Entrega Parcial

Procesadores del Lenguaje

Alberto Penas Díaz:
NIA: 100471939
Correo: 100471939@alumnos.uc3m.es
Titulación: Ingeniería informática

Héctor Álvarez Marcos:
NIA: 100495794
Correo: 100495794@alumnos.uc3m.es
Titulación: Ingeniería informática

Índice

Analizador Léxico

Para el analizador léxico, hemos definido la secuencia de tokens que se han especificado en el enunciado, teniendo en cuenta como tokens los comentarios simples y los comentarios multilínea, para llevar un conteo de la línea del código actual, usada para llevar a cabo una forma trazable de recuperación de errores. Se han definido, además, una serie de palabras reservadas, de nuevo, las especificadas en el enunciado.

A tener en cuenta, esta vez se exporta un fichero .token con los tokens hallados en el fichero de entrada. Cabe mencionar que se espera que el fichero de entrada tenga extensión .vip, ya que de no ser así, el fichero de tokens reescribirá el fichero de entrada.

Analizador Sintáctico

Para el desarrollo de la gramática, se ha seguido cuidadosamente los requisitos y limitaciones del enunciado, habiendo optado (por simplicidad y limpieza de código) no usar las indentaciones propuestas, si no los *brackets* de apertura y cierre.

Definición formal de la gramática

De manera formal, nuestro diseño de gramática es el siguiente:

G = {NT, T, P, S}
$\sum_{NT}$={ S,program, statement_list, statement, declaration, assignment, if_statement, while_statement, register, var_list, var_decl, decl_assign, reference, rest_ref, funct_decl, funct_call, arg_funct_call, type_funct, arg_funct, arg_funct2, arg_funct_rec, extra, another, block_funct, funct_ret, newlines, else, block, expression}
$\sum_{T}$={ INT_TYPE, FLOAT_TYPE, CHAR_TYPE, BOOL_TYPE, ID, TYPE, TRUE, FALSE, COMMENT, MLCOMMENT, DEF, RETURN, IF, ELSE, WHILE, DOT, LBRACKET, RBRACKET, LBRACE, RBRACE, LPAREN, RPAREN, COMMA, COLON, SEMICOLON, PLUS, MINUS, TIMES, DIVIDE, GT, LT, GE, LE, EQ, EQUALS, AND, OR, NOT, DECIMAL, BINARY, OCTAL, HEXADECIMAL, FLOAT_CONST, CHAR_CONST,NEWLINE }

P = {
S ::= program
program ::= statement_list
| λ
statement_list ::= statement_list statement | statement
statement ::= declaration NEWLINE | assignment NEWLINE | if_statement | TYPE register
| while_statement | COMMENT NEWLINE | MLCOMMENT NEWLINE
| funct_decl | funct_call NEWLINE | NEWLINE

declaration ::= INT_TYPE var_list | FLOAT_TYPE var_list | CHAR_TYPE var_list
| BOOL_TYPE var_list
| ID var_list

register ::= ID COLON block

var_decl ::= ID decl_assign | LBRACKET expression RBRACKET ID decl_assign

decl_assign ::= EQUALS expression | λ

var_list ::= var_list COMMA var_decl | var_decl

assignment ::= reference EQUALS assignment | reference EQUALS expression

reference ::= ID rest_ref

rest_ref ::= λ | DOT ID rest_ref | LBRACKET expression RBRACKET rest_ref

funct_decl ::= DEF type_funct ID LPAREN arg_funct RPAREN COLON block_funct

funct_call ::= ID LPAREN arg_funct_call RPAREN

arg_funct_call ::= expression COMMA arg_funct_call | expression | λ

type_funct ::= INT_TYPE | FLOAT_TYPE | BOOL_TYPE | CHAR_TYPE | ID

arg_funct ::= type_funct arg_funct2 | λ

arg_funct2 ::= ID extra another

arg_funct_rec ::= type_funct ID extra another

extra ::= COMMA ID extra | λ

another ::= SEMICOLON arg_funct_rec | λ

block_funct ::= newlines LBRACE statement_list funct_ret RBRACE NEWLINE

funct_ret ::= RETURN expression newlines

newlines ::= NEWLINE | λ

if_statement ::= IF expression COLON block else_

else_ ::= newlines ELSE COLON block | NEWLINE

while_statement ::= WHILE expression COLON block

block ::= newlines LBRACE statement_list RBRACE

expression ::= expression PLUS expression | expression MINUS expression
  | expression TIMES expression | expression DIVIDE expression | expression EQ expression
  | expression GT expression | expression LT expression | expression GE expression
  | expression LE expression | expression AND expression | expression OR expression
  | NOT expression | MINUS expression | PLUS expression | LPAREN expression RPAREN
  | DECIMAL | BINARY | OCTAL | HEXADECIMAL | FLOAT_CONST | TRUE | FALSE
  | CHAR_CONST | funct_call | reference
}

