Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
Traductores e Interpretadores
Trimestre Abril-Julio 2025

Etapa I: Análisis Lexicográfico

Esta entrega consiste en la implementación de un analizador lexicográfico para el lenguaje imperativo definido en el enunciado del proyecto. La entrega constituye la primera fase del traductor que se desea construir a lo largo del trimestre. Como entrada, debemos aceptar cualquier secuencia de caracteres, y como salida, los tokens relevantes reconocidos. Si no se reconocen los caracteres como parte de nuestro lenguaje, entonces debemos arrojar un error. Todo token debe ir acompañado de su número de fila y columna.

1. Tokens

Entre los tokens relevantes del lenguaje se encuentran:

- Cada palabra reservada del lenguaje: `if`, `while`, `end`, etc. En este caso, los tokens deben llamarse: `Tk<Palabra Clave>`, donde `<Palabra Clave>` es la palabra clave del lenguaje, con su primera letra en mayúscula. Por ejemplo, para la palabra clave `if`, su token sería `TkIf`.
- Los identificadores de variables. Todos los identificadores corresponden a un token llamado `TkId`. Este token siempre tendrá asociado como atributo el identificador reconocido. Ejemplo: al leer: `id`, su token correspondiente sería `TkId("id")`.
- Los números enteros. Ellos son agrupados bajo el token `TkNum`. Por ejemplo: `TkNum(30)`.
- Las cadenas de caracteres encerradas entre comillas. Debe ser un único *token* cuyo contenido sea la cadena leída. Ejemplo: al leer `"hola mundo"`, su token correspondiente sería `TkString("hola mundo")`.
- Las constantes booleanas. El usado para `true` será: `TkTrue` y el asociado a `false` será: `TkFalse`.
- Cada símbolo utilizado para denotar separadores, los cuales se presentan a continuación:
  - "`{`" - `TkOBlock`
  - "`}`" - `TkCBlock`
  - "`..`" - `TkSoForth`
  - "`,`" - `TkComma`
  - "`(`" - `TkOpenPar`
  - "`)`" - `TkClosePar`
  - "`:=`" - `TkAsig`
  - "`;`" - `TkSemicolon`
  - "`-->`" - `TkArrow`
  - "`[]`" - `TkGuard`
- Cada símbolo utilizado para denotar operadores aritméticos, booleanos, relacionales o de manipulación de funciones y cadenas de caracteres, los cuales se presentan a continuación:

- “+” - `TkPlus`
- “-” - `TkMinus`
- “*” - `TkMult`
- “or” - `TkOr`
- “and” - `TkAnd`
- “!” - `TkNot`
- “<” - `TkLess`
- “<=” - `TkLeq`
- “>=” - `TkGeq`
- “>” - `TkGreater`
- “==” - `TkEqual`
- “<>” - `TkNEqual`
- “[” - `TkOBracket`
- “]” - `TkCBracket`
- “:” - `TkTwoPoints`
- “.” - `TkApp`

Otras consideraciones:

- Los espacios en blanco, tabuladores, salto de línea y comentarios deben ser ignorados. Es inaceptable manejarlos como *tokens*.
- Se debe preservar la diferencia entre mayúsculas y minúsculas.

2. Formato de salida y entrada del programa

Su analizador lexicográfico debe llamarse `lexer` y recibirá como primer argumento el nombre del archivo a analizar con extensión `.imperat` (su programa `lexer` debe revisar si la extensión del archivo es correcta). La salida debe mostrar todos los tokens reconocidos de manera legible e incluyendo para cada uno, la línea y columna donde se encontró. Por ejemplo:

Suponiendo que el contenido del archivo `programa.gcl` es:

```
{
    int a, b, c;
    function[..2] d, e, f;
    a := b + 3;
    print e
    // Esto es un comentario. Debe ser ignorado.
}
```

Entonces al correr `lexer` con el argumento `programa.imperat`, se debe arrojar como salida la secuencia de tokens, todos acompañados de dos números (el primero de ellos la fila en donde se encuentra el token y el segundo la columna en donde empieza el mismo):

```
TkOBlock 1 1
TkInt 2 5
TkId("a") 2 9
TkComma 2 10
TkId("b") 2 12
TkComma 2 13
TkId("c") 2 15
TkSemicolon 2 16
TkFunction 3 5
TkOBracket 3 13
TkSoForth 3 14
TkNum(2) 3 16
TkCBracket 3 17
TkId("d") 3 19
TkComma 3 20
TkId("e") 3 22
TkComma 3 23
TkId("f") 3 25
TkSemicolon 3 26
TkId("a") 4 5
TkAsig 4 7
TkId("b") 4 10
TkPlus 4 12
TkNum(3) 4 14
TkSemicolon 4 15
TkPrint 5 5
TkId("e") 5 11
TkCBlock 7 1
```

El analizador lexicográfico solo es capaz de reconocer secuencias arbitrarias de tokens, a pesar de que esas secuencias pueden estar sintácticamente incorrectas.

En cuanto a los errores; los únicos errores detectables a nivel lexicográfico, corresponden a frases incorrectas o mal formadas, que no corresponden a ningún token válido. Por ejemplo para el programa:

```
{
    in|t a, b, c;
    function[..2] d, e, f;
    a :,= b + 3;
    print e
    // Esto es un comentario. Debe ser ignorado.
}
```

Se debe imprimir

```
Error: Unexpected character "|" in row 2, column 7
Error: Unexpected character "=" in row 4, column 9
```

Su analizador lexicográfico debe reportar todos los errores léxicos, en caso de haberlos. Cuando un error es encontrado, los tokens se hacen irrelevantes (ya que no corresponden a un programa correcto), por lo que no deben ser mostrados.

3. Tecnología

Su analizador lexicográfico debe estar programado en el lenguaje Python. La herramienta generadora de analizadores lexicográficos para Python que se utilizará, es a la vez la misma que genera analizadores sintácticos. Por lo tanto, deberán manejarse algunas nociones de gramáticas libres de contexto antes de tiempo para poder trabajar con la misma. Esta herramienta se llama *PLY* y puede ser encontrada desde la siguiente dirección Web: `http://www.dabeaz.com/ply/`

4. Detalles de la entrega

La entrega del proyecto es el viernes de la semana 4 al correo electrónico fflaviani@usb.ve. Su entrega debe incluir lo siguiente:

- Un archivo comprimido `tar.gz` con el código fuente de su proyecto, debidamente documentado. El nombre del archivo debe ser **Etapa1-XX-YY.tar.gz** donde **XX-YY** son los carné de los integrantes del grupo.

  El no cumplimiento de los requerimientos podría resultar en el rechazo de su entrega.

Federico Flaviani / fflaviani@usb.ve / Mayo 2025