Project Phase 1A: Front end parser and lexer

Total points: 50

Due Date: 09/14/2020, 8 am (via Canvas)

Table of Contents

Overview

The purpose of Project 1A is to build a lexer and parser for the Tiger language that uses a longest match lexer and an LL parser. You will generate both using ANTLR4. ANTLR v4 is a production tool used by professionals and it can generate lexer and parser in C++ or Java as output using the input lexical and grammatical specifications. First you will study the ANTLR using lots of examples and documentation that are provided with it. After studying a few examples and reading the documentation, you will be ready to tackle building the compiler front end using ANTLR.

Please refer to the language specification for Tiger. It is a small language with properties that you are familiar with: functions, arrays, records, integer and float types, and control flow. The syntax and semantics of Tiger constructs are described in detail in the language specification. The specification document includes sample Tiger programs.

Scanner

First, you need to build a scanner using ANTLR that will scan the input file containing the Tiger program and 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, invoke the scanner through the parser and check the stream of the tokens generated.

Lexical definition

Tiger has three token types – punctuation; keywords; and identifiers, int, and float literals.

Punctuation

ASSIGN	COMMA	COLON	SEMI	LPAREN	RPAREN
LBRACK	RBRACK	LBRACE	RBRACE	PERIOD	PLUS
MINUS	MULT	DIV	EXP	EQ	NEQ
LESSER	GREATER	LESSEREQ	GREATEREQ	AND	OR
TEQ					

Keywords

MAIN	ARRAY	RETURN	BREAK	DO	ELSE
END	FOR	FUNCTION	IF	IN	LET
OF	THEN	TO	TYPE	VAR	WHILE
ENDIF	BEGIN	END	ENDDO		

Identifiers, integer and float literals

ID	INTLIT	FLOATLIT

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 the 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 a part of the current token. Lesson 1 slides demonstrate the working of this algorithm on “main”.

Scanner details

The scanner reads in char by char from the input file and performs token generation per longest match algorithm. Thus, it is able to read in a stream of characters and either returns the next matched `<token type, token>` tuple or throws an error. For example, given this 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”>
```

The scanner implements the lexical specification of Tiger. It reads in the program’s stream of characters and returns the correct token tuple on each request. For lexically malformed Tiger programs, the scanner throws an error which prints the line number in the file, the partial prefix of the erroneous string (from the input file), and the malformed token (putting it in quotes pin-pointing the culprit character that caused the error). The scanner is capable of catching multiple errors in one pass. That is, it does not quit but continues on after catching the first error. It throws away the bad characters and restarts the token generation from the next one that starts a legal token in Tiger.

Requirements

You will write the lexical specification of Tiger in a form acceptable to ANTLR v4 and generate the lexer program in C++ or Java as per your choice of implementation. You will then test it for both 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.

Parser

You will write a parser for Tiger that calls the scanner above which supplies it the tokens. This parser is automatically built by using ANTLR. For this purpose, you will be rewriting the grammar that goes as an input to the ANTLR. This consists of three parts:

- Rewrite the grammar given in the Tiger language specification below to remove the ambiguity by enforcing operator precedence and left and right associativity for different operators. This part is to be done by hand.
- Modify the grammar obtained in step 1 to support LL(k) parsing, 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.
- After you have the grammar in the correct form, you will input it to ANTLR and generate the parser in C++ or Java as per your choice. 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 input Tiger program generates a parse tree which can be potentially visualized with a suitable IDE plug-in. For more information, see 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, which should include: 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. In addition, your parser should also output the sequence of token types it sees, as it receives them from the scanner when you turn on a debug flag in your code. This will help us in verifying your solution. For example, given the stream "`var x := 1 + 1`", the parser would output "`VAR ID ASSIGN INTLIT PLUS INTLIT`". You should use the error recovery mechanisms provided in the 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 of this project.

Turn-in

Correctness

You will be provided with some simple programs for testing. You will also be provided with several test inputs.

Grading (50 points)

Deliverables for part 1:

- Hand-modified Tiger grammar in appropriate LL(k) grammar form (25 points)
- Generated Parser code (15 points)
- Testing and output report (10 points)

Project Phase 1A will be graded twice. The first time will be on the first due date. It will be graded again when you resubmit it with Project Phase 1B. Your final grade will be the average of the two grades.

Format

The graders will build your code. You must include a *makefile* in your source code directory so that we can build your code by simply running *make*. Since we need to run your code on different tests, include an example about how to run your code in your report.

As previously described, your generated parser code should generate correct codes and print accurate information including:

- < tokentype, token > tuple on successful scanning, one-per-line
- The sequence of token types on successful parsing, e.g. "VAR ID ASSIGN INTLIT PLUS INTLIT", all in a line.
- Error messages, either lexical or synthesis error as mentioned above, if the test code contains errors.

Resources

https://github.com/antlr/antlr4/blob/master/doc/getting-started.md

"Semantic Analysis"
https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/180%20Semantic%20Analysis.pdf

The Definitive ANTLR 4 Reference, 2nd Edition (available on https://learning.oreilly.com/ using your GaTech sign-in)

The TA will tell you the exact version of ANTLER to be used. The TA might also provide a Vagrant build with the ANTLR version.