Lab 1. Implementando scanners

Compiladores

Introducción

El lenguaje B-Minor es un lenguaje "pequeño". B-Minor incluye expresiones, control de flujo básico, funciones recursivas y verificación de tipos estricta. B-Minor es lo suficientemente similar a C como para que resulte familiar, pero presenta suficientes variaciones para permitir una discusión sobre cómo las diferentes decisiones de diseño del lenguaje afectan la implementación. Por ejemplo, la sintaxis de tipos de B-Minor es más cercana a la de Pascal o SQL que a la de C.

Objetivo

Construir un analizador léxico para el lenguaje B-Minor que lea un archivo de origen y produzca una lista de cada token uno por uno, anotado con el tipo de token (identificador, entero, cadena, etc.) y la ubicación (línea). Si se detecta una entrada no válida, producir un mensaje, recuperar el error y continuar.

Crear un conjunto de pruebas para probar todos los casos difíciles, como comentarios, cadenas, caracteres de escape, etc.

Tareas

- ☐ Identificar los tokens que se necesitarán para la implementación de lenguaje.
- ☐ El código fuente se leerá desde un archivo o un elemento de área de texto.
- ☐ Implementar funciones de lectura de caracteres del input.
    - ☐ **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.

Tokens

- En B-Minor, el espacio en blanco se define como cualquier combinación de los siguientes caracteres: **tabulaciones, espacios, saltos de línea**. La ubicación del espacio en blanco no es significativa en B-Minor.
- Los comentarios al estilo de C y al estilo de C++ son válidos en B-Minor:

  /* Un comentario al estilo de C */ a=5;

  // Un comentario al estilo de C++

- Los identificadores (es decir, nombres de variables y funciones) pueden contener **letras, números y guiones bajos**. Los identificadores deben comenzar con una letra o un guión bajo. Estos son ejemplos de identificadores válidos:

  `i x mystr fog123 BigLongName55`

- Las siguientes cadenas son palabras clave en B-Minor y no pueden usarse como identificadores:

`array boolean char else false for function if integer print return string true void while`

Tipos

B-Minor tiene cuatro tipos atómicos: **enteros, booleanos, caracteres y cadenas.**

Una variable se declara con un nombre seguido de dos puntos, luego un tipo y un inicializador opcional. Por ejemplo:

```
x: integer; y: integer = 123; b: boolean = false; c: char = 'q';
s: string = "hello world\n";
```

Un entero es siempre un valor firmado de 64 bits. El tipo boolean puede tomar los valores literales `true` o `false`. El tipo `char` es un único carácter ASCII de 8 bits. El tipo `string` es una cadena constante entre comillas dobles que está terminada en nulo y no puede ser modificada. (Nota: a diferencia de C, `string` no es un arreglo de `char`, es un tipo completamente separado.)

B-Minor también permite arreglos de tamaño fijo. Estos pueden declararse sin valores, lo que hace que contengan todos ceros:

a: array [5] integer;

O bien, el arreglo completo puede recibir valores específicos:

a: array [5] integer = {1,2,3,4,5};

Expresiones

B-Minor tiene muchos de los operadores aritméticos que se encuentran en C, con el mismo significado y nivel de precedencia:

`[] f()`	array subscript, function call
`++ --`	postfix increment, decrement
`- !`	unary negation, logical not
`^`	exponentiation
`* / %`	multiplication, division, modulus
`+ -`	addition, subtraction
`< <= > >= == !=`	comparison
`&& ||`	logical and, logical or
`=`	assignment

- [ ] Cuando se detecte un error, informar con detalles útiles, incluido dónde se encontró, qué salió mal exactamente y cómo el programador podría solucionarlo.
- [ ] Incluir una funcionalidad de salida detallada que rastree las etapas del scanner.

Notas

- El trabajo es en grupos de 2.
- No se puede usar Lex ni ningún generador de scanner.
- El lenguaje para la implementación es libre. El código debe usar y demostrar las buenas prácticas.

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.

Ejemplos:

- Símbolo que no pertenezca a la especificación de ningún token.
- Verificar tamaño máximo permitido para literales.

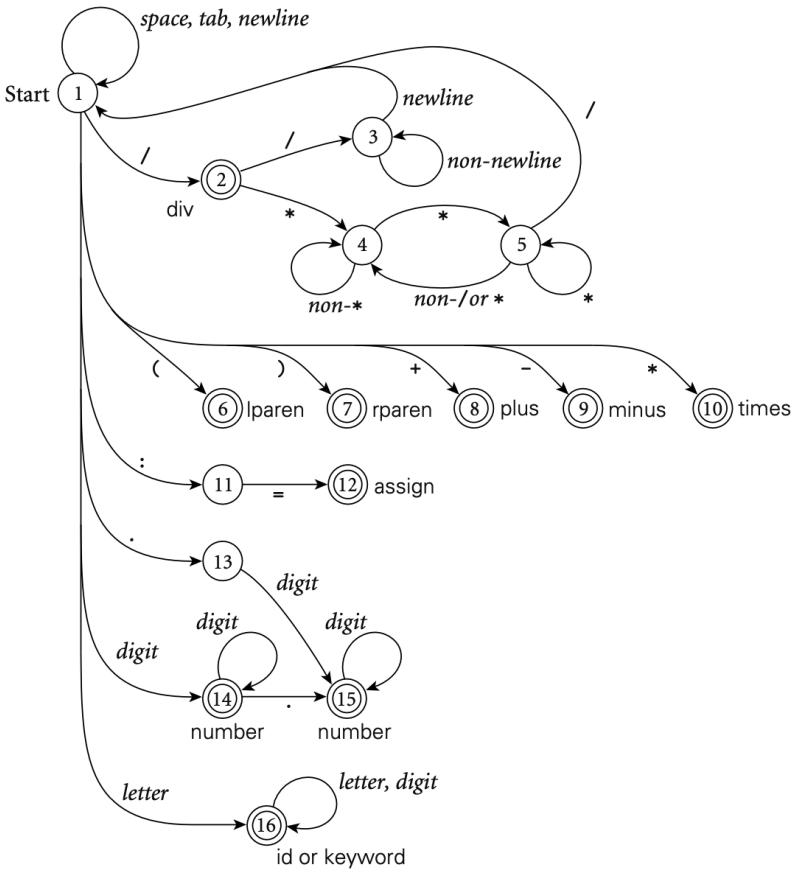
Ejemplo

```
a = 2 +7
print (a) # este es un comentario
$

INFO SCAN - Start scanning...
DEBUG SCAN - ID [ a ] found at (1:1)
```

```
DEBUG SCAN - ASSIGN_OP [ = ] found at (1:3)
DEBUG SCAN - INT [ 2 ] found at (1:5)
DEBUG SCAN - ADD_OP [ + ] found at (1:7)
DEBUG SCAN - INT [ 7 ] found at (1:8)
DEBUG SCAN - PRINT_KEY [ print ] found at (2:1)
DEBUG SCAN - OPEN_PAR [ ( ] found at (2:7)
DEBUG SCAN - ID [ a ] found at (2:8)
DEBUG SCAN - CLOSE_PAR [ ) ] found at (2:9)
DEBUG SCAN - EOP [ $ ] found at (3:1)
INFO SCAN - Completed with 0 errors
```

Autómata de calculadora



Código 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
```