CS 4240: Compilers and Interpreters, Project 3, Spring 2019

Assigned: April 8, 2019, 10% of course grade

(100 points total, with an extra credit option for 15 points)

Due in Canvas by 11:59pm on April 23, 2019

(Automatic penalty-free extension till 11:59pm on April 30, 2019)

1 Project Overview

In this project, you will build a **scanner** and a **parser** for the *Tiger* language. In practice, a complete compiler front-end will also include symbol table creation, semantic analysis, and intermediate representation (IR) code generation, but these are not required given the limited time available for this project (though some of these components can be implemented for extra credit).

2 Scanner and Parser

The goal of this project is to build a scanner and parser for the *Tiger* language that uses a longest match scanner and LL parser; both will be generated using the ANTLR v4 (`https://github.com/antlr/antlr4/blob/master/doc/getting-started.md`). ANTLR v4 is a production tool used to generate lexer & parser in C++, Java, Python, Go, etc., as output, using the input lexical and grammatical 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 scanner and parser for the *Tiger* language.

Please refer to the language specifications of the *Tiger* language given in Appendix A. *Tiger* is a small language with properties that you are familiar with, including functions, arrays, integer and float types, and control flow.

First, you need to build a scanner that will scan the input file containing a *Tiger* program, perform lexical actions, and return tokens one by one on demand to the parser. You will first test it by writing *Tiger* programs as per the grammar, invoking the scanner through the parser and checking the stream of the tokens generated. The scanner will be built using ANTLR.

2.1 Scanner

The scanner reads in one character at a time from the input file and performs token generation per longest match algorithm. Thus, it is able to read in a stream of characters and, return the next matched token or an error.

The scanner implements the lexical specification of *Tiger*: It reads in the program's stream of characters and return the correct token on each request. For lexically malformed *Tiger* programs, the scanner throws 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 scanner 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 the token generation from the next one that starts a legal token in *Tiger*.

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.

Notes:

- The keywords are recognized as a subset of the identifiers – that is, first the scanner 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 scanner uses the longest match algorithm when recognizing tokens. That is, the scanner 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.

2.2 Parser

First, you will be writing a parser for *Tiger* that calls the scanner above which supplies it the tokens. This parser too will be automatically built by using ANTLR. You will rewrite the grammar provided in Appendix A.1 and use the new grammar as an input to ANTLR. This consists of three parts:

1. Rewrite the grammar to remove the ambiguity by enforcing operator precedences and left and right associativity for different operators. This part is to be done by hand.
2. Modify the grammar obtained in step 1 to support LL(k) parsing, where k must be minimized, so that the parser is LL(1). This could include removing left recursion and performing left factoring on the grammar obtained in step 1 above. You are not allowed to rewrite the grammar except using these two techniques. This part is to be done by hand.

3. Once you have the grammar in the correct form, you will input it to ANTLR. ANTLR will generate the parser in the language of your choice (it should be supported by ANTLR, though). You will then test the parser by feeding it *Tiger* programs which are used as test cases. You can 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 potentially visualized with a suitable IDE plug-in (`http://www.antlr.org/tools.html`).

For syntactically correct *Tiger* programs, the parser should output "successful parse" to stdout. For programs with lexical issues, the scanner is already responsible for throwing an error. For programs with syntactic problems, however, the parser is responsible for raising its own errors. In these cases, the output should be some 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 perhaps what the parser was expecting there.

You will just use the error recovery mechanisms provided by ANTLR and test and report their outcomes on different input test cases. More advanced error recovery mechanisms than the above are not expected to be built and are out of scope for this project.

3 Grading Criteria for Base Project

In this section, we summarize the grading criteria for the base project (100 points). Appendix B summarizes the opportunities to implement an extra credit option for 15 points.

3.1 Correct Implementation (75 Points)

The hand-modified *Tiger* grammar is worth 25 points.

You will be provided with some test programs that contain lexical errors (10 points), syntactic errors (20 points). To get the points for those test programs, your front-end should report the errors with reasonable error messages, when errors are encountered.

You will also be provided with some legal test programs along with input/output pairs (20 points). To get those points, your front-end should report no error for these programs.

