CS 8803-O08: Project Phase 1

Santosh Pande
santosh.pande@cc.gatech.edu

Georgia Institute of Technology — Due: Sep 19, 2022 11:59 PM AOE

Overview

The purpose of Phase 1 is to build the first stage of a compiler for the Tiger language. The first stage is comprised of a longest match scanner and an LL(*) parser. You will generate both using ANTLR4, a popular tool used commercially. ANTLR4 can generate scanners and parsers in C++ or Java directly from a lexical and grammatical specification. Because of its popularity, ANTLR4 has plenty of examples and documentation available online. After reviewing these, you will be ready to tackle the creation of the compiler front end!

Requirement 1

Use either C++ or Java for the implementation of your compiler.

In this phase, you will be writing the ANTLR4 lexical and grammatical specification for the Tiger language. Tiger is a small language with properties you should be familiar with: functions, arrays, integer and float types, and control flow. The syntax and semantics are described in the Tiger Language Specification. Tiger source programs should end with the suffix **.tiger**.

Requirement 2

Accept Tiger source code as an input file using the flag `-i <path/to/file.tiger>`.

1 Scanner

The first step is to build a scanner using ANTLR4. The scanner will scan the input file containing the Tiger program and convert the stream of characters into a stream of tokens. You can test the scanner by writing Tiger programs and checking the tokens generated.

1.1 Lexical Definition

As expressed in the language specification, Tiger contains keywords, identifiers, operators, punctuation, and literals. The keywords are recognized as a subset of the identifiers - first the scanner recognizes an

identifier (ID), and then it checks the ID against a list of keywords. If it matches, the scanner returns the corresponding keyword token and not an ID.

The scanner uses the longest match algorithm when recognizing tokens. It keeps matching the input character to the current token until encountering one which is not part of the current token. At this point, the token is completed using the last legal character and is returned to the parser. Next time around, the token generation restarts from the first character which was not part of the current token.

Requirement 3

Write a lexical specification for Tiger using ANTLR4's syntax. This specification must be included in the file named **Tiger.g4**.

1.2 Scanner Details

Reading the input file character by character, the scanner either returns the next matched tuple (`<token type, "token value">`) or outputs an error. Given the character stream **var x := 1+1**, the first request to the scanner for a matching token returns `<VAR, "var">`. The complete list is:

```
<VAR, "var">
<ID, "x">
<ASSIGN, ":=">
<INTLIT, "1">
<PLUS, "+">
<INTLIT, "1">
```

Warning: It is critical for grading that the token type strings match those provided in the language specification exactly. Comments and white-space should be discarded by the scanner.

Requirement 4

When the `-l` flag is provided, scan the input file and write the stream of tokens to a file. The output file should have the same name and path as the input file with the extension changed to `.tokens`. Output one tuple per line using the syntax `<token type, "token value">`.

