ENTREGA FINAL

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

Introducción

Esta memoria recoge el desarrollo completo del ejercicio final de la asignatura de Procesadores del Lenguaje. En ella se detallan las distintas fases de implementación, así como las correcciones y mejoras llevadas a cabo con respecto a entregas anteriores. El objetivo principal ha sido construir un compilador funcional que integre correctamente las fases léxica, sintáctica y semántica, cumpliendo con los requisitos establecidos.

Además de cumplir con el enunciado, se han incluido distintas mejoras para aumentar la robustez del sistema y facilitar su mantenimiento. Entre ellas, se encuentran mejoras en la detección y recuperación de errores, una reducción significativa de la complejidad gramatical y la introducción de un sistema de preprocesado previo al análisis léxico.

En los siguientes apartados se explican en detalle las decisiones tomadas, la estructura de la gramática final, el enfoque seguido en el análisis semántico y los mecanismos implementados para el manejo del ámbito y la gestión de errores. También se incluye una sección con funcionalidades adicionales implementadas que no eran obligatorias, pero que consideramos útiles o interesantes dentro del contexto del ejercicio.

Corrección de Errores

Hemos corregido una cantidad significativa de pequeños errores realizados en la segunda entrega de la asignatura para no acarrearlos en la entrega final.

Análisis Léxico

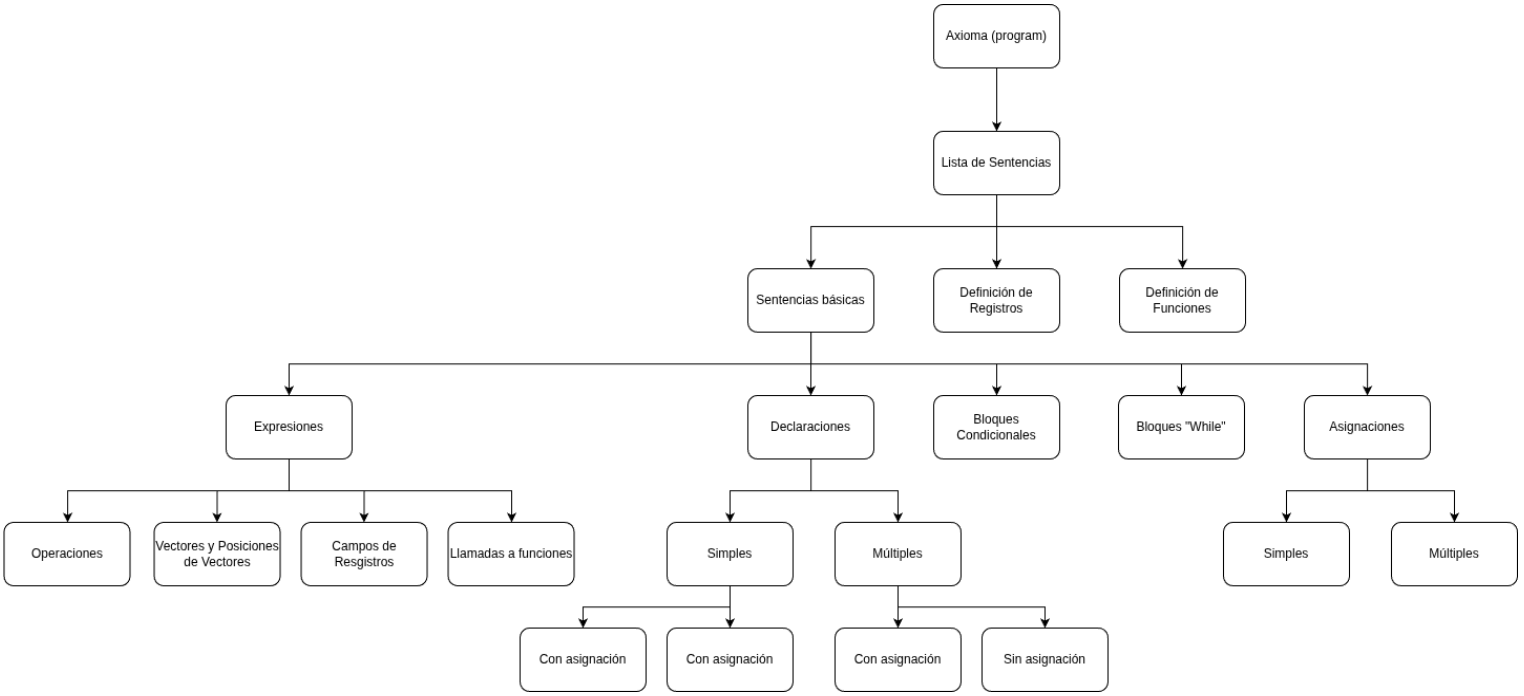
- Ya no se sustituye la extensión “.vip” por “.token” en el nombre del fichero de entrada, si no que se anexa.
- Ahora se tienen en cuenta los números en notación científica.
- Ahora se tienen en cuenta los ceros no significativos, considerándolos como tokens separados al valor significativo. Por ejemplo, la cadena de caracteres “00001,3” se descompondría en cuatro tokens “INT” de valor “0” y un token “FLOAT” de valor “1,3”.
- Se han reducido la cantidad de tokens, ahora se agrupan los valores hexadecimales, binarios, octales y decimales en el token único “INT”.
- Se ha corregido en análisis léxico de los caracteres individuales. Ahora únicamente se aceptan caracteres en ASCII extendido.

