Esta práctica consistirá en generar un prototipo de las dos primeras etapas de un intérprete (analizador léxico y sintáctico) para el lenguaje COOL.

Análizador léxico («lexing»)

Aunque es posible escribir analizadores léxicos en cualquier lenguaje, existen multitud de herramientos que generan código muy eficiente. En esta primera parte de la práctica vamos a «tokenizar» utilizando la libreria SLY. El lenguaje objetivo es el lenguaje COOL (Classroom Object Oriented Language). El lenguaje COOL es un lenguaje diseñado para ser fácil escribir un compilador, tal es el caso que se calcula que hay más compiladores para el lenguaje COOL que programas escritos en este lenguaje. En el siguiente link está un transpilador para javascript. En la siguiente sección daremos un resumen de los tokens del lenguaje y comportamiento esperado del compilador.

Especificación léxica básica

Las unidades léxicas del lenguaje «Cool» con el correspondiente nombre de token son:

- enteros (INT_CONST)
- identificadores de tipos (TYPEID)
- identificadores de objetos (OBJECTID)
- notación especial,
  - LE (<=)
  - DARROW (=>)
  - ASSIGN (<-)
- «strings» (STR_CONST)
- palabras reservadas (IF, FI, THEN, NOT, IN, CASE,ELSE, ESAC, CLASS, INHERITS, ISVOID, LET, LOOP, NEW, OF, POOL, THEN, WHILE, TRUE, FALSE)
- espacios
- símbolos literales (==, +, -, *, /, (,), <, >, ., , „ ;, :, @, ,)

Los enteros son cadenas de caracteres formadas por dígitos. Los identificadores consisten en cadenas de caracteres formadas por letras, números y el guión bajo. Los identificadores de tipos empiezan por una letra mayúscula y los identificadores de objetos empiezan por una letra minúscula.

Los «strings» están delimitados por comillas dobles. Dentro de un «string», la regla general es que $\backslash c$ se sustituye por el carácter c. Las siguientes excepciones no se les debe quitar la contrabarra:

- $\backslash b$, además es la representación del carácter «backspace»
- $\backslash t$, además es la representación del carácter «tab»
- $\backslash n$, además es la representación del carácter salto de linea
- $\backslash r$, además es la representación del carácter retorno de carro

El carácter salto de linea y las comillas no pueden aparecer en un string, a menos que estén escapados. Tampoco puede encontrar ni los caracteres de fin de fichero y el carácter $\backslash 0$.

El límite de longitud de una cadena de caracteres es de 1024 caracteres.

Hay dos formas de escribir un comentario en un fichero, utilizando $--$ *comentarios de linea* y encerrándolo entre los símbolos ($*$ y $*$). Los comentarios de varias lineas pueden estar anidados, por lo

que hay que contar aquellos el número de elementos de cierre. Notad que los comentarios de cierre se pueden escapar, pero entonces no cuentan como cierre de comentario.

Las palabras reservadas pueden estar en mayúscula y minúscula, menos la palabras **true** y **false** que deben empezar por minúscula para ser tokens del tipo *BOOL_CONST*, ya que si no se cumple esto, deben ser tratados como identificadores de tipos.

1) *Programe un analizar léxico, de forma que todos los archivos de ejemplo sean analizados y sigan la implementación base modificando únicamente el archivo Lexer.py*

Análizador sintáctico («parser»)

Con los tokens definidos, es necesario especificar el lenguaje COOL mediante una gramática libre de contexto. La siguiente gramática esta especificada utilizando la notación EBNF (notación extendida Backus-Naur).

La Notación de Backus-Naur extendida, también conocida como EBNF (del inglés Extended Backus–Naur Form), es un metalenguaje utilizado para expresar gramáticas libres de contexto). Es una extensión de la Notación de Backus-Naur.