2 Parser

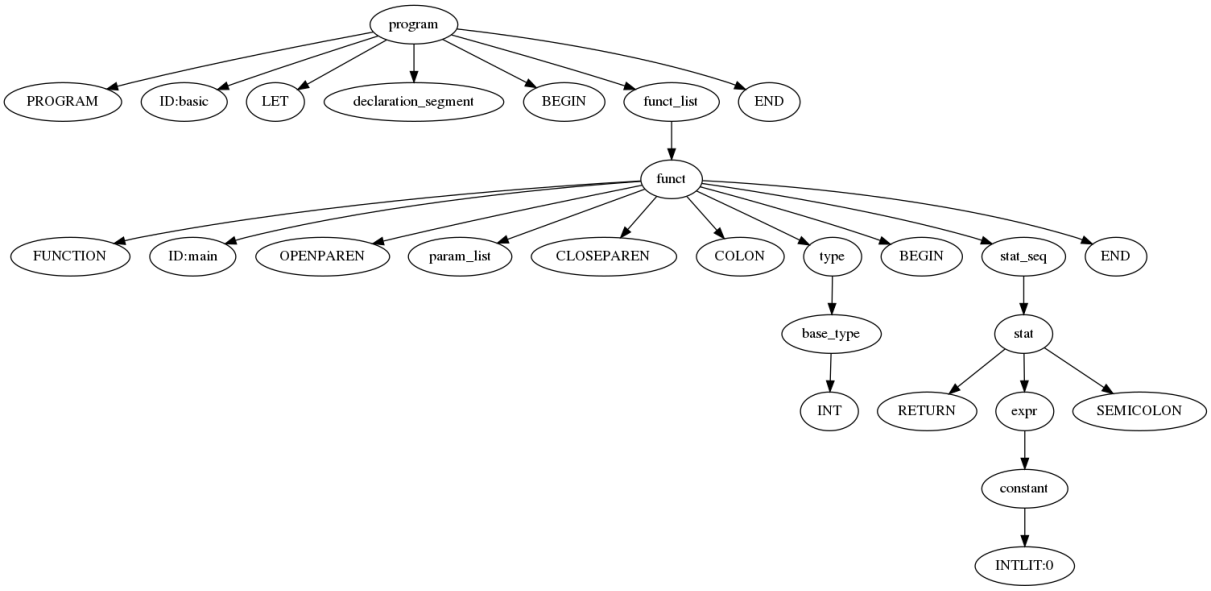
The second step is to build an LL(*) parser using ANTLR4. This requires converting the BNF grammar located in the Tiger Specification into a grammar that ANTLR4 understands. You may utilize ANTLR4's advanced syntax when creating your ruleset to aid in enforcing precedence and associativity rules. You can test your parser by writing Tiger programs and viewing the generated parse tree. ANTLR4 provides many plugins for popular IDEs which may prove extremely useful during development and testing.

Requirement 5

Write a grammar specification for Tiger using ANTLR4's syntax to generate a parser. This specification must be included, along with the lexer specification, in the file named **Tiger.g4**

Requirement 6

When the `-p` flag is provided, parse the input file and write the parse tree to a file in GraphViz DOT format. The output file should have the same name and path as the input file with the extension changed to `.tree.gv`. Further details of the format can be of your own design as long as it is valid GraphViz, and it displays the structure of the parse tree. An example is illustrated below.



Requirement 7

The data used to constuct the Graphviz output must be collected by walking the parse tree using either one of the mechanisms provided by Antlr, or one of your own design.

Requirement 8

The generated parse tree must visibly display the correct Tiger rules of association and precedence.

3 Error Handling

3.1 Lexical and Syntactic Error Messages

For lexical and syntactic errors, the ANTLR4 library will automatically print the errors to standard error by default. The scanner is capable of catching multiple errors in one pass. After catching the first error, it throws away the bad characters and restarts the token generation from the next character that starts a legal token. The auto-generated error messages are sufficient for Phase 1.

3.2 Error Return Codes

Your compiler is expected to be robust when provided with incorrect input or invalid arguments. It should exit gracefully and return the appropriate error code:

- 0: No errors encountered
- 1: Error in program arguments, e.g., file not found, unknown argument
- 2: Lexical Error Found, e.g., invalid token
- 3: Parse Error Found, e.g., invalid sentence

Requirement 9

Return the appropriate error code on exit. If there are multiple errors, return the lowest error code.

4 Grading

Your project should be submitted to Gradescope for grading. It is important that your project follows the correct build process and naming conventions outlined below.

4.1 Building

In order to facilitate the building and testing of your code, you must include a *Makefile* at the root of your source code directory. Your *Makefile* should generate the lexer and parser code, then compile and build a single output file under the `cs8803_bin` folder. If using Java, the output file should be a JAR file called `tigerc.jar`. If using C++, the output file should be an executable file called `tigerc`. You can assume that the following programs are available via the `PATH` variable:

- ANTLR 4.9.3
- Java 11.0.11
- g++ 7.5.0
- GNU Make 4.1
- cmake 3.10.2
- Apache Maven 3.6.0

Requirement 10

Include a *Makefile* which generates the lexer and parser and then compiles and builds your code.

