Consideraciones de YALex

Descripción

YALex [Yet Another Lex] se encuentra basado al estilo de Lex y toma inspiración sobre la implementación de `ocamllex`, herramienta escrita para OCaml. El objetivo principal de YALex es la generación de Analizadores Léxicos a partir de una Definición Regular o conjunto de Expresiones Regulares**.** La llamada esperada sería:

```
yalex lexer.yal -o thelexer
```

Donde `lexer.yal` es un archivo escrito en Lenguaje YALex. Esta instrucción genera un archivo [`thelexer`] escrito en el lenguaje a su elección, el cuál implementa el Analizador Léxico a partir de lo definido en `lexer.yal`. Este archivo se utilizará en conjunto con el parser generador por YAPar para implementar de forma completa un módulo de parseo o traducción.

Estructura de Archivo [yal]

La estructura de un archivo escrito en YALex es la siguiente:

```
{ header }
let ident = regexp …
rule entrypoint [arg1… argn] =
      regexp { action }
      | …
      | regexp { action }
{ trailer }
```

- Un archivo en YALex acepta comentarios delimitados por (* y *)
- La sección de {header} y {trailer} son secciones opcionales. Si estas secciones existen, su contenido es copiado al principio y al final del archivo generado.
- Entre {header} y `entrypoint` se pueden especificar nombres a expresiones regulares de uso muy frecuente en la especificación de tokens [`let ident = regexp`], los cuales pueden ser utilizados luego en la definición de posteriores expresiones regulares en el `entrypoint`.
- *entrypoint* es el identificador de una función dentro del archivo generado, que acepta n argumentos arg1, ..argn que servirá para el punto de entrada del reconocedor de componentes léxicos. Esta función será la responsable de consumir el buffer de símbolos o caracteres de entrada, los cuales buscarán tener concordancia con algún patrón definido por *regexp*. Al encontrar una concordancia de un prefijo de la entrada con una expresión regular, la función ejecutará la acción [action] correspondiente.
- Si un prefijo hace concordancia con más de una expresión regular, se buscará siempre el lexema más largo que haga concordancia. En el caso de empate se priorizará por orden de definición de las expresiones.

Expresiones Regulares

La escritura de las expresiones regulares se basan a un estilo similar a Lex, tomando en cuentas las siguientes consideraciones léxicas y sintácticas:

- 'regular-char | escape-sequence'
  - Denota un símbolo constante o una secuencia de escape válida.
- _
  - Denota cualquier símbolo
- "string-character"
  - Denota una constante tipo cadena. Puede contener secuencias de escape válidas.
- [character-set]
  - Denota un conjunto de símbolos. Un conjunto válido de símbolos puede denotar como un símbolo simple, *'c'*; como un rango de símbolos *'c1'* - *'c2'* [todos los símbolos entre c1 y c2 inclusive]; o la unión de dos o más símbolos denotados por medio de una constante tipo cadena, "abcd" [conjunto que contiene a, b, c, d]
- [^character-set]
  - Denota cualquier símbolo que no pertenece al *character-set*
- *regexp1 # regexp2*
  - Dado que *regexp1* y *regexp2* son *character-set*, denota el confjunto diferencia entre *regexp1* y *regexp2*.
- *regexp**
  - Cerradura Kleene de *regexp*
- *regexp+*
  - Cerradura Positiva de *regexp*
- *regexp?*
  - Define la existencia de *regexp* o *cadena vacía.*
- *regexp1 | regexp2*
  - Define *regexp1 ó regexp2*
- *regexp1 regexp2*
  - Concatenación de *regexp1* y *regexp2*
- *(regexp)*
  - Define *regexp*
- *ident*
  - Representa la expresión regular definida previamente por `let ident = regexp`

La precedencia de los operadores es la siguiente:

- # mayor precedencia
- *, +, ?
- Concatenación
- |

Acciones

El segmento {action} puede ocurrir o no dentro de un archivo yal. La ocurrencia representa instrucciones en código de programación (alineado al lenguaje de programación elegido)

arbitrarias que se deben de ejecutar al momento de encontrar una concordancia con la expresión regular que acompaña.

Ejemplo [bajo el supuesto de generar código Python]

```
(* ejemplo.yal *)
  {
  import myToken
  }
  rule gettoken =
      [' ' '\t']      { return lexbuf }      (* skip blanks *)
    | ['\n' ]         { return EOL }
    | ['0'-'9']+      { return int(lxm) }
    | '+'             { return PLUS }
    | '-'             { return MINUS }
    | '*'             { return TIMES }
    | '/'             { return DIV }
    | '('             { return LPAREN }
    | ')'             { return RPAREN }
    | eof             { raise( 'Fin de buffer' )}
```