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. Software engineering challenges and issues that arose and how you resolved them

3. Any known outstanding bugs or deficiencies that you were unable to resolve before the project submission
4. 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.
- 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

⟨*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 left 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	

A.2.7 Punctuation Symbols

`, : ; ( ) [ ]`

A.2.8 Binary Operators

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

A.2.9 Assignment operator

`:=`

A.3 Semantics

A.3.1 Operators

Binary operators take scalar (integer or float) operands and return a scalar value.

Comparison operators compare their operands, which may be either both integer or both float, and produce the integer value 1 if the comparison holds (indicating a true) and integer 0 otherwise (indicating a false).

Arithmetic operators (+, -, *, /) require two operands of the same type and return a result of this type.

The operators & and | are bit-wise "and" and "or" operators. They take integer operands and produce an integer result.

For an integer, zero is considered false; non-zero values are considered true.

A.3.2 Types

Two named-types `int` and `float` are predefined.

We also have static array types in Tiger. An array type of `int` or `float` may be made by `array [intlit] of <type-id>` of length `intlit`.

A.3.3 Assignment

The assignment expression `lvalue := expr` evaluates the expression then binds its value to the contents of the lvalue.

A.3.4 Array Initialization

The optional initialization of an array will fill the entire array with the same value.

A.3.5 Control Flow

The `if-then-else` expression, written as `if expr then <stat-seq> else <stat-seq> endif`, evaluates the first expression, which must return an integer. If the result is non-zero, the statements under the `then` clause are evaluated, otherwise the third part under the `else` clause is evaluated.

The `if-then` expression, `if expr then <stat-seq> endif` evaluates its first expression, which must return an integer. If the result is non-zero, it evaluates the statements under `then` clause.

The `while-do` expression, `while expr do <stat-seq>` evaluates its first expression, which must return an integer. If it is non-zero, the body of the loop is evaluated, and the first expression is evaluated again.

The `for` expression, `for ID := expr to expr do <stat-seq>` evaluates the first and second expressions, which are loop bounds and must be integers. Then, for each integer value between the values of these two expressions (inclusive), the third part `<stat-seq>` is evaluated with the integer variable named by `ID` bound to the loop index. This part is not executed if the loop's upper bound is less than the lower bound.

A.3.6 Scoping Rules

The scopes follow the classic block structure.

- A declaration of a variable with a given name in a given scope hides its declaration from outer scopes (if any).
- When a scope is closed, all entities declared in that scope are automatically destroyed. That is, the the lifetime of entities is limited to the scope in which they are declared and when the scope is closed, the entities declared in that scope cease to exist.
- Lookup rules: The binding of a name is decided in the following manner: first the name is looked up in the current scope, if the lookup finds the declared name in the current scope, it is bound to the corresponding entity. If not found, the lookup proceeds to the outer scopes one by one until the name is found. If a declaration of the name is not found anywhere, it means the entity is undeclared and constitutes a semantic error. Once a name is found in a given scope, the lookup stops and does not proceed to outer scopes.

A.3.7 Variables

A variable declaration declares a new variable and its initial value (optional). The lifetime of a variable is limited to its scope. Variables and functions share the same name space. Redeclaration of the same name in the same scope is illegal.

A.3.8 Functions

The first form is a "procedure" declaration (no return type); the second is a "function" (with return type). Functions return a value of the specified type; procedures are only called for their side-effects. Both forms allow the specification of a list of zero or more typed arguments. Scalar arguments are passed by value and array arguments are passed by reference.

A.3.9 Standard Library

Signature	Semantics
`function readi():int`	`Read an integer from the standard input.`
`function readf():float`	`Read a float from the standard input.`
`function readchar():int`	`Read a character as its ASCII value from the standard input.`
`function printi(i:int)`	`Print the integer i to the standard output.`
`function printf(f:float)`	`Print the float f to the standard output.`
`function printc(c:int)`	`Print the character encoded as the ASCII value c to the standard output.`

Appendix B Extra Credit Options: Semantic Analysis, IR Code Generation

For extra credit (15 points), you can implement one of the following combinations:

- Symbol Table + Semantic Analysis, so that your extended front-end can report semantic errors in addition to lexical and syntax errors. You don't need to generate IR code for this combination.
- Symbol Table + IR Code generation, so that your extended front-end can generate IR code that can be input to your Project 1 and Project 2 implementations. You don't need to check for semantic errors for this combination. (For this combination, just assume that your front-end will always be tested with programs that have no semantic errors, though they may have lexical and syntax errors.)

