"El saber de mis hijos
hará mi grandeza"

Universidad de Sonora

Lexer Artesanal

Lenguajes de Programación

Gutierrez Navarro Gustavo

October 7, 2023

1 Descripción de las funciones del *Lexer Artesanal*:

1.1 read-char* input

- **Función:**
  Dado un valor de entrada *input* se llama a la función $read-char$ proveniente del propio Racket y lo que hace es guardar el valor del primer carácter en la variable *ch* y lo elimina para el flujo de datos que se esta leyendo.Posteriormnete se devuelve el valor de *ch*.
  Adicionalmnete, si la variable global *debug*? esta con valor verdadero, se imprime en pantalla un mensaje que informa al usuario el dato de entrada que recibio y el carácter que leyó.

- **Errores:**

1.2 peek-char* input

- **Función:**
  Esta función opera exactamente igual que la anterior, excepto por la función que se llama al momento de leer el carácter. A diferencia de $read-char$, $peek-char$ no consume el carácter que se lee.

- **Errores:**
  Los mismos que $read-char*$.

1.3 char-digit? ch

- **Función:**
  Conceptualmente, determina si la variable *ch* (con valor esperado de un carácter) es un dígito. La forma en que lo hace es mediante la función *char <=?* que determina si los parámetros dados se encuentran de manera no descendiente, esta particularidad se usa para colocar a la variable *ch* en medio del carácter 0 y el 9, por lo tanto, en teoría solo aceptará valores que se encuentren en este rango, es decir dígitos.

- **Errores:**

1.4 char-varletter? ch

- **Función:**
  Determina si un carácter es una variable valida, en este caso puede ser x, y o z, si lo es devuelve a la lista apartir del elemento encontrado, en caso contrario devuelve falso.

- **Errores:**

1.5 char-delimiter? ch

- **Función:**
  Revisa un carácter y checa si es un delimitador. Los simbolos que se consideran delimitadores son el carácter especial eof, espacio en blanco o los paréntesis. El primer delimitador se verifica con la función $eof - object?$ de Racket, el segundo con otra función de Racket llamada $char - whitespace?$ y por ultimo el tercero con la función *member* (previamente explicada en la función anterior) usando como paramétro una lista que contiene los dos paréntesis (abierto y cerrado).

- **Errores:**

1.6 token-open-paren

- **Función:**
  Crea una estructura de tipo token de tipo $open - paren$ con valor falso y dentro de los campos de fila y columna escribe los que se encuentran actualmente en las variables globales de $lex - line$ y $lex - col$ haciendo referencia a la posición del lexer dentro del documento. Finalmnete devuelve la estructura creada.

- **Errores:**

1.7 token-close-paren

- **Función:**
  Crea un token muy similar al anterior con todos los campos iguales excepto por el primer parámetro ya que este token es del tipo $close - paren$. Al finalizar se regresa la estructura.

- **Errores:**

1.8 token-binop type

- **Función:**
  Se crea un token de tipo *binop* correspondiente a una operación binaria, en este caso tambien se pide el tipo, es decir que operación es, ya que puede ser suma o multiplicación, este tipo que es pasado como parámetro en la función es escrito dentro de la estructura, las filas y columnas sigue en mismo estilo que las dos funciones anteriores y tambien se devuelve.

- **Errores:**

1.9 token-number value col

- **Función:**
  Se crea una estructura de tipo token y se devuelve. Esta estructura es de tipo *number*, su valor esta determinado mediante el valor pasado como parámetro en la función, la linea en donde se encuentra coincide con la del $lex - line$, sin embargo el campo de columna esta ocupado por un parámetro *col* que es pasado a la función, se hace de esta manera para poder iterar dentro del número, ya que esta compuesto de varios digitos.
- **Errores:**

1.10 token-number/+ token

- **Función:**
  Dada una estructura token de tipo *number* pasada como parámetro, se crea una copia de la estructura con los mismos valores, excepto por el valor de la columna, el cual actualiza su valor al restarle 1. Esta función permite eliminar un posible signo + que pudiera estar escrito en el documento pero que para realizar operaciones seria un estorbo. Al finalizar se devuelve la estructura creada.
- **Errores:**
  El parámetro *token* de la función eclipsa a la estructura *token* por lo que, no permite crear un token.

