of the Ledgard constructs are given below. If you put your generated code into a file with a `.s` extension (`program.s` for example, you can use the Linux assembler to create an object file (`program.o`),

```
as32 -o program.o program.s
```

the loader to link it with the Ledgard library routines and create an executable (program),

```
ld32 -o program program.o precompiled/ledgard-lib.o
```

which you can then execute directly

```
./program
```

The precompiled directory for this phase contains the `.s` files that should be generated for each of the test programs, as well as executable versions.

Code to generate for a program

```
.section .bss
<code for declaration>
.section .text
.globl _start
_start:
<code for statements>
        xorl    %ebx,%ebx
        movl    $1,%eax
        int     $0x80
```

Code to generate for a declaration

For each identifier declared, generate the following:

```
        .lcomm  <identifier>, <size of type>
```

Code to generate for an assignment statement

```
<code for the variable's address>
<code for the expression>
        popl    %eax
        popl    %ebx
        movl    %eax,(%ebx)
```

Code to generate for an exchange statement

```
<code for the left variable's address>
<code for the right variable's address>
        popl    %ebx
        popl    %ecx
        movl    (%ebx),%eax
        movl    (%ecx),%edx
        movl    %eax,(%ecx)
        movl    %edx,(%ebx)
```

Code to generate for an 'if' statement

```
<code for the condition>
        popl    %eax
        test    %eax,%eax
        jz      label1
<code for then-part statements>
        jmp     label2
label1 :
<code for else-part statements>
label2 :
```

Code to generate or a ‘while’ statement

```
label1 :
<code for the condition>
        popl    %eax
        test    %eax,%eax
        jz      label2
<code for the body statements>
        jmp     label1
label2 :
```

Code to generate for an ‘input’ statement

For each variable listed, generate the following:

```
<code for the variable’s address>
        call    <read dec if variable is integer, read bool otherwise>
        popl    %ebx
        movl    %eax,(%ebx)
```

Code to generate for an ‘output’ statement

For each variable listed, generate the following:

```
<code for the variable’s value>
        popl    %eax
        call    <print dec if variable is integer, print bool otherwise>
```

and then generate

```
        call    print_newline
```

Code to generate for a ‘<’ operation

```
<code for left operand>
<code for right operand>
        popl    %ebx
        popl    %eax
        xorl    %ecx,%ecx
        cmpl    %ebx,%eax
        jnl     1f
        incl    %ecx
1:      pushl   %ecx
```

Code to generate for a ‘<=’ operation

```
<code for left operand>
<code for right operand>
        popl    %ebx
        popl    %eax
        xorl    %ecx,%ecx
        cmpl    %ebx,%eax
        jnle    1f
        incl    %ecx
1:      pushl   %ecx
```

Code to generate for a ‘==’ operation

```
<code for left operand>
<code for right operand>
        popl    %ebx
        popl    %eax
        xorl    %ecx,%ecx
        cmpl    %ebx,%eax
        jne     1f
        incl    %ecx
1:      pushl   %ecx
```

Code to generate for a ‘<>’ operation

```
<code for left operand>
<code for right operand>
        popl    %ebx
        popl    %eax
        xorl    %ecx,%ecx
        cmpl    %ebx,%eax
        je      1f
        incl    %ecx
1:      pushl   %ecx
```

Code to generate for a ‘>=’ operation

```
<code for left operand>
<code for right operand>
        popl    %ebx
        popl    %eax
        xorl    %ecx,%ecx
        cmpl    %ebx,%eax
        jnge    1f
        incl    %ecx
1:      pushl   %ecx
```

Code to generate for a ‘>’ operation

```
<code for left operand>
<code for right operand>
        popl    %ebx
        popl    %eax
        xorl    %ecx,%ecx
        cmpl    %ebx,%eax
        jng     1f
        incl    %ecx
1:      pushl   %ecx
```

Code to generate for a ‘+’ operation

```
<code for left operand>
<code for right operand>
```

```
        popl    %ebx
        popl    %eax
        addl    %ebx,%eax
        jo      overflow
        pushl   %eax
```

Code to generate for a ‘-’ operation

```
<code for left operand>
<code for right operand>
        popl    %ebx
        popl    %eax
        subl    %ebx,%eax
        jo      overflow
        pushl   %eax
```

Code to generate for a ‘texttt*’ operation

```
<code for left operand>
<code for right operand>
        popl    %ebx
        popl    %eax
        imull   %ebx,%eax
        jo      overflow
        pushl   %eax
```

Code to generate for a ‘/’ operation

```
<code for left operand>
<code for right operand>
        popl    %ebx
        test    %ebx,%ebx
        jz      divide0
        popl    %eax
        cltd
        idivl   %ebx
        pushl   %eax
```

Code to generate for an ‘and’ operation

```
<code for left operand>
        movl    (%esp),%eax
        testl   %eax,%eax
        jz      label1
        popl    %eax
<code for right operand>
label1 :
```

Code to generate for an ‘or’ operation

```
<code for left operand>
        movl    (%esp),%eax
```

```
        testl   %eax,%eax
        jnz     label1
        popl    %eax
<code for right operand>
label1 :
```

Code to generate for a ‘not’ operation

```
<code for operand>
        popl    %eax
        xorl    $01,%eax
        pushl   %eax
```

Code to generate for an integer literal

```
        pushl   $<the literal's value>
```

Code to generate or a boolean literal

```
        pushl   $<the literal's value,  0 for false, 1 for true>
```

Code to generate for a variable’s value

```
<code for the variable's address>
        popl    %eax
        pushl   (%eax)
```

Code to generate for a variable’s address

First generate the single instruction

```
        xorl    %esi,%esi
```

Then, for each subscript expression and its corresponding base type,

```
        pushl   %esi
    <code for subscript expression>
        popl    %eax
        popl    %esi
        cmpl    $<corresponding index's upper bound>,%eax
        ja      out_of_range
        subl    $<corresponding index's lower bound>,%eax
        jb      out_of_range
        imull   $<size of current base type>,%eax
        addl %eax,%esi
```

and, finally,

```
        leal    <the variable's identifier >(%esi),%eax
        pushl   %eax
```

8 Optimization (optional)

You may notice that the code generated according to this scheme is not very efficient. In particular, it generates quite a lot of redundant pushes and pops. If you're feeling adventurous, you might want to try generating more optimal code. Removing the redundant pushes and pops, for example, reduces the number of generated assembly language lines by about 20%.

Your "optimized" code must, of course, produce the same results as would the original, "unoptimized" code.

9 Extra Credit

If you add more to your compiler, I will give you extra credit. For instance, writing a code optimizer will increase your score. So would adding other features to the Ledgard language. The harder your chosen task, the greater the points.

Enjoy!