Tarea Programada Número 1

Sintaxis de Triangle

```
Command             ::=  single-Command
                       |  Command ; single-Command

single-Command      ::=  V-name := Expression
                       |  Identifier ( Actual-Parameter-Sequence )
                       |  begin Declaration in single-Command
                       |  if Expression
                            then single-Command
                            else single-Command
                       |  while Expression do single-Command

Expression          ::=  second-Expression
                       |  let Declaration in Expression
                       |  if Expression then Expression else Expression

second-Expression   ::=  primary-Expression
                       |  second-Expression Operator primary-Expression

primary-Expression  ::=  Integer-Literal
                       |  Character-Literal
                       |  V-name
                       |  identifier ( Actual-Parameter-Sequence )
                       |  Operator primary-Expression
                       |  ( Expression )
                       |  { Record-Aggregate }
                       |  [ Array-Aggregate ]

record-Aggregate    ::=  Identifier ~ Expression
                       |  Identifier ~ Expression , record-Aggregate

array-Aggregate     ::=  Expression
                       |  Expression , array-Aggregate

V-name              ::=  Identififer
                       |  V-name . Identifier
                       |  V-name [ Expression ]
```

```
Declaration          ::=  single-Declaration
                       |  Declaration ; single-Declaration

single-Declaration   ::=  const Identifier ~ Expression
                       |  var Identifier : Type-denoter
                       |  proc Identififer ( Formal-Parameter-Sequence )
                            ~ single-Command
                       |  func Identififer ( Formal-Parameter-Sequence )
                            : Type-denoter ~ Expression
                       |  type Identifier ~ Type-denoter

Formal-Param-Seq     ::=  ϵ
                       |  proper-FP-Sequence

proper-FP-Sequence   ::=  Formal-Parameter
                       |  Formal-Parameter , proper-FP-Sequence

Formal-Parameter     ::=  Identifier : Type-denoter
                       |  var Identifier : Type-denoter
                       |  proc Identifier ( Formal-Param-Seq )
                       |  func Identifier ( Formal-Param-Seq )
                            : Type-denoter

Actual-Param-Seq     ::=  ϵ
                       |  proper-AP-Sequence

proper-AP-Sequence   ::=  Actual-Parameter
                       |  Actual-Parameter , proper-AP-Sequence

Actual-Parameter     ::=  Expression
                       |  var V-name
                       |  proc Identifier
                       |  func Identifier

Type-denoter         ::=  Identifier
                       |  array Integer-Literal of Type-denoter
                       |  record Record-Type-denoter end

Record-Type-denoter  ::=  Identifier : Type-denoter
                       |  Identifier : Type-denoter , Record-Type-denoter
```

Lexicon/Tokens de Triangle

```
Program         ::= (Token | Comment | Blank)*
Token           ::= Integer-Literal | Character-Literal | Identifier | Operator
                  | array | begin | const | do | else | end
                  | func | if | in | let | of | proc | record
                  | then | type | var | while
                  | . | : | ; | , | := | ~ | ( | ) | [ | ] | { | }
Integer-Literal ::= Digit Digit*
Char-Literal    ::= 'Graphic'
Identifier      ::= Letter ( Letter | Digit )*
Operator        ::= Op-character Op-character*
Comment         ::= ! Graphic* end-of-line
Blank           ::= space | tab | end-of-line
Graphic         ::= Letter | Digit | Op-character | space | tab |…
Letter          ::= a..z | A..Z
Digit           ::= 0..9
Op-character    ::= + | - | * | / | = | < | > | \ | & | @ | % | ^ | ?
```

Actividad

En esta tarea se provee tanto la gramática de los lexicons (tokens) como del lenguaje en general. Para desarrollar esta tarea usted debe hacer dos actividades.

1. Implementar un programa escrito "`tokenize`" en Rust que tome como input en la linea de comandos un archivo de texto y escriba en un archivo de salida, que por defecto llamaremos `tokens.out`, la secuencia de tokens leído del archivo de texto original. Si por alguna razón el archivo produce error, debe reportar dicho error al standard error y abortar el proceso de compilación.
    1. Su programa debe aceptar redireccionar la salida a otro archivo mediante la directiva -o
       ```
       > tokenize input.tri –o output.tok
       ```
    2. Cada token debe incluir la linea y columna del archivo fuente del cual fue leído para futuro reporte de errores por una subsiguiente fase del compilador.
2. Implementar un programa "`tokens`" que acepte en la linea de comandos un archivo de tokens y genere un listado en el standard output de los tokens leídos con sus respectivos tipos asociados, y la posición en que se encuentran dentro del archivo fuente.
    1. Si el programa list no recibe parámetro en la linea de comandos debe asumir que el archivo a leer es `tokens.out`
3. Modificar la gramática de Triangle para que pueda ser procesada por un parser LL(1), es es:
    1. Quede en formato EBNF
    2. Factorice por la izquierda todos las reglas
    3. Elimine recursión por la izquierda

Los primeros dos puntos son programas en Rust que usted debe programar, el tercero es un documento en PDF que debe entregar conjunto a los otros programas, todo en un archivo .zip.

Bajo los estándares de Rust, puede hacer que los dos primeros

Fecha de Entrega

13 de Septiembre 2024, 3pm