Report Title

REPORT SUBTITLE

Javier Armunia Hinojosa | 622877 | Procesadores de Lenguajes

Análisis léxico:

Detallar y explicar la expresión regular utilizada para los identificadores.

```
< tIDENTIFICADOR :
  ([ "a"-"z", "_" ]) ([ "a"-"z", "_", "0"-"9" ])* ([ "a"-"z", "0"-"9" ])
| ([ "a"-"z", "_" ]) >
```

La expresión regular utilizada para los identificadores consta de dos opciones separadas por un or, la primera opción es para los identificadores de mas de un carácter, nos aseguramos de que no puedan empezar por número, pero sí por "_" y también aseguramos que no pueden acabar por "_". En la segunda opción permitimos los identificadores de 1 carácter y que empiece por "_".

Detallar y explicar cómo se han procesado los comentarios simples y multilínea.

```
< DEFAULT >
SKIP :
{
  " "
| "\r"
| "\t"
| "\n"
| < [ "%" ] ~[ "%" ] > : COMENT
| < [ "%" ] [ "%" ] > : COMENT_MULTI
}
```

En los tokens de skip introducimos dos opciones extra, una que se selecciona con "%" y otra que se selecciona con "%%", al seleccionarse, cada una nos lleva a un contexto diferente:

```
< COMENT >
SKIP :
{
  "\n" : DEFAULT
}

< COMENT >
SKIP :
{
  < ~[ ] >
}
```

En el contexto del comentario simple “capturamos” (Mas bien saltamos) dos tipos de tokens, los saltos de línea y todo lo demás, cuando se encuentra un salto de línea se vuelve al contexto default.

```
< COMENT_MULTI >
SKIP :
{
  "%%" : DEFAULT
}

< COMENT_MULTI >
SKIP :
{
  < ~[ ] >
}
```

El contexto del comentario multilínea funciona de manera similar al simple, pero en este, hasta que no encontramos “%%” no volvemos al contexto default.

Detallar y explicar cómo se ha implementado el modo “verbose”.

El modo verbose se ha implementado utilizando una tabla hash en la que almacenamos cada aparición de cada token, al final de la ejecución, si estamos en modo verbose, se imprime la tabla por pantalla.

TOKEN	APARICIONES
tCARACTER	1
tDIV	1
tENT	2
tENTACAR	1
tENTERO	20
tESCRIBIR	4
tFALSE	1
tFIN	1
tFIN_SENTENCIA	13
tFMQ	2
tFSI	2
tIDENTIFICADOR	9
tIGUAL	1
tMENOS	2
tMOD	1
tMQ	2
tNI	1
tOPAS	6
tPARENTESIS_DCHA	6
tPARENTESIS_IZQ	6
tPOR	6
tPRINCIPIO	1
tPROGRAMA	1
tSI	2
tSTRING	4
tTRUE	1

Detallar y explicar la función de error léxico implementada.

```
catch (Error e)
{
  System.err.println("ERROR LEXICO en la linea " + token.beginLine + ", columna " +
  token.beginColumn + ". Simbolo no reconocido: " + token.image);
}
```

El mensaje de error léxico introducido es bastante sencillo, se muestra la línea y la columna del último token leído antes del error.

Análisis sintáctico

Modificaciones sobre la gramática propuesta y decisiones de diseño adoptadas.

1. Para permitir la invocación y declaración de acciones sin paréntesis:

parametros_formales ::= ((<tPARENTESIS_IZQ> parametros) (<tFIN_SENTENCIA> parametros)* <tPARENTESIS_DCHA>)?

invocacion_accion ::= (argumentos)? <tFIN_SENTENCIA>

   He utilizado "?" en la expresión regular para indicar que son opcionales.

2. Para que leer y escribir reciban al menos un parametro:

lista_asignables ::= <tIDENTIFICADOR> (<tCOMA> <tIDENTIFICADOR>)*

lista_escribibles ::= (expresion | <tSTRING>) (<tCOMA> (expresion | <tSTRING>))*

3. Para utilizar look-ahead = 1:

sentencia ::= (leer <tFIN_SENTENCIA> | escribir <tFIN_SENTENCIA> | <tIDENTIFICADOR> (asignacion | invocacion_accion)? | seleccion | mientras_que)

Tuve que unir la opción de asinación e invocación acción, ya que las dos empiezan por tIDENTIFICADOR, en un principio puse tFIN_DE_SENTENCIA fuera de asignación e invocación_acción, pero para poder permitir la invocación de acciones sin paréntesis tuve que ponerlo dentro, ya que si no, asignación podía hacer match con la cadena vacía.

