Práctica de Analizador Semántico

Objetivos

El objetivo de la práctica es construir el analizador semántico para el lenguaje diseñado. Para simplificar y estructurar esta práctica, se dividirá en dos partes:

Parte I: Ampliación de las prácticas de PDL I

En esta parte de la práctica se modificará el analizador sintáctico para poder realizar acciones semánticas avanzadas según se especifica en las siguientes partes de la práctica.

I.1 Tareas a realizar.

1. Ampliar la gramática especificada para permitir el anidamiento de procedimientos y/o funciones en el lenguaje especificado. No existirá límite en el anidamiento y la declaración de procedimientos podrás hacerse tanto al final de la zona de declaración de variables como dentro de una zona de declaraciones común.

2. Ampliar el número de tipos simples en el lenguaje especificado. Debe existir al menos el tipo *entero, booleano, char* y *real*. El tipo *cadena* de caracteres también se exige y se considera tipo básico. En el caso de disponer del tipo *cadena* como único tipo avanzado, los alumnos deberán considerar la inclusión de un tipo avanzado diferente, como pueden ser las listas, pilas, colas, árboles, conjuntos, tablas hash, etc…; así como añadir alguna funcionalidad al tipo avanzado.

3. Eliminar las producciones de error de la especificación sintáctica.

4. De forma **optativa**, el alumno podrá implementar las operaciones del tipo avanzado que dispongan como si se tratase de operaciones en notación infijo. Por ejemplo, en vez de utilizar el operador *inserta(a,l)* (a:tipo_simple, l:lista), se podrá hacer algo así: *a -> l.*

Parte II: Implantación de la tabla de símbolos

En esta parte de la práctica se introducirán las acciones YACC necesarias para construir y visualizar la tabla de símbolos. La visualización debe presentar la actualización de la tabla de símbolos a medida que se analiza cada bloque. Dichas acciones se incluirán junto a las producciones que definan las declaraciones. Esta segunda parte sólo será de utilidad para la puesta a punto de la práctica completa, suponiendo que será de gran utilidad por parte de los alumnos.

La única comprobación semántica que se puede realizar, en esta fase, es detectar si una variable ha sido declarada, en un mismo bloque, más de una vez.

II.1 Tareas a realizar.

1. Determinar cuales son todas las acciones semánticas necesarias para poder realizar las comprobaciones (ver sección de comprobaciones semánticas), y en que reglas de la gramática abstracta deben ser insertadas esas comprobaciones.

2. Una vez obtenido lo anterior, diseñar las estructuras de datos (ver sección de estructuras de datos) y las operaciones que vamos a realizar sobre dichas estructuras de datos, con el fin de realizar las acciones semánticas establecidas.

3. Insertar, en la especificación YACC, las acciones necesarias para construir y **visualizar** la tabla de símbolos, así como realizar la comprobación semántica propia de esta parte..

II.2 Prueba sugerida para la primera parte de la práctica.

Se sugiere que cada grupo de prácticas pruebe esta primera parte usando el programa prueba confeccionado para la prueba de la práctica anterior (la correspondiente a PL-I), en donde podrá comprobar el comportamiento de la tabla de símbolos. También deberían introducir errores en declaración de variables y se comprobarán que los mensajes de error son los adecuados.

Parte III: Comprobaciones Semánticas. Implantación.

En esta parte de la práctica se introducirán las acciones YACC necesarias para el resto de las comprobaciones semánticas.

II.1 Tareas a realizar.

Extender, y modificar en caso de ser necesario, la especificación YACC obtenida en la parte anterior para realizar las comprobaciones semánticas propias de cada lenguaje. Para ello, se añadirá el código necesario para manejar las estructuras de datos diseñadas en el apartado anterior y realizar las acciones semánticas necesarias. El programa EJECUTABLE obtenido debe realizar todo el análisis semántico y sintáctico en una sola pasada.

II.2 Prueba de la práctica.