Análisis Sintáctico

- Se ha reducido la complejidad de la gramática (Véase apartado Gramática)
- Ya no se permite la declaración de funciones dentro de funciones o la declaración de tipos dentro de tipos.

Gramática Final

Para la gramática, hemos seguido las indicaciones que se propusieron en las clases de práctica de la asignatura, corrigiendo errores encontrados en la segunda entrega parcial de la asignatura y simplificando de manera significativa la formación de la gramática. De forma, abstracta, nuestra gramática se puede ejemplificar con el siguiente diagrama:



De esta manera, creemos que se abstrae de forma correcta, sencilla y escalable el razonamiento que sigue el lenguaje del problema. De manera formal, la gramática de nuestro ejercicio es la siguiente:

<program> ::= <statement_list>
| ε

<statement_list> ::= <statement_list> <statement>
| <statement>

<statement> ::= <sentence> NEWLINE
| <type_definition> NEWLINE
| <function_definition> NEWLINE
| NEWLINE

<sentence_list> ::= <sentence_list> <sentence> NEWLINE
| <sentence> NEWLINE

```
<sentence> ::= <expression>
            | <assignment>
            | <declaration>
            | <if_statement>
            | <while_statement>

<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>
              | INT
              | FLOAT
              | CHAR
              | TRUE
              | FALSE
              | ID LPAREN <function_call_argument_list> RPAREN
              | LPAREN <expression> RPAREN
              | ID <reference>

<function_call_argument_list> ::= <expression> COMMA <function_call_argument_list>
                                | <expression>
                                | ε

<reference> ::= LBRACKET <expression> RBRACKET <reference>
             | DOT ID <reference>
             | ε

<assignment> ::= ID <reference> EQUALS <expression>
              | ID <reference> EQUALS <assignment>

<declaration> ::= INT_TYPE <variable_list>
```

| FLOAT_TYPE <variable_list>
| BOOL_TYPE <variable_list>
| CHAR_TYPE <variable_list>
| ID <variable_list>

<declaration_list> ::= <declaration_list> <declaration> NEWLINE
| <declaration> NEWLINE

<variable_list> ::= <variable_list> COMMA <variable_declaration>
| <variable_declaration>

<variable_declaration> ::= ID <assignment_declaration>
| LBRACKET <expression> RBRACKET ID

<assignment_declaration> ::= EQUALS <expression>
| ε

<if_statement_header> ::= IF <expression> COLON

<if_statement> ::= <if_statement_header> LBRACE NEWLINE <sentence_list> RBRACE
| <if_statement_header> LBRACE NEWLINE <sentence_list> RBRACE ELSE COLON LBRACE NEWLINE <sentence_list> RBRACE

<while_header> ::= WHILE <expression> COLON

<while_statement> ::= <while_header> LBRACE NEWLINE <sentence_list> RBRACE

<type_definition> ::= <type_definition_header> <type_definition_body>

<type_definition_header> ::= TYPE ID COLON LBRACE NEWLINE

<type_definition_body> ::= <declaration_list> RBRACE

<function_header> ::= DEF <function_type> ID

<function_header_and_parameters> ::= <function_header> LPAREN <argument_list> RPAREN

<function_before_body> ::= <function_header_and_parameters> COLON LBRACE NEWLINE

<function_definition> ::= <function_before_body> <sentence_function> RETURN <expression> <newlines> RBRACE

<newlines> ::= NEWLINE
| ε

<sentence_function> ::= <sentence_list>
| ε

<function_type> ::= INT_TYPE
| FLOAT_TYPE
| BOOL_TYPE
| CHAR_TYPE
| ID

<argument_list> ::= <declaration> SEMICOLON <argument_list>
| <declaration>
| ε

Para evitar conflictos desplazamiento/reducción en la parte de if/else, hemos decidido omitir el salto de linea al comenzar el bloque if, por lo que ésta palabra reservada debe encontrarse en la misma línea en la que se encuentra el *bracket* de cierre del *if*.

