# Trabajo Práctico

## Teoría de Lenguajes

Primer cuatrimestre de 2012

# 1 Introducción

En este trabajo práctico construiremos un formateador de código Jay. El mismo recibirá como entrada un programa en lenguaje Jay y escribirá como salida un archivo HTML con el código indentado según la estructura del programa, y asignándole distintos atributos visuales a los diferentes elementos sintácticos que lo componen.

# 2 Descripción del lenguaje de entrada

Jay es un lenguaje imperativo, con una sintaxis similar a la del lenguaje C. Fue definido por Tucker y Noonan (2002) para ilustrar algunos principios de los lenguajes de programación.

Al igual que en la mayoría de los lenguajes de programación modernos, los espacios en blanco, tabulaciones y saltos de línea no afectan el significado del programa. Sin embargo, se suelen respetar convenciones en la escritura para facilitar la lectura del código.

La siguiente definición léxica y sintáctica de Jay fue extraída de (Tucker y Noonan 2002, apéndice B).

## 2.1 Componentes léxicos de Jay

| | | |
|---|---|---|
| *InputElement* | → | *WhiteSpace* \| *Comment* \| *Token* |
| *WhiteSpace* | → | *space* \| `\t` \| `\r` \| `\n` \| `\f` \| `\r\n` |
| *Comment* | → | `// any string ended by \r or \n or \r\n` |
| *Token* | → | *Identifier* \| *Keyword* \| *Literal* \| *Separator* \| *Operator* |
| *Identifier* | → | *Letter* \| *Identifier Letter* \| *Identifier Digit* |
| *Letter* | → | `a` \| `b` \| ... \| `z` \| `A` \| `B` \| ... \| `Z` |
| *Digit* | → | `0` \| `1` \| `2` \| ... \| `9` |
| *Keyword* | → | `boolean` \| `else` \| `if` \| `int` \| `main` \| `void` \| `while` |

$$
\begin{array}{rcl}
\textit{Literal} & \rightarrow & \textit{Boolean} \mid \textit{Integer} \\
\textit{Boolean} & \rightarrow & \texttt{true} \mid \texttt{false} \\
\textit{Integer} & \rightarrow & \textit{Digit} \mid \textit{IntegerDigit} \\
\textit{Separator} & \rightarrow & \texttt{(} \mid \texttt{)} \mid \texttt{\{} \mid \texttt{\}} \mid \texttt{;} \mid \texttt{,} \\
\textit{Operator} & \rightarrow & \texttt{=} \mid \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{<} \mid \texttt{<=} \mid \\
 & & \texttt{>} \mid \texttt{>=} \mid \texttt{==} \mid \texttt{!=} \mid \texttt{\&\&} \mid \texttt{||} \mid \texttt{!}
\end{array}
$$

Los tokens que pueden aparecer en un programa en Jay se pueden dividir en cinco clases: *Identifier*, *Keyword*, *Literal*, *Separator* y *Operator*. El resto (*WhiteSpace* y *Comment*) es ignorado durante el análisis léxico.

## 2.2 Sintaxis de Jay

$$
\begin{array}{rcl}
\textit{Program} & \rightarrow & \texttt{void main ( )} \ \{ \ \textit{Declarations Statements} \ \} \\
\textit{Declarations} & \rightarrow & \lambda \mid \textit{Declarations Declaration} \\
\textit{Declaration} & \rightarrow & \textit{Type Identifiers} \ \texttt{;} \\
\textit{Type} & \rightarrow & \texttt{int} \mid \texttt{boolean} \\
\textit{Identifiers} & \rightarrow & \textit{Identifier} \mid \textit{Identifiers} \ \texttt{,} \ \textit{Identifier} \\
\textit{Statements} & \rightarrow & \lambda \mid \textit{Statements Statement} \\
\textit{Statement} & \rightarrow & \texttt{;} \mid \textit{Block} \mid \textit{Assignment} \mid \textit{IfStatement} \mid \\
 & & \textit{WhileStatement} \\
\textit{Block} & \rightarrow & \{ \ \textit{Statements} \ \} \\
\textit{Assignment} & \rightarrow & \textit{Identifier} \ \texttt{=} \ \textit{Expression} \ \texttt{;} \\
\textit{IfStatement} & \rightarrow & \texttt{if (} \ \textit{Expression} \ \texttt{)} \ \textit{Statement} \mid \\
 & & \texttt{if (} \ \textit{Expression} \ \texttt{)} \ \textit{Statement} \ \texttt{else} \ \textit{Statement} \\
\textit{WhileStatement} & \rightarrow & \texttt{while (} \ \textit{Expression} \ \texttt{)} \ \textit{Statement} \\
\textit{Expression} & \rightarrow & \textit{Conjunction} \mid \textit{Expression} \ \texttt{||} \ \textit{Conjunction} \\
\textit{Conjunction} & \rightarrow & \textit{Relation} \mid \\
 & & \textit{Conjunction} \ \texttt{\&\&} \ \textit{Relation} \\
\textit{Relation} & \rightarrow & \textit{Addition} \mid \\
 & & \textit{Relation} \ \texttt{<} \ \textit{Addition} \mid \\
 & & \textit{Relation} \ \texttt{<=} \ \textit{Addition} \mid \\
 & & \textit{Relation} \ \texttt{>} \ \textit{Addition} \mid \\
 & & \textit{Relation} \ \texttt{>=} \ \textit{Addition} \mid \\
 & & \textit{Relation} \ \texttt{==} \ \textit{Addition} \mid \\
 & & \textit{Relation} \ \texttt{!=} \ \textit{Addition} \\
\textit{Addition} & \rightarrow & \textit{Term} \mid \\
 & & \textit{Adition} \ \texttt{+} \ \textit{Term} \mid \\
 & & \textit{Adition} \ \texttt{-} \ \textit{Term} \\
\textit{Term} & \rightarrow & \textit{Negation} \mid \\
 & & \textit{Term} \ \texttt{*} \ \textit{Negation} \mid \\
 & & \textit{Term} \ \texttt{/} \ \textit{Negation} \\
\textit{Negation} & \rightarrow & \textit{Factor} \mid \texttt{!} \ \textit{Factor} \\
\textit{Factor} & \rightarrow & \textit{Identifier} \mid \textit{Literal} \mid \texttt{(} \ \textit{Expression} \ \texttt{)}
\end{array}
$$

