Práctica 4: Generación de código intermedio para *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:

- Reforzar el conocimiento de las técnicas de generación de código
- Traducir programas *adac* a código para la P-máquina virtual

2. Introducción

Esta práctica se desarrollará a lo largo de dos sesiones de laboratorio.

3. Trabajo a desarrollar

En esta práctica deberás implementar un generador de código para la máquina P, que traduzca programas escritos en *adac*. Como resultado, deberás generar un compilador completo que realiza las fases de análisis léxico, análisis sintáctico, análisis semántico y generación de código.

La generación de código debe integrarse con el código de la práctica anterior. Es decir, partiendo del analizador semántico de *adac*, la práctica consiste en añadir la generación de código en las acciones que corresponda. Tu compilador debe cumplir las siguientes características:

- Si el programa fuente es correcto, el compilador sólo debe mostrar por pantalla el siguiente mensaje:

```
Compilación finalizada. Se ha generado el fichero <nombreFichero>.pcode
```

El fichero .pcode contendrá el código generado para la máquina P.

- Si el programa fuente tiene errores, se mostrarán igual que en las prácticas anteriores, y no se generará ningún fichero .pcode.

Si el programa es correcto, el fichero .pcode generado se puede ejecutar con el intérprete que implementa la máquina P, y que se os proporciona como parte de la práctica. Eso sí, antes de pasárselo como entrada al intérprete, debéis pasarlo por el ensamblador (se proporciona), que se encargará de pasar el fichero .pcode, que contiene texto, a un fichero .x, que contiene el mismo código pero en binario. El ensamblador e intérprete que se os proporcionan son binarios que funcionan en Hendrix. Podéis generar el código (.pcode) en vuestra máquina, pero el ensamblado y ejecución (intérprete) deben funcionar en Hendrix.

Por tanto, el funcionamiento normal es el siguiente:

1. Compilar `<fuente>.adac`
2. Si es correcto: se genera un `<fuente>.pcode`
3. Ensamblar `<fuente>.pcode` (ejecutando `ensamblador <fuente>`, en Hendrix)
4. Interpretar `<fuente>.x` (ejecutando `maquinap <fuente>`, en Hendrix)

Además, se os proporcionan los ficheros .pcode de tres de los programas de prueba, para que podáis comprobar el correcto funcionamiento tanto del ensamblador como del intérprete, y tengáis una referencia del código que se debe generar.

4. Materiales proporcionados

Como parte del material necesario para esta práctica se os proporcionan un paquete auxiliar escrito en Java para la gestión de las instrucciones, comentarios, etiquetas, etc., así como los binarios del ensamblador y del intérprete en la página del curso (Moodle). Ambos programas están compilados para funcionar en hendrix.

4.1. Paquete `lib.tools.codeGeneration`

Dentro de este paquete distinguimos tres elementos principales:

- `CGUtils`: es una librería estática que proporciona una factoría de etiquetas *libres* (no usadas anteriormente) mediante el método *newLabel*, y contiene una variable estática auxiliar para gestionar los bloques de memoria llevando cuenta del tamaño de cada bloque de activación (*memorySpaces*).

- `CodeElement`: clase abstracta que representa un elemento de código. Tiene como subclases `PCodeInstruction`, `Comment`, `Label` y `XMLTag`. Está pensado para utilizar únicamente CodeBlock (no llamar a los constructores directamente).
- `CodeBlock`: clase que representa un bloque de código (pudiendo contener instrucciones, etiquetas, comentarios y etiquetas XML). Contiene métodos para:
  - Añadir instrucciones, comentarios y etiquetas.
  - Concatenar bloques de código.
  - Limpiar un bloque de código y comprobar si está vacío.
  - Etiquetar un bloque de código con etiquetas XML (añadiendo al principio y al final del bloque).

CodeBlock proporciona un método para convertir la lista en texto (`toString`), el cual delega en cada `CodeElement` su representación. Para controlar si se generan etiquetas XML (útiles para poder estructurar el código en editores que permitan visualizar XML), existe una variable estática de configuración *generationMode*, que toma los valores del enumerable `BlockMode` (PLAIN o XML).

