Grammar of TeaPL

Note: The precedence and associativity of operators in TeaPL are the same as those in C language.

Each program is composed of variable declarations, function declarations, function definitions, and comments.

```
program := (varDeclStmt | structDef | fnDeclStmt | fnDef | comment | < ; >)*
```

Basic Identifiers, Values, Expressions, and Assignments

Each identifier begins with an alphabat and contains only alphabats and digits, e.g., alice, a0.

```
id := [a-z_A-Z][a-z_A-Z0-9]*
```

TeaPL allows integers, e.g., 123

```
num := [1-9][0-9]* | 0
```

Arithmatic Expressions

An expression is a composd of identifiers, values, and operators, e.g., 1+2, a*(b+c). For simplicity, we donot support unary operators, such as ++, +=.

```
arithExpr := arithExpr arithBiOp arithExpr | exprUnit
exprUnit :=  num | id | < ( > arithExpr < ) > | fnCall | leftVal < [ > id | num < ] > |
leftVal < . > id | arithUOp exprUnit
arithBiOp := < + > | < - > | < * > | < / >
arithUOp := < - >
```

Condition Expressions

```
boolExpr := boolExpr boolBiOp boolExpr | boolUnit
boolUnit := exprUnit comOp exprUnit | < ( > boolExpr < ) > | boolUOp boolUnit // we
restrict the operands of comparison operators to be exprUnit instead of rightVal to
avoid confusing the precedence.
boolBiOp := < && > | < || >
boolUOp := < ! >
comOp := < > > | < < > | < >= > | < <= > | < == > | < != >
```

Assignment

We restrict neither the left value nor right value can be assignments.

```
assignStmt := leftVal < = > rightVal < ; >
leftVal := id | leftVal < [ > id | num < ] > | leftVal < . > id
rightVal := arithExpr | boolExpr
```

Function Call

```
fnCall := id < ( > rightVal (< , > rightVal)* | ϵ < ) >
```

Variable Declarations

TeaPL allows declaring one variable each time, which can be either a primitive or array type. Developers can initializate the variable during declaration. For example, it supports the following variable declaration samples.

Primitive Types

```
let a:int; // declare a variable of type int; the type field can be ignored.
let b:int = 0; // declare a variable of int and init it with value 0.
```

One-level Array

```
let c[10]:int; // declear a variable of integer array.
let d[10]:int = {0}; // declear a variable of integer array and initialize it with
zero.
```

The grammar is defined as follows.

```
varDeclStmt := < let > (varDecl | varDef) < ; >
varDecl := id < : > type |  id < [ > num < ] >< : > type | id |  id < [ > num < ] >
varDef :=  id < : > type < = > rightVal | id < = > rightVal  //primitive type
         | id < [ > num < ] >< : > type < = > < { > rightVal (< , > rightVal)* | ϵ < }
> | id < [ > num < ] > < = > < { > rightVal (< , > rightVal)* | ϵ < } > //array
type := nativeType | structType
nativeType := < int >
structType := id
```

Define A New Structure

Developers can define new customized types with the preserved keyword struct, e.g.,

```
struct MyStruct {
    node:int,
    len:int
}
```

The grammar is defined as follows.

```
structDef := < struct > id < { > (varDecl) (< , > varDecl)* < } >
```

Function Declaration and Definition

Each function declaration starts with the keyword fn.

```
fn foo(a:int, b:int)->int;
fn foo();
```

The grammar is defined as follows.

```
fnDeclStmt := fnDecl < ; >
fnDecl := < fn > id < ( > paramDecl < ) > //without return value
        | < fn > id < ( > paramDecl < ) > < -> > type //with return value
paramDecl := varDecl (< , > varDecl)* | ϵ
```

Function Definition
We can also define a function while declaring it.

```
fn foo(a:int, b:int)->int {
    return a + b;
}
```

The grammar is specified as follows.

```
fnDef := fnDecl codeBlock
codeBlock :=  < { > (varDeclStmt | assignStmt | callStmt | ifStmt | whileStmt |
returnStmt | continueStmt | breakStmt | < ; > )* < } >
returnStmt : = < ret > rightVal < ; > | < ret > < ; >
continueStmt := < continue > < ; >
breakStmt := < break > < ; >
```

We have already defined the grammar of varDeclStmt and assignStmt. The callStmt is simply a function call terminated with an colon.

```
callStmt := fnCall < ; >
```

Next, we define the grammar of each rest statement type.

Control Flows

If-Else Statement
The condition should be surrounded with a paired parenthesis, and we further restrict the body should be within a paired bracket. The following shows an example.

```
if (x >0) {
    if (y >0) {
        x++;
    }
    else {
        x--;
    }
} else {

}
```

Besides, we restrict the condition expression to be explicit logical operations, e.g., x >0; we donot allow implicit expressions like x, which means. We define the grammar as follows.

```
ifStmt := < if > < ( > boolExpr < ) > codeBlock ( < else > codeBlock | ϵ )
```

While Statemet

Used for the representability of complicated loops.

Example:

```
while (x  > 0) {
    x--;
}
```

Definition:

```
whileStmt := < while > < ( > boolExpr < ) > codeBlock
```

Code Comments

Similar to most programming languages, TeaPL allows line comments with "//" and scope comments with "/* ... */".

```
int a = 0; // this is a line comment.

/*
    Feature: this is a scope comment
*/
fn foo(){
    ...
}
```

```
comment :=  < // > .* | < /* > [^]* < */ >
```

`.`: This is a special character that matches almost any character except `\n`.

`[^]`: This is a character class that matches any character not in the brackets(`[^ABC]`: A single character that is not 'A', 'B', or 'C'). Since there are no characters in the brackets, it matches any character.

`*`: This is a quantifier that means "zero or more of the preceding element".

`.*`: This regular expression matches any number (including zero) of almost any character, except for newline characters.

`[^]*`: This regular expression matches any number (including zero) of any character, including newline characters.