De igual manera, a favor de que la gramática final quedara lo más limpia y clara posible, hemos reutilizado reglas para ahorrar espacio y favorecer la simplicidad. Por ejemplo, reutilizamos la regla "declaration" en la especificación de atributos en las variables de tipo registro o en la especificación de argumentos en funciones. Si bien es cierto que, por tal y como hemos definido nuestra gramática, las declaraciones permiten internamente asignaciones y esto es incompatible en algunos casos, hemos decidido delegar la responsabilidad de comprobar estas asignaciones ilegales al analizador semántico. Esta parte está explicada en detalle en el siguiente apartado.

Decisiones de Diseño

Con respecto a las decisiones de diseño tomadas, hemos seguido una filosofía similar tanto para la parte léxica como sintáctica de los ejercicios anteriores. Para el análisis semántico, hemos seguido una jerarquía de objetos y herencia en Python, aprovechando al máximo la programación orientada a objetos.

Por ejemplo, para el análisis semántico, hemos definido clases para las Literales, Llamadas a Funciones, Expresiones binarias o unarias, etc, heredan de la clase Expresión, que contiene el método clave "*infer_type*" Este método permite conocer el tipo de valor final que devuelve

una expresión formada de cualquier manera. Es aquí, además, donde se tiene en cuenta la transformación de valores automáticos. Por ejemplo, el siguiente bloque de sentencias:

```
Unset
char a = 'A'

int b = a + 4
```

La variable "a" es convertida automáticamente cuando se trata de operar con ella detectando que su resultado se quiere almacenar en una variable de tipo entero.

Con respecto a la devolución de valores en las funciones, también se comprueba que el tipo de la expresión que devuelve la función es igual al tipo especificado. En este caso, no hay problema con el valor de retorno de una función recursiva ya que, dada la especificación del enunciado, sólo se puede emplear una sentencia *return* por cada función, por lo que no sería posible salir de una función con una única expresión return que volviera a llamar a la función.

También, aunque no se especifique explícitamente en el enunciado, no permitimos realizar declaraciones con asignaciones dentro de la definición de registros o en la especificación de los argumentos que va a aceptar una función. Es por ello que devolvemos un error semántico cuando esto ocurre pero permitimos que se continúe con el análisis semántico del fichero. Por ejemplo, en el siguiente caso:

```
Unset
type Point : {
       int x = 0
       int y = 0
}
```

En este caso, no se permite el valor "por defecto" de los campos del registro "Point", por lo que el analizador semántico devuelve un error **pero sigue analizando el fichero, ignorando por completo el valor 0 que se ha pretendido asignar**.

Pruebas

Para las pruebas de la práctica, hemos diseñado un fichero programado en bash que se encarga de lanzar y testear todas las pruebas. Cada prueba consiste en un fichero localizado en /test_files/input/. Todos los ficheros de este directorio se ejecutan y se genera, por cada fichero de test, cuatro ficheros adicionales que se almacenan en /test_files/output/. Estos ficheros son:

1. fichero .token: Fichero con todos los Tokens generados por el analizador léxico.
2. fichero .symbol: Fichero con todos los símbolos generados por el analizador semántico.
3. fichero .record: Fichero con todos los registros generados en el fichero de entrada.
4. fichero .error: Fichero con la salida estándar de la ejecución del fichero. Si el funcionamiento del programa no ha generado errores, este fichero debería estar vacío.

El script, compara estos cuatro ficheros con los 4 ficheros que tienen el mismo nombre pero se localizan en el directorio /test_files/expected/. Si los ficheros generados son idénticos a los esperados, el test pasa, en caso contrario, el test fallará.

Para ejecutar este fichero, basta con poner en la terminal el siguiente comando (desde el directorio del proyecto):

>> ./run.sh

Estas pruebas se han diseñado probar principalmente el analizador semántico, ya que asumimos que el sintáctico y léxico ya han sido privados en la práctica anterior.

