Smart HOme Language

Teoría de Lenguajes

Guillermo Barco Muñoz

May 16, 2016

Contents

1 Puesta en marcha

En la carpeta del código ejecutar el comando "make", esto generará el archivo ejecutable SHoL.

Para utilizar el interprete hay que ejecutar el comando "./SHoL fichero.sol", siendo "fichero" un nombre cualquiera. Se generará un archivo ".cpp" con el mismo nombre.

Como es lógico, para poder compilar es necesario tener instalado el compilador de c++. El proyecto utiliza funcionalidad de c++ 11.

2 Arquitectura

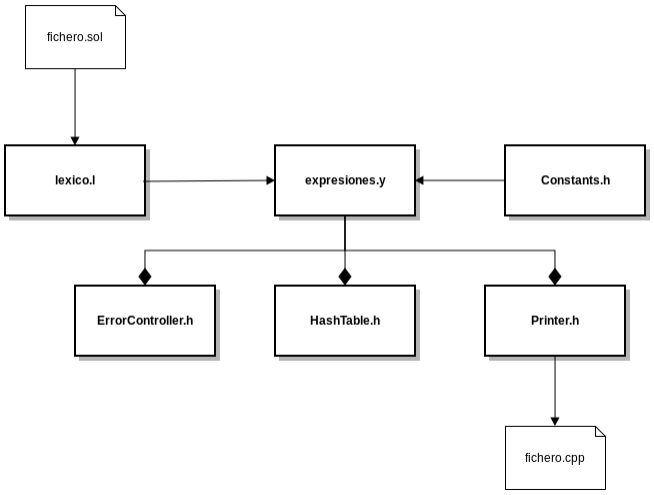

Figure 1: Diagrama de clases

2.1 ErrorController

Es una clase encargada de gestionar los errores de tipo semántico.

Tiene un único método encargado de obtener un código de error y una serie de parámetros opcionales dependiendo de los que sean necesarios para mostrar suficiente información sobre el error.

```
1 void ErrorController::errorCatcher(short errorCode,
2       const char* parameter1, const char* parameter2,
```

```
3       const char* parameter3) {
4
5    cout << "Error semntico en la linea " << n_lineas << ", ";
6
7
8    switch (errorCode) {
9
10   case constants::ERRORPOSITIONINARITHMETIC:
11      cout << "se ha utilizado la variable " << parameter1
12           << " de tipo posicion en una operacion aritmetica"
13           << endl;
14      break;
```

2.2 Constants

Simplemente es un archivo ".h" que define el namespace "constants" donde se encuentran todos los valores numéricos que hay a lo largo del código (tipos de variables en la tabla de símbolos, tipos de error, etc). De esta manera, todas las referencias a un valor están unificadas y se pueden cambiar cuando se desee y además no hay valores numéricos desperdigados por el código, sino referencias a constantes con sentido semántico.

```
1  namespace constants {
2
3  const short TYPEINTEGER = 0;
4  const short TYPEREAL = 1;
5  const short TYPEBOOLEAN = 2;
6  const short TYPEPOSITION = 8;
7  const short TYPESENSOR = 9;
8  const short TYPEACTUATOR = 10;
9
10 const short ERRORPOSITIONINARITHMETIC = 3;
11 const short ERRORNONDECLARED = 4;
12 const short ERRORREDEFINED = 5;
13
14 const short TYPETEMPERATURE = 11;
15 const short TYPEBRIGHTNESS = 12;
16 const short TYPESMOKE = 13;
17 const short TYPEALARM = 14;
18 const short TYPELIGHT = 15;
19
20 const short OK = 7;
21
22 const short PRINTMARKSENSOR = 20;
23 const short PRINTDISABLEACTUATOR = 21;
24 const short PRINTENABLEACTUATOR = 22;
25 const short PRINTVALUESENSOR = 23;
```

```
26 const short PRINTPAUSE = 24;
27 const short PRINTMESSAGE = 25;
28
29 const short PAUSETIME = 1;
30
31 }
```

```
1 /* Determines which kind of sensor or actuator is*/
2 sensorOrActuator: SENSORTEMPERATURE {$$ = constants::TYPETEMPERATURE;}
```

2.3 Printer

Clase que se encarga de abrir el fichero de salida, volcar las instrucciones que recibe del analizador sintáctico y generar el fichero ".cpp" si no existen errores.

