Proyecto FKScript

Implementación mediante ANTLR y C#
de un compilador y una máquina virtual
para un lenguaje de script sencillo

Versión 0.1 – 12/03/2008
(Borrador)

www.sgoliver.net

ÍNDICE DEL DOCUMENTO

1

Introducción al proyecto FKScript

En este primer capítulo haremos una breve introducción al proyecto FKScript. Comentaremos como surge la idea, los requisitos que definiremos para el sistema final y las herramientas con las que contaremos para su desarrollo.

Mis primeros pasos en el mundo de las herramientas de generación de compiladores fueron con *Flex* y *Bison* (versiones GNU de *Lex* y *Yacc*). Estas herramientas, aunque potentes, resultaban algo tediosas de utilizar tanto por sus características intrínsecas como por el código generado, en lenguaje C. A pesar de todo, mis primeros trabajos con estas herramientas me proporcionaron enormes conocimientos sobre este bonito campo del desarrollo de software, aunque eso sí, pasando por muchas dificultades durante el aprendizaje debido a la falta de documentación en español sobre la generación de procesadores de lenguaje y sobre las herramientas concretas utilizadas en el proceso.

Más tarde, mis comienzos con ANTLR (allá por la versión 2.7) no fueron mejores. Me fascinaba un artilugio que unificara en una sola herramienta todos los procesos de generación de un compilador: el análisis léxico y análisis sintáctico mediante autómatas LL, la generación y recorrido de árboles de sintaxis abstracta (AST) para el análisis semántico, y la generación de código tradicional o mediante plantillas (*templates*), todo con una sintaxis común y con una integración total entre todos los módulos sin tener que recurrir a herramientas de terceros o desarrolladas *ad hoc*. Sin embargo, la curva de aprendizaje para ANTLR resultó más pronunciada de lo esperado, ya no por la falta de documentación técnica de referencia (extensa pero en inglés) sino por la falta de documentación práctica, es decir, documentación sobre cómo enfocar el desarrollo de un traductor o compilador con esta nueva herramienta.

Por aquel entonces un compañero de promoción realizó como proyecto fin de carrera un fantástico estudio de la versión 2.7.2 de ANTLR (ANTLR v2), abordando el tema desde un punto de vista completamente práctico, es decir, desarrollando desde cero un compilador para un lenguaje relativamente sencillo aunque bastante completo. Este trabajo resultó ser un recurso inmejorable de información útil sobre ANTLR 2.

Sin embargo, con la llegada de ANTLR v3 todo volvió a cambiar. La documentación de la versión anterior había quedado obsoleta ya que los cambios entre una versión y otra son notables, la documentación pública y gratuita de la versión 3 brillaba por su ausencia (aunque poco a poco se va incrementando en forma de *wiki*, por supuesto en inglés) y el único documento estructurado con información sobre la herramienta era el libro "*The Definitive ANTLR Reference: Building Domain-Specific Languages*" del propio autor de ANTLR, Terence Parr. Y por supuesto que recomiendo adquirir el libro a todo aquel que esté realmente interesado en conocer a fondo ANTLR, pero no deja de ser una fuente de información que no está al alcance de todos de forma gratuita como cabría esperar de una herramienta de código libre como es ANTLR (licencia BSD).

Y aquí es precisamente donde pensé que podría aportar mi granito de arena a todo este asunto. Aprovechando un proyecto personal en el que hacía uso de ANTLR v3 para generar el procesador de un pequeño lenguaje de script decidí escribir un pequeño documento sobre el desarrollo de un compilador para un lenguaje aún más pequeño. Un lenguaje sin complicaciones innecesarias, lo más básico posible para no perdernos en explicaciones. Un documento para comenzar a conocer ANTLR 3 y saber cómo empezar a aplicarlo al desarrollo de un compilador y una máquina virtual sencillos utilizando como lenguaje base C#. En definitiva, algo que asiente las bases necesarias para, a partir de ahí, poder seguir profundizando en el tema todo lo que se desee.

Y así nace este documento que aún no sé cómo llamar: manual de ANTLR 3, tutorial de ANTLR 3, guía práctica de ANTLR 3... quizá documentación práctica de iniciación a ANTLR v3 con C#.

¿A quiénes está dirigido este tutorial? Pues en principio a cualquiera que quiera conocer los entresijos básicos de un sistema como el que se pretende construir, y que desee utilizar como

herramientas de apoyo ANTLR 3 y C#. Sin embargo, aunque al principio del documento se intenta describir una foto del proceso general que siguen estos sistemas, ayudará mucho conocer al menos unos principios básicos de construcción de compiladores y máquinas virtuales. En cuanto a ANTLR v3, en este manual no se describen sus características generales por lo que se recomienda consultar documentación adicional si aún no se ha tenido ningún contacto con la herramienta. Serán de mucha utilidad conocimientos previos sobre otras herramientas de generación de compiladores (como Flex/Bison, Lex/Yacc, JavaCC...), ya que aunque sus principios no coincidan con los de ANTLR, sí que compartirán muchos conceptos comunes.