Nota importante: todas las pruebas, así como el desarrollo en su completitud del proyecto se ha desarrollado en un entorno **Linux**. Ejecutar este fichero en un entorno diferente puede generar incompatibilidades e incoherencias con respecto a las salidas esperadas. Si se creyera conveniente, se podría ejecutar el proyecto en las Aulas Virtuales de la Universidad, en las que se dispone de máquinas virtuales con Linux en las que el fichero de pruebas funciona perfectamente.

Cabe mencionar, además, que la primera vez que se ejecuten las pruebas es probable que estas fallen al comparar el fichero .error si no **están generados previamente los ficheros parsetab.py y parser.out** ya que éstos generan un mensaje en la salida estándar al generarse, lo que falla al compararse con el fichero esperado (en el que se asume que estos ficheros ya han sido creados)

Contenido Extra

Scope de funciones (variables locales)

Como contenido extra, hemos tenido en cuenta el scope en las funciones para poder asignar variables de manera local, que se eliminan automáticamente al finalizar la función. Por ejemplo, en el siguiente bloque de código:

```
Unset
def int foo (int a):{
       int i
       return 0
}

i = 10
```

No estaría permitido porque la variable “i” ha sido declarada en un scope distinto. De igual manera, sí es posible re-declarar una variable que ha sido declarada en la scope global en una función.

Recuperación de errores

Como ya se ha mencionado previamente, cada vez que hay un error en el fichero, ya sea léxico, semántico o sintáctico, se continúa analizando el fichero. No obstante, cabe destacar que cuando se detecta un error léxico (por ejemplo) éste error se arrastra y es probable que genere una gran cantidad de errores gramaticales y semánticos causantes por un error inicial menor.

Este comportamiento es más exagerado cuando el error está en la primera “capa” de análisis (léxico) y mucho menos exagerado cuando éste está en la última “capa” (semántico).

Preprocesado del fichero

Como apartado complementario, hemos decidido incluir la capacidad de realizar un preprocesado del fichero inicial incluyendo dos funciones claras muy inspiradas en el lenguaje de programación C.

- **%append <path>** : esta función incluye todo el contenido del fichero especificado en path en el lugar donde esté la palabra reservada append (precedida directamente del símbolo %)
- **%supplant <old> <new>** : esta función recorre todo el fichero y sustituye todas las coincidencias de <old> con <new>

Este preprocesado se realiza antes de pasar el fichero al Lexer. De forma que si, por ejemplo tenemos los dos siguientes ficheros:

foo.vip

```
Unset
type Circle : {
	float radius
}
```

main.vip

```
Unset
%append foo.vip

%supplant PI 3.141592

Circle c
c.radius = 1

float perimeter = 2 * c.radius * PI
```

Si se habilita el preprocesado, (argumento opcional "*allow_preprocess*" de la clase ViperLexer en el archivo de código *lexer.py* y por defecto se encuentra apagado) Se preprocesa el fichero de entrada y se genera un nuevo fichero con el nombre del fichero de entrada con la extensión .postprocessed.

Este fichero es que más adelante se pasará al Lexer para realizar el funcionamiento normal del programa. Con el ejemplo anterior, el fichero que queda después del preprocesado es el siguiente:

main.postprocessed

```
Unset
type Circle : {
	float radius
}

Circle c
c.radius = 1

float perimeter = 2 * c.radius * 3.141592
```

Conclusiones

La práctica nos ha parecido útil para consolidar los conocimientos teóricos adquiridos en la asignatura, especialmente en lo que respecta a la integración de las distintas fases de un compilador: análisis léxico, sintáctico y semántico. A lo largo de su desarrollo, hemos podido comprobar la complejidad real de construir un sistema robusto capaz de detectar y manejar errores de forma controlada, así como la importancia de seguir un diseño modular y reutilizable.

A pesar de ello, es importante señalar que el periodo en el que se ha planteado esta entrega ha coincidido con la etapa de exámenes de otras asignaturas, lo que ha limitado significativamente el tiempo que hemos podido dedicar al refinamiento de ciertos aspectos. Por esta razón, somos conscientes de que algunos detalles podrían mejorarse y pulirse, pero consideramos que el resultado final es funcional, coherente y aceptable para los objetivos planteados.

En resumen, la práctica ha supuesto una buena oportunidad para poner en práctica conceptos fundamentales del diseño de lenguajes y ha servido para reforzar competencias tanto técnicas como organizativas.