

Course Project

Instructor: Dr. Fahed Jubair
Computer Engineering Department
University of Jordan



We Will Implement A Simple Compiler



- Our goal in this project is to build a compiler for a simple programming language, called *MICRO*
- *MICRO* is popularly used for teaching students how to build compilers in many universities
- Our compiler will translate a *MICRO* program into a simple two-address assembly program, called *Tiny*





MICRO Language

- For the purpose of this course, we will use a modified version of *MICRO*
- *MICRO* syntax is a bit similar to C
- MICRO is case-sensitive
- Keywords are written using capital letters only
- Function “main” is where the execution starts
- Three data types are used: **INT**, **FLOAT**, and **STRING**
 - STRING variables are read-only and must be initialized when declared
 - INT and FLOAT variables can be read and written and are not initialized when declared

“Hello” Program in *MICRO*



```
PROGRAM hello
BEGIN
    STRING str := “Welcome to MICRO!” ;

    FUNCTION void main ( )
        WRITE ( str ) ;           -- print on the screen
    END

END
```



Tiny Assembly Code

- The output of our compiler is *Tiny*: a two-address assembly code
- We will use a **simulator** to execute and test translated *Tiny* codes
- I will go into more details about *Tiny* later

Compilation is Done in Multiple Passes



- During the translation, compilers **pass** over the source code or the IR several times
- Each pass usually comprises a single task
 - For example, the first pass is scanning, next pass is parsing, next pass is symbol table generation, etc
- The output of each pass is the input of the next pass
- Multi-pass compilers are very popular
 - **We will use the multi-pass scheme in our project**
- Single-pass compilers (where everything is done in one pass) do exist but less commonly used

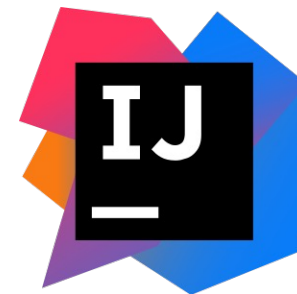
Coding Language is Java



- We will implement the compiler in Java
- Java is platform-independent
 - We need not worry about OS compatibility 😊
- Java is also known for being easy to learn, write, compile and debug
- Multiple nice IDEs that support Java are available



NetBeans





Coding Tips

- Do the design first, then write the code
- Take advantage of object-oriented programming
- Some of the good programming practices:
 - Keep the code simple
 - Use good naming scheme
 - Keep the code portable and extendable
 - Eliminate redundancy
 - Encapsulate what varies into classes



Project Steps

- Step0: forming teams
- Step1: implementing the scanner
- Step2: implementing the parser
- Step3: building the symbol table
- Step4: generating the IR
- Step5: generating *Tiny* code
- Step6 (not included for BSc): implementing live variables dataflow analysis
- Step7 (not included for BSc): implementing register allocation

Step 0: Forming Teams



- Each team should consist of 2 students
- You may ask other teams questions about Java or the IDE but not about the compiler design or code
- It is really common sense to tell what is cheating and what is not
- Team Registration link: TBD

Step1: Building The Scanner



- We will use ANTLR to automatically obtain the code of the scanner (*isn't that great?!*)
- ANTLR's input is a text file that specifies the regular expression of each token, as well as the language grammar
 - We will worry about the grammar in Step 2
- To complete step1, you need to learn
 - How to express regular expressions with ANTLR
 - How to use ANTLR tool to generate the scanner code
 - How to integrate the scanner code with your compiler code

Token Types in *MICRO*



- KEYWORD
- OPERATOR
- IDENTIFIER
- INTLITERAL, FLOATLITERAL, STRINGLITERAL
- COMMENT



Keywords in *MICRO*

PROGRAM	→	declare a program
BEGIN	→	Similar to an open bracket '{' in C
END	→	Similar to a closed bracket '}' in C
FUNCTION	→	declare a function
RETURN	→	Similar to return in C
READ	→	Read the input from the user
WRITE	→	Print on the screen
IF	}	IF statement keywords
ELSE		
ENDIF		
FOR	}	FOR statement keywords
ENDFOR		
INT	}	Data type declarations
VOID		
STRING		
FLOAT		



Operators in *MICRO*

:=	→	Assignment operator
+	→	Plus
-	→	Minus
*	→	Multiply
/	→	Divide
=	→	Test if equal
!=	→	Test if not equal
<	→	Test if smaller
>	→	Test if greater
<=	→	Test if less or equal
>=	→	Test if greater or equal
;	→	Same as C: a code statement must end with a semicolon
,	→	Same as C: separate parameters using a comma
(→	Left parenthesis
)	→	Right parenthesis



Identifiers in *MICRO*

- An identifier token will begin with a letter, followed by any number of letters and numbers
 - Letters can be small or capital
 - A number consists of characters 0, 1, ..., 9
- Example on valid identifier names:
N, m, STR, str, Str1, a100
- Example on non-valid identifier names:
8N, STR!, 1