Se partirá del programa confeccionado para la defensa de la práctica anterior (correspondiente a PL-I) sin errores.

Prueba de la práctica para su defensa:

1º.- Se pasará el analizador sobre el programa prueba comprobando que está libre de errores.

2º.- El profesor introducirá los siguientes errores:

- Se introduce un error léxico en un tipo, generando error, semántico y sintáctico (0.1 puntos),

- Dentro de una función/procedimiento se declara una variable igual a un argumento (0.8 puntos).

- Se hacen llamadas a funciones/procedimientos con el número y tipo de argumentos erróneo y de ámbitos distintos (1 puntos).

- Se introducen error de tipo en expresiones lógicas y aritméticas (0.6 puntos),

- Se introducen errores en expresiones complejas del tipo compuesto (0.5 puntos).

- Se introducen errores en otros aspectos del lenguaje, como pueden ser la definición de tipos, constantes, etc. (1 puntos).

Nota: NO SE DEBE PRESENTAR LA TABLA DE SÍMBOLOS. SOLO LOS ERRORES LÉXICOS, SINTÁCTICOS Y SEMÁNTICOS, a no ser que el profesor lo pida de manera expresa.

3º.- Adicionalmente, el alumno podrá realizar comprobaciones semánticas para las características especiales de su lenguaje o definición de lenguaje. Por ejemplo, comprobaciones semánticas en la definición, uso de tipos complejos, estructuras *switch,* etc… (0.5 puntos).

La puntuación se obtendrá como suma de los puntos asignados para cada tipo de error introducido que hayan sido detectados (**puntuación máxima de 4 puntos, incluidos los puntos por comprobaciones extra**). Esta práctica es superada si la calificación alcanzada es igual o mayor 2.

II.3 Documentación a presentar.

Previa a la realización de la práctica, el alumno o el grupo (máximo de 2 personas) deberá informar al profesor de las características específicas de su lenguaje enviando un fichero de texto a una actividad creada en ILIAS donde se especifique la información que corresponda en los apartados siguientes:

- español o inglés
- C o Pascal
- funciones o procedimientos
- estructuras de control (for, while)
- tipos básicos del lenguaje (enteros, reales, cadenas, punteros, ...)
- tipo de datos complejos (listas, pilas, ...)
- definición de tipos complejos (SI/NO)
- definición de constantes (SI/NO)
- inclusión de ficheros (SI/NO)

Tras la realización de la práctica y el mismo día de la defensa: listado del código fuente correspondiente a la implementación del analizador semántico. Esto incluye tanto las acciones incluidas en la especificación YACC como las implementaciones de dichas acciones en las rutinas de usuario.

III Anexos

III.1 Comprobaciones Semánticas

Hay que tener en cuenta que, según las opciones escogidas por cada grupo de prácticas, algunas de las comprobaciones, acciones o elementos de la estructura de datos no van a ser necesarias. Cada grupo debe razonar cuáles son los elementos que se necesitan y cuáles no.

Por ámbito de un identificador (variables, parámetros, funciones y procedimientos) entendemos el bloque en el que está declarado y todos aquellos bloques incluidos en él, directa o indirectamente.

En el caso de parámetros de procedimientos y funciones, entendemos que el bloque en el que están declarados es el que forma el cuerpo del subprograma. En adelante, supondremos que cada expresión, sub-expresión o término que aparece en el programa tiene un tipo determinado de entre los permitidos. El lenguaje realizará comprobación fuerte de tipos, al estilo de Pascal.

Las comprobaciones semánticas que debe realizar nuestro compilador van a ser las siguientes:

- El punto en que se usa un identificador pertenece a su ámbito.

- En el ámbito de un identificador puede declararse otro con el mismo nombre, pero no en el mismo bloque en el que está declarado el primero.

- En el caso de asignaciones, el tipo en la parte izquierda coincide con el tipo de la expresión en la parte derecha.

- En el caso de llamadas a procedimientos y funciones, el tipo, número y orden de las expresiones que forman los parámetros actuales coinciden con el tipo, número y orden de los parámetros formales especificados en su declaración.