Conviene familiarizarse muy a fondo con el juego de instrucciones de la máquina P; en `PCodeInstruction` hay un tipo definido por enumeración con los códigos de instrucción que es el que hay que usar para poder añadir instrucciones.

```
// ejemplo de añadido de instrucciones a CodeBlock
import lib.tools.codeGeneration.CodeBlock;
import lib.tools.codeGeneration.PCodeInstruction.OpCode;
....
block.addInst(OpCode.ADD);
block.addInst(OpCode.JMP, label);
....
```

```
// ejemplo de tratamiento del operador NOT
void factor(Attributes at) :
{  ...
    Attributes expressionInfo = new Attributes();
    ...
}
{
    t = < tNOT >
    factor(expressionInfo)
        {
            // comprobaciones semánticas y propagaciones de información
            ...
            // añadimos el código de la expresión
            at.code.addBlock(expressionInfo.code);
            // añadimos la negación de ese resultado
            at.code.addInst(OpCode.NGB);
        }
    ...
```

En general todas las instrucciones o bien tienen una lista de enteros como argumentos o bien una etiqueta. La única excepción es OSF, para la que se ha añadido en CodeBlock un método adhoc extra (`addOSFInst`) que permite crearla con una etiqueta como tercer argumento (en caso de trabajar con las direcciones de las funciones/procedimientos, ese tercer argumento sería un entero y se añadiría como una instrucción normal).

4.2. Ensamblador e intérprete

- El ensamblador toma como entrada el nombre de un fichero con extensión .pcode, y genera como resultado un fichero binario .x que contiene la traducción de las instrucciones resultantes del generador a un lenguaje binario, similar a los bytecodes en Java. Su formato de ejecución es `ensamblador <nombre_fichero_sin_extension>`
- El intérprete toma como entrada el nombre de un fichero con extensión .x y ejecuta las instrucciones que contiene dicho fichero mediante una máquina de pila. Su formato de ejecución es `maquinap <nombre_fichero_sin_extension>`.

A continuación, se muestra un ejemplo de ejecución completa, partiendo de que tenemos un programa escrito en *adac* y llamado `nprimos1.adac`:

```
$ <compilador adac> nprimos1.adac // primero generamos
// el código intermedio
$ ./ensamblador nprimos1 // ahora generamos el binario
$ ./maquinap nprimos1 // finalmente, interpretamos el
// binario mediante una máquina de pila
2 es primo.
3 es primo.
4 es primo.
5 es primo.
7 es primo.
11 es primo.
13 es primo.
17 es primo.
...
43 es primo.
47 es primo.
53 es primo.
59 es primo.
61 es primo.
67 es primo.
71 es primo.
73 es primo.
79 es primo.
83 es primo.
89 es primo.
97 es primo.

Terminación normal.
```

5. Entrega de resultados de la práctica

5.1. Lo que hay que entregar

Como resultado de la práctica se deberá entregar:

- El fichero `practica_4.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 (análogo a como se hizo para la práctica anterior).
- 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 información que la pareja considere oportuna

5.2. Recordatorio de los niveles

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**. Recordad asimismo que se añadió a ***adac*** **el poder definfir 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.

5.3. 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.

5.4. Plazos de entrega

Como se ha comentado anteriormente, esta práctica está pensada para ser desarrollado durante las sesiones 7 y 8. Los dos miembros de la pareja deben entregarla, como muy tarde, el 1 de junio de 2022.

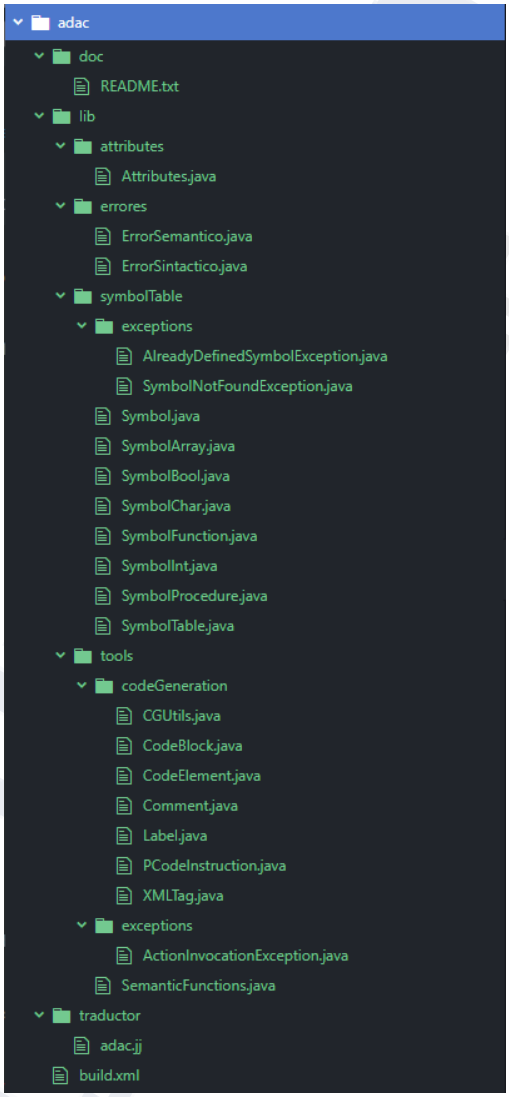

Figura 1: Estructura del proyecto incluyendo el paquete `lib.tools.codeGeneration` proporcionado.