CS 4240: Compilers and Interpreters, Project 3, Spring 2022

Assigned: March 28, 2022

Due in Canvas by 11:59pm on April 20, 2022

1 Project Overview

In this project, you will build a **lexer** and a **parser** for the ***Tiger*** language.

Tiger is a small language with properties that you are familiar with, including functions, arrays, integer and float types, and control flow. Syntactic & lexical specifications of the ***Tiger*** language are available in Appendix A.

You will be using the ANTLR v4 tool to generate a lexer and a parser for the *Tiger* language. You will write a program that invokes the generated lexer and parser.

ANTLR v4 is a production tool used to generate lexers and parsers based on the input lexical and syntactic specifications. We recommend that you start by studying ANTLR using examples and documentation that are provided with it. After that, you will be ready to build the lexer and parser for the *Tiger* language. Download the latest version of ANTLR (v4.9.3) from `https://www.antlr.org/download.html`.

A complete compiler front-end also includes symbol table creation, semantic analysis, and intermediate representation (IR) code generation, but you will **not** implement those components in this project.

2 Lexer (Scanner)

The lexer implements the lexical specification of *Tiger*: It reads in the program's stream of characters (one character at a time) and returns the correct token on each request from the parser.

You will write the lexical specification of *Tiger* (Appendix A.2) in a form acceptable to ANTLR v4 and generate the lexer program in any of the following languages (Java, C++, C#, Python (2 & 3), JavaScript, Go, Swift) as per your choice of implementation.

You will then test it for errors and correct production of stream of tokens. As far as the errors are concerned, you can just produce the errors generated by the ANTLR generated lexer.

For lexically malformed *Tiger* programs, your lexer will throw an error which prints: line number in the file, the partial prefix of the erroneous string (from the input file) and the malformed token with the culprit character that caused the error. The lexer is capable of catching multiple errors in one pass, i.e., it does not quit but continue on after catching the 1st error. It will throw away the bad characters and restart token generation from the next character that starts a legal token in *Tiger*.

Notes:

- The keywords are recognized as a subset of the identifiers – that is, first the lexer recognizes an identifier (ID) and then checks the string against a list of keywords, if it matches you return corresponding keyword token and not an ID.
- The lexer uses the **longest match algorithm** when recognizing tokens. That is, the lexer keeps matching the input character to the current token, until you encounter one which is not a 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 the part of the last token.

2.1 Parser

The parser implements the syntactic specification of *Tiger*. Rewrite the grammar provided in Appendix A.1 and use the new grammar as an input to ANTLR, in order to generate your parser that invokes your lexer to generate tokens. This part of the project consists of three parts:

1. **Rewrite the grammar to remove any ambiguity** by enforcing operator precedences and left/right associativity for different operators. **A grammar is ambiguous if it can produce 2 or more distinct parse trees for the same input sentence.** You can either manually modify the grammar by hand or write code that manipulates the grammar.
2. **Modify the grammar obtained in step 1 to make it LL(1).** You are only allowed to modify the grammar by removing left recursion or performing left factoring. Follow the instructions in `https://github.gatech.edu/CS-4240-Spring-2022/Project-3-Antlr/tree/master/how-to-verify-that-your-grammar-is-ll1.md` on how to use ANTLR to verify that your grammar is LL(1). You can either manually modify the grammar by hand or write code that manipulates the grammar.
3. Input your grammar to ANTLR. ANTLR will generate the parser in the language of your choice (which needs to be supported by ANTLR). Then test the parser by feeding it *Tiger* programs which are used as test cases. Iterate and revise the grammar repeating steps 1 and 2 until you get it right. The generated parser when invoked on an input *Tiger* program will generate a parse tree which can be visualized with a suitable IDE plug-in (`http://www.antlr.org/tools.html`).

For syntactically correct *Tiger* programs, your program (which invokes the parser) should print "successful parse" to stdout. For programs with lexical problems, the lexer is already responsible for throwing an error. **For programs with syntactic problems, the parser is responsible**

for raising its own errors. In these cases, the output should be a reasonable message about the error including: the input file line number where it occurred, a partial sentence which is a prefix of the error, the erroneous token, and what the parser was expecting there. Use the error recovery mechanisms provided by ANTLR to print out error messages.

3 Grading Criteria for Base Project

In this section, we summarize the grading criteria for the project (100 points).

3.1 Correct Implementation (75 Points)

The modified *Tiger* grammar (as .g4 file) is worth 35 points.

- Your grammar is not ambiguous - (15 points)
- Your grammar is LL(1) - (20 points)

Correct execution of your lexer/parser is worth 40 points.

