Assignment 2: MiniC Grammar Implementation (10%)

Due: Feb. 2, 2022

In this assignment, you will learn how to follow the MiniC language specifications and write MiniC grammar rules in the `g4` format. The assignment is based on the environment built in Assignment 1.

ANTLR4 Workflow in the Minic Compiler

In the skeleton code, use `grammars/Minic.g4` to fill in the MiniC grammar. When building, `antlr4-runtime` processes `Minic.g4` grammar and generates Minic parser source files. The parser files are under `build/grammars/Minic` folder. Then, `src/ASTChecker.cpp` & `src/ASTBuilder.cpp` uses the Minic parser API to generate the parse tree and AST (AST will be implemented in Assignment 3). The two files are already written for you and you don't need to modify them.
If you are interested in how the ANTLR4 is compiled for MiniC, you can take a look at `CMakelist.txt` in each folder.

Defining .g4 Files

The `.g4` format follows lexer rules and parser rules defined by ANTLR4. The details are in the ANTLR 4 Documentation. The simplified version is shown below in Table 1.
Here is a simple example, `Expr.g4`. The file name must be the same as the grammar name. There are 4 rules in grammar `Expr`: `prog`, `expr`, `NEWLINE` and `INT`. Each rule definition is stated after `":"`. E.g., the definition of `prog` is the string with 0 or more patterns of `expr NEWLINE`. You can learn about the other three rules according to Table 1.

grammar <name of the language>
so here the name of our grammar language is "Expr".

```
grammar Expr;
prog: (expr NEWLINE)* ;
expr: INT
    | expr ('*' | '/') expr
    | expr ('+'|'-') expr
    | '(' expr ')'
    ;
NEWLINE: [\r\n]+ ;
INT:     [0-9]+ ;
```

Simple Testing

Install `antlr4` following UNIX installation from ANTLR4 Github. The `antlr-4.9-complete.jar` mentioned in the website is the same as `third_party/antlr/antlr-4.9-complete.jar` in skeleton code. The test commands for `Expr.g4` are shown on Listing 1.

Syntax	Description
T	Invoke lexer rule T; recursion is allowed in general, but not left recursion. T can be a regular token or fragment rule.
'literal'	Match that character or sequence of characters. E.g., 'true' or '='.
[char set]	Match one of the characters specified in the character set. E.g. ID : [a-zA-Z] [a-zA-Z0-9]* ;
'x'..'y'	Match any single character between range x and y, inclusively. E.g., 'a'..'z'. 'a'..'z' is identical to [a-z].
{action}*	The lexer executes the actions at the appropriate input position, according to the placement of the action within the rule. E.g., END : ('endif'|'end') {System.out.println("found an end");} ; ANTLR copies the action's contents into the generated code verbatim.
∼x	Match any single character not in the set described by x. E.g., COMMENT: '//' (∼[\r \n])* − > skip; A 'skip' command tells the lexer to get another token and throw out the current text.
$*	Attribute
@init{action}*	Action for initialization
:	Rule definition
;	End rule
|	Alternative
.	Wildcard
[...] *	Argument or return value spec
+	1 or more
∗	0 or more
?	Optional or semantic predicate
!	Don't include in AST
− >	Rewrite rule
// ...	Single-line comment
/∗ ... ∗/	Multi-line comment

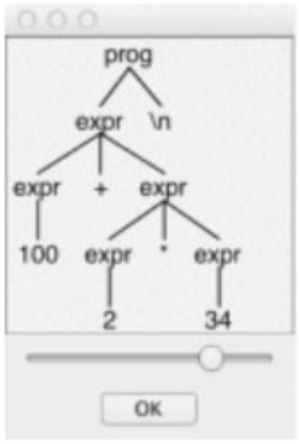
* **These will be implemented in Assignment 3**

Table 1: Rule Elements

```
$antlr4 Expr.g4
$javac Expr*.java
$grun Expr prog -gui
>>100+2*34
>>^D
```

Listing 1: Expr.g4 testing

A simple parse tree is shown below:



From the GUI output, the priority of `'*'` is higher than `'+'`. The reason is that in `Expr.g4`, `'*','/'` subrule is written first, and then `'+','-'`. Therefore, the order of subrules will determine operator priority.

Minic.g4 implementation

You will follow the MiniC language specifications from `language.pdf` (which can be downloaded from `Files/handouts/language.pdf` on Quercus) to write MiniC grammar rules. The grammar structure in `language.pdf` will give you an idea for the `Minic.g4`. You cannot directly copy-paste the rules to `Minic.g4` otherwise you may get the wrong solution. (Hint: You can reorganize the language structure to make it legal.)

In the skeleton, `Minic.g4`, the default rule declarations are set. **Do not remove any default rules**. You can add additional rules as necessary. `varlistentry` & `parameterentry` are optional rules that you may or may not choose to use.

Public Autotester

Besides the simple testing mentioned above, we have provided a public autotester `a2tester.zip` on Quercus. After compiling your code, change line 11 in `asst2.py` to your minicc executable file path. Then simply run:

```
    $./asst2.py
OR in verbose mode
    $./asst2.py -v
```

The A2 tester will examine the number of statements, etc with different MiniC benchmarks. Check `src/GrammarStatVisitor.*` for more details. You could add more counting categories in `GrammarStatVisitor.*`. Also, You can add your own benchmarks for the public tester, but please revert it back for your submission.

Deliverables

Compress the whole project folder as a `zip` file. On Markus, please submit:

- The `zip` file. You should only need to modify the `Minic.g4` file in this assignment. We will build your submission from source and run it against public and private tests.

- The brief explanation of your implementation and testing plan as a `txt` file.

If you have any questions or issues, please post on the course Piazza website.