**University of Toronto**

**CSC 488 / CSC2107 Compilers and Interpreters** **Winter 2022**

# Assignment 3: MiniC AST Implementation (11%)

## Due: Feb. 16, 2022

In this assignment, you will learn how to write compiler actions to assemble the AST in the `g4` format. The assignment is based on the Minic Grammar rules in `MiniC.g4` in Assignment 2.

## AST Workflow in the MiniC Compiler (cont'd from Assignment 2 ANTLR4 Workflow)

In `src/ASTBuilder.cpp` or `src/ASTChecker.cpp`, the `parser` object takes input tokens and implements the actions of rules or subrules defined in the grammar `MiniC.g4`. The actions connect AST nodes with each other. Every rule represents `ASTNode` or inherited `ASTNode`, which will be discussed more later.
Then the AST node representing `prog` is retrieved from `parser`. Finally `ASTVisitor` or inherited `ASTVisitor` objects are capable of visiting all of AST nodes starting at `prog` recursively.

## Defining .g4 Files

Besides the g4 syntax elements defined in Assignment 2, Table 1 defines more elements you may use in Assignment 3. The details are in the ANTLR4 Actions and Attributes Documentation, and we **strongly recommend** you to read it.

## A Simple Example in Java

Here is a simple example: `DynScope.g4`. Note that the actions are written in `java` (We will be using **C++** actions for MiniC). The `DynScope` grammar should be somewhat familiar from Assignment 2. In terms of actions and attributes, each rule is a Class object after ANTLR4 processes the g4 file. In the `block` rule, the return value is `symbols`, which is a member variable of object `block`. Adding the $ attribute allows use of local variables and rule objects. Within each scope `{..}`, the actions can be implemented in target language. In this case, `block` collects all declarations identifiers and prints symbols. `stat` identifies inner block and prints the scope.

```
grammar DynScope;
prog: block ;
block returns [List<String> symbols]
    @init {
        $symbols = new ArrayList<String>();
    }
    : '{' (decl{
        $symbols.add($decl.symbol);
    })*
    stat+ '}'
    {
        System.out.println("symbols="+$symbols);
    }
    ;
decl returns [String symbol]
    : 'int' ID {$symbol = $ID.text;} ';'
    ;
stat: ID '=' INT ';'
    | b=block {
        System.out.println("New scope contains: " + $b.symbols);
    }
    ;
ID : [a-z]+ ;
INT : [0-9]+ ;
WS : [ \t\r\n]+ -> skip ;
```

Listing 1: DynScope.g4 Example

## Testing our Example

The test commands for `DynScope.g4` are shown on Listing 2.

```
$antlr4 DynScope.g4
$javac DynScope*.java
$grun DynScope prog
>>{int i;int j;
>>i=0;j=3;
>>{int z;int h;
>>z = 1;
>>}}
>>^D
symbols=[z, h]
New scope contains: [z, h]
symbols=[i, j]
```

Listing 2: DynScope.g4 testing

| **Syntax** | **Description** |
|---|---|
| ? | 0 or 1 of the pattern |
| {action} | The lexer executes the actions at the appropriate input position, according to the placement of the action within the rule. actions are in C++ syntax.<br>E.g., END : ('endif'\|'end') {System.out.println("found an end");} ;<br>ANTLR copies the action's contents into the generated code verbatim. |
| $ | Attribute |
| @init{action} | Action for initialization |
| @header{action} | Actions for header including |
| [...] | Argument or return value spec. The arguments inside the square brackets are in C++ syntax |
| r [arg1,arg2,...] | Match rule r at current input position, passing in a list of arguments just like a function call. |
| l1=r1 l2=r1 l3=r2 ... | Using l1, l2... to label rule elements using the = operator to add fields to the rule context objects. |
| r [args] returns [retvals] locals [localvars] : ... ; | `retvals` is the return values of r. `localvars` is the local varibles of r. These variables can be used in the actions inside r or subrules of r. |

Table 1: Rule Elements (Continued from A2)

## Minic.g4 implementation

You will add compiler actions in `MiniC.g4`. As mentioned above, all of rules are inherited from `ASTNode`. E.g., in `src/Declarations.h`, `VarDeclaration` and `FuncDeclaration` are inherited from `Declaration` which is inherited by `ASTNode`. You should invoke related member functions to form AST. Besides, different inherited classes have their own members and you should take care of them. Here are some tips you should consider:

- In Class `Program`, `SyslibFlag` should be raised if the `minicio` header is included.
- In Class `FuncDeclaration`, `HasBody` should be raised if the function has a body.
- In Class `ScopeStatement`, `NumVarDecl` defines the number of variable declarations in the scope.
- In the Minic language, some rules contain the subrule which is a sequence of the same type rule. E.g., `decl: VarDeclaration FuncDeclaration`. You may think of a way to initialize a AST node to represent `decl`.
- `ctx->start->getLine()` returns the line number of current rule/subrule in the input Minic code.
- `ctx->start->getCharPositionInLine()` returns the column number of current rule/subrule in the input Minic code.
- You can edit and add other source/header files **except src/AST* files**. If you are adding files, make sure it can be compiled with the Makefile, or it may fail the autotester.

## Public Autotester

We have provided a public autotester `a3tester.zip` on Quercus. After compiling your code, change line 12 in `asst3.py` to your minicc executable file path. Then simply run:

```
    $./asst3.py
OR in verbose mode
    $./asst3.py -v
```

The A3 tester will examine AST structure by using `src/ASTPrinter.*` with different MiniC benchmarks. Please revert `src/AST*` back for your submission.

## Debugging

Unfortunately, `grun` is only supported in Java and for Java actions. Since we will be defining C++ actions for MiniC, `grun` may not be as useful for this Assignment. You can try to debug your actions using a debugger (e.g. `gdb`) or by adding print statements if you wish. You can also try to modify the `src/AST*` files for debugging, but please revert it back before submitting.

## Deliverables

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

- The `zip` file. **DO NOT modify the src/AST* files**. 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.