Las variables están escritas entre ángulos y los tokens están en negrita. Además, se utiliza símbolos propios, que recuerdan a las expresiones regulares como

- $*$ para representar 0 o más veces la expresión anterior
- $+$ para representar 1 o más veces la expresión anterior
- $|$ para representar la elección entre dos expresiones
- las expresiones entre corchetes son opcionales
- los paréntesis sirven para delimitar las expresiones

⟨*Programa*⟩ ::= ⟨*Clase*⟩+

⟨*Clase*⟩ ::= `CLASS TYPEID` [`inherits TYPEID`] `{` (⟨*Atributo*⟩ | ⟨*Metodo*⟩)* `} ;`

⟨*Atributo*⟩ ::= `OBJECTID : TYPEID` [`ASSIGN` ⟨*Expresion*⟩]`;`

⟨*Metodo*⟩ ::= `OBJECTID ( ) : TYPEID {` ⟨*Expresion*⟩ `} ;`
| `OBJECTID (` (⟨*Formal*⟩ `,`)* ⟨*Formal*⟩ `) : TYPEID {` ⟨*Expresion*⟩ `} ;`

⟨*Formal*⟩ ::= `OBJECTID : TYPEID`

⟨*Expresion*⟩ ::= `OBJECTID ASSIGN` ⟨*Expresion*⟩
| ⟨*Expresion*⟩ `+` ⟨*Expresion*⟩
| ⟨*Expresion*⟩ `-` ⟨*Expresion*⟩
| ⟨*Expresion*⟩ `*` ⟨*Expresion*⟩
| ⟨*Expresion*⟩ `/` ⟨*Expresion*⟩
| ⟨*Expresion*⟩ `<` ⟨*Expresion*⟩
| ⟨*Expresion*⟩ `<=` ⟨*Expresion*⟩
| ⟨*Expresion*⟩ `=` ⟨*Expresion*⟩
| `(` ⟨*Expresion*⟩ `)`
| `NOT` ⟨*Expresion*⟩
| `ISVOID` ⟨*Expresion*⟩
| `~` ⟨*Expresion*⟩
| ⟨*Expresion*⟩ `@ TYPEID . OBJECTID ( )`
| ⟨*Expresion*⟩ `@ TYPEID . OBJECTID (` (⟨*Expresion*⟩ `,`)* ⟨*Expresion*⟩ `)`
| [⟨*Expresion*⟩ `.`] `OBJECTID (` (⟨*Expresion*⟩ `,`)* ⟨*Expresion*⟩ `)`
| [⟨*Expresion*⟩ `.`] `OBJECTID ( )`

| IF ⟨*Expresion*⟩ THEN ⟨*Expresion*⟩ ELSE ⟨*Expresion*⟩ FI
| WHILE ⟨*Expresion*⟩ LOOP ⟨*Expresion*⟩ POOL
| LET OBJECTID : TYPEID [<- ⟨*Expresion*⟩] (, OBJECTID : TYPEID [<- ⟨*Expresion*⟩])* IN ⟨*Expresion*⟩
| CASE ⟨*Expresion*⟩ OF (OBJECTID : TYPEID DARROW <Expresion>)+ ; ESAC
| NEW TYPEID
| { (⟨*Expresion*⟩ ;) + }
| OBJECTID
| INT_CONST
| STR_CONST
| BOOL_CONST

La clase *Parser* de la librería SLY proporciona un generador de analizadores sintácticos. Para la realización de un analizador sintáctico que **solo compruebe que una entrada está en el lenguaje** es necesario que cada regla no devuelva nada.

La gramática anterior es *ambigüa*, ya que algunos operadores no tienen definida la precedencia. El manual de COOL fija la precedencia de las operaciones de mayor a menor en este orden:

- .
- @
- ISVOID * / + -
- LE < = NOT
- ASSIGN

Todas las operaciones binarias son asociativas a la izquierda, excepto la asignación, que es asociativa a la derecha, y las tres operaciones de comparación, que no se asocian.

2) *Programe un analizador sintáctico utilizando la gramática anterior, teniendo en cuenta los tokens definidos en el analizador léxico. Para ello es necesario convertir la gramática a forma BNF para que sea soportada en la librería SLY. Modifique solo el fichero Parser.py.*

Cada variable está relacionada con una clase en el archivo Clase.py. Notad que la clase *Expresion* tiene varias clases hijas, dependiendo de si la expresión representa una suma, o una multiplicación, un bloque de expresiones, etc.

3) *Modifique el analizador anterior para que el analizador sintáctico devuelva un objeto de la clase Programa, si la entrada cumple la especificación dada por la gramática.*