1.11 token-number/- token

- **Función:**
  Siguiendo la misma lógica que la función pasada, sin embargo ahora el valor númerico si se ve afectado, ahora se agrega el signo - para que internamente, Racket lo pueda manejar como un número negativo.
- **Errores:**
  El parámetro *token* de la función eclipsa a la estructura *token* por lo que, no permite crear un token.

1.12 token-identifier symbol col

- **Función:**
  Crea un token de tipo $identifier$ con el simbolo que es pasado como parámetro, ademas como ocurria con la lectura de números, la linea de lectura permanece igual a la global, sin embargo la columna es dada como parámetro para tener mas libertad de iteración debido a que no consta de un simple carácter. Se devuelve el token.
- **Errores:**

1.13 token-define col

- **Función:**
  Siendo del mismo tipo que las anteriores funciones para crear tokens, en este caso su tipo corresponde a $define$ con valor de verdad falso, columna global por parte de $lex - line$ y una columna arbitraria pasada como parámetro y finalmente se devuelve.

- **Errores:**

1.14 lex-open-paren chars

- **Función:**
  Sirve como un lexer de las expresiones de tipo paréntesis abierto. Comienza leyendo el char, crea un token de tipo paréntesis abierto, cambia la columna del lector a 1 mas de la previa y posteriormente llama a la función $stream-cons$ con el token creado y la llamada a función de lex con el parámetro $chars$. Es importante recalcar que al momento de leer los carácteres se utiliza la función previamente definida en este archivo $read-char*$, dicha función aparte de leer el carácter se encarga de borrarlo, por lo tanto al volver a llamar a la función lex, los carácteres que van a ser evaluados seran los mismos que antes pero, eliminando lo que se acaba de leer, y sumando el $cons$, consigue hacer una recursión para en teoría terminar con una lista de tokens.

- **Errores:**

1.15 lex-close-paren chars

- **Función:**
  Hace exactamnete lo mismo que la función anterior, pero se crea un token de tipo paréntesis cerrado y no un paréntesis abierto.

- **Errores:**

1.16 lex-whitespace chars

- **Función:**
  Esta función se encarga de procesar los espacios en blanco. Comienza leyendo el caracter, posteriormente se pregunta si el carácter que se acaba de leer es $n\#newline$ (un salto de linea), si es el caso, entonces las variables globales que se encargan de la posicion del lexer cambian, ahora se cambia a la siguiente fila (se suma 1 al valor actual) y se pone en 0 la columna; en otro caso simplemente se salta a la siguiente columna y finalmente llama denuevo a lex con el flujo de carácteres. En resumen, lee un espacio en

blanco, si es un salto de linea se pasa a leer a la siguiente fila, y si no lo es, entonces simplemente se brinca al siguiente carácter.

- **Errores:**

1.17 lex-sum-or-number chars

- **Función:**
  Similar a las funciones pasadas, pero un poco mas complicado. En este caso un simbolo + puede significar dos cosas, el operador binario de suma o un número positivo. Comienza igual que las otras funciones del mismo tipo, leyendo el carácter (y borrandolo del flujo), posteriormente se revisa el siguiente carácter en el flujo (mediante *peek* y no *read*) y se guarda en una variable auxiliar *ch*. Si *ch* es un digito (se comprueba mediante la función $char-digit?$) entonces significa que presuntamente se esta leyendo un número que empieza con el signo +, entonces, para empezar se coloca la columna en donde empieza el número (uno mas de donde esta el signo mas) y despues se llama a la función $lex-plain-number$ que en resumen se encarga de generar el token que queremos del número, para finalmente seguir mandando a $stream-cons$ con el token y la recursión, sin embargo aqui se realiza mediante un flujo de datos diferente al que entro, esto se debe a que internamnete, el método para identificar números es un poco mas complejo ya que no son cáracteres individuales ; si no empieza con un digito, eso quiere decir que se trata del operador de suma, por lo tanto se crea un token correspondiente a dicho simbolo y se hace el protocolo estandar, pasar a la siguiente columna y llamar recursivamente a $stream-cons$ con el token creado y la llamada a función de *lex*.

- **Errores:**
  Al momento de dar un signo singular signo de +, el algoritmo asume que lo que se le dara a continauación es un carácter, por lo tanto no contempla que podria ser el fin del archivo.