Example *Makefiles* are included in the Appendix at the end of this document.

4.2 Testing

We will test your lexer and parser on multiple test inputs. Some of the test cases will be provided to you before the due date. For example, we will test your parser implementation against a test case as follows:

Command Line

```
$ pwd
/cs8803/phase1/astudent3
$
$ make
$ ./cs8803_bin/tigerc -i input1.tiger -l
...
```

4.2.1 Test Environment

The environment we use to test your code is based on vanilla Ubuntu 18.04 with ANTLR4 libraries installed. You can build a local test environment using Vagrant or Docker from the provided setup scripts. The test environment provides a convenient platform to verify your project builds and to directly run individual test cases while troubleshooting.

4.2.2 Gradescope

You can also test your compiler by submitting early to Gradescope. Gradescope will run **make** to generate the executable, and then run a series of automated tests. Note that additional points may be deducted for errors that are not caught by the automated testing. You can submit your compiler to Gradescope as many times as you would like prior to the deadline. It is highly recommended to **submit early and often**.

4.3 Deliverables

Your project should be submitted as a .zip file to Gradescope. The .zip file should contain:

- Makefile - Makefile to compile and build your code
- Tiger.g4 - ANTLR4 lexer/parser specification
- src - Directory containing your compiler source code (C++/Java)

4.4 Rubric

Warning: Your submission is expected to compile cleanly from source, **including ANTLR classes**. If a submission fails to compile, contains pre-built class files or binaries, or artificially bypasses the auto-grader, it can result in a **zero**.

- Identify invalid input and generate appropriate error code **20 points**
- Scanner test case output **12 points**
- Parser test case output with correct precedence and association **18 points**

5 Appendix: Makefile

Example Java and C++ *Makefiles* to build your project are provided below. These are povided for reference only and will need modifications to be used in your project.

5.1 Example Java *Makefile*

```
ANTLR := /usr/local/lib/antlr-4.9.3-complete.jar
BUILD_DIR := build
COMPILER_JAR := tigerc.jar
GRAMMAR := Tiger.g4
JAR_DIR := cs8803_bin
MAIN_CLASS_NAME := Main

ANTLR_JAVA_FILES := \
  src/TigerBaseListener.java \
  src/TigerBaseVisitor.java \
  src/TigerLexer.java \
  src/TigerListener.java \
  src/TigerParser.java \
  src/TigerVisitor.java

ANTLR_FILES := \
  src/Tiger.interp \
  src/Tiger.tokens \
  src/TigerLexer.interp \
  src/TigerLexer.tokens \
  $(ANTLR_JAVA_FILES)

ANTLR_LIBS := \
  $(BUILD_DIR)/javax \
  $(BUILD_DIR)/org

SOURCES := \
  src/Foo.java \
  src/Bar.java \
  src/Main.java

.PHONY:
all: $(COMPILER_JAR)

$(COMPILER_JAR): $(SOURCES) $(ANTLR_JAVA_FILES) $(ANTLR_LIBS)
  @mkdir -p $(BUILD_DIR) $(JAR_DIR)
  @javac -d $(BUILD_DIR) -cp "src:$(ANTLR)" $(SOURCES) \
  $(ANTLR_JAVA_FILES)
  @cd $(BUILD_DIR) && jar cfe ../$(JAR_DIR)/$(COMPILER_JAR) \
  $(MAIN_CLASS_NAME) *.class org javax && cd ..

$(ANTLR_JAVA_FILES): $(GRAMMAR)
  @java -jar $(ANTLR) -o src -visitor $(GRAMMAR)

$(ANTLR_LIBS):
  @mkdir -p $(BUILD_DIR)
  @cd $(BUILD_DIR) && jar xf $(ANTLR) && cd ..
  @rm -rf $(BUILD_DIR)/META-INF
```

```
.PHONY:
clean:
  @rm -f $(JAR_DIR)/$(COMPILER_JAR) $(ANTLR_FILES) \
  $(BUILD_DIR)/*.class
  @rm -rf $(ANTLR_LIBS)
```