Literals in *MICRO*

- Three types of literals can be used:

1. INTLITERAL

- Any integer number

2. FLOATLITERAL

- Any real number with the format xxx.xxx or .xxx
- integer and fraction parts can have any number of digits

3. STRINGLITERAL

- any sequence of characters except ' " ' between ' " ' and ' " '
- Valid string examples: "Hello!\n" and "*****"



Comments in *MICRO*

- Any string that starts with "--" and lasts till the end of line
- The scanner must recognize comments but then ignore them (i.e., skip them)
- Valid comment examples:
 - this is a comment
 - \$%#4-34455003-7354530
 - 999--



Micro.g4

- Micro.g4 will be the input file of the ANTLR tool
- Below is skeleton of Micro.g4 code (your job is to finish it)

grammar Micro;

KEYWORD:

OPERATOR:

IDENTIFIER:

INTLITERAL:

FLOATLITERAL:

STRINGLITERAL:

COMMENT:

WS : [\t\r\n]+ -> skip ; *// skip spaces, tabs, newlines*

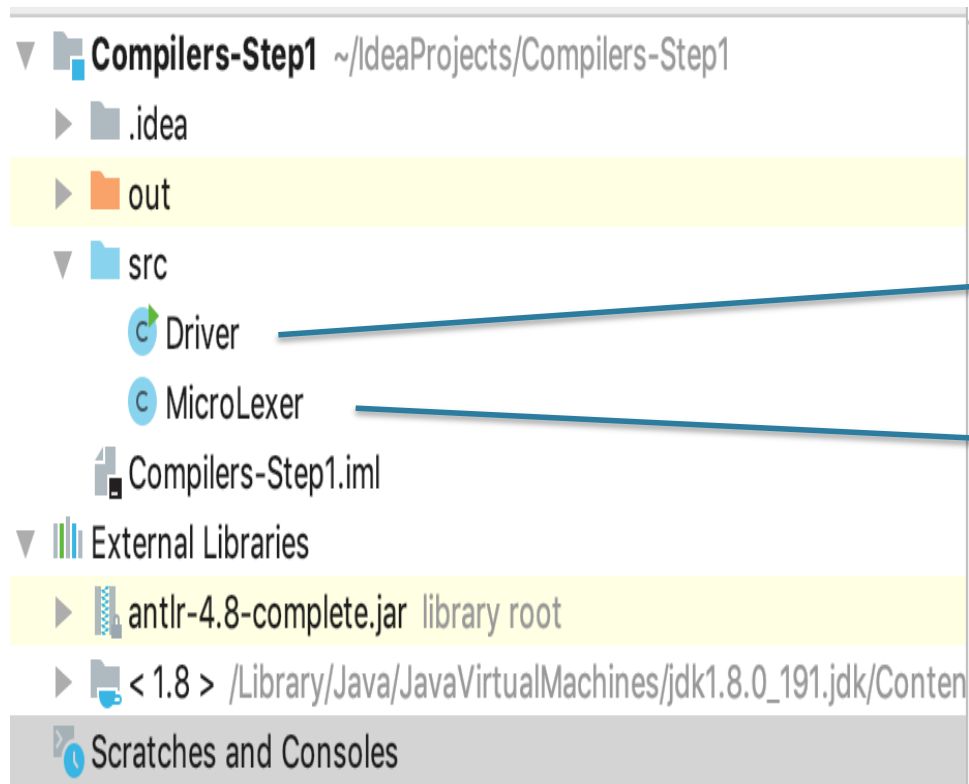
Refer to the ANTLR v4
reference book to learn
how write regular
expressions with ANTLR



Using ANTLR Tool

- Download ANTLR: <https://www.antlr.org/download/antlr-4.13.1-complete.jar>
- Use the following command
`java -jar antlr-4.13.1-complete.jar Micro.g4`
- Scanner source code is then found in `MicroLexer.java`
- To use the scanner, simply integrate the scanner source code files with your code

Example: Integrating Scanner Code With Eclipse Project



This is the test file

add scanner class
to the project

Do not forget to add
ANTLR to the build path



Driver.java

```
// import antlr
import org.antlr.v4.runtime.ANTLRInputStream;

import java.io.FileInputStream;
import java.io.InputStream;
public class Driver {
    public static void main(String[] args) throws Exception{
        // read input MICRO code
        InputStream is=null;
        try{
            String inputFile;
            inputFile = args[0];
            is = new FileInputStream(inputFile);
        }
        catch ( Exception e){
            System.out.println("You must specify an input file");
            System.exit( status: 0);
        }

        ANTLRInputStream input = new ANTLRInputStream(is);
        MicroLexer lexer = new MicroLexer(input);

        // add code here to print each token's type and value

    }
}
```

