CS 4240: Compilers and Interpreters, Project 3, Fall 2025

ANTLR Version

Assigned: October 29, 2025, 5% of course grade

Due in Gradescope by 11:59pm on November 24, 2025

1 Overview

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

Tiger is a small language with properties you are familiar with, including functions, arrays, integer and float types, and control flow. Syntactic and 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 input lexical and syntactic specifications. We recommend that you start by studying ANTLR using the 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 (v4.13.2) from `https://www.antlr.org/download.html`.

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

2 Project Steps

The project repository is available at

`https://github.gatech.edu/CS-4240-Fall-2025/Project-3-Antlr`.

The `demo` directory in this repository contains a detailed readme file which additional guidance on building and using ANTLR.

2.1 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 and generate the lexer program in one of the following languages: Java, C++, C#, Python.

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 the line number in the file, the partial prefix of the erroneous string, 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 continues on after catching the first error. It will throw away the bad characters and restart token generation from the next character that starts a legal token in *Tiger*.

Some notes on the lexer:

- 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, it returns the 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 characters to the current token until it encounters 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.

2.2 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 a parser that invokes your lexer to generate tokens. This part of the project consists of three parts.

1. **Remove ambiguity.** You must 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 two 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 to achieve this. If you write code to do so, you must submit it with your project.
2. **Make the grammar LL(1).** You must modify the grammar to ensure that it is LL(1). To this end, you are only allowed to modify the grammar by removing left recursion or performing left factoring. Again, you may manually modify the grammar by hand or write code that manipulates the grammar. In the latter case, you must submit the code you wrote.
3. **Generate a parser.** Input your grammar to ANTLR and generate a parser in the language of your choice (among Java, C++, C#, and Python). 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` to stdout. For programs with lexical or syntactic errors, ANTLR's lexer and parser will automatically detect and report them through their error listeners. You should provide custom error listeners that tag these messages appropriately:

- Lexical errors will be reported as `[LexicalError] line <row>:<col> <msg>`
- Syntactic errors will be reported as `[SyntacticError] line <row>:<col> <msg>`

Use ANTLR's built-in error recovery mechanisms to print these messages and continue parsing gracefully when possible.

3 Grading

The total possible number of points for Project 3 is 100. Points are awarded according to the following rubric.

You must submit a single ZIP file that contains:

- A grammar file named "tiger.g4" in the root directory.
- A directory named "antlr_generated" which contains the ANTLR–generated lexer and parser for your grammar. It's fine if you leave this directory empty and regenerate its contents each time "run.sh" is run, but your report should indicate if this is the case.
- A file named "run.sh" in the root directory which runs your parser.
- Any other source code for your project, including any code you wrote to manipulate the grammar.
- Any new test cases you developed for the project in a directory named "new_test_cases".
- A document named "report.pdf".

3.1 Correct grammar (35 points)

You must submit a modified *Tiger* grammar in the .g4 format. This grammar must be in a file named "tiger.g4" in the root directory of your submitted zip file. Points are allocated as follows.

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

3.2 Correct lexer/parser (40 points)

You must submit the ANTLR–generated lexer and parser for your project in a directory named "antlr_generated" in the root of your ZIP file. You may leave this directory empty and regenerate its contents with ANTLR as part of your "run.sh" script—if you choose to do this, please document it in your report. If you do submit a parser the "antlr_generated" directory, please include instructions on how to use ANTLR to regenerate your parser in your submitted `report.pdf`. In this case, **the parser you submit must be replicable.**

In addition, include a script named "run.sh" in the root directory of your submission which runs your parser. It should take a single argument, a *Tiger* program and output to stdout either `successful` in the case that parsing was successful, or an appropriate error as described in Section 2 if it was not. For example,

```
$ run.sh path/to/correct_case.tiger
successful
```

This will be graded as follows.

- Lexer prints reasonable error message for malformed *Tiger* programs with lexical errors (10 points)
- Parser prints reasonable error message for malformed *Tiger* programs with syntactic errors (15 points)
- Lexer/Parser doesn't report errors when provided with legal *Tiger* programs (15 points)

Unlike projects 1 and 2, all of the test cases (*Tiger* programs) for this project are public, available in the `test_cases` directory of the project repository.

Note that in some cases, a lexical error may also trigger subsequent syntactic errors during error handling and recovery. Such behavior is possible and acceptable, but you are responsible for verifying that your reported errors are reasonable and consistent with your grammar.

3.3 Design report (25 Points)

Your submission should include a document named "report.pdf" which briefly describes the following:

1. A summary of your implementation strategy, including how you modified the *Tiger* grammar to make it unambiguous and LL(1), e.g. how you identified and resolved ambiguities
2. Which language you chose for your parser, and either an indication that your "run.sh" script recreates the contents of the "antlr_generated" directory or instructions on how to regenerate the contents of that directory using ANTLR
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

4 Collaboration

The collaboration policy and expectations are the same as in project 1.

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

`:=`