Práctica 3: Análisis semántico de programas *adac*

Procesadores de lenguajes

Dpto. de Informática e Ingeniería de Sistemas,
Grado de Ingeniería Informática
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

1. Objetivos

Los objetivos para esta práctica son:

- Introducirse en las técnicas de análisis semántico
- Entender y manejar una tabla de símbolos
- Añadir análisis semántico para *adac*
- Aprender a gestionar acciones semánticas en *javacc*

2. Introducción

Esta práctica, a desarrollar durante tres sesiones de laboratorio, está dividida en dos partes. En la primera se trabajará con la tabla de símbolos como mecanismo para la gestión de los símbolos declarados en un programa. En la segunda se integrará el análisis semántico en el compilador de *adac* en desarrollo.

3. Alcance del compilador

El desarrollo de las tareas de un compilador, especialmente en lo concerniente al análisis semántico y las fases de síntesis, requiere la consideración de muchos aspectos que, si bien con la comprensión correcta de los conceptos no son de gran dificultad, son bastante trabajosos. Por eso se han establecido cuatro niveles diferentes de lenguaje, que se

corresponden con cuatro niveles de dificultad y de carga de trabajo, que se reflejan en la calificación final a la que se puede aspirar. Los niveles son los siguientes:

Nivel 1: El lenguaje no considera el uso de parámetros en procedimientos y funciones, aunque los procedimientos y funciones pueden tener variables locales. La calificación para este nivel de restricción será de hasta **6.0/10.0**.

Nivel 2: El lenguaje permite el uso de parámetros escalares (tipos simples) tanto por valor como por referencia en procedimientos y funciones, pero no admite parámetros de tipo vector. La calificación para este nivel de restricción será de hasta **7.0/10.0**.

Nivel 3: El lenguaje permite el uso de parámetros escalares (tipos simples) tanto por valor como por referencia en procedimientos y funciones, y de parámetros de tipo vector por referencia. La calificación para este nivel de restricción será de hasta **8.0/10.0**.

Nivel 4: El lenguaje permite el uso de parámetros escalares y de vectores, tanto por valor como por referencia en procedimientos y funciones. La calificación para este nivel de restricción será de hasta **10.0/10.0**.

El nivel al que corresponde el compilador entregado se determinará con el nivel de la entrega final. Por otra parte, y con el objetivo de incidir en algunos aspectos semánticos interesantes, vamos a añadir una nueva característica al lenguaje: ***adac* permite procedimientos y funciones anidados**. Es decir, se pueden declarar procedimientos y funciones dentro de procedimientos y funciones, sin límite en el nivel de anidamiento. Esta característica debe tenerse en cuenta en todos los niveles de compilador.

4. La tabla de símbolos

Como hemos comentado en clases de teoría, una tabla de símbolos es la estructura de datos que vamos a utilizar para almacenar la información de cada símbolo declarado en un programa. Esta información es imprescindible para poder llevar a cabo tanto el análisis semántico como las fases de síntesis (generación y "optimización" de código).

Dado que una tabla de símbolos es un tipo de dato con unos requisitos muy bien definidos, su implementación no deja de ser un ejercicio de implementación de un tipo abstracto de datos que respete los requisitos. Por eso se suministra una tabla de símbolos básica ya implementada que podéis usar. Podéis desarrollar la vuestra propia, si queréis.

Las características fundamentales de la implementación suministrada ya se comentaron en clase. También se mostraron algunos ejemplos de uso. La siguiente figura muestra cómo está organizado el desarrollo de las clases involucradas en su implementación.

Los fuentes de la tabla de símbolos se encuentran en el fichero *symbolTable.zip* que se puede descargar desde moodle. Para manejar los símbolos del programa debemos completar el analizador sintáctico con las acciones correspondiente a la inserción y eliminación de símbolos de la tabla. Para ello debemos:

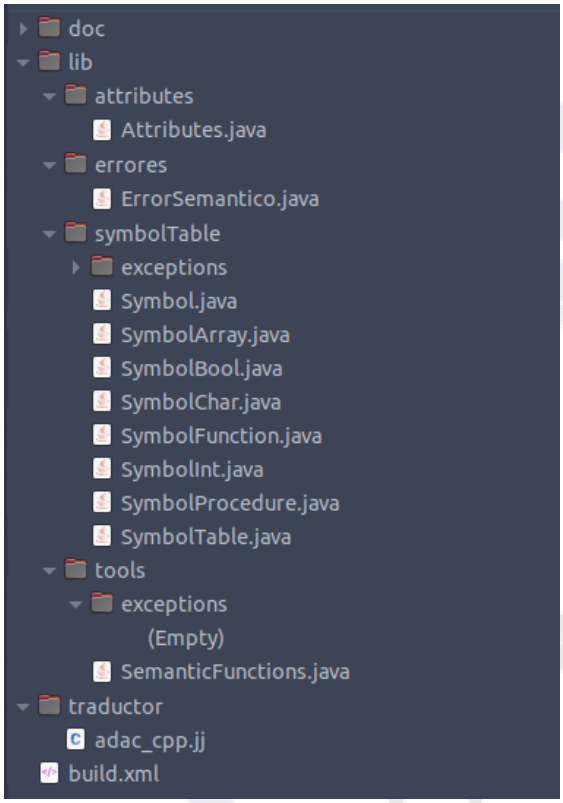

Figura 1: Estructura del proyecto incluyendo la tabla de símbolos. Incluye, además de la tabla de símbolos, otras clases para la gestión de atributos, errores o procedimientos semánticos.

