Lab 1. Implementando scanners

Compiladores

Objetivo

Implementar un analizador léxico (scanner) para el lenguaje propuesto (*ChocoPy*).
Fecha de entrega: 12 - abril - 2023

Estructura léxica

Un análisis léxico lee un archivo de entrada y produce una secuencia de tokens. Los tokens se encuentran en la cadena de entrada mediante reglas léxicas que se expresan mediante expresiones regulares.
Cuando existe ambigüedad, un token comprende la cadena más larga posible que forma un token legal, cuando se lee de izquierda a derecha.
Existen las siguientes categorías de tokens:

- Estructura de línea.
- Identificadores.
- Palabras clave.
- Literales.
- Operadores y Delimitadores.

Estructura de línea

En ChocoPy, como en Python, el espacio en blanco puede ser significativo tanto para terminar una declaración como para reconocer el nivel de indentación de una instrucción.
Para acomodar esto, ChocoPy define tres tokens léxicos que se derivan del espacio en blanco: **NEWLINE, INDENT** y **DEDENT**.

1. **Líneas físicas**: Una línea física es una secuencia de caracteres terminada por una secuencia de final de línea (**\n**). El final de la entrada también sirve como un terminador implícito para la línea física final.

2. **Líneas lógicas**: Una línea lógica es una línea física que contiene por lo menos un token que no es un whitespace o un comentario. El final de la línea lógica se representa por el token **NEWLINE**. Los enunciados (statements) no pueden cruzar los límites de la línea lógica, excepto cuando la sintaxis permite **NEWLINE** (por ejemplo, entre instrucciones en estructuras de flujo de control, como bucles while).

3. **Comentarios:** Un comentario comienza con un carácter hash **(#)** que no forma parte de un string y termina al final de una línea física. **Los comentarios son ignorados por el analizador léxico; No se emiten como tokens.**

4. **Líneas en blanco:** Una línea física que contiene solo espacios, tabs y, posiblemente, un comentario, se ignora (es decir, **no se genera token NEWLINE**).
5. **Indentación:** Los whitespace al inicio de una línea lógica son usados para calcular el nivel de indentación de una línea, usado para determinar la agrupación de los enunciados. Las tabulaciones se reemplazan (de izquierda a derecha) por uno a ocho espacios de tal manera que el número total de caracteres hasta e incluyendo el reemplazo es un múltiplo de ocho (esto pretende ser la misma regla que usa Unix). El número total de espacios que preceden al primer carácter que no está en blanco determina la indentación de la línea. Los niveles de indentación de líneas consecutivas se utilizan para generar tokens **INDENT y DEDENT**, utilizando un stack.
    a. *Before the first line of the input program is read, a single zero is pushed on the stack; this will never be popped off again. The numbers pushed on the stack will always be strictly increasing from bottom to top. At the beginning of each logical line, the line's indentation level is compared to the top of the stack. If it is equal, nothing happens. If it is larger, it is pushed on the stack, and one INDENT token is generated. If it is smaller, it must be one of the numbers occurring on the stack; all numbers on the stack that are larger are popped off, and for each number popped off a DEDENT token is generated. At the end of the input program, a DEDENT token is generated for each number remaining on the stack that is larger than zero."*

6. **Espacios entre tokens:** Los espacios (o tabs) que no están al inicio de línea no son tokens y son ignorados. Sólo es necesario escribir espacios entre dos tokens si su concatenación se interpreta como un nuevo token.

Identificadores

- **[a-z|A-Z][a-z|A-Z|0-9|_]***

Keywords

- ***False, None, True, and, as, assert, async, await, break, class, continue, def, del, elif, else, except, finally, for, from, global, if, import, in, is, lambda, nonlocal, not, or, pass, raise, return, try, while, with, yield***

Literales

- ***Strings***: Los literales de cadena son simplemente una secuencia de caracteres ASCII delimitados por (e incluyendo) comillas dobles: "...". Los caracteres ASCII deben estar dentro del rango decimal 32-126 inclusive, es decir, mayor o igual que el carácter de espacio y hasta tilde. El string puede contener comillas dobles usando una barra diagonal inversa **\"**. Las demás secuencias de escape no son válidas.