This code will invoke the
lexer automatically



Step 1 Output

- Your code should read an input *MICRO* code, **scan** it, and then produce an output file that shows each token's type and value
- I uploaded a set of input *MICRO* code examples on the course website
- I also uploaded the expected output for each input code example
- Due: TBD

Step2 : Building The Parser



- We will use ANTLR to automatically obtain the code of the parser
- To complete step 2, you need to:
 1. Use the grammar file (Micro.g4) to write the full grammar of Micro (the language grammar is available on the course webpage)
 2. Use ANTLR to obtain the scanner and parser codes and integrate them with your project
 3. Modify [Driver.java](#) to invoke the parser and check for syntax errors



Micro.g4

grammar Micro;

KEYWORD:

.....

WS : [\t\r\n]+ -> skip ;



These are the regular
expressions that you wrote in
step 1

program: 'PROGRAM' id 'BEGIN' pgm_body 'END' ;

id: IDENTIFIER ;

....



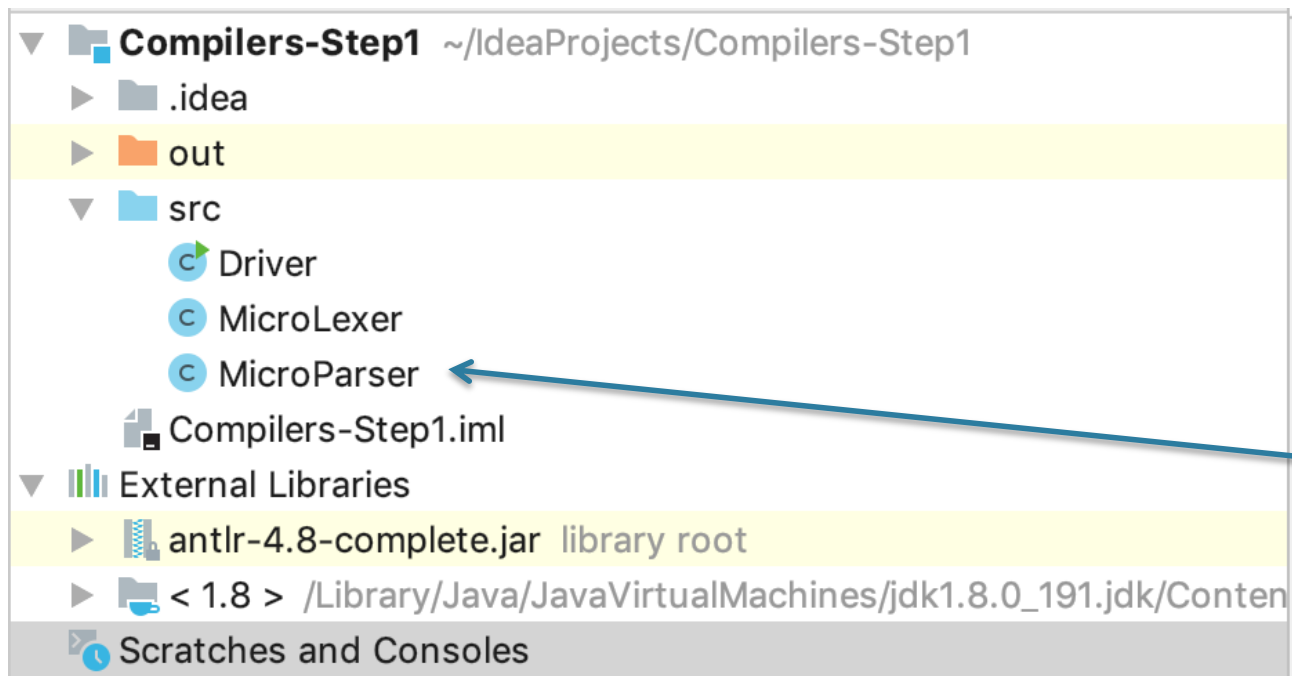
Now, write the
grammar rules



Using ANTLR Tool

- Use the following command
`java -jar antlr-4.13.1-complete.jar -no-listener Micro.g4`
- The scanner code is found in [MicroLexer.java](#)
- The parser code is found in [MicroParser.java](#)

Integrating Parser Code



Add the parser code
to the project you
already created in
step1

Driver.java



```
public class Driver {  
  
    public static void main(String[] args) throws Exception{  
        // read input MICRO code  
        InputStream is=null;  
        try{  
            String inputFile;  
            inputFile = args[0];  
            is = new FileInputStream(inputFile);  
        }  
        catch ( Exception e){  
            System.out.println("You must specify an input file");  
            System.exit( status: 0);  
        }  
  
        ANTLRInputStream input = new ANTLRInputStream(is);  
        MicroLexer lexer = new MicroLexer(input);  
        MicroParser parser = new MicroParser(new CommonTokenStream(lexer));  
        System.out.println("number of errors is " + parser.getNumberOfSyntaxErrors());  
    }  
}
```