1. En las producciones en que se declare un nuevo símbolo en el programa (variable, parámetro, procedimiento o función), añadir el código para introducir el símbolo en la tabla. En caso de que dicho símbolo ya exista, se debe capturar la excepción y mostrar un mensaje de error. El análisis deberá continuar.

2. En las producciones en que se cierran bloques de programa (fin de procedimiento o función), añadir el código para eliminar de la tabla los símbolos que fueron declarados en el bloque.

Con el objeto de comprobar que el rellenado de la tabla de símbolos se lleva a cabo correctamente, durante las pruebas se puede mostrar el contenido de la tabla de símbolos cada vez que se vaya a cerrar un bloque de procedimiento o función, incluyendo el principal. Por ejemplo, ejecutando *System.err.println(st.toString())*, donde *st* es la tabla de símbolos).

A modo de ejemplo, si consideramos el programa mostrado en el anexo, la salida debería ser del estilo siguiente:

```
**********************
```

```
Procesando b_dec
**********************
------------------------------------------------
enB=(enB,INT,0,NONE,0)
cambio_base=(cambio_base,PROCEDURE,[],0)
base=(base,INT,0,NONE,0)
b_dec=(b_dec,FUNCTION,[(n,INT,0,VAL,1), (b,INT,0,VAL,1)],INT,NONE,0)
num=(num,INT,0,NONE,0)
    b=(b,INT,0,VAL,1)
    n=(n,INT,0,VAL,1)
    uc=(uc,INT,0,NONE,1)
    valRec=(valRec,INT,0,NONE,1)
------------------------------------------------

**********************
Procesando dec_b
**********************
------------------------------------------------
enB=(enB,INT,0,NONE,0)
cambio_base=(cambio_base,PROCEDURE,[],0)
dec_b=(dec_b,FUNCTION,[(n,INT,0,VAL,1), (b,INT,0,VAL,1)],INT,NONE,0)
base=(base,INT,0,NONE,0)
b_dec=(b_dec,FUNCTION,[(n,INT,0,VAL,1), (b,INT,0,VAL,1)],INT,NONE,0)
num=(num,INT,0,NONE,0)
    b=(b,INT,0,VAL,1)
    n=(n,INT,0,VAL,1)
    resto=(resto,INT,0,NONE,1)
    valRec=(valRec,INT,0,NONE,1)
------------------------------------------------

**********************
Procesando cambio_base
**********************
------------------------------------------------
enB=(enB,INT,0,NONE,0)
cambio_base=(cambio_base,PROCEDURE,[],0)
dec_b=(dec_b,FUNCTION,[(n,INT,0,VAL,1), (b,INT,0,VAL,1)],INT,NONE,0)
base=(base,INT,0,NONE,0)
b_dec=(b_dec,FUNCTION,[(n,INT,0,VAL,1), (b,INT,0,VAL,1)],INT,NONE,0)
num=(num,INT,0,NONE,0)
------------------------------------------------
```

La interpretación de la información mostrada corresponde a la función *toString()* de la implementación de la tabla de símbolos y de las clases de los distintos símbolos (véase el código fuente).

5. Análisis semántico

Como resultado de esta práctica, tenéis que integrar el análisis semántico en vuestro compilador de *adac*.

Se detallan a continuación los aspectos del lenguaje que deben ser considerados. El análisis semántico tiene que asegurarse de que todos ellos son respetados:

Tipos:

- En *adac* hay cuatro tipos de constantes literales: `integer, boolean, character, string`.
- En *adac* hay tres tipos de variables y parámetros: `integer, boolean y character`
- Los vectores, cuyas componentes han de ser de uno de esos tres tipos, empiezan en la componente de índice 0.
- Es un sistema de tipos rígido: no hay compatibilidad entre tipos distintos para ninguna operación.
- Hay dos funciones para convertir un valor simple de un tipo a otro: `int2char` (devuelve el carácter cuyo valor ASCII es el parámetro, si el valor es apropiado) y `char2int` (devuelve el valor ASCII del parámetro).
- Dos vectores se consideran del mismo tipo si, y solo si, tienen la misma dimensión y el mismo tipo base.
- Las variables simples, los parámetros simples, y las componentes de variables o de parámetros de tipo vector son asignables (pueden aparecer en la parte izquierda de una asignación o en un paso por referencia). Ni los procedimientos ni las funciones ni el procedimiento principal son asignables.

