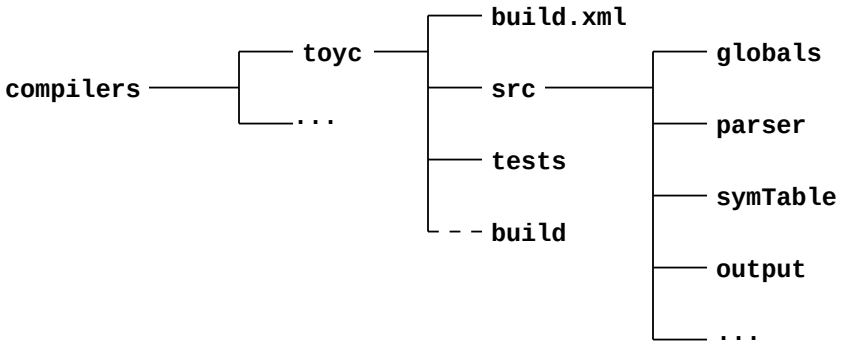
EGRE591 Compiler Project
Java & C++ Helps
Spring 2024

1 Program Development

The customs and practices for software development are somewhat different for Java and C++; therefore the directory structures, files used, and the way your project will be organized will vary depending on which language you choose. Unless you have considerable prior experience in organizing and managing a project of this scope, it is strongly recommended that you follow the organization suggestions provided in the project handouts. The instructor may provide both Java and C++ versions of a sample compiler written in the manner presented in the following sections.

2 Java Projects

2.1 Directory Structure & Buildfile



Your base/project directory should be `compilers/toyc` . The purpose of these directories/packages (more on most all of this will come later):

src all of your source code

tests the toyc source code files used to test your compiler

build all of your class files; created by '`ant compile`', entire directory removed by '`ant clean`'

src/globals contains all definitions needed globally throughout your compiler

src/parser most of the source code for your scanner & parser plus your drivers (e.g. `tc.java`)

src/symTable your symbol table maintenance routines

src/output screen output; code that is called to report compiler messages, including errors & warnings

Following standard practice name your ant buildfile `build.xml` . Your buildfile should have at least 3 targets: `clean`, `compile`, and `test`. Target `compile` depends on `clean` and should create and place all class files in directory `toyc/build/classes`. Issuing an '`ant clean`' should remove the entire subdirectory `toyc/build`.

Target `test` (which should be the ant default target and depend upon `compile`) executes your scanner driver (main method) located in file `parser/tc.java`. You should define two properties to control the flags and source code file names; these properties must be named `flags` and `source`. You would define these flags from the command line in the following manner:
`ant -Dflags="verbose" -Dsource="tests/test1.tc" test`

2.2 Compiler Interfaces

The `compilers` directory contains several interfaces that you might implement (some in Part #1, some later) in your compiler. The code for these interfaces will be made available to you off of the class web pages. You may modify this code for your project as appropriate if necessary.

Part #1 interface ...	**is implemented by ...**
`compilers/Lexer.java`	`parser/TCscanner.java`
`compilers/Token.java`	`parser/TCtoken.java`

2.3 Suggested Source Code Files

Generally all source code files (and therefore classes) related to the front-end (i.e. scanning & parsing) of your compiler should be prepended with '`TC`', e.g. `TCscanner.java` .

2.3.1 File `globals/TCGlobals.java`

(All source listings given in this handout are suggestions as a starting point for your coding; you are not required to use them in the form listed.)

```
package globals;

public class TCGlobals {

public static String inputFileName=null;
public static String outputClassFileName=null;
public static String targetFileName=null;

public static final String ASM_FILE_EXTENSION = "j";

//public static Abstract.Program abstractTree = null;
//public static SymTable.SymTable symTable = null;
//public static CodeGen.TargetCode objectCode = null;

public final static String COMPILER = "";
public final static String VERSION = "";
```

```
public final static String AUTHOR = "";

public static boolean debug_scanner = false;
public static boolean debug_parser = false;
public static boolean debug_codeGen = false;

public static boolean dump_abstractCode = false;
public static boolean dump_objectCode = false;
public static boolean dump_symbolTable = false;

public static boolean verbose = false;
}
```

The variable `debug_scanner` should be set to `false` for Part #1 of the project (users will have the option to set it to true on the command line; more on this later). When `debug_scanner` is `false` (i.e. check for this), you should output nothing except possible warning or error messages (i.e. if all tokens are properly formed, no output will appear). When `debug_scanner` is `true` you should print out all of the token name *and* lexeme pairs as well as the number of correctly-formed tokens in the file (see example below). Note that when `verbose` is true *all* debugging messages are printed.

2.3.2 File `parser/TCscanner.java`

This class will contain the instance method that returns a token when called. Though it's not required, source handling routines (i.e. the code to access the character stream from the source code file) can be defined in this class as well.

2.3.3 File `parser/TCtoken.java`

This class should define your enumerated token type which will be used throughout the project.