- Lexer prints reasonable error message for malformed *Tiger* programs with lexical errors
- Parser prints reasonable error message for malformed *Tiger* programs with syntactic errors
- Lexer/Parser doesn't report errors when provided with legal *Tiger* programs

Note that you need to provide a program that invokes the generated lexer & parser. You will also need to provide a `build.sh` (if building is necessary) and `run.sh`. We will test your code by running `./run.sh test_file`. Please make sure you follow these execution instructions.

You will get 0% of the score for a test case if

- Your program does not compile or build.
- The `run.sh` does not run with the test input, e.g. it crashes or has some bug etc.

Note that all test cases (*Tiger* programs) for this project will be public at https://github.gatech.edu/CS-4240-Spring-2022/Project-3-Antlr/tree/master/test_cases.

3.2 Design Report (25 Points)

In your design report, briefly describe the following:

1. High-level architecture of your front-end, including the algorithm(s) implemented, and why you chose that approach
2. Summary of your implementation strategy (including how you implemented extensions to the code that was provided, if applicable)
3. Software engineering challenges and issues that arose and how you resolved them

4. Any known outstanding bugs or deficiencies that you were unable to resolve before the project submission
5. Build and usage instructions for your front-end

4 Submission

On Canvas, submit a single ZIP file that contains:

- The complete source code of your project. Please make sure to include your `build.sh` (if building is necessary) and `run.sh`.
- The report.pdf file.
- (Optional) Any new test cases that you developed for this project.

5 Collaboration

We will award identical grades to each member of a given project team, unless members of the team directly register a formal complaint. We assume that the work submitted by each team is their work solely. Any clarification question about the project handout should be posted on the course's public Piazza message board. Any non-obvious discussion or questions about design and implementation should be either posted on the course's Piazza message boards privately for the instructors or presented in person during office hours. If the instructors determine that parts of the discussion are appropriate for the entire class, then they will forward selections. Under no condition is it acceptable to use code written by another team, or obtained from any other source. As part of the standard grading process, each submitted solution will automatically be checked for similarity with other submitted solutions and with other known implementations.

Appendix A *Tiger* Language Reference Manual

A.1 Grammar (syntactic rules)

⟨*tiger-program*⟩	::=	'`main`' '`let`' ⟨*declaration-segment*⟩ '`in`' '`begin`' ⟨*stat-seq*⟩ '`end`'
⟨*declaration-segment*⟩	::=	⟨*var-declaration-list*⟩ ⟨*funct-declaration-list*⟩
⟨*var-declaration-list*⟩	::=	⟨*empty*⟩
⟨*var-declaration-list*⟩	::=	⟨*var-declaration*⟩ ⟨*var-declaration-list*⟩
⟨*var-declaration*⟩	::=	'`var`' ⟨*id-list*⟩ '`:`' ⟨*type*⟩ ⟨*optional-init*⟩ '`;`'
⟨*funct-declaration-list*⟩	::=	⟨*empty*⟩
⟨*funct-declaration-list*⟩	::=	⟨*funct-declaration*⟩ ⟨*funct-declaration-list*⟩
⟨*funct-declaration*⟩	::=	'`function`' `ID` '`(`' ⟨*param-list*⟩ '`)`' ⟨*ret-type*⟩ '`begin`' ⟨*stat-seq*⟩ '`end`'