1.18 lex-negative-number chars

- **Función:**
  Funciona casi igual que el anterior, las pequeñas diferencias son ciertas llamadas a funciones y los casos del condicional, para empezar al momento de detectar el cáracter, tambien se guarda la columa en donde se inicio para un posible error que se vera mas adelante, ademas de esto, al crear el token del número, se llama a la función $token-number/-$ para que internamente se maneje como un número negativo, la última diferencia se encuentra en que si el despues de leer el carácter - no se encuentra un digito, marca un error, esto debido a que en este lenguaje la operación de la resta no esta implementada (aquí se usa la variable de *col* para indicar donde se encontro el error).

- **Errores:**

1.19 lex-mult chars

- **Función:**
  Siguiendo con la estructura de los operadores, en este caso el signo de la multiplicación solo puede significar una y solo una cosa (la operación binaria), por lo tanto obligatoriamente, despues del simbolo, debe de ir un delimitador, si lo hay entonces se crea el token y se sigue la función recursivamente; si no tira un error.

- **Errores:**

1.20 lex-identifier-or-keyword chars

- **Función:**
  Se crea una función auxiliar que tendra como objetivo leer la cadena de texto y ademas indicar si se trata de una palabra clave o un identificador. La manera en la que esta función trabaja es la siguiente: Recibe un puerto de cadenas que servirara como un bloc de notas para ir escribiendo los carácteres que llevamos y ademas un valor booleano para saber si lo que se esta leyendo es un identificador. Al inicio del proceso, se lee (y consume) el primer cáracter del flujo de *chars* y se escribe en el puerto de cadenas, se avanza en uno la columna y empieza el sistema de condiciones. Se da un vistazo al siguiente carácter y dependiendo del contenido de este se toma una decisión.

  - Si es un delimitador, se devuelve el valor de $is-identifier?$ por que ya se termino de leer la cadena.

  - Si es un digito, simplemente se sigue leyendo la cadena mandando a llamar otra vez a la función con los mismos parametros (posiblemente mutados como strport).

  - Si es una letra aceptada como variable, entonces se la misma forma se manda a llamar recursivamente para contiunuar leyendo.

  - Si no es ninguna, entonces quiere decir que no es un identificador ya que no cumple con la condición definida previamente, por lo tanto se continua leyendo la cadena pero con valor de verdad falso ya que no es un identificador (podria ser una palabra clave)

De esta manera, la función se encarga de dos cosas, tener la cadena de texto leida y saber si es un identificador o no. Ahora si, dando comienzo a la función en si, se declaran varias variables, para comenzar se marca la columna por si llega a ocurrir un error poder referenciarlo, tambien se abre un puerto de cadenas, despues se llama a la función auxiliar $is-identifier$ con el puerto recien abierto y con el valor de verdad preguntando si el primer carácter del texto corresponde a una letra valida para ser variable (x, y o z), una vez leida la informacion, es almacenada en la variable *str* mediante una función *get* para los puertos de cadenas.Finalmente, la función termina con una condicional:

- Si la variable $is-identifier?$ es verdadera, eso quiere decir que se leyo el texto y se encontro que fue valida, por lo tanto se crea un token de tipo $identifier$ y se llama recursivamente a la función $stream-cons$ y lex para continuar la lectura de todo el documento.

- Si no es una variable, entonces tiene que ser una palabra clave, por el momento la única palabra clave que se encuentra definida es $define$, por lo tanto se comprueba si str es $define$ y si lo es se crea un token de tipo $token-define$ y se sigue el procedimiento recursivo de la clausula anterior.

- Si no es ninguno de estos marca error por que se esperaba un define o un identificador.

- **Errores:**
  Al leer los carácteres, el resultado de la función resulta contraproducente al momento de generar la cadena de texto.

1.21 lex-plain-number chars

- **Función:**
  Esta función es la encargada de leer el número compuesto de varios digitos para posteriormente devolverlo en forma de token. Para realizar esto se tiene definida una función interna llamada $build-integer$ que recibe como único parámetro *value*, esta función auxiliar se encarga de primero leer el carácter y lo transforma a entero, despues avanza la columna para que coincida con el carácter que se acaba de leer, ahora mediante un *peek* se ve el siguiente elemento para entrar en una serie de condicionales.