These combinations can be implemented be using the ParseTree Walking mechanisms provided by the ANTLR (`https://github.com/antlr/antlr4/blob/4.7.1/doc/listeners.md`).

For convenience, we describe all three components needed for a complete front-end below, but keep in mind that you only need to implement the Symbol Table with one of the other components to obtain one of the extra credit options listed above. These descriptions will also be useful for any students who may, out of interest, want to try and integrate all tehir projects into a single compiler after the end of the semester.

B.1 Symbol Table

The symbol table is a useful structure generated by the compiler. The semantic checking will make extensive use of it. The symbol table holds declarative information about variables, constants, user defined types, functions and their expected parameters and so on that make up a program. The following are the typical entries (not exhaustive) for a symbol table:

- variable names
- defined constants
- procedure and function names and their parameters
- literal constants and strings
- source text labels
- compiler-generated temporaries

Attributes for each of the above entries will usually be stored in the symbol table. Typical attributes include:

- textual name
- data type

- array bounds
- declaring function
- lexical level of declaration (scoping)
- if parameter, by-reference or by-value?
- number and type of arguments to functions

You have some flexibility in deciding these implementation details, and the above lists are by no means requirements. The guiding principle you should use for building your symbol table and the information you keep in it should be that of utility: store what you will need for the next parts.

Scoping is a key aspect of symbol tables. By this we mean the following two things:

1. The most closely nested rule (i.e. references always apply to the most closely nested block) – please implement the scoping rules described in the language definition.
2. Declaration of a variable before use.

Thus, insertion into your symbol table cannot overwrite previous declarations but instead must mask them. Subsequent lookup operations that you run against the symbol should return the most recently inserted variable definition. This handles rule 1. For rule 2, a lookup operation should never fail. If it does, it means the symbol table has not yet seen a declaration for the reference you are attempting to check and it is a semantic error to use an undeclared value. The scoping rule we are going to use is the same as used in all block structured languages: inner declaration of a variable name hides the outer declaration; in a given scope, the innermost declaration is the one that is visible and a name used in that scope is bound to that declaration.

As you consider scoping, you are free to implement your symbol table on a scope by scope basis or as a global symbol table in which declarations are entered upon entering a scope and deleted upon exiting it.

Here are the suggested steps:

1. Carefully read the grammatical and semantic specification of *Tiger* and decide which values and which of their attributes are going to be held in symbol table.
2. Design a symbol table with hash maps, chains, etc.
3. Implement the symbol table generation.

For the above purposes you will implement the listener and visitor objects that interface to the parse tree (`https://github.com/antlr/antlr4/blob/master/doc/listeners.md`) and write suitable tree walking code to gather and then store the relevant information into the symbol table as discussed above.

B.2 Semantic Checking

This phase consists of semantic checks. It implements semantic checks by walking the Parse Tree using listener and visitor objects. You will develop the necessary walking and checking code here. You should first read the semantic specification of *Tiger* and then implement the checking as follows.

Typically, the first step in semantic checking is the binding analysis. It determines which variable name binds to which symbol table entry. This is done using look-up mechanism as per the block structure rule. The first step is to decorate the respective nodes of the parse tree with attributes elicited from the symbol table. The usage of a variable must occur as per its declaration. Finally, the type checking is done. Some checks might involve making sure the type and number of parameters of a function match actual arguments. There are several cases in *Tiger* where type checking must occur:

- Agreement between binary operands
- Agreement between function return values and the function's return type
- Agreement between function calls and the function's parameters

Refer to Appendix A.3 for the rules about type semantics.

For correct programs, your front-end should pass this part silently and emit nothing. For programs with semantic problems, your front-end should emit an error message stating the problem and relative place in the source.

B.3 Intermediate Code Generation

The final part of this phase is to convert the program into intermediate representation (*Tiger-IR*). You will need to implement necessary tree walks to generate IR code including labels and temporaries. For each correct program fed into your front-end, you should print the corresponding IR code to an output file. Don't print any IR code for programs with errors.