4. Detallar y explicar el manejo de excepciones y la función de error sintáctico implementada.

He utilizado bloques try/catch en cada función, de forma que la excepción lanzada se capture dentro de la regla correspondiente.
La función de error sintáctico muestra información detallada sobre el error ocurrido, muestra la fila, columna, el token que ha causado el error y los tokens que se esperaban. Un ejemplo:

```
ERROR SINTACTICO en la linea 33, columna 5. Simbolo obtenido: entero
Se esperaba uno de:
        "fin"
        "si"
        "mq"
        "escribir"
        "leer"
        <tIDENTIFICADOR>
```

Tabla de símbolos

Detallar y explicar la función de hash implementada.

He utilizado la función de hash de Pearson:

```
h = 0 ;
for i in 1..n loop
    h := T[h xor C[i]];
end loop ;
return h
```

La cual consiste en siendo T una permutación aleatoria de un array de números del 0 al 255. Para cada carácter de la clave, se selecciona el valor h de una posición de T, obtenida haciendo una operación xor entre h y el carácter de la clave. La hash será el valor final de h, con el que realizaremos la operación módulo con el tamaño de nuestra tabla para obtener la posición final.

Detallar y explicar la implementación de la tabla y el manejo de colisiones.

Ya que solamente puede existir una tabla de símbolos, se ha utilizado el patrón de diseño Singleton para asegurarnos que esta restricción se cumple.

He utilizado una tabla de símbolos implementada mediante un ArrayList de ArrayList de símbolos, inicializado el primer ArrayList con el tamaño de la tabla. Para introducir elementos en la tabla se calcula la clave hash y se comprueba si existe un elemento con el mismo nombre y nivel dentro de la tabla, si no existe se introduce, si existe se devuelve nulo.

Para eliminar todos los elementos de un nivel, se comprueba cada posición de la tabla para eliminar los elementos del nivel deseado. Los parámetros de un nivel que se va a borrar no se ocultan, se borran ya que tenemos una referencia a ellos en el símbolo de la acción. De esta forma no tenemos que preocuparnos porque se llamen igual los parámetros de acciones diferentes.

Análisis semántico

Detallar los errores y avisos (warnings) que se han implementado.

Se realizan las siguientes comprobaciones:

- Comprobación de tipos correctos en operaciones, asignaciones e invocaciones de acciones.
- Colisiones en la tabla de símbolos.
    - No se pueden declarar dos símbolos con el mismo nombre en el mismo nivel.
- Identificadores desconocidos.
    - Se introducen en la tabla de símbolos con tipo desconocido tras dar el error.
- Overflow en sumas, multiplicaciones, restas e índices de vector.
    - Se comprueba que no haya errores de overflow cuando es posible.
- División por o en resta y modulo.
- Comprobación al invocar acciones, de que el símbolo sea realmente una acción.
- Comprobación del numero de parámetros al invocar acciones.
- Comprobación de tipos de parámetros.
- Comprobaciones parámetros por valor o referencia.
    - No se puede pasar por referencia un valor.
    - No se puede asignar un valor.
    - No se puede utilizar un valor en leer.
- Comprobaciones para no asignar a símbolos que sean acciones.
    - No se puede realizar una asignación a una acción.
- En mientras que se comprueba que la condición sea un booleano, que el bucle no sea infinito o que no se entre nunca.
- En la selección se comprueba que la condición sea un booleano, que la condición no sea siempre cierta o falsa.
- En entacar no se pueden utilizar valores que no estén entre 0 y 255.
- Comprobación de longitud de vectores al pasar por parámetro.
- No se pueden realizar operaciones con vectores enteros, pero con componentes del vector sí.