Step 2 Output

- By the end of this step, your compiler should be able to take a source Micro code, parse it and show number of syntax errors
- I uploaded a set of input *MICRO* code examples on the course website, as well as the expected output for each code example
- Due: One week from today

Step3 : Building The Symbol Table



- In this step, you will implement a **semantic action pass** for building the symbol table
- Our symbol table only needs to store information about declared variables in the program
 - For *INT* and *FLOAT* variables, the symbol table should store their types and names
 - For *STRING* variables, the symbol table should store their types, names and values

Variables Declaration In Micro



- In Micro, there are multiple *scopes* where variables can be declared:
 - Variables declared at the beginning of the program before any functions are *global* – visible to all functions
 - Variables declared as part of a function's parameter list are *local* to this function, and cannot be accessed by any other function
 - Variables declared at the beginning of a function body are *local* to this function as well.
 - Variables declared at the beginning of an ***if***, ***else*** or ***while*** blocks are *local* to the block itself. Other blocks, even in the same function, cannot access these variables.

Symbol Table Organization



- You will implement an individual symbol table for each **scope** that store its *local* variables
- Variables with the same name cannot appear in the same scope
- Duplicate variable names across different scopes are allowed
- Note that *scopes* can be nested, therefore, variables in outer scopes are implicitly visible to inner scopes
- The compiler should keep track of the relationship between nested scopes

Implementing Semantic Actions



- Semantic actions are routines that the compiler invokes while traversing the **parse tree**
- ANTLR provides multiple mechanisms for implementing semantic actions:
 - Inserting actions in the grammar
 - Inserting actions in a parse-tree visitor ←
 - Inserting actions in a parse-tree listener

We will use
this method



Calculator Grammar

- I will use the following grammar to demonstrate how to implement semantic actions
- The grammar describes an algebraic expression of integers that are added or subtracted to each other

// Calculator.g4

```
grammar Calculator;
```

```
expr : INT expr_tail;
```

```
expr_tail : '+' INT expr_tail  
          | '-' INT expr_tail  
          ;
```

```
INT : [0-9] | [1-9] [0-9] ([0-9]?); // up to 3-digit integers
```

```
WS : [ \t\r\n]+ -> skip; // skip spaces, tabs, newlines
```

Our goal is to write semantic actions that compute the sum of the integers in the expression

Embedding Semantic Actions Into **Parse-Tree Visitors**



- To generate the Visitor class, use the following option when running ANTLR tool:

```
$ java -jar antlr.jar -no-listener -visitor Calculator.g4
```

↑
This option
deactivate Listener

↑
This option generates
Visitor classes

- Output files are:

- CalculatorLexer.java
- CalculatorLexer.tokens
- CalculatorParser.java

- CalculatorVisitor.java
- CalculatorBaseVisitor.java

Looking Inside CalculatorBaseVistor.java



- By default, ANTLR generate only one visit function for each **non-terminal**, regardless of how many rules it has
- However, as we studied in class, it is better to generate a separate visit function for each **rule**
- We can add directives in the grammar file to guide ANTLR into generating separate visit functions for separate rules (see next slide)

Embedding Semantic Actions Into **Parse-Tree Visitors**



- Conceptually, ANTLR Visitor is similar to the Visitor pattern we studied in lecture 7

// **Calculator.g4**

grammar Calculator;

expr : **operand=INT** expr_tail;

expr_tail : '+' **operand=INT** expr_tail **#AdditionOperation**
 | '-' **operand=INT** expr_tail **#SubtractionOperation**
 | **#EmptyOperation**
 ;

INT : [0-9] | [1-9] [0-9] ([0-9]?); // **up to 3-digit integers**

WS : [\t\r\n]+ -> skip; // **skip spaces, tabs, newlines**

Let us do a demo

© All Rights Reserved.



You Are Ready

- Use ANTLR's Visitor to implement semantic actions for building the symbol table
- Hints:
 - **Look carefully at all the rules inside Micro grammar and decide**
 - Which rules that indicate that a new scope has started
 - Which rules that indicate that a scope has ended
 - Which rules that indicate that new variables are being declared
 - Add hashtags to your grammar file to generate rule-based visit functions
 - **Think about the design of your code**
 - Is it a good idea to create a class for Symbol? If yes, what are the methods needed in that class?
 - Is it a good idea to create a class for Scope? If yes, what are the methods needed in that class?

Step 3 Test Cases and Output



- To test the correctness of this step, your compiler should print the symbol table for each scope
- I uploaded a set of input *MICRO* code examples on the course website, as well as the expected output for each code example
- Due: TBA