2.3.4 File `output/TCoutput.java`

All error and diagnostic output should be produced or generated through calls to reporting routines in `output/TCoutput.java`.

2.3.5 File `parser/tc.java`

This will contain your main method i.e. it is the "driver" program for your compiler. For Part #1 of the project your driver will just repeatedly call your scanner tokenizing the input, i.e. (in pseudo-code):

```
main()
begin_main
  process the command line
  instantiate a scanner object
  while the next token != end_of_file do
```

```
    ; // nothing! the scanner reports tokens only if appropriate flag is set
end_main;
```

2.4 Command Line Arguments

2.4.1 Java Projects

Your program must provide a certain level of control from the command-line, as specified below. Note that you will be adding legal command line options as your compiler is developed stage by stage.

```
liberty:~/compilers/toyc/> java -cp build/classes:../interfaces.jar parser.tc -help
Usage: java [classpath] parser.tc [options] toyc_source_file

where options include:
   -help           display this usage message
   -debug <level>  display messages that aid in tracing the
                   compilation process. If level is:
                      0 - all messages
                      1 - scanner messages only
   -verbose        display all information

liberty:~/compilers/toyc/>
```

If your scanner is working correctly (after an '`ant compile`') and all the tokens in your toyc source code file `source.tc` are properly formed, the command '`java -cp build/classes:../interfaces.jar parser.tc source.tc`' will produce no output. Issuing the command '`java -cp build/classes:../interfaces.jar parser.tc -debug 0 source.tc`' will output all the token/lexeme pairs according to the specification (i.e. global variable `debug_scanner` will be set to true and evaluated in your code). For Part #1 of your project a debug level of 0, 1, or issuing the '`-verbose`' option will all have the same effect. The `verbose` option takes precedence over all other output options. The global variable `inputFileName` (or equivalent in your code) should be set to the file name given on the command line; if no name is given, report an error and output a usage report (i.e. the message printed when issuing the `-help` option) with the exception that the help option can legally exist alone on the command line with no file name. Any ill-formed or illegal options (including specifying an input file that does not exist) should report an error message and then give the usage report.

2.4.2 C++ Projects

```
liberty:~/compilers/toyc/> bin/tc -help
Usage: tc [options] toyc_source_file

where options include:
   -help           display this usage message
   -debug <level>  display messages that aid in tracing the
                   compilation process. If level is:
```

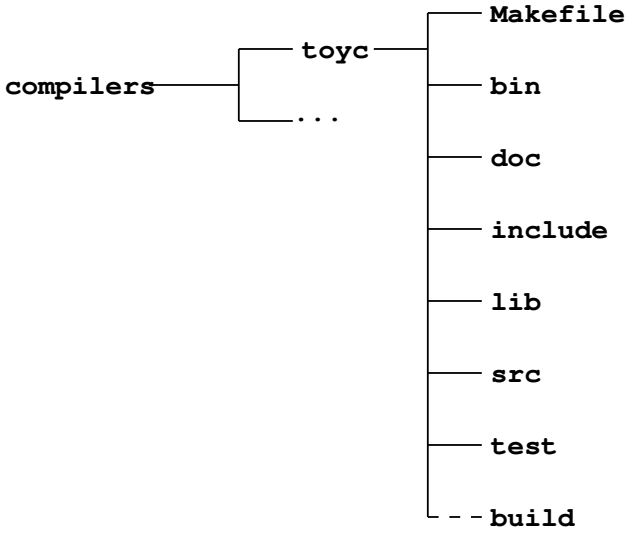
```
                    0 - all messages
                    1 - scanner messages only
  -verbose        display all information

liberty:~/compilers/toyc/>
```

If your scanner is working correctly (after a '`make`') and all the tokens in your toyc source code file `test/source.tc` are properly formed, the command '`bin/tc test/source.tc`' will produce no output. Issuing the command '`bin/tc -debug 0 test/source.tc`' will output all the token/lexeme pairs according to the specification. For Part #1 of your project a debug level of 0, 1, or issuing the '`-verbose`' option will all have the same effect. The `verbose` option takes precedence over all other output options. The global variable `inputFileName` (or equivalent in your code) should be set to the file name given on the command line; if no name is given, report an error and output a usage report (i.e. the message printed when issuing the `-help` option) with the exception that the help option can legally exist alone on the command line with no file name. Any ill-formed or illegal options (including specifying an input file that does not exist) should report an error message and then give the usage report.

3 C++ Projects

3.1 Directory Structure & Buildfile



Your base/project directory should be `compilers/toyc` . The purpose of these directories (more on most all of this will come later):

bin location of your `tc` binary executable (executable is removed by '`make clean`')

doc documentation directory (will be empty)

include location of all your header (`*.h`) files

lib location of all libraries used (probably will be empty)

src all of your C++ source code

test the toyc source code files used to test your compiler

build all of your object code (`*.o`) files; created by your makefile, entire directory removed by 'make clean'

Following standard practice name your make buildfile `Makefile`. Your buildfile should have at least 3 targets: `all`, `clean`, and `test`. Target `all` first do a `clean` before compiling all of your source files and creating an executable. Issuing an '`make clean`' should remove the entire subdirectory `toyc/build` plus remove any executables in `bin`.