  - Si es un delimitador devuelve la suma de *value* multiplicado por 10 y el dígito actual.

- Si es un dígito, se manda a llamar a la misma función recursivamente ($build - integer$) pero con el parámetro de la suma de $value$ multiplicado por 10 y el dígito actual.

- Si no es ninguan de las dos, es un error y lo marca.

Mediante el uso de la función previamente descrita, se manda a llamar con el valor inicial de 0 para calcular el valor del número que se quiere leer, posteriormente se crea un token de número con todas las especificaciones y se devuelve siguiendo la regla para continuar con el flujo de datos.

- **Errores:**

1.22 lex-end-of-file chars

- **Función:**
  Esta función es llamda cuando se encuentra el fin del archivo y sirve para cerrar el canal que se abrio cuando se abrio el archivo de texto (o cadena) por primera vez, esto se hace mediante la función $close - input - port$. Al finalizar devuelve un $empty - stram$ que es simplemente un tipo de dato proporcionado por Racket que devuelve un flujo de datos vacio.

- **Errores:**

1.23 signal-lex-error message line col

- **Función:**
  A traves de un mensaje de error, una linea y una columna, esta función devuelve un error con un mensaje diseñado para indicar el tipo de error y en que posición se encontro, correspondiente a la información pasada en los parámetros.

- **Errores:**

1.24 lex chars

- **Función:**
  Simplemente a traves de hacer un *peek* al siguiente carácter en el flujo, determina cual de todos los lexers hay que usar.

- **Errores:**

1.25 lex-from-file path

- **Función:**
  Comienza la lectura del lexer dentro de un archivo de texto. Inicialmnete se cambia el valor de $lex - path$ para saber la dirección del archivo a leer. Se cambia a la primera linea y se llama a la función lex con el parámetro de llamar a la función $open - input - file$, esta función se encarga de abrir un canal de comunicación para poder leer el documento, esto se hace con el *path* que fue pasado como parámetro y ademas se abre especificando que es un documento de texto.

- **Errores:**

1.26 lex-from-string str

- **Función:**
  Se encarga de iniciar la lectura del lexer a partir de una cadena. Establece la variable $lex - line$ en 1 para dar inicio con la lectura del primer renglón y llama a la función lex, usando al función $open - input - string$ para mandar como parámetro a la cadena como si fuera un flujo de datos, imitando el funcionamiento de un archivo.

- **Errores:**

2 Descripción de las variables del *Lexer Artesanal*:

2.1 debug? #f

- **Función:**
  Esta variable global tiene la funcionalidad de indicar cuando el usuario quiere imprimir en consola las acciones que se estan realizando internamente, es por esto que se llama *debug*?, ya que actúa como una herramienta para entender el funcionamiento del programa y encontrar posibles fallos.

- **Errores:**

2.2 lex-path "<unknown>"

- **Función:**
  Sirve para tener guardado la dirección del archivo en caso de que se este leyendo un archivo de texto.

- **Errores:**

2.3 lex-line 0

- **Función:**
  Es una variable global que sirve para identificar en que renglón o fila se encuentra el lexer dentro del documento, es inciada con 0 para indicar que todavia no se ha iniciado la lectura.
- **Errores:**

2.4 lex-col 0

- **Función:**
  Es una variable global que sirve para identificar en que columna se encuentra el lexer dentro del documento, es inciada con 0 para indicar que todavia no se ha iniciado la lectura.
- **Errores:**

2.5 zero-char-val char->integer #\0

- **Función:**
  Se encarga de definir el 0 para su utilización en otras funciones.
- **Errores:**

3 Descripción de la estructura del *Lexer Artesanal*:

3.1 token

- **Función:**
  Es una estructura (como las vistas en C) que se encarga de almacenar 4 datos. El tipo del token, su valor, su linea y columna (estos ultimos dos son referentes a la documento que se esta revisando). Este token se usara dentro de las funciones de forma auxiliar.
- **Errores:**

4 Errores adicionales del *Lexer Artesanal*:

4.1 provide

- **Función:**
  Indica los elementos del archivo que seran exportados cuando se use en forma de modulo.

- **Errores:**
  La estructura token no esta definida, por lo que al momento de intentar crear un token fuera del archivo lexer.rkt da error.