  "Hell\o" nos devuelve : (error: "\o" not recognized)

- ***Integers***: Los literales enteros en ChocoPy se componen de una secuencia de uno o más dígitos 0-9, donde el dígito más a la izquierda sólo puede ser 0 si es el único carácter de la secuencia. Es decir, los literales enteros con valores distintos de cero no pueden tener ceros iniciales. El valor entero de tales literales se interpreta en base 10. El máximo interpretado El valor puede ser $2^{31} - 1$ (2147483647). Un literal con un valor mayor que este límite da como resultado un error léxico.

Operadores y delimitadores

- **+**
- **-**
- *****
- **//**
- **%**
- **<**
- **>**
- **<=**
- **>=**
- **==**
- **!=**
- **=**
- **(**
- **)**
- **[**
- **]**
- **,**
- **:**
- **.**
- **->**

Gramática

program ::= ⟦*var_def* | *func_def* | *class_def*⟧* *stmt**

class_def ::= **class** ID (ID) : NEWLINE INDENT *class_body* DEDENT

class_body ::= pass NEWLINE
| ⟦*var_def* | *func_def*⟧+

func_def ::= **def** ID (⟦*typed_var* ⟦, *typed_var*⟧*⟧?) ⟦-> type⟧? : NEWLINE INDENT *func_body* DEDENT

func_body ::= ⟦*global_decl* | *nonlocal_decl* | *var_def* | *func_def*⟧* *stmt*+

typed_var ::= ID : *type*

type ::= ID | IDSTRING | [*type*]

global_decl ::= **global** ID NEWLINE

nonlocal_decl ::= **nonlocal** ID NEWLINE

var_def ::= *typed_var* = *literal* NEWLINE

stmt ::= *simple_stmt* NEWLINE
| **if** *expr* : *block* ⟦**elif** *expr* : *block* ⟧* ⟦**else** : *block*⟧?
| **while** *expr* : *block*
| **for** ID **in** *expr* : *block*

simple_stmt ::= **pass**
| *expr*
| **return** ⟦*expr*⟧?
| ⟦ *target* = ⟧+ *expr*

block ::= NEWLINE INDENT *stmt*+ DEDENT

literal ::= **None**
| **True**
| **False**
| INTEGER
| IDSTRING | STRING

expr ::= *cexpr*
| **not** *expr*
| *expr* ⟦**and** | **or**⟧ *expr*
| *expr* **if** *expr* **else** *expr*

cexpr ::= ID
| *literal*
| [⟦*expr* ⟦, *expr*⟧*⟧?]
| (*expr*)
| *member_expr*
| *index_expr*
| *member_expr* (⟦*expr* ⟦, *expr*⟧*⟧?)
| ID (⟦*expr* ⟦, *expr*⟧*⟧?)
| *cexpr bin_op cexpr*
| - *cexpr*

bin_op ::= + | - | * | // | % | == | != | <= | >= | < | > | **is**

member_expr ::= *cexpr* . ID

index_expr ::= *cexpr* [*expr*]

target ::= ID
| *member_expr*
| *index_expr*

Tareas

- [ ] El código fuente se leerá desde un archivo o un elemento de área de texto.
- [ ] Implementar la lectura del input.
    - [ ] ¿Se leerá línea por línea? ¿Buffer?
    - [ ] ¿Cómo calcular la indentación?
    - [ ] **getchar()**: devuelve el siguiente carácter del input y mueve el puntero del carácter al siguiente.
    - [ ] **peekchar()**: devuelve el siguiente carácter sin mover el puntero.