Target `test` executes your scanner driver `tc` located in `toyc/bin` on a specified test file. You should define two properties to control the flags and source code file names; these properties must be named `flags` and `source`. You would define these flags from the command line in the following manner:

```
make -Dflags="verbose" -Dsource="test/test1.tc" test
```

3.2 Compiler Header Files

The `include` directory contains all of your header (`.h`) files for your project. Generally you will have a header specification file for every C++ source code file. The following are strong suggestions but you are free to organize your code in any manner you wish.

Part #1 header ...	**is implemented by ...**
`include/TClexer.h`	`src/TClexer.cpp`
`include/TCtokens.h`	(none)
`include/TCtoken.h`	`src/TCtoken.cpp`
`include/TCglobals.h`	`src/TCglobals.cpp`

3.3 Suggested Source Code Files

Generally all source code files related to the front-end (i.e. scanning & parsing) of your compiler should be prepended with '`TC`', e.g. `TClexer.cpp` .

3.3.1 File `include/TCglobals.h`

All source listings given in this handout are suggestions as a starting point for your coding; you are not required to use them in the form listed or at all.

```
namespace toyc {

   extern std::string inputFileName;

   extern std::string COMPILER;
   extern std::string VERSION;
   extern std::string AUTHOR;
```

```
   extern bool debug_scanner;
   extern bool debug_parser;
   extern bool debug_codeGen;
   extern bool verbose;

   extern bool dump_abstractCode;
   extern bool dump_objectCode;
   extern bool dump_symbolTable;

   void turnVerboseOn();
   void turnVerboseOff();

}
```

The variable `debug_scanner` should be set to `false` for Part #1 of the project (users will have the option to set it to true on the command line). When `debug_scanner` is `false` (i.e. check for this), you should output nothing except possible warning or error messages (i.e. if all tokens are properly formed, no output will appear). When `debug_scanner` is `true` you should print out all of the token name *and* lexeme pairs as well as the number of correctly-formed tokens in the file (see sample run example). Note that when `verbose` is true *all* debugging messages are printed.

3.3.2 File `include/TClexer.cpp`

```
#ifndef TCLEXER_H
#define TCLEXER_H

#include "TCtoken.h"

namespace toyc {
  class TClexer {
  public:
  TClexer(std::string);
  TCtoken* getToken();

  std::string getLine();
  std::string getLexeme();
  int getLineNum();
  int getPos();
  };
}
#endif
```

This header will specify the function that returns a token when called. Though it's not required, source handling routines (i.e. the code to access the character stream from the source code file) can be defined in this class as well.

3.3.3 File include/TCtokens.cpp

```
#ifndef TCTOKENS_H
#define TCTOKENS_H

namespace toyc {

   enum tokens{ ... };

}

#endif
```

This class should define your enumerated token type which will be used throughout the project.

3.3.4 File include/TCtoken.cpp

```
#ifndef TCTOKEN_H
#define TCTOKEN_H

namespace toyc {
  class TCtoken{
  public:
    TCtoken();
    TCtoken(int);
    TCtoken(int,std::string);
    int getTokenType();
    std::string getLexeme();
    std::string toString();

  private:
    int tokenType;
    std::string lexeme;
  };

}
#endif
```

3.3.5 File src/tc.cpp

This will contain your `main` function i.e. it is the "driver" program for your compiler. For Part #1 of the project your driver will just repeatedly call your scanner tokenizing the input, i.e. (in pseudo-code):

```
main()
begin_main
  process the command line
```

```
  instantiate a scanner object
  while the next token != end_of_file do
    ; // nothing! the scanner reports tokens only if appropriate flag is set
end_main;
```

4 Example Run

Date file `tests/scanTest.tc`:

```
hello char int while <= !=
123 "hello" /*
* / && /* ||
> */ */ *
// this is a comment
```

Run of scanner:

```
liberty:~/compilers/toyc/> java -cp build/classes:../interfaces.jar parser.tc -debug 1 tests/scanTest.tc
[SCANNER] (<ID>,"hello")
[SCANNER] (<CHAR>,"char")
[SCANNER] (<INT>,"int")
[SCANNER] (<WHILE>,"while")
[SCANNER] (<RELOP>,"<=")
[SCANNER] (<RELOP>,"!=")
[SCANNER] (<NUMBER>,"123")
[SCANNER] (<STRING>,""hello"")
[SCANNER] (<MULOP>,"*")
[SCANNER] (<EOF>,"EOF")
[SCANNER] Total tokens: 10
liberty:~/compilers/toyc/>
```

Your token/lexeme pairs must have exactly the format shown above and use the token names given in the specification.

Note that for C++ you will issue the command (if you are in the `compilers/toyc` directory) '`bin/tc -debug 1 test/scantest.tc`'.