- En el caso de expresiones que incluyan un operador (ya sean unários, binarios o ternarios), comprobar que el operador es compatible con el tipo de las sub-expresiones sobre las que actúa.

III.2 Estructura de la tabla de símbolos

Por tabla de símbolos (t.s.) entenderemos una estructura de datos que contendrá, en cada momento, la información necesaria acerca de todos los identificadores (o símbolos) cuyo ámbito incluya el punto actual de compilación (esto es, todos los identificadores que se pueden usar). Por información necesaria entendemos todos los datos requeridos para hacer las comprobaciones semánticas. Esta información se agrupará en forma de registro, al que denominaremos *entrada*, y que tendrá una estructura fija independientemente del tipo de identificador que describe.

La anterior definición hace necesario incluir en la t.s. una entrada cada véz que un nuevo identificador es declarado, así como eliminar de la t.s. una entrada cuando finalice el ámbito del identificador que describe. Lo primero ocurre siempre en la sección de declaraciones de un bloque, y lo segundo al final de los bloques. Todo esto nos lleva a estructurar la t.s. como una pila de entradas. En ella metemos y sacamos entradas a medida que comienzan o terminan los ámbitos de los identificadores. Al encontrarnos con la declaración de una variable, construimos una nueva entrada y hacemos un *push* de ella en la t.s. Cuando un bloque finaliza, debemos hacer *pop* de todas las entradas de identificadores declarados en dicho bloque.

Para realizar esto último, podemos definir un tipo de entrada especial que marca en la pila el comienzo de un bloque, con lo cual al desactivar el bloque eliminaremos todas las entradas hasta la primera marca que encontremos. Al entrar en un bloque, lo primero que haremos será un *push* de esta marca en la pila.

Algunas entradas llevarán información sobre un identificador que referencia a un procedimiento o función, y por tanto será necesario incluir información sobre sus parámetros. Para realizar esto, se incluirán los nombres y tipos de los parámetros formales de un sub-programa en las sucesivas entradas de la tabla de símbolos posteriores a la correspondiente a una función. Estas entradas especiales estarán marcadas para indicar que no corresponden a identificadores activos. Se usarán al comprobar los parámetros actuales de las llamadas a subprogramas con respecto a los parámetros formales de la aplicación.

Algo parecido ocurre con la lista de rangos de una matriz, en el caso de matrices multi-dimensionales. En la entrada correspondiente a la matriz incluiremos el rango de la primera dimensión. En el caso de que el número de dimensiones sea mayor que 1, en las sucesivas entradas se indicarán los rangos de las sucesivas dimensiones. Estas otras entradas especiales estarán debidamente marcadas.

Según el tipo de lenguaje, existen dos opciones de estructura de la sección de declaración de variables, que son:

- En algunos casos, la lista de identificadores precede a la descripción del tipo de la variable. Al incluir una entrada en la tabla de símbolos, no sabremos aún su tipo. Por tanto será necesario incluirla con una marca especial, que indica que el tipo aún no ha sido asignado. Al procesar la

declaración del tipo, se deben actualizar las entradas de la tabla de símbolos cuyo tipo no hubiese sido asignado aún.

- En otros casos, la lista de identificadores va después de la descripción del tipo. Aquí debemos almacenar el tipo en una variable intermedia (que llamaremos *TipoTmp*). Después, según van apareciendo los identificadores, se dan de alta en la tabla de símbolos, tomando el tipo de dicha variable intermedia.

Todas las consideraciones anteriores nos llevan a definir las entradas de la tabla de símbolos como una estructura con los siguientes componentes:

***TipoEntrada**:* Indica el tipo de entrada. Este valor será de un tipo enumerado, con los siguientes valores posibles:

> *marca* Indica que la entrada es una marca de principio de bloque
>
> *procedimiento* La entrada describe un procedimiento
>
> *funcion* La entrada describe una función
>
> *variable* La entrada describe una variable local
>
> *parametro-formal* La entrada describe un parámetro formal de un procedure o funcion situado en una entrada anterior de la tabla
>
> *rango-array* La entrada describe el rango en una dimensión de un array situado en una entrada anterior de la tabla.