⟨*type*⟩	::=	⟨*type-id*⟩
⟨*type*⟩	::=	‘`array`’ ‘`[`’ `INTLIT` ‘`]`’ ‘`of`’ ⟨*type-id*⟩
⟨*type-id*⟩	::=	‘`int`’ | ‘`float`’
⟨*id-list*⟩	::=	`ID`
⟨*id-list*⟩	::=	`ID` ‘`,`’ ⟨*id-list*⟩
⟨*optional-init*⟩	::=	⟨*empty*⟩
⟨*optional-init*⟩	::=	‘`:=`’ ⟨*const*⟩
⟨*param-list*⟩	::=	⟨*empty*⟩
⟨*param-list*⟩	::=	⟨*param*⟩ ⟨*param-list-tail*⟩
⟨*param-list-tail*⟩	::=	⟨*empty*⟩
⟨*param-list-tail*⟩	::=	‘`,`’ ⟨*param*⟩ ⟨*param-list-tail*⟩
⟨*ret-type*⟩	::=	⟨*empty*⟩
⟨*ret-type*⟩	::=	‘`:`’ ⟨*type*⟩
⟨*param*⟩	::=	`ID` ‘`:`’ ⟨*type*⟩
⟨*stat-seq*⟩	::=	⟨*stat*⟩
⟨*stat-seq*⟩	::=	⟨*stat*⟩ ⟨*stat-seq*⟩
⟨*stat*⟩	::=	⟨*lvalue*⟩ ‘`:=`’ ⟨*expr*⟩ ‘`;`’
⟨*stat*⟩	::=	‘`if`’ ⟨*expr*⟩ ‘`then`’ ⟨*stat-seq*⟩ ‘`endif`’ ‘`;`’
⟨*stat*⟩	::=	‘`if`’ ⟨*expr*⟩ ‘`then`’ ⟨*stat-seq*⟩ ‘`else`’ ⟨*stat-seq*⟩ ‘`endif`’ ‘`;`’
⟨*stat*⟩	::=	‘`while`’ ⟨*expr*⟩ ‘`do`’ ⟨*stat-seq*⟩ ‘`enddo`’ ‘`;`’
⟨*stat*⟩	::=	‘`for`’ `ID` ‘`:=`’ ⟨*expr*⟩ ‘`to`’ ⟨*expr*⟩ ‘`do`’ ⟨*stat-seq*⟩ ‘`enddo`’ ‘`;`’
⟨*stat*⟩	::=	⟨*opt-prefix*⟩ `ID` ‘`(`’ ⟨*expr-list*⟩ ‘`)`’ ‘`;`’
⟨*opt-prefix*⟩	::=	⟨*lvalue*⟩ ‘`:=`’
⟨*opt-prefix*⟩	::=	⟨*empty*⟩
⟨*stat*⟩	::=	‘`break`’ ‘`;`’
⟨*stat*⟩	::=	‘`return`’ ⟨*expr*⟩ ‘`;`’
⟨*stat*⟩	::=	‘`let`’ ⟨*declaration-segment*⟩ ‘`in`’ ⟨*stat-seq*⟩ ‘`end`’
⟨*expr*⟩	::=	⟨*const*⟩
	|	⟨*lvalue*⟩
	|	⟨*expr*⟩ ⟨*binary-operator*⟩ ⟨*expr*⟩
	|	‘`(`’ ⟨*expr*⟩ ‘`)`’

⟨*const*⟩ ::= `INTLIT`

⟨*const*⟩ ::= `FLOATLIT`

⟨*binary-operator*⟩ ::= ‘`+`’ | ‘`-`’ | ‘`*`’ | ‘`/`’ | ‘`=`’ | ‘`<>`’ | ‘`<`’ | ‘`>`’ | ‘`<=`’ | ‘`>=`’ | ‘`&`’ | ‘`|`’

⟨*expr-list*⟩ ::= ⟨*empty*⟩

⟨*expr-list*⟩ ::= ⟨*expr*⟩ ⟨*expr-list-tail*⟩

⟨*expr-list-tail*⟩ ::= ‘`,`’ ⟨*expr*⟩ ⟨*expr-list-tail*⟩

⟨*expr-list-tail*⟩ ::= ⟨*empty*⟩

⟨*lvalue*⟩ ::= `ID` ⟨*lvalue-tail*⟩

⟨*lvalue-tail*⟩ ::= ‘`[`’ ⟨*expr*⟩ ‘`]`’

⟨*lvalue-tail*⟩ ::= ⟨*empty*⟩

A.1.1 Precedence (Highest to Lowest)

```
( )  *  /  +  -  =  <>  > <  >=  <=  &  |
```

A.1.2 Associativity

Binary operators are **right associative**.

A.2 Lexical Rules

A.2.1 Case Sensitivity

Tiger is a case-sensitive language.

A.2.2 Identifier (ID)

An identifier is a sequence of one or more letters, digits, and underscores. It must start with a letter, followed by zero or more of letter, digit or underscore.

A.2.3 Comment

A comment begins with “/*” and ends with “*/”. Nesting is not allowed.

A.2.4 Integer Literal (INTLIT)

An integer literal is a non-empty sequence of digits.

A.2.5 Float Literal (FLOATLIT)

A float literal must consist of a non-empty sequence of digits, a radix (i.e., a decimal point), and a (possibly empty) sequence of digits.

A.2.6 Reserved (Key)words

main	array	break	do	if	else	for
function	let	in	of	then	to	var
while	endif	begin	end	enddo	return	
int	float					

A.2.7 Punctuation Symbols

`,   :   ;   (   )   [   ]`

A.2.8 Binary Operators

`+   -   *   /   =   <>   <   >   <=   >=   &   |`

A.2.9 Assignment operator

`:=`