MINI (v1.6) Grammar

Jingke Li

Portland State University

A Naive Grammar for MINI

```
Program     -> ClassDecl {ClassDecl}

ClassDecl   -> "class" <ID> ["extends" <ID>]
                 '{' {VarDecl} {MethodDecl} '}'

MethodDecl -> "public" Type <ID> '( [Formals] ')' MethodBody
            |  "public" "static" "void" "main"
                 '(' "String" '[' ']' <ID> ')' MethodBody
MethodBody -> '{' {VarDecl} {Statement} '}'
Formals     -> Type <ID> {',' Type <ID>}

VarDecl     -> Type <ID> ['=' Expr] ';'

Type        -> Type '[' ']'
            |  <ID>
            | "int" | "float" | "boolean"
            | "void"
```

A Naive Grammar for MINI (cont.)

```
Statement  -> '{' {Statement} '}'
           |  Expr '=' Expr ';'
           |  Expr '(' [Args] ')' ';'
           |  "if" '(' Expr ')' Statement ["else" Statement]
           |  "while" '(' Expr ')' Statement
           |  "System.out.println" '(' [Expr | <STRVAL>] ')' ';'
           |  "return" [Expr] ';'

Expr       -> "new" Type '[' <INTVAL> ']'
           |  "new" <ID> '(' [Args] ')'
           |  Expr Binop Expr
           |  Unop Expr
           |  Expr '[' Expr ']'
           |  Expr '(' [Args] ')'
           |  Expr '.' "length" '(' ')'
           |  Expr '.' <ID>
           |  '(' Expr ')'
           |  <ID>
           |  Literal
...
```

What Is Wrong with the Naive Grammar?

- It is too general! It allows illegal types, statements and expressions.

```
void x;
SomeClass[] y;
int[][] z;
3 + 2 = 7;
f(1) * f(2) = g(3);
y = new int[10].length() + 4;
z = 2 * this.x [new c(1,2)];
```

  - MINI does not support arrays of class objects, nor multi-dimentional arrays; void is not a valid data type.
  - MINI (as well as most languages) only allow expressions that denote storage locations (i.e. *l-values*) on the left-hand side of an assignment.
  - MINI does not allow array and object allocation expressions to be used in other expressions.

- It is ambiguous.

Refining the Grammar

Refine the syntax of types and explicitly capture the syntax of l-values:

```
MethodDecl -> "public" Type <ID> '( [Formals] ')' MethodBody
           |  "public" "void" <ID> '( [Formals] ')' MethodBody
           ...

Type       -> BasicType ['[' ']']
           |  <ID>

BasicType  -> "int" | "float" | "boolean"
```

An Improved Expression Grammar

```
Statement  -> '{' {Statement} '}'
           |  Lvalue '=' InitExpr ';'
           |  Lvalue '(' [Args] ')' ';'
              ...
InitExpr   -> "new" BasicType '[' <INTVAL> ']'
           |  "new" <ID> '(' [Args] ')'
           |  Expr
Expr       -> Expr Binop Expr
           |  Unop Expr
           |  '(' Expr ')'
           |  Lvalue '(' [Args] ')'
           |  Lvalue '.' "length" '(' ')'
           |  Lvalue
           ...
```

Note that this grammar still allows illegal expressions: e.g. arbitrary exprssions as array indices. For these cases, a sematic solution is more effective than a syntatic solution.

Eliminating Expression Ambiguity

```
Expr       -> Expr "||" AndExpr
           -> AndExpr

AndExpr    -> AndExpr "&&" RelExpr
           -> RelExpr

RelExpr    -> ...

ArithExpr  -> ...

Term       -> ...

Factor     -> ...
           -> Literal
```

highest	`new`, `()`
	`[]`, `.`(selector), call
	`-`, `!`
	`*`, `/`
	`+`, `-`
	`==`, `!=`, `<`, `<=`, `>`, `>=`
	`&&`
lowest	`||`

Eliminating Left-Recursion

Use the standard transformation:

Replace $A \rightarrow A\alpha \mid \beta \mid \cdots \mid \gamma$

with
$$A \rightarrow \beta A' \mid \cdots \mid \gamma A'$$
$$A' \rightarrow \alpha A' \mid \epsilon$$

```
Expr      -> AndExpr OrTail
OrTail    -> [ "||" AndExpr OrTail ]

AndExpr   -> RelExpr AndTail
AndTail   -> [ "&&" RelExpr AndTail ]

...
```

Easy Left-Factoring Cases

Before:

```
Statement  -> '{' {Statement} '}'
           |  Lvalue '=' InitExpr ';'
           |  Lvalue '(' [Args] ')' ';'
              ...
InitExpr   -> "new" BasicType '[' <INTVAL> ']'
           |  "new" <ID> '(' [Args] ')'
           |  Expr
BasicType  -> "int" | "float" | "boolean"
```

After:

```
Statement  -> '{' {Statement} '}'
           |  Lvalue ('=' InitExpr | '(' [Args] ')') ';'
              ...
InitExpr   -> "new" ( BasicType '[' <INTVAL> ']'
                    | <ID> '(' [Args] ')' )
```

Hard Left-Factoring Cases

```
VarDecl    -> Type <ID> ['=' InitExpr] ';'
Type       -> BasicType ['[' ']']
BasicType  -> "int" | "float" | "boolean"
Statement  -> Lvalue ('=' InitExpr | '(' [Args] ')') ';'
Lvalue     -> ["this" '.'] <ID> '.' <ID> ['[' Expr ']']
MethodDecl -> "public" (Type | "void") <ID> '(' [Formals] ')'
                '{' {VarDecl} {Statement} '}'
```

Bother `VarDecl` and `Statement` can start with an `<ID>`.

```
class a { ...}
public int f() {
  a x = new a();
  ...
```

```
int a;
public int f() {
  a = 2;
  ...
```

Now consider the input: `public int f() { a ...`

Is "a" the start of a `VarDecl` or a `Statement`? — It is not easy to factor out `<ID>` from `VarDecl` and `Statement`, so using an additional lookahead is acceptable for this case.