- Los índices de los vectores deben ser enteros.
- No se pueden transformar vectores enteros con entacar.
- No se pueden transformar vectores enteros con caraent.
- Para asignar vectores, los dos deben tener la misma longitud.
- Comprobación al asignar vectores, que ambos símbolos en la asignación sean vectores.
- El índice del vector debe ser positivo.
- No se puede acceder a posiciones mayores que el tamaño del vector.

Detallar y explicar los controles que se han implementado sobre los tipos de datos (desbordamientos, booleanos, etc.).

Mediante RegistroExpr, en los casos que es posible, calculo el resultado de las expresiones y lo propago utilizando este registro. De esta forma es posible realizar las comprobaciones de tipos, desbordamientos y demás. Para los desbordamientos he supuesto una máquina de 16 bits y enteros con signo. Cuando en la expresión intervenían variables, se consideraba que no era posible realizar los cálculos.

Detallar y explicar cómo se ha implementado la propagación y el manejo de valores constantes en las expresiones.

Mediante el uso de RegistroExpr, una clase que sirve como struct, en la que se almacena el tipo del resultado de las expresiones, su valor si es posible, si es un parámetro, la clase de parámetro que es y si es un vector y su longitud. Finalmente también he incluido que pueda propagar símbolos.

Detallar y explicar cómo se han implementado las comprobaciones sobre parámetros VAL/REF. ¿Dónde se realizan estas comprobaciones? Adjuntar ejemplos que validen la funcionalidad implementada.

Los ejemplos de esta sección se han sacado del programa de prueba test_semantico1.ml

En RegistroExpr se propaga la clase del parámetro cuando es necesario, además de en los propios símbolos.

```
accion acc(val entero valor, valor1 ; ref entero referencia);
```

Al realizar asignaciones, se comprueba que el símbolo de la izquierda de la asignación no sea un parámetro por valor.

```
referencia:= 10;
valor := 10; % Error
referencia := valor;
referencia := referencia + 10;
```

Al invocar acciones, se comprueba la clase de los parámetros, de forma que no se pueda pasar un parámetro por valor por referencia.

```
acc(1, 10, referencia);
acc(referencia, referencia, referencia);
acc(valor, valor, valor); % Error, no se puede pasar un valor por referencia
acc(valor, valor, 19); % Error
```

Ilustración 1 Dentro de la acción, invocandola de forma recursiva

```
principio
    acc(10, variable);
    acc(variable, variable);
    acc(10, 10); % Error
    leer(10); % Error
fin
```

Ilustración 2 Desde el programa principal

También se realizan comprobaciones para no poder utilizar en la función leer() parámetros por valor, pero sí en escribir():

```
leer(valor); % Error
escribir(valor, referencia);
```

Generación de código

Detallar y justificar la elección del esquema de generación de código utilizado (secuencial/AST).

He elegido el esquema secuencial principalmente por su sencillez, ya que no me quedaba mucho tiempo para la realización de esta parte, pero me hubiera gustado utilizar AST ya que ofrece mas posibilidades de optimización.

Mejoras introducidas

Detallar las mejoras y puntos opcionales que se han desarrollado sobre la propuesta inicial. Especificar detalles de implementación, decisiones de diseño y pruebas/ejemplos de las funcionalidades añadidas. Separar las mejoras en bloques

según correspondan al análisis léxico, sintáctico, semántico o a la generación de código

Se han implementado vectores, la parte léxica, sintáctica, semántica y de generación de código. Se pueden utilizar componentes de vectores exactamente igual que una variable normal. Los vectores enteros se pueden asignar a otros vectores del mismo tamaño, además de pasar por parámetros como valor o referencia.

Pruebas realizadas

Detallar la metodología seguida para verificar el funcionamiento del compilador desarrollado.

Para la verificación del funcionamiento del compilador se ha ido comprobando que funcionaba bien cada cosa a medida que se implementaba, desde el analizador léxico, comprobando que detecte cada token, el analizador sintáctico, comprobando que detecte cada expresión, el semántico, comprobando que realice todas las comprobaciones correctamente y la generación de código, comparando con los ficheros proporcionados y ejecutando en Hendrix.

Para la comprobación de los vectores he realizado pruebas en Hendrix y comprobando los ficheros de código generados.