Operadores:

- Los operadores aritméticos (`+, -, *, div, mod`) sólo admiten operandos simples de tipo integer.
- Los operadores booleanos (`and, or, not`) sólo admiten operandos simples de tipo boolean.
- Los operadores relacionales (`>, >=, <, <=, ==, <>`) sólo admiten operandos simples del mismo tipo.

get/put/put_line/skip_line:

- La instrucción de lectura `get` admite como argumentos una lista no vacía de asignables. Cada asignable ha de ser de uno de los tipos `integer, character`.
- La instrucción de lectura `skip_line` no admite argumentos. En su ejecución se saltan caracteres de la entrada estándar hasta haber leído un carácter `newline` (ASCII 10).

- La instrucción de escritura `put` requiere una lista no vacía de expresiones. Cada expresión ha ser de uno de los tipos `integer, boolean, character, string`.
- La instrucción de escritura `put_line` requiere una lista (posiblemente vacía) de expresiones. Cada expresión ha ser de uno de los tipos `integer, boolean, character, string`.

Asignación:

- La instrucción de asignación `<asignable> := <expresion>` requiere que el asignable y la expresión sean del mismo tipo.
- Los vectores no son asignables (aunque sus componentes sí lo son).

if/while:

- La guarda de las instrucciones de selección y de iteración sólo puede ser simple booleana.

Estructura:

- *adac* es un lenguaje estructurado en bloques. Los bloques son los procedimientos y funciones.
- No puede haber dos símbolos homónimos en el mismo bloque.

Procedimientos y funciones:

- Los parámetros por valor, `val`, de un procedimiento o función sólo admiten argumentos del mismo tipo.
- Los parámetros por referencia, `ref`, de un procedimiento o función sólo admiten asignables del mismo tipo.
- En la invocación de un procedimiento o función deben coincidir el número de parámetros reales y el de parámetros formales y, uno a uno, ser del mismo tipo.
- El procedimiento principal no es invocable.

6. Entrega de resultados de la práctica

6.1. Lo que hay que entregar

Como resultado de la práctica se deberá entregar:

- El fichero `practica_3.zip`. Este, una vez descomprimido, tendrá la estructura mostrada en la figura 1. El nombre del fichero con el fuente deberá ser `adac_1.jj`, `adac_2.jj`, `adac_3.jj` o `adac_4.jj` en función al nivel de trabajo al que se aspire.

- Todos los fuentes requeridos para que la ejecución de `ant` genere todo lo necesario para la correcta compilación del compilador. Al igual que en las prácticas anteriores, el fichero `jar` deberá generarse en directorio `dist`, y llamarse `adac_1.jar`, `adac_2.jar`, `adac_3.jar` o `adac_4.jar`, según corresponda.
- El fichero `README.txt`, del directorio `Doc`, deberá incluir, además de la información proveniente de la práctica 2, información sobre las características del lenguaje aceptado (en función al nivel atacado), así como una descripción de la organización del proyecto: clases introducidas, organización en directorios, etc. Este último aspecto se irá matizando a lo largo del desarrollo de las sesiones de laboratorio.

6.2. Método de entrega

La entrega se debe hacer en *hendrix* mediante la ejecución del programa `someter`, análogamente a como se hizo en las prácticas anteriores.

6.3. Plazos de entrega

Como se ha comentado anteriormente, esta práctica está pensada para ser desarrollado durante las sesiones 4, 5 y 6. Los dos miembros de la pareja deben entregarla, como muy tarde, la víspera del día en que tiene la sesión 7 de laboratorio, a las 20:00. Respectivamente, los días límites para los grupos 1, 2 y 3 son el 16, 17 y 18 de mayo.

Un programa sencillo en *adac*

```
procedure cambio_base is
    integer num, base;
    integer enB;
-----------------------------------------------------
function integer b_dec(val integer n,b) is
    integer uc;
    integer valRec;
begin
    if n = 0 then
        return 0;
    else
        uc := n mod 10;
        valRec := b_dec (n div 10, b);
        return valRec*b + uc;
    end
end
-----------------------------------------------------
function integer dec_b(val integer n,b) is
    integer resto,valRec;
begin
    if (n < b) then
        return n;
    else
        resto := n mod b;
        valRec := dec_b (n div b, b);
        return valRec*10 + resto;
    end
end
-----------------------------------------------------
begin
    num := 4;
    base := 2;

    put_line ("Este␣programa␣convierte␣n=",num,"␣a␣base␣b=", base,
              ",␣y␣luego␣efecúa␣la␣conversión␣inversa.");
    put_line();
    put_line("n:␣",num);
    put_line("b:␣",base);
    enB := dec_b(num,base);
    put_line ("dec_b(",num,",",base,"):␣",enB);
    put_line ("b_dec(",enB,",",base,"):␣",b_dec(enB,base));
end
```