- [ ] Implementar el scanner
    - [ ] **gettoken()**: llama al scanner. Su salida es un token.
      Token:
      tokentype: tipo de token
      tokenval : valor de token

    - [ ] Determinar si lo que estamos leyendo:
        - [ ] Es una palabra reservada o id (primero verificar si pertenece al conjunto de palabras reservadas, si no considerarlo id).
        - [ ] Es un literal (número o string)
        - [ ] Es un símbolo o delimitador. (Especial cuidado con operadores dobles como ==)
    - [ ] ¿En qué momento se incluyen los tokens de NEWLINE, INDENT Y DEDENT?

    - [ ] Cuando se detecte un error, informar con detalles útiles, incluido dónde se encontró, qué salió mal exactamente y cómo el programador podría solucionarlo.
        - [ ] Probar con diferentes posibles errores

    - [ ] Incluir una funcionalidad de salida detallada que rastree las etapas del scanner.

Notas

- El trabajo es en pares.
- No puede usar Lex ni ningún generador de scanner.
- El lenguaje es libre. El código debe usar y demostrar las buenas prácticas.
- Los comentarios serán ignorados por el SCAN.

Errores

- En algunos casos, el siguiente carácter de entrada puede no ser una continuación aceptable del token actual ni el comienzo de otro token. En tales casos, el escáner debe imprimir un mensaje de error y realizar algún tipo de recuperación para que la compilación pueda continuar, aunque solo sea para buscar errores adicionales. Afortunadamente, los errores léxicos son relativamente raros (la mayoría de las secuencias de caracteres corresponden a secuencias de tokens) y relativamente fáciles de manejar.
- El enfoque más común es simplemente
    - desechar el token actual no válido;
    - saltar hacia adelante hasta que se encuentre un carácter que pueda legítimamente comenzar un nuevo token;
    - reiniciar el algoritmo de escaneo
    - contar con el mecanismo de recuperación de errores del analizador para hacer frente a cualquier caso en el que la secuencia de tokens resultante no sea sintácticamente válida.
-