¿Qué pretendemos construir? Los requerimientos a grandes rasgos del lenguaje de script y la máquina virtual que pretendemos implementar durante esta guía son los siguientes:

- El programa principal se escribirá en un solo fichero y estará formado por una sola función principal, es decir, no será necesario la implementación de llamadas a funciones internas (aunque no se descarta su inclusión en futuras versiones).
- Las variables del lenguaje tendrán un tipo declarado de forma explícita y se deberá comprobar en tiempo de compilación que han sido declaradas y que sus tipos concuerdan dentro de una expresión o una asignación.
- Deberán existir los siguientes tipos de datos: entero, real, lógico y cadena.
- El lenguaje deberá proporcionar las instrucciones clásicas: asignaciones, condicionales y bucles.
- El lenguaje deberá proporcionar las expresiones aritméticas y lógicas básicas.
- La máquina virtual se debe poder integrar fácilmente con cualquier aplicación que exponga una API, y desde el programa script se podrá interactuar con esta aplicación mediante llamadas a las funciones de esta API.

El sistema se escribirá completamente en C# y estará formado por los siguientes módulos:

- Un compilador que transformará el código script a un lenguaje intermedio.
- Un ensamblador que generará el fichero binario ejecutable a partir del código intermedio.
- Una máquina virtual capaz de ejecutar el fichero generado por el ensamblador.

ANTLR será usado para construir el compilador, que estará formado a su vez por los analizadores léxico y sintáctico, un analizador semántico para el cálculo y comprobación de tipos, y un generador de código a partir del *árbol de sintaxis abstracta* (AST) construido durante las fases anteriores. Por su parte, tanto el ensamblador como la máquina virtual se escribirán sin utilizar ninguna herramienta de apoyo, dado que son relativamente sencillos de implementar.

2

Introducción a ANTLR

En este capítulo presentaremos la herramienta ANTLR, que nos servirá de base para el desarrollo de todos los componentes del compilador para nuestro lenguaje FKScript.

Tomando la definición de su propia web, ANTLR es una herramienta que proporciona un marco de trabajo para la construcción de reconocedores, intérpretes, compiladores y traductores de lenguajes a partir de gramáticas enriquecidas con acciones. En resumen proporciona todo lo necesario para el desarrollo de este tipo de sistemas, entre los más importantes:

- Construcción de analizadores léxicos.
- Construcción de analizadores sintácticos.
- Mecanismos de construcción y recorrido de árboles de sintaxis abstracta (AST).
- Mecanismos de tratamiento de plantillas.
- Mecanismos de detección y recuperación de errores.

Como ventajas adicionales que diferencian a ANTLR de otras herramientas similares podemos citar la posibilidad de generar el código de salida en diferentes lenguajes como Java, C, C++, C# o Python, y el hecho de disponer de un entorno de desarrollo propio llamado ANTLRWorks que nos permitirá construir de una forma bastante amigable las gramáticas de entrada a la herramienta, proporcionando representaciones gráficas de las expresiones y árboles generados, e incluyendo un intérprete y depurador propio.

Como recursos para empezar a conocer esta herramienta recomiendo los siguientes:

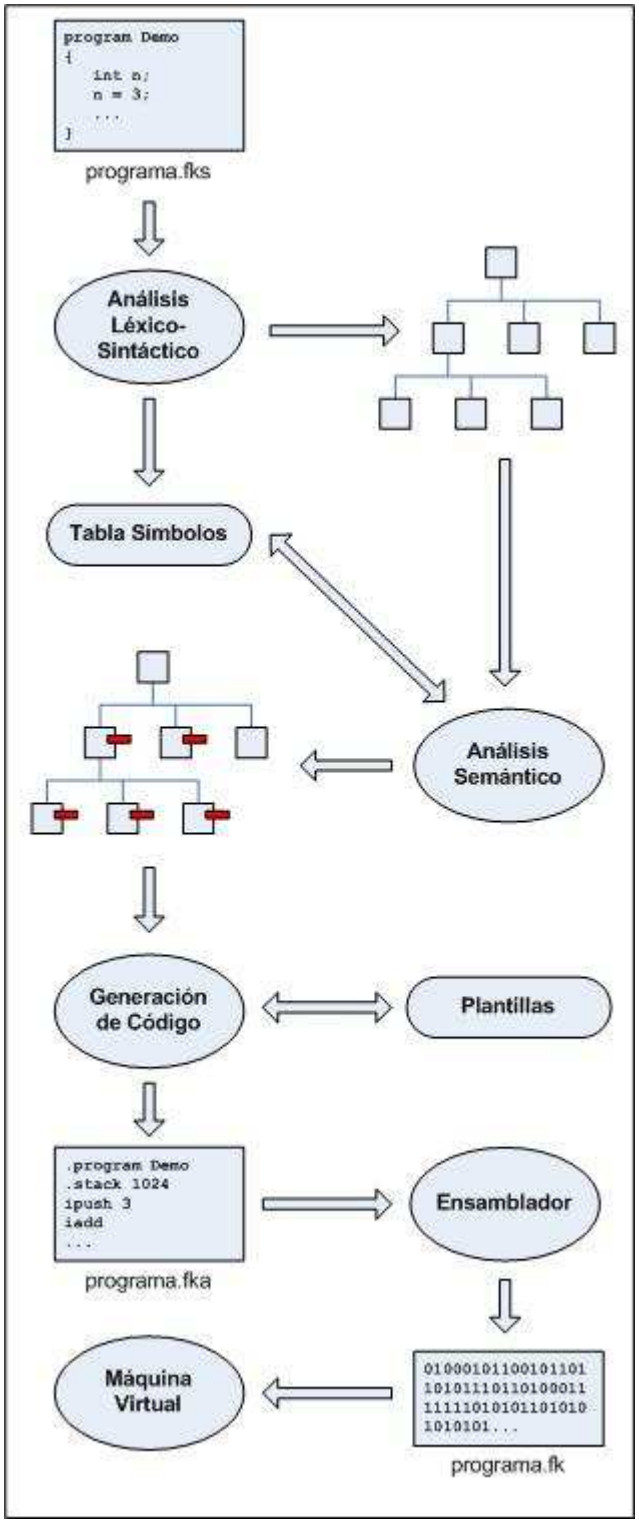
Web principal de ANTLR
Web principal de ANTLRWorks
Wiki de documentación (Docs, Tutoriales, Ejemplos...)
Libro: The Definitive ANTLR Reference - Building Domain-Specific Languages
A medida de avancemos en la construcción de nuestro sistema de scripts trataré de ir comentando muchas de las posibilidades que ofrece ANTLR para el desarrollo de cada módulo, publicando ejemplos y por supuesto los fuentes completos del sistema.

3

Proceso General FKScript

En este capítulo intentaremos ofrecer una visión de conjunto de too el sistema que queremos construir, desde el código de script escrito en el fichero de entrada hasta los resultados del programa ejecutable. Veremos por tanto dónde y cómo ubicar las fases de compilación, ensamblado y ejecución de un programa FKScript.

De cara a tener una visión global del sistema que vamos a construir vamos a describir brevemente el proceso general que se seguirá durante la compilación, ensamblado y ejecución de nuestro script, y cada uno de los módulos que van a intervenir en el proceso.



Como punto de partida se tomará el fichero con el código escrito en lenguaje FKScript, el cual será procesado por el compilador para generar un fichero en lenguaje intermedio FKIL. El proceso de compilación se desarrollará íntegramente utilizando ANTLR 3 y se dividirá en las siguientes fases:

1. Análisis léxico-sintáctico.
2. Análisis semántico.
3. Generación de código.

En primer lugar, los **análisis léxico y sintáctico** parsearán todo el código, detectando y reportando posibles errores de sintaxis, y se generará un **árbol de sintaxis abstracta** (**AST**) con todos los elementos relevantes del código del programa y su estructura completa. Además, durante el análisis se creará la tabla de símbolos, que contendrá todas las variables del programa junto a su tipo y su número de orden en el programa.

Si los análisis léxico y sintáctico se realizan sin errores se procederá al **análisis semántico**, que a partir de la **tabla de símbolos** y el **AST** generados en el paso anterior se encargará de enriquecer dicho AST con el tipo de cada una de las expresiones referenciadas en el programa y verificará que no existen errores de tipo en el código.

Posteriormente, y si el análisis semántico se realiza correctamente, se ejecutará la fase de **generación de código**, que tomará como entrada el **AST enriquecido** y generará a partir de él el **código intermedio** obtenido del programa, utilizando para ello una serie de plantillas construidas a priori.

Una vez generado el fichero con el código intermedio se llamará al **ensamblador**, que transformará dicho código en un fichero binario ejecutable por la **máquina virtual**. Durante el proceso de ensamblado se realizarán las siguientes tareas:

1. Validación del código intermedio, detectando posibles errores.
2. Conversión de constantes de tipo cadena a referencias a memoria dinámica.
3. Conversión de etiquetas a referencias al segmento de código.
4. Generación del fichero binario ejecutable.

Por último, una vez generado el fichero binario final se pasará éste como entrada a la **máquina virtual** para que sea ejecutado. Ésta ejecutará cada una de las instrucciones del programa y devolverá como resultado el último dato contenido en la pila al final del programa. Tanto el ensamblador como la máquina virtual se desarrollarán en C# sin hacer uso de la herramienta ANTLR.