Nombre: Un puntero a una cadena de caracteres (char *) que contiene los caracteres que forman el identificador. En el caso de que *TipoEntrada* sea *marca* o *rango-array*, no se usará

TipoDato: En el caso de que *TipoEntrada* sea *funcion*, *variable*, o *parametro-formal*, indica el tipo del dato al que hace referencia, en otro caso no se usa. Será un tipo enumerado cuyos valores son uno de entre los siguientes: *booleano, entero, array, pila, conjunto, cadena* y además:

> *desconocido* Indica que no se conoce el tipo de dato, debido a un error de sintaxis en la descripción del tipo en su declaración.
>
> *no-asignado* Para lenguajes en los que en las declaraciones el tipo siga a los identificadores, indica que aun no se ha alcanzado la descripción del tipo y por tanto aun no ha sido asignado tipo a esta entrada.

Parámetros: Es un valor entero positivo que indica el numero de parámetros formales, solo si el *tipo* es *procedimiento* o *función*, en otro caso no se usa.

***Dimensiones**:* En el caso de que *TipoDato* sea *array*, es un valor entero positivo que indica el numero de dimensiones del array.

***MinRango** y **MaxRango**:* Son dos valores enteros que indican el rango posible de valores del índice de un array. Solo tiene sentido si *TipoDato* es *array*, o bien *TipoEntrada* es *rango-array*.

Una vez definidas las entradas, podemos definir la tabla de símbolos como un array unidimensional de entradas como la descrita (lo llamaremos *TS*). Este array tendrá un tamaño máximo fijo predefinido, que se hará lo suficientemente grande como para que se pueda procesar cualquier programa de tamaño prueba (no superior a 1000 líneas). Además del array, existirá una variable

entera (*Tope*) inicializada a -1 y que indica en cada momento la última entrada del array usada. *Tope* debe declararse como un entero largo (*long int*).

Para introducir una entrada en *TS*, bastará incrementar el valor de *Tope* en uno y después asignar valores a los elementos de la entrada *TS[Tope]*. Para extraer el último elemento de *TS*, solo es necesario disminuir tope en uno. En el caso de que *Tope* alcance el valor máximo prefijado, se producirá un error y se abortara la compilación. Para ello se debe definir una función mediante la cual se incremente *Tope*, realizando la comprobación descrita.

Línea: línea del código en la que se declaró (en el caso de lenguajes con declaración explícita) o se usó por primera vez (para lenguajes con declaración implícita) un objeto (variable, procedimiento, constante, ...) del lenguaje.

III.3 Estructura de los atributos sintetizados

Al usar YACC, disponemos de una pila (gestionada internamente) de atributos sintetizados por los símbolos de la gramática. El usuario de YACC tiene la posibilidad de redefinir este tipo de dato para adaptarlo a sus necesidades. En nuestro caso, usaremos los atributos sintetizados fundamentalmente con dos fines:

- Para informar al analizador sintáctico de los atributos de los tokens que encuentra el analizador léxico (a través de la variable *yylval*). En este caso, los atributos sintetizados pueden ser de los siguientes tipos:

- Puntero a una cadena de caracteres, para identificadores y literales

- Valores enteros, para constantes enteras y booleanas

- Para sintetizar el tipo en las expresiones, términos y variables de asignaciones (a través de *$$*).

Aquí necesitaremos la suficiente información para describir completamente el tipo de una expresión.

Con estas consideraciones, podremos definir los atributos sintetizados como una estructura tipo *union* de C, que contiene cada una de las alternativas posibles detalladas arriba. En el fuente BYACC incluiremos la directiva de redefinición del tipo de la pila, como sigue:

```
typedef struct {
        int entero ;
        char * cadena ;
        struct dtipo tipo ;
} atributos ;

#define YYSTYPE atributos
```

De esta forma, las variables de YACC yylval, *$$*, *$0*, *$1*, *$2*, etc..., serán del tipo *atributos*.