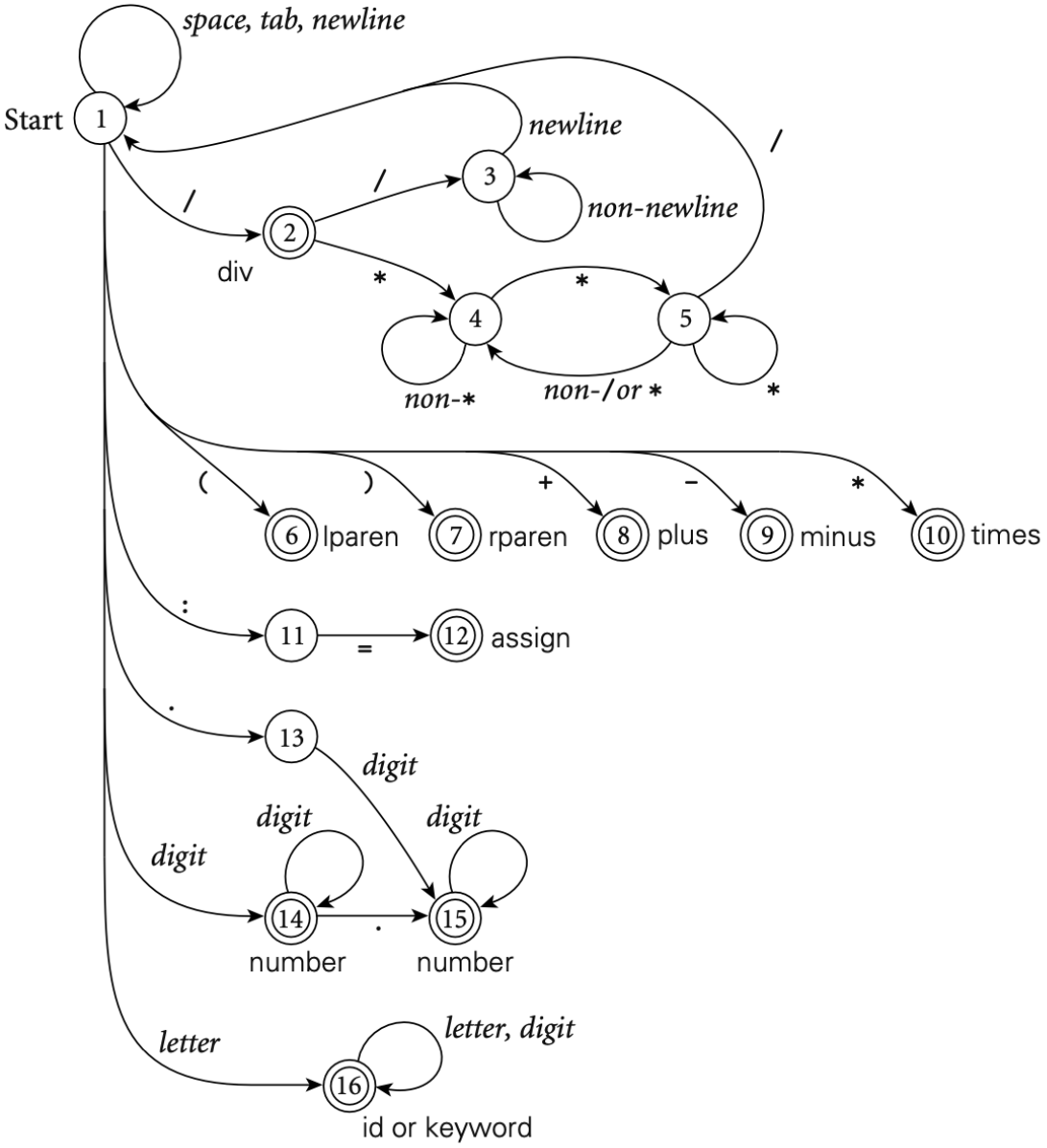
Ejemplo

```
# A broken program
def is_even(x:
) -> bool:
    if x % 2 == 1:
        return 0      # FIXME
    else:
        return True

print(is_even("3"))   # FIXME
```

```
INFO SCAN - Start scanning…
DEBUG SCAN - KEY           [ def ]     found at (2:1)
DEBUG SCAN - ID            [ is_even ] found at (2:5)
DEBUG SCAN - OPEN_PAR [ ( ]      found at (2:12)
DEBUG SCAN - ID            [ x ]       found at (2:13)
DEBUG SCAN - OP            [ : ]       found at (2:14)
DEBUG SCAN - KEY           [ int ]     found at (2:15)
DEBUG SCAN - CLO_PAR  [ ) ]      found at (2:18)
DEBUG SCAN - OP            [ -> ]      found at (2:20)
DEBUG SCAN - KEY           [ bool ]    found at (2:23)
DEBUG SCAN - OP            [ : ]       found at (2:27)
DEBUG SCAN - NEWLINE  []          found at (2:28)
DEBUG SCAN - IDENT    []          found at (3:1)
…
INFO SCAN - Completed with 0 errors
```

EJEMPLO DE CALCULADORA



```
skip any initial white space (spaces, tabs, and newlines)
if cur_char ∈ {'(', ')', '+', '-', '*'}
    return the corresponding single-character token
if cur_char = ':'
    read the next character
    if it is '=' then return assign else announce an error
if cur_char = '/'
    peek at the next character
    if it is '*' or '/'
        read additional characters until "*/" or newline is seen, respectively
        jump back to top of code
    else return div
if cur_char = .
    read the next character
    if it is a digit
        read any additional digits
        return number
    else announce an error
if cur_char is a digit
    read any additional digits and at most one decimal point
    return number
if cur_char is a letter
    read any additional letters and digits
    check to see whether the resulting string is read or write
    if so then return the corresponding token
    else return id
else announce an error
```