4

Análisis Léxico y Sintáctico de FKScript

En este capítulo comenzaremos a describir el primero de los módulos que formarán parte del compilador de FKScript, el analizador léxico-sintáctico. Comentaremos aspectos generales de esta fase de la compilación y entraremos en detalle con su implementación sobre ANTLR v3.

En esta sección comentaremos la construcción mediante ANTLR 3 de los analizadores léxico y sintáctico para nuestro lenguaje FKScript.

En primer lugar escribiremos los analizadores básicos, es decir, sin acciones, de forma que podamos reconocer fragmentos de código escritos en FKScript, y posteriormente definiremos las acciones a incluir en cada regla para la construcción del árbol de sintaxis abstracta (AST) y la tabla de símbolos que serán utilizados en las fases posteriores de nuestro compilador (análisis semántico y generación de código). Por último también hablaremos un poco sobre cómo se informa de los posibles errores al usuario y cómo los contabilizamos.

Empecemos. En primer lugar cabe destacar que ANTLR v3 permite definir los analizadores léxico (*lexer*) y sintáctico (*parser*) en ficheros independientes o en un solo fichero donde aparezcan ambos. Dado que en nuestro caso ambos analizadores son relativamente sencillos vamos a optar por la segunda opción.

Lo primero que debemos especificar en el fichero ANTLR será el tipo de gramática a definir (`lexer grammar`, `parser grammar`, o `grammar` para combinar análisis léxico y sintáctico en un mismo fichero) y el nombre que recibirá la gramática, en nuestro caso "FKVM".

En segundo lugar indicaremos las opciones del analizador, donde sólo incluiremos la opción `language`, que indica el lenguaje en el que se generará el código de los analizadores, utilizaremos C#, y el tipo de salida producida por el analizador sintáctico, en nuestro caso un árbol AST (opción `output`) de tipo `CommonTree` (opción `ASTLabelType`).

```
grammar FKVM;

options {
    output=AST;
    ASTLabelType=CommonTree;
    language=CSharp;
}
```

1. Analizador Léxico

El analizador léxico se encargará de reconocer y separar convenientemente los elementos básicos (*tokens*) del lenguaje que estamos construyendo. En la mayoría de los casos deberemos distinguir: literales de los distintos tipos de datos que existan en el lenguaje, identificadores (ya sean palabras clave o no) y deberemos definir qué se entenderá como espacio en blanco entre elementos.

Literales

En FKScript existen 4 tipos de datos: enteros, reales, lógicos y cadenas. Deberemos definir por tanto cómo reconocer cada uno de estos literales en nuestro código. Para ayudar a que las definiciones de estos elelementos no sean demasiado complejas y repetitivas definiremos dos reglas auxiliares (para las cuales se utiliza la palabra clave `fragment`) para reconocer dígitos y letras. Una vez definidas estas reglas auxiliares, el resto son muy sencillas. Por ejemplo, un literal entero estará formado por cualquier combinación de 1 o más dígitos (lo que se indica con el operador '+' de ANTLR). Los literales de tipo cadena estarán formados por una doble comilla seguida de cualquier combinación de caracteres distintos a dobles comillas, saltos de línea o tabuladores y por último otra doble comilla de cierre. Por su parte, un literal lógico tan sólo podrá tomar dos valores: `true` o `false`.

```
fragment LETRA : 'a'..'z'|'A'..'Z' ;
fragment DIGITO : '0'..'9' ;

LIT_ENTERO : DIGITO+ ;

LIT_REAL : LIT_ENTERO '.' LIT_ENTERO ;

LIT_CADENA : '"' (~('"'|'\n'|'\r'|'\t'))* '"' ;

LIT_LOGICO : 'true'|'false' ;
```

Identificadores

Los identificadores en FKScript, como ya se indicó en la especificación del lenguaje, estarán formados por cualquier letra o carácter subrayado seguida de cualquier combinación de dígitos, letras o caracteres de subrayado.

```
IDENT : (LETRA|'_')(LETRA|DIGITO|'_')* ;
```

Comentarios

Los comentarios permitidos serán de una sola línea y se utilizará la sintaxis de C++ o Java, es decir, precederlos de los caracteres "//". Como puede observarse en la regla se le indicará además a ANTLR que estos *tokens* no deben ser pasados al analizador sintáctico ya que no son de utilidad. Esto lo indicaremos mediante la acción `$channel=HIDDEN;`

```
COMENTARIO : '//' (~('\n'|'\r'))* '\r'? '\n' {$channel=HIDDEN;};
```

Espacio en blanco

Se considerará espacio en blanco entre elementos del lenguaje a toda combinación de caracteres espacio, saltos de línea o tabuladores. Además, estos elementos tampoco serán pasados como tokens al analizador sintáctico.

```
WS : (' '|'\r'|'\n'|'\t')+ {$channel=HIDDEN;} ;
```

2. Analizador Sintáctico

A partir de los tokens pasados por el analizador léxico, el analizador sintáctico se encargará de reconocer las combinaciones correctas de tokens que forman instrucciones o expresiones válidas en nuestro lenguaje. Por tanto deberemos definir cuál es la estructura general de un programa FKScript y la estructura concreta de cada una de las construcciones (instrucciones y expresiones) aceptadas en el lenguaje.