Respecto al campo *dtipo*, incluirá un valor de un tipo enumerado para indicar el tipo básico de la expresión (*entero*, *booleano*, *conjunto*, *pila*, *array*). Podemos usar el mismo tipo enumerado que usamos en el campo *TipoDato* en las entradas de la tabla de símbolos.

En la mayoría de los casos, esta información es suficiente. En el caso de array, necesitamos además incluir información sobre el número de dimensiones y la lista de rangos posibles del índice en cada dimensión. Para hacer ésto podemos usar listas enlazadas en memoria dinámica, y almacenar dentro de los registros *dtipo* un puntero a la cabecera de dicha lista.

III.4 Acciones semánticas para la construcción de la tabla de símbolos.

Una vez definidas las estructuras de datos necesarias, veremos cuales son las acciones semánticas asociadas a las reglas sintácticas.

1. En el análisis de léxico, al reconocer un identificador, se debe de guardar en *yylval* un puntero a la cadena de caracteres que lo forman, de forma que el analizador sintáctico tenga acceso a esa información. Para realizar ésto, habría que guardar en el campo cadena de *yylval* una copia (obtenida con *strdup*) del contenido de *yytext* (es importante no usar referencias directas a *yytext*, puesto que su contenido se reescribe cada vez que se reconoce un token en la entrada). En el fuente YACC, este valor es leído como el atributo sintetizado por el símbolo terminal "identificador".

2. Al reconocer un identificador en una declaración de variables es necesario introducir una nueva entrada en *TS.* Si hubiese otro identificador en *TS* con el mismo nombre y posterior a la última marca de bloque, se produce un error de identificador duplicado.

3. Al entrar en un bloque, se debe incluir en la tabla de símbolos una marca de inicio de bloque. En el caso de que el bloque corresponda al cuerpo de un subprograma, los parámetros formales (que se encuentran por encima de la marca en la tabla de símbolos) deberán ser introducidos de nuevo en *TS* como si fuesen variables locales.

4. Al reconocer el tipo en una declaración de variables, será necesario asignar tipo a las variables de *TS* que no lo tuviesen asignado (para lenguajes con el tipo después de los nombres de las variables) o bien asignar la variable temporal de transito *TipoTmp* con este tipo. Para lo anterior será necesario que el no terminal asociado con la descripción de tipo tenga como atributo sintetizado precisamente el registro descriptor de tipo (*dtipo*) dentro de la estructura *atributos.*

5. Al alcanzar el final de un bloque, se deben eliminar de *TS* todas las entradas hasta la última marca de inicio bloque, incluida ésta.

6. Al reconocer un identificador que contenga el nombre de un procedimiento, en su declaración, se insertará en la tabla de símbolos una nueva entrada. El número de parámetros se pondrá inicialmente a cero.

7. Al reconocer un identificador en la lista de parámetros formales de la cabecera de un procedimiento o función, se insertará una entrada en la tabla de símbolos, con el nombre y tipo del identificador.

Además, será necesario incrementar el número de parámetros de dicho subprograma, que se encontrará en una entrada anterior de la TS.

III.5 Acciones para las comprobaciones semánticas.

1. Todos los identificadores detectados deben aparecer en la TS.

2. En las sentencias de control de flujo, como *if*, *while*, etc. se debe comprobar que la expresión que forma la condición es de tipo booleano.

3. En las sentencias de asignación, el tipo de la variable en la parte izquierda debe coincidir con el tipo de la expresión en la parte derecha. Esto es también así para las asignaciones al contador en las sentencias tipo *for.*

4. En las expresiones compuestas de sub-expresiones o términos relacionados con operadores, se debe comprobar que los tipos son concordantes, y se debe sintetizar el tipo ascendentemente.