Decisiones principales de diseño de la gramática

Las decisiones de diseño principales de la gramática han sido el cómo consideramos nosotros el tratamiento de los "newlines". Básicamente consideramos tanto uno como n newlines seguidos un único token newline, por lo que por ejemplo, en los casos de control de flujo if/else, entre el cierre de llaves del if y del else puede haber tantos \n como guste, como ocurre en otros lenguajes, no hemos querido hacer limitación en eso. Otro punto es el tema ya mencionado de la elección de brackets, ya que para mantener una estructura limpia, todos los tokens de llave derecha "}", que implican el cierre de una serie de sentencias, de un controlador de flujo o de un bloque de función van a ir seguidos de un salto de linea, con el fin de evitar nuevas declaraciones o sentencias en la misma línea del cierre de llaves. El único caso en el que esto no funciona así es en el bucle if/else, ya que se permite que justo después del cierre del bloque if, sí y sólo si va un "else", se permita ponerlo a la misma altura. De esta forma:

```
if b: {
  x = x + 1
}else :
{
  x = x - 1
}
```

Este es el único caso en el que se puede escribir algo que no sea \n después de un cierre de llaves.

Tampoco se podrá realizar el cierre de múltiples bloques de llaves de forma consecutiva("}}}...") ya que como se ha dicho, después de cada llave tiene que existir un salto de línea.

Otro tema a considerar es que nosotros dentro de los índices de un vector permitimos al lenguaje encapsular todo tipo de expresiones, incluidas por ejemplo las de tipo float y booleano, pese a que sabemos que luego en el semántico va a producir error porque se necesita una expresión de tipo entero, pero entendemos que eso es un punto a tratar en la próxima parte de la práctica. Por ende, el resto *de momento* son válidos. Así pues, se puede asignar al índice de un vector la llamada a una función. Luego será cosa del sintáctico el verificar por ejemplo que esa función tenga una sentencia de retorno que sea de tipo entero, pero a ese punto no tenemos que llegar todavía, simplemente se han descartado las opciones que son sintácticamente incorrectas.

Cabe mencionar, además, que cuando existe algún error sintáctico en el fichero de entrada, se imprime la línea en la que se encuentra, de forma en la que se señala exactamente dónde se encuentra el error y se recupera del mismo pasando directamente a la siguiente línea.

Batería de Pruebas

Para la batería de pruebas, hemos generado un *shell script* con distintos casos de prueba. Estos casos siguen la estructura "v[n]_[referencia_a_funcionalidad_test]" $\forall$ n $\in \mathbb{N}$ y se basan en la generación de un archivo temporal en el directorio /temp (el cual es eliminado al finalizar la ejecución de las pruebas) y la comparación de este fichero con la salida esperada almacenada en los ficheros con estructura "v[n]_[referencia_a_funcionalidad_test]_expected" $\forall$ n $\in \mathbb{N}$.

Si se quisiera visualizar la salida de cada uno de los ficheros de entrada del programa, basta con comentar o eliminar la <u>última</u> sentencia del fichero de test *run.sh*, que es la que se encarga de eliminar el directorio temporal que se ha creado. Esto se puede hacer de la siguiente manera:

```
# rm -rf temp
```

Por otra parte, los casos inválidos tratan de explorar los "casos límite" de nuestra implementación, revisando la gestión de errores tanto en la apertura del archivo de entrada, como errores reconocidos en el ámbito del léxico, del sintáctico, como errores puramente matemáticos, como la no división por 0. Estos casos de error siguen la misma estructura que los válidos, pero con nombre "i[n]_[referencia_a_funcionalidad_test].vip" $\forall$ n $\in \mathbb{N}$.

En cada uno de los ficheros de los casos de prueba se explica más en detalle qué se está tratando de comprobar específicamente, todos ellos situados en el directorio test_files.