## 3 Descripción de la salida

La salida será un documento HTML (`http://www.w3.org/TR/html4`), utilizando hojas de estilo externas (`http://www.w3.org/TR/REC-CSS2/`). El código del programa de entrada deberá aparecer formateado con una línea por sentencia, y con cada línea indentada según el nivel de anidación del bloque de código al que pertenece. A cada clase de token se le deben poder asignar distintos atributos visuales.

Una manera de implementar esto último es que cada token quede incluido en un elemento de tipo `SPAN` que indique de qué clase es. Por ejemplo, `<SPAN class="Key">if<\SPAN>`, `<SPAN class="Lit">32<\SPAN>`, etc.

Para indentar el código se puede usar un elemento de tipo `DIV` de una clase para la que se defina un margen izquierdo. Por ejemplo:

```
<SPAN class="Sep">{</SPAN>
<DIV class="CodeBlock">
este código irá indentado
</DIV>
<SPAN class="Sep">}</SPAN>
```

El siguiente es un ejemplo de hoja de estilo para visualización en pantalla que produce un aspecto similar al de los viejos entornos de Borland:

```
BODY { background-color: navy; font-family: monospace }
DIV.CodeBlock { margin-left: 2 em }
SPAN.Id { color: yellow }
SPAN.Key { color: white }
SPAN.Lit { color: fuchsia }
SPAN.Sep { color: white }
SPAN.Op { color: white }
```

Para incorporar la información de estilo, el documento HTML podría tener la siguiente estructura:

```
<HTML>
<HEAD>
  <LINK href="jay.css" rel="stylesheet" type="text/css">
</HEAD>
<BODY>
código del programa
</BODY>
</HTML>
```

# 4 Implementación

Hay dos grupos de herramientas que se pueden usar para generar los analizadores léxico y sintáctico, y agregar las acciones requeridas para generar el documento de salida:

- Uno utiliza expresiones regulares y autómatas finitos para el análisis lexicográfico y la técnica LALR para el análisis sintáctico. Ejemplos de esto son `lex` y `yacc`, que generan código C o C++, o `JLex` y `CUP`, que generan código Java.
  `flex` y `bison` son implementaciones libres y gratuitas de `lex` y `yacc`. Son parte de todas las distribuciones Linux. Para Windows pueden ser instalados como parte de Cygwin, (`http://www.cygwin.com/`) o de DJGPP (`http://www.delorie.com/djgpp/`). También se pueden conseguir en forma independiente en `http://www.gnu.org/`.
  `JLex` y `CUP` se pueden conseguir en
  `http://www.cs.princeton.edu/~appel/modern/java/`

- El otro grupo utiliza la técnica LL(k) tanto para el análisis léxico como para el sintáctico, generando parsers descendentes recursivos. Ejemplos son `JavaCC`, que genera código Java, y `ANTLR`, que está escrito en Java pero puede generar código Java, C++ o C#. `ANTLR` se puede conseguir en `http://www.antlr.org/` y `JavaCC` en
  `http://javacc.java.net/`.

La entrega debe incluir:

- Un programa que reciba como entrada un programa en Jay y verifique si es léxica y sintácticamente correcto. En ese caso, debe producir la salida descripta. En caso contrario debe informar la ubicación del error.

- El código fuente que se le ingresa a las herramientas generadoras de analizadores.

- Ejemplos de programas correctos e incorrectos, resultados obtenidos en ambos casos y conclusión.

# Referencias

[1] A. Tucker, R. Noonan. *Programming Languages: Principles and Paradigms.* Mc Graw Hill, 2002.