Universidad Rafael Landívar
Tradición Jesuita en Guatemala

Facultad de Ingeniería
Compiladores
Segundo Ciclo 2018
Mgtr. Diana Gutiérrez

Proyecto - Fase #2 - Analizador sintáctico

Mini C#

Fecha de entrega: 11 de octubre, 07:00 pm

El objetivo

La segunda fase del proyecto consistirá en analizar sintácticamente un programa escrito en Mini C#. En esta fase, lo primero que harán es conocer el generador de analizadores sintáctico CUP y la manera en que se debe especificar una estructura en dicho generador. Además deberán integrarlo con su analizador léxico de la fase anterior. Al final de esta fase debe ser posible determinar que el programa fuente escrito en Mini C# está sintácticamente correcto.

Estructura Lexicográfica

Todas las especificaciones del proyecto No. 1, con excepción de:

- Palabras clave (reservadas):

Existentes:

void int double bool string class interface null this extends implements for while if else return break New NewArray

Deben agregar:

Print ReadInteger ReadLine Malloc

Gramática Mini C#

Generalidades de la gramática:

- x significa que x es un terminal, un token devuelto por el analizador léxico. Los terminales están en minúscula excepto aquellos que son palabras reservadas
- *x* en itálico es un no terminal. Todos los no terminales tienen la primera letra en mayúscula
- <x> significa cero o una ocurrencia de x, es decir, x es opcional
- x* significa cero o más ocurrencias de x
- x^+ significa una o más ocurrencias de x
- | significa las alternativas de las producciones
- ε significa épsilon, la ausencia de tokens.

La gramática del lenguaje Mini C# es la siguiente:

Program	::=	*Decl*+
Decl	::=	*VariableDecl* | *FunctionDecl* | *ClassDecl* | *InterfaceDecl*
VariableDecl	::=	*Variable* ;
Variable	::=	*Type* `ident`
Type	::=	`int` | `double` | `bool` | `string` | `ident` | *Type* `[]`
FunctionDecl	::=	*Type* `ident` `(` *Formals* `)` *StmtBlock* | `void` `ident` `(` *Formals* `)` *StmtBlock*
Formals	::=	*Variable*+`,` | ε
ClassDecl	::=	`class` `ident` ⟨`extends` `ident`⟩ ⟨`implements` `ident`+`,`⟩ { *Field** }
Field	::=	*VariableDecl* | *FunctionDecl*
InterfaceDecl	::=	`interface` `ident` { *Prototype** }
Prototype	::=	*Type* `ident` `(` *Formals* `)` ; | `void` `ident` `(` *Formals* `)` ;
StmtBlock	::=	{ *VariableDecl** *Stmt** }
Stmt	::=	⟨*Expr*⟩ ; | *IfStmt* | *WhileStmt* | *ForStmt* | *BreakStmt* | *ReturnStmt* | *PrintStmt* | *StmtBlock* |
IfStmt	::=	`if` `(` *Expr* `)` *Stmt* ⟨`else` *Stmt*⟩
WhileStmt	::=	`while` `(` *Expr* `)` *Stmt*
ForStmt	::=	`for` `(` ⟨*Expr*⟩ ; *Expr* ; ⟨*Expr*⟩ `)` *Stmt*
ReturnStmt	::=	`return` ⟨*Expr*⟩ ;
BreakStmt	::=	`break` ;
PrintStmt	::=	`Print` `(` *Expr*+`,` `)` ;
Expr	::=	*LValue* `=` *Expr* | *Constant* | *LValue* | `this` | *Call* | `(` *Expr* `)` | *Expr* `+` *Expr* | *Expr* `-` *Expr* | *Expr* `*` *Expr* | *Expr* `/` *Expr* | *Expr* `%` *Expr* | `-` *Expr* | *Expr* `<` *Expr* | *Expr* `<=` *Expr* | *Expr* `>` *Expr* | *Expr* `>=` *Expr* | *Expr* `==` *Expr* | *Expr* `!=` *Expr* | *Expr* `&&` *Expr* | *Expr* `||` *Expr* | `!` *Expr* | `New(ident)` | `NewArray(` *Expr*`,` *Type* `)` | `ReadInteger()` | `ReadLine()` | `Malloc(`*Expr*`)`
LValue	::=	`ident` | *Expr*`.``ident` | *Expr* `[` *Expr* `]`
Call	::=	`ident(`*Actuals*`)` | *Expr*`.``ident(`*Actuals*`)` | *Expr*`.`*LibCall*`(`*Actuals*`)`
LibCall	::=	`GetByte(`*Expr*`)` | `SetByte(`*Expr*`,` *Expr*`)`
Actuals	::=	*Expr*+`,` | ε
Constant	::=	`intConstant` | `doubleConstant` | `boolConstant` | `stringConstant` | `null`

Implementación

1. Crear un archivo de entrada para su analizador sintáctico que contenga la definición de la gramática del lenguaje Mini C#.
2. El parser leerá la estructura del archivo anterior y generará un programa en Java. Este programa debe ser compilado en vivo.
3. El programa compilado lee un archivo de entrada en lenguaje Mini C# y debe indicar en pantalla únicamente si está sintácticamente correcto o no. No debe realizar ninguna acción adicional si está correcto.
4. De encontrar algún error, el escáner debe continuar hasta el final de archivo.
    a. Los errores deberá reportarlos en pantalla indicando:

    Línea y columna, símbolo que provocó el error, Error: <descripción del error>.

Requerimientos generales

- El ejecutable debe ser llamado minic y debe pedir el archivo de entrada y escribir todos los resultados en pantalla.
- El proyecto se entrega en el mismo repositorio en GitHub de la fase anterior, sobre la rama master o principal. Por tanto, se deberá crear una rama con su analizador léxico y una nueva donde incorpore esta fase. Se calificará el proyecto hasta el último commit realizado antes de la hora de entrega. Se revisará el historial del repositorio. No se aceptarán cambios en el momento de la calificación.
- En el repositorio deberá incluir un archivo README.txt donde se detalle cómo su analizador maneja los errores.

¡Buena suerte!