Aclaraciones Importantes

IMPORTANTE: esta práctica se ha desarrollado en equipos linux. Para ejecutar un fichero de Python en equipos con arquitectura linux, basta con poner en terminal `python3 <nombre del archivo>`. Sin embargo esto puede cambiar en equipos con MacOS o Windows, ya que es posible que para ejecutar un fichero de python en estos equipos haya que introducir por terminal `python <nombre del archivo>`. Si este fuera el caso, solo hace falta cambiar la linea 22 del fichero run.sh de la siguiente manera:

Cambiar:

```
python3 main.py "$input_file" > "$output_file"
```

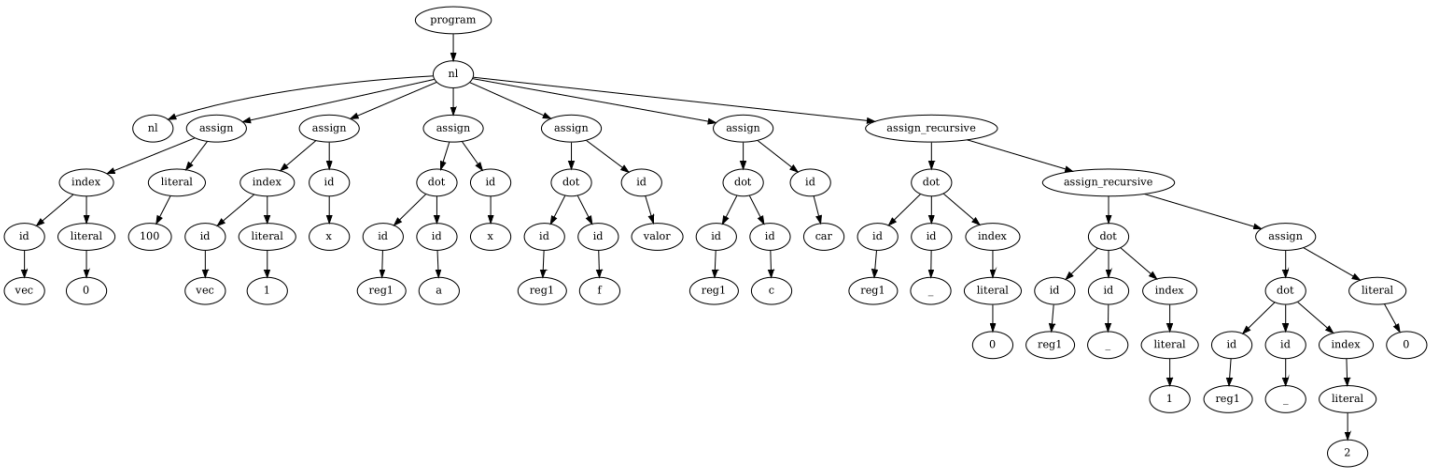
por:

```
python main.py "$input_file" > "$output_file"
```

Además, es importante mencionar que si no se puede ejecutar el fichero run.sh, es posible que haya que darle permisos de administrador con el comando `chmod +x run.sh`

Contenido Extra: Visualización del Árbol Sintáctico

Para facilitar el desarrollo del árbol sintáctico, se ha utilizado una herramienta para poder visualizar de forma gráfica el árbol resultante de la siguiente manera:



Éste árbol nos sirve como base para ver cómo se está formando la gramática **pero no es explícitamente el árbol que genera un analizador sintáctico LALR**. Se ha generado para ayudarnos a ver gráficamente cómo se producen las derivaciones de las reglas de producción. Para la ejecución del mismo, se ha modificado levemente el protocolo de lanzamiento del

programa, de forma que si se quiere exportar dicho árbol, se ha de introducir por terminal la siguiente sentencia:

```
python3 main.py true
```

Es necesario que el tercer parámetro de la sentencia sea la palabra “true”. En caso de no serlo, se imprimirá por terminal un mensaje de error, explicando cómo se tiene que ejecutar. Es muy importante mencionar, además, que esta visualización depende del paquete “graphviz”, por lo que para usarlo se deberá tener instalado. Lo que no quiere decir que el programa no pueda usarse en caso de no tenerlo: es puramente opcional.
Los resultados obtenidos se guardarán en el directorio /tree_gen/ en formato pdf