Programa

Tal como se indicó en la especificación del lenguaje, un programa FKScript estará formado por una serie de declaraciones de funciones API (opcionales) seguidas del programa principal que se identifica mediante la palabra clave `program` seguida de un identificador que indicará el nombre del programa y una lista de instrucciones delimitadas por llaves ('{' y '}'). Las instrucciones a su vez podrán ser: declaraciones, asignaciones, condicionales *if*, bucles *while* o instrucciones de retorno. Abajo podemos ver lo sencillo que resulta definir todo esto en ANTLR v3, y lo que es mejor, utilizando la misma sintaxis usada ya para el analizador léxico.

```
programa : declaraciones_api principal ;

declaraciones_api : declaracion_api* ;

declaracion_api : 'api' tipo IDENT '(' lista_decl ')' ';' ;

principal : 'program' tipo IDENT '{' lista_instrucciones '}' ;

lista_instrucciones : instruccion* ;

instruccion : inst_decl
            | inst_asig
            | inst_if
            | inst_while
            | inst_return
            | inst_expr;
```

Instrucciones

La definición de las instrucciones no implica ninguna dificultad. Así, por ejemplo, la instrucción IF de nuestro lenguaje estará formada por la palabra clave `if`, una expresión (regla que definiremos más tarde) entre paréntesis, una lista de instrucciones para el caso de que se cumpla la condición, y opcionalmente (operador `?` de ANTLR) otra lista de instrucciones para el caso de que no se cumpla dicha condición precedida de la palabra clave `else`. Nótese como las palabras clave y símbolos de puntuación los indicamos directamente entre comillas simples, los tokens devueltos por el analizador léxico con mayúsculas y las reglas del analizador sintáctico en minúsculas.

```
inst_decl : tipo IDENT ';' ;

inst_asig : IDENT '=' expresion ';' ;

inst_if : 'if' '(' expresion ')' '{' lista_instrucciones '}' else_otras? ;

else_otras : 'else' '{' lista_instrucciones '}' ;

inst_while : 'while' '(' expresion ')' '{' lista_instrucciones '}' ;

inst_return : 'return' expresion ';' ;

inst_expr : expresion ';' ;
```

Expresiones

La definición de las expresiones en ANTLR sí es más interesante. Dado que ANTLR 3 no porvee ningún mecanismo explícito para indicar la precedencia de cada uno de los operadores debemos buscar la forma de que ésta se tenga en cuenta por la propia definición de las reglas. Para ello, el método utilizado será definir cada una de las posibles expresiones (según operadores) de forma que un tipo de expresión quede definido en función del siguiente tipo en el orden de precedencia (siempre de menor a mayor). Así, por ejemplo, como las operaciones de multiplicación y división tienen mayor precedencia que la suma y la resta definiremos ésta última en función de la primera, así nos aseguramos de que las operaciones se asocien siempre de forma correcta.

```
expMasMenos : expMultDiv (
        ('+'|'-') expMultDiv)* ;
```

Para el resto de expresiones se seguiría la misma técnica:

```
expresion : expOr ;

expOr : expAnd ('||'^ expAnd)* ;

expAnd : expComp ('&&'^ expComp)* ;

expComp : expMasMenos (
        ('=='|'!='|'>'|'<'|'>='|'<=') expMasMenos)* ;

expMasMenos : expMultDiv (
        ('+'|'-') expMultDiv)* ;

expMultDiv : expMenos (
        ('*'|'/') expMenos)* ;

expMenos : '-' expNo
         | '+'? expNo ;

expNo : '!'? acceso ;

acceso : IDENT
       | literal
       | llamada
       | '(' expresion ')' ;
```

Llamadas a funciones

Las llamadas a función seguirán la sintáxis clásica compuesta por el nombre de la función seguido de una lista de parámetros (expresiones) entre paréntesis y separados por comas.

```
llamada : IDENT '(' lista_expr ')' ;

lista_expr : expresion (',' expresion)*
           | //Sin parámetros ;
```

Otras reglas

Por último, tan sólo quedan definir el resto de reglas auxiliares, como los tipos de datos o los tipos de literales. Estas reglas suelen ser muy sencillas ya que se limitan a enumerar cada uno de los elementos aceptados en el lenguaje.

```
tipo : 'int'|'float'|'string'|'bool'|'void' ;

literal : LIT_ENTERO
        | LIT_REAL
        | LIT_CADENA
        | LIT_LOGICO ;
```