5.2 Example C++ *Makefile*

```
BIN = cs8803_bin
RUNTIME = runtime
BUILD = build
SRC = src
ANTLR_RUNTIME = /usr/local/include/antlr4-runtime

INCLUDE = \
    -I$(RUNTIME) \
    -I$(SRC) \
    -I$(ANTLR_RUNTIME)

CXXFLAGS = -std=c++17 -g -Wno-attributes
LIBANTLR = /usr/local/lib/libantlr4-runtime.a

GENERATED_SOURCES = \
    $(RUNTIME)/TigerParser.cpp \
    $(RUNTIME)/TigerLexer.cpp \
    $(RUNTIME)/TigerBaseVisitor.cpp \
    $(RUNTIME)/TigerVisitor.cpp \
    $(RUNTIME)/TigerBaseListener.cpp \
    $(RUNTIME)/TigerListener.cpp

GENERATED_HEADERS = \
    $(RUNTIME)/TigerParser.h \
    $(RUNTIME)/TigerLexer.h \
    $(RUNTIME)/TigerBaseVisitor.h \
    $(RUNTIME)/TigerVisitor.h \
    $(RUNTIME)/TigerBaseListener.h \
    $(RUNTIME)/TigerListener.h

GENERATED_FILES = \
    $(GENERATED_SOURCES) \
    $(GENERATED_HEADERS) \
    $(RUNTIME)/Tiger.tokens \
    $(RUNTIME)/Tiger.interp \
    $(RUNTIME)/TigerLexer.tokens \
    $(RUNTIME)/TigerLexer.interp

GENERATED_OBJECTS = \
    $(BUILD)/TigerParser.o \
    $(BUILD)/TigerLexer.o \
    $(BUILD)/TigerVisitor.o \
    $(BUILD)/TigerBaseVisitor.o \
    $(BUILD)/TigerBaseListener.o \
    $(BUILD)/TigerListener.o

HEADERS = \
    $(SRC)/YourFilesHere.h
```

```
SOURCES = \
    $(SRC)/YourFilesHere.cpp

OBJECTS = \
    $(BUILD)/YourFilesHere.o

MAIN_OBJECT= \
    $(BUILD)/tigerc.o

all: dirs $(BIN)/tigerc

$(BIN)/tigerc: $(MAIN_OBJECT) $(OBJECTS) $(GENERATED_OBJECTS)
    g++ -o $@ $(MAIN_OBJECT) $(OBJECTS) $(GENERATED_OBJECTS) $(LIBANTLR)

$(MAIN_OBJECT): $(BUILD)/%.o: $(SRC)/%.cpp $(GENERATED_HEADERS) $(HEADERS)
    $(CXX) -c $(CXXFLAGS) $(INCLUDE) $< -o $@

$(OBJECTS): $(BUILD)/%.o: $(SRC)/%.cpp $(GENERATED_HEADERS) $(HEADERS)
    $(CXX) -c $(CXXFLAGS) $(INCLUDE) $< -o $@

$(GENERATED_FILES): Tiger.g4
    antlr -o $(RUNTIME) -Dlanguage=Cpp -visitor -Xexact-output-dir $<

$(GENERATED_OBJECTS): $(BUILD)/%.o: $(RUNTIME)/%.cpp
    $(CXX) -c $(CXXFLAGS) $(INCLUDE) $< -o $@

dirs: cs8803_bin build runtime

cs8803_bin:
    mkdir -p $(BIN)
build:
    mkdir -p $(BUILD)
runtime:
    mkdir -p $(RUNTIME)

.PHONY:
clean:
    @rm -f $(MAIN_OBJECT) $(OBJECTS) $(GENERATED_FILES)
    @rm -f cs8803_bin/tigerc
```