Tiene un método principal que es "print" que recibe un código dependiendo del tipo de instrucción a imprimir y la clave de la tabla de simbolos que guarda la información de la variable.

```
1  void Printer::print(short typeOfSentence, string key) {
2
3     VariableDetail variableDetail;
4
5     switch (typeOfSentence) {
6
7     case constants::PRINTMARKSENSOR:
8
9        variableDetail = hashTable->getValueByKey(key);
10       outputFlow << "marca_sensor(" << variableDetail.position1 << ","
11            << variableDetail.position2 << ","
12            << hashTable->sensorActuatorInfo(variableDetail.specificType)
13            << ",\"" << key << "\");" << endl;
14       break;
```

2.4 HashTable

Es la clase encargada de almacenar y gestionar la tabla de símbolos. Tiene una estructura de tipo unordered_map (c++ 11), es un tipo map que almacena clave valor y tiene un acceso mediante hashing con coste O(1) (ver `http://www.cplusplus.com/reference/unordered_map/unordered_map/`).

La clave es un string con el nombre de la variable y el valor es un struct que almacena toda la información necesaria de una variable.

```
struct VariableDetail {

   short type;
   short specificType;
   float value;
   float position1;
   float position2;
};

unordered_map <string,VariableDetail> table;
```

Al finalizar la ejecución se muestra por consola la tabla de símbolos si no ha habido ningún error.

```
Creating myfile.cpp file
Symbol table
+--------------+--------------+--------------+--------------------+
|Name          |Type          |Value         |Mas                 |
+--------------+--------------+--------------+--------------------+
|B             |actuador      |<100, 320>    |Alarma              |
|L_2           |actuador      |<250, 200>    |Luz                 |
|H             |sensor        |<100, 180>    |Humo                |
|P_INICIAL     |posicion      |<0, 0>        |                    |
|P_S_HUMO      |posicion      |<100, 180>    |                    |
|VALOR_UMBRAL  |real          |5             |                    |
|L_1           |actuador      |<250, 100>    |Luz                 |
|FILA          |entero        |500           |                    |
|A             |sensor        |<100, 100>    |Temperatura         |
|COLUMNA       |entero        |400           |                    |
+--------------+--------------+--------------+--------------------+
```

Figure 2: Tabla de símbolos

3 Conflictos

Existe un único conflicto de desplazamiento reducción que es imposible de eliminar. A la hora de escribir un mensaje, es posible que la palabra reservada 'escribir' vaya seguida por una posición (token VARIABLE) y posteriormente por una expresión aritmética.

Puesto que una expresión aritmética puede contener también una variable, bison no puede determinar si la variable que se encuentra pertenece al parámetro opcional que determina la posición o es parte de la expresión aritmética.

```
State 99

   46 action: WRITE VARIABLE . text
   58 arithmeticExpression: VARIABLE .

    VARIABLE shift, and go to state 29
```

```
    TEXT     shift, and go to state 100
    ENTERO   shift, and go to state 30
    REAL     shift, and go to state 31
    '-'      shift, and go to state 33
    '('      shift, and go to state 34

    '-'      [reduce using rule 58 (arithmeticExpression)]
    $default reduce using rule 58 (arithmeticExpression)

    text                go to state 119
    arithmeticExpression go to state 103
```

4 Control de errores

Aunque la calculadora, proyecto del que partía este interprete controlaba algún error más, me he limitado a controlar únicamente los errores requeridos en el enunciado del proyecto para intentar mantener la claridad y legibilidad del código.

- Usar una variable, sensor o actuador que no ha sido previamente definido

```
   case constants::ERRORNONDECLARED:
      cout << "la variable " << parameter1 << " no ha sido definida"
           << endl;
      break;
```

- Cambiar el tipo de una variable ya definida

```
   case constants::ERRORREDEFINED:
      cout << "se ha cambiado el tipo de la variable " <<
          parameter1 << " de tipo "
           << parameter2 << endl;
      break;
```

- Realizar operaciones aritméticas con variables de tipo posición

```
   case constants::ERRORPOSITIONINARITHMETIC:
      cout << "se ha utilizado la variable " << parameter1
           << " de tipo posicin en una operacin aritmtica"
           << endl;
      break;
```

- Utilizar posiciones fuera de la zona de trabajo permitida.

  Este error no he sabido como controlarlo puesto que el plano puede tener

diversas formas geométricas y sería necesario calcular su área y posteriormente si el punto está dentro del plano

5 Ampliaciones

Existen dos ampliaciones.

5.1 Plano

Se crea un plano que determina la forma y dimensiones de la casa. Es una producción del tipo:

```
plano: PLANO = { puntos del plano };
```

Los puntos del plano pueden ser un número indeterminado de variables de tipo posición o pares de valores que a su vez pueden ser expresiones aritméticas.

Como es lógico para poder formar un plano, al menos son necesarios tres puntos. Por cada pareja de puntos se genera una línea en el código en c++.

```
linea(350,400,350,350);
```

El primer y el segundo punto forman la primera línea, la segunda está formada por el segundo y el tercer punto, así sucesivamente hasta el último punto que forma una línea con el primero.

La solución que he implementado ha sido crear una variable de tipo pair que almacena la primera posición y otra que almacena la posición anterior a la que se está procesando. De esta forma se van creando las líneas según se procesa cada punto y al llegar al final se crea la línea con el primer pair.

Otra de las opciones que barajé fue la de crear una estructura de datos auxiliar para guardar los puntos y luego generarlos todos pero la primera me pareció más sencilla.

```
1  /* Here is detected plano, a set of positions given by pairs or
       positions */
2  plano: PLANO '=' '{' planoFirstPoint ',' planoPoints '}' ';'
       {printer->printPlanoLinePair(previousPosition,firstPosition);}
3
4  planoPoints: planoPoint;
5    | planoPoints ',' planoPoint;
6
```

```
planoPoint: VARIABLE
    {printer->printPlanoLinePosition(previousPosition,$1);
    updatePreviousPositionWithPosition($1);}
  | '<' arithmeticExpression ',' arithmeticExpression '>'
      {printer->printPlanoLinePair(previousPosition,make_pair($2,$4));
      updatePreviousPositionWithPair($2,$4);}

planoFirstPoint: VARIABLE {createFirstPositionWithPosition($1);}
  | '<' arithmeticExpression ',' arithmeticExpression '>'
      {createFirstPositionWithPair($2,$4);}
```

5.2 Bucles dentro de condiciones y viceversa

Puesto que almaceno en una lista las sentencias que hay dentro de una producción "mientras" para repetirlas al llegar al final, la solución es sencillamente no almacenar en la lista aquellas sentencias que se procesen cuando la variable "ejecutar" este puesta a falso. La variable "ejecutar" se iguala a falso cuando no se cumple la condición de una producción de tipo "si", "sino".

```
  if(execute) { /*If there is an if acting, do or not do the action*/
    if(!inWhile){ /*If there is a while acting repeats the
        instruction*/
      printer->print(constants::PRINTENABLEACTUATOR,$2);
    }
    else {

      whileInstructions.push_back(make_pair(constants::PRINTENABLEACTUATOR,$2));
    }
  }
```

6 Histórico de la realización del proyecto

He realizado un total de diez commits, el primero el día 7 de Mayo y el último el día 16 de Mayo. Aunque había trabajo hecho previo al primer commit.

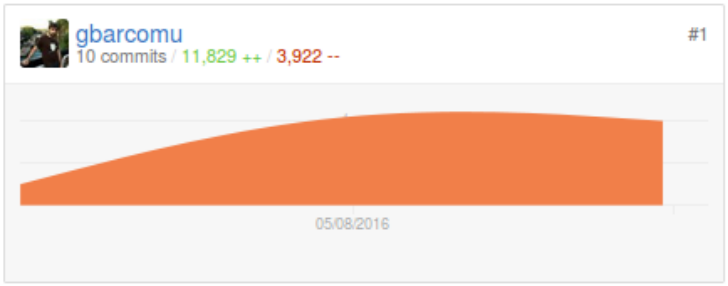

Figure 3: Gráfico de la cantidad de trabajo a lo largo del tiempo

El código se puede encontrar en `https://github.com/gbarcomu/SHoL`

7 Fortalezas y debilidades

7.1 Fortalezas

- Documentación interna.
- Variables largas y descriptivas.
- Sólo un conflicto gramatical.
- Bien modularizado.
- Control de versiones mediante "Github".

7.2 Debilidades

- No están implementadas tantas ampliaciones como me hubiera gustado.
- Aunque estoy satisfecho con el sistema de control de errores, no lo estoy con la cantidad de errores que se controlan.
- Poco tiempo para hacer pruebas.

8 Conclusiones

- El proyecto en general me ha gustado mucho, me ha resultado muy interesante la creación de un interprete de un lenguaje de programación inventado.

- Me hubiera gustado profundizar mucho más en el proyecto, haber realizado más ampliaciones, pruebas, etc. Sin embargo me ha sido imposible por la falta de tiempo.
- Me ha costado bastante, pero creo que he conseguido luchar contra el código spaghetti y hacer código modularizado y legible.
- Finalmente integré el proyecto en el IDE Eclipse tocando algunos parámetros de configuración, gestionar este proyecto con un editor de texto resulta extremadamente complicado y tedioso.