Programa de prueba

Hemos finalizado nuestros analizadores básicos. Con esto ya deberíamos ser capaces de comprobar si la sintaxis de un programa escrito en FKScript es o no válida. Para ello, necesitamos disponer de un programa de prueba, en el que se llame convenientemente a las clases generadas por ANTLR a partir de nuestra gramática.

```
using System;
using Antlr.Runtime;
using Antlr.Runtime.Tree;
using Antlr.StringTemplate;

namespace PruAntlr
{
    class Program
    {
        static void Main(string[] args)
        {
            ANTLRFileStream input = new ANTLRFileStream("prueba.fks");
            FKVMLexer lexer = new FKVMLexer(input);

            CommonTokenStream tokens = new CommonTokenStream(lexer);
            FKVMParser parser = new FKVMParser(tokens);
            FKVMParser.programa_return result = parser.programa();

            Console.WriteLine("Analisis finalizado.");
        }
    }
}
```

De esta forma, utilizando el programa principal anterior y un fichero de entrada válido como el que se muestra a continuación deberíamos obtener como única salida del programa el mensaje de "Analisis finalizado", ya que si los analizadores no encuentran ningún error no muestran nada a la salida.

```
program Prueba
{
    int a;
    a = 1;
}
```

Sin embargo, si introducimos un error en el fichero de entrada, por ejemplo en este caso hemos eliminado la variable 'a' de la primera declaración, el analizador sintáctico debería mostrarnos el mensaje de error correspondiente.

```
program Prueba
{
    int ;
    a = 1;
}
```

El resultado del análisis debería ser el siguiente:

```
line 2:5 mismatched input ';' expecting IDENT
Analisis finalizado.
```

En el mesaje se informa al usuario de que en la posición 5 de la línea 2 del fichero de entrada se ha encontrado un carácter ';' cuando se esperaba un identificador.

3. Recuento y reporte de errores

Vamos a ir ahora un poco más allá y además de mostrar los errores al usuario vamos a mostrar un último mensaje con el número de errores encontrados por los analizadores léxico y sintáctico. Para ello, la técnica que vamos a utilizar será sobescribir los métodos de ANTLR encargados de generar los mensajes de error (`GetErrorMessage`). La modificación que vamos a realizar a estos métodos

será simplemente incrementar una variable propia cada vez que sean llamados y llamar por último al metodo padre para que ANTLR realice el resto de tareas necesarias. El código de estos métodos y la declaración de nuestras variables lo incluiremos en las secciones `@lexer::members` y `@members`

```
@lexer::members {
    public int numErrors = 0;

    public override string GetErrorMessage(RecognitionException e, string[]
tokenNames)
    {
        numErrors++;
        return base.GetErrorMessage(e,tokenNames);
    }
}

@members {
    public int numErrors = 0;

    public override string GetErrorMessage(RecognitionException e, string[]
tokenNames)
    {
        numErrors++;
        return base.GetErrorMessage(e,tokenNames);
    }
}
```

A partir de este momento, desde nuestro programa principal tendremos acceso a nuestra nueva variable que tras llamar a los analizadores contendrán el número de errores encontrados. Así, podríamos modificar la última línea de nuestro programa principal por la siguiente:

```
Console.WriteLine("Análisis finalizado. Errores: " + parser.numErrors);
```

Y ante este fichero de entrada:

```
program Prueba
{
    int ;
     = 1;
}
```

La salida sería la siguiente:

```
line 2:5 mismatched input ';' expecting IDENT
line 3:2 mismatched input '=' expecting '}'
Análisis finalizado. Errores: 2
```

4. Construcción del AST

Una vez que ya somos capaces de reconocer correctamente ficheros de entrada válidos para nuestro lenguaje y de informar de los posibles errores en caso de producirse el siguiente paso será modificar convenientemente la gramática para construir durante el análisis el árbol de sintaxis abstracta (AST) y la tabla de símbolos que serán utilizados por el analizador semántico y el generador de código.

ANTLR v3 proporciona dos mecanismos básicos de construcción de árboles AST:

1. Mediante operadores.
2. Mediante reglas de reescritura.

Ambos mecanismos pueden mezclarse en una misma gramática y el uso de uno u otro dependerá de la complejidad del árbol que queramos construir. En nuestro caso mezclaremos ambos métodos por lo que veamos en primer lugar un par de ejemplos.

La construcción mediante operadores consiste en incluir en las propias reglas del analizador una serie de operadores que indiquen a ANTLR cómo debe tratarlos para construir el árbol devuelto por la regla. Estos operadores son tan sólo dos: `^` y `!`. El primero de ellos se utilizará para indicar qué elemento de la regla se utilizará como raíz del árbol (todos los demás pasarán a ser hijos directos de éste) y el segundo operador se añadirá a los elementos que no deben formar parte del árbol.

Así, por ejemplo, en la regla `inst_while` podremos indicar que la palabra clave `while` se utilizará como raíz del árbol y que los paréntesis y llaves no deben incluirse en el árbol ya que no aportan ningún valor para las fases siguientes.

```
inst_while : 'while'^ '('! expresion ')'! '{'! lista_instrucciones '}'! ;
```

En reglas más complejas o donde haya que utilizar *nodos ficticios* (elementos que no aparecen en la regla pero se incluyen en el árbol AST por conveniencia) se pueden utilizar las reglas de reescritura de árboles. Éstas se incluyen a la derecha de la regla precedida por el operador `->` y utilizan la sintaxis siguiente:

```
regla -> ^(raiz hijo1 hijo2 ...)
```

En nuestro caso podemos poner como ejemplo la regla para definir las asignaciones `inst_asig`, donde indicaremos mediante una regla de reescritura que queremos construir un árbol con un nodo raíz ficticio llamado `ASIGNACION` y dos hijos, uno con el identificador y otro con la expresión asignada.

```
inst_asig : IDENT '=' expresion ';' -> ^(ASIGNACION IDENT expresion);
```

Los nodos ficticios deben declararse al principio de la gramática en la sección `tokens`. En nuestro caso utilizaremos los siguientes:

```
tokens {
    PROGRAMA;
    DECLARACION;
    DECLARACIONPARAM;
    LISTADECLARACIONESAPI;
    DECLARACIONAPI;
    ASIGNACION;
    MENOSUNARIO;
    LISTAINSTRUCCIONES;
    LLAMADA;
    LISTAEXPRESIONES;
    LISTAPARAMETROS;
}
```

5. Construcción de la Tabla de Símbolos

Otra de las tareas a realizar durante el análisis sintáctico será la construcción de la tabla de símbolos. Esta estructura deberá contener al finalizar el análisis una relación fácilmente accesible que contenga todos los identificadores utilizados en el programa junto cualquier información asociada a dicho identificador que sea relevante para las etapas posteriores, como por ejemplo el tipo de dato de la variable.

En nuestro caso, construiremos una tabla de símbolos que contenga cada identificador junto a su tipo y su número de orden dentro del programa. Para ello definiremos en primer lugar una clase para encapsular toda la información asociada a cada identificador:

```
public class Symbol
{
    public int numvar;  //Número de orden la variable
    public string type; //Tipo de la variable

    //Constructor de la clase
    public Symbol(string t, int n)
    {
        type = t;
        numvar = n;
    }
}
```

El siguiente paso será declarar la estructura que contendrá la tabla de símbolos. Nosotros utilizaremos una colección genérica de tipo `Dictionary` con claves de tipo `string` para el nombre de a variable y valores de tipo `Symbol` que acabamos de crear. La declaración de esta estructura debe incluirse en la sección `@members` y los *includes* necesarios en la sección `@header` de la gramática.

```
@header {
using System.Collections.Generic;
}

@members {
    public Dictionary<string,Symbol> symtable = new Dictionary<string,Symbol>();
    int numVars = 0;

    ...
}
```

En nuestro lenguaje FKScript, el único lugar válido donde pueden definirse variables es dentro de las declaraciones, por lo que la única regla donde debemos ir actualizando la tabla de símbolos será `inst_decl`. Por tanto, dentro de esta regla, además de la regla de reescritura para construir el árbol AST deberemos incluir una acción (entre llaves) donde se añada el identificador declarado a la tabla de símbolos. Esta acción se limitará a comprobar si el identificador ya existe en la tabla y añadirlo en caso de no existir. Veamos cómo:

```
inst_decl : tipo IDENT ';' {
     if(!symtable.ContainsKey($IDENT.text))
     {
           symtable.Add($IDENT.text, new Symbol($tipo.text, numVars++));
     }
} -> ^(DECLARACION tipo IDENT);
```

6. Finalizando el analizador léxico-sintáctico

Una vez completada la implementación de nuestra gramática tan sólo nos queda probarla mediante el programa principal que ya comenzamos en apartados anteriores. En esta ocasión vamos a modificarlo para que al finalizar el análisis nos muestre el árbol AST construido y la tabla de símbolos con nuestras variables y su información asociada.

```
using System;
using Antlr.Runtime;
using Antlr.Runtime.Tree;
using Antlr.StringTemplate;
using System.Collections.Generic;

namespace PruAntlr
{
    class Program
    {
        static void Main(string[] args)
        {
            ANTLRFileStream input =
                      new ANTLRFileStream("C:\\pruantlr\\prueba.fks");
            FKVMLexer lexer = new FKVMLexer(input);

            CommonTokenStream tokens = new CommonTokenStream(lexer);
            FKVMParser parser = new FKVMParser(tokens);
            FKVMParser.programa_return result = parser.programa();

            if (parser.numErrors == 0)
            {
                CommonTree t = (CommonTree)result.Tree;

                Console.WriteLine("Arbol AST:");
                Console.WriteLine(t.ToStringTree() + "\n");

                Console.WriteLine("Tabla de Simbolos:");
                foreach (string k in parser.symtable.Keys)
                {
                    Console.WriteLine(((Symbol)parser.symtable[k]).numvar +
                      " - " + k + " -> " + ((Symbol)parser.symtable[k]).type);
                }
            }
            else
            {
                Console.WriteLine("Errores: " + parser.numErrors);
            }

            Console.ReadLine();
        }
    }
}
```

El árbol AST lo obtenemos mediante la propiedad `Tree` del objeto que representa a la regla principal de nuestra gramática `programa_return` y lo imprimimos por pantalla mediante el método `toStringTree()`. Por su parte, a la tabla de símbolos podemos acceder como a un atributo más del objeto `parser` ya que la declaramos en la sección `@members`.

De esta forma, para el siguiente fichero de entrada:

```
program Prueba {
    int a;
    float b;

    a = 1;
}
```

Obtendríamos la siguiente salida (la formateo para que sea legible):

```
Arbol AST:
(program Prueba
    (LISTAINSTRUCCIONES
         (DECLARACION int a)
         (DECLARACION float b)
         (ASIGNACION a 1)
     )
)

Tabla de Simbolos:
0 - a -> int
1 - b -> float
```

5

Análisis Semántico de FKScript

En este capítulo describiremos la segunda fase dentro del proceso de compilación de FKScript, el análisis semántico. Comenzaremos describiendo qué tipo de tareas se realizan durante esta fase y detallaremos su implementación sobre ANTLR v3.

Dedicaremos esta sección a comentar todos los aspectos prácticos del desarrollo de la etapa de análisis semántico del compilador para nuestro lenguaje FKScript. Veremos en primer lugar las modificaciones necesarias que hay que realizar al árbol AST construido en la fase anterior, posteriormente describiremos el tipo de comprobaciones que se realizarán en esta fase del análisis y por último veremos cómo podemos implementarlas con ANTLR.

Todas las tareas del analizador semántico se realizarán mediante el recorrido de árboles AST, lo que en ANTLR se llama `tree grammar`. Veremos más adelante cómo implementar un analizador de este tipo con ANTLR v3.

1. Tareas del análisis semántico

Durante la etapa de análisis semántico nuestro compilador realizará una serie de comprobaciones que ya nada tienen que ver con la sintaxis del lenguaje, y que en nuestro caso serán las siguientes:

- Comprobación de la existencia de variables y funciones.
- Cálculo de tipos en expresiones.
- Chequeo de tipos en instrucciones.
- Chequeo de tipos en expresiones.

Iremos enumerando todas esas comprobaciones a medida que vayamos comentando la implementación sobre ANTLR por lo que no entraremos por el momento en más detalle.

2. Enriqueciendo los nodos del árbol AST

Por defecto, el tipo de árbol construido por ANTLR es del tipo `CommonTree` cuyos nodos únicamente contienen un tipo (atributo `Type`) y un valor (atributo `Text`). Esta información no suele ser suficiente en la mayoría de las ocasiones por lo que se hace necesario definir un tipo de árbol personalizado con toda la información que necesite nuestro compilador.

En nuestro caso vamos a necesitar para los elementos simples (identificadores y literales) toda la información contenida en la clase `Symbol` que ya definimos y dos campos adicionales para almacenar el tipo de las expresiones compuestas `expType` y el tipo de sus subexpresiones `expSecType`. Para conseguir esto simplemente tendremos que definir una clase derivada de `CommonTree` que contenga toda esta información y un constructor adecuado que inicialice todo lo necesario y llame al constructor de la clase padre. Veamos cómo quedaría esta clase a la que llamaremos `FkvmAST`:

```
using Antlr.Runtime;
using Antlr.Runtime.Tree;

public class FkvmAST : Antlr.Runtime.Tree.CommonTree
{
    public Symbol symbol;
    public string expType = "";
    public string expSecType = "";

    public FkvmAST(IToken t) : base(t)
    {
        //Nada que inicializar
    }
}
```

Una vez tenemos definida la clase que describirá nuestros nodos del árbol AST debemos indicar a ANTLR que use ésta para construir el árbol durante la fase de análisis sintáctico. Para ello deberemos modificar el valor asignado a la opción `ASTLabelType` indicando nuestra nueva clase.

```
options {
        output=AST;
        ASTLabelType=FkvmAST;
        language=CSharp;
}
```

Pero no nos basta con esto. Debemos crear también un adaptador para nuestro nuevo tipo de árbol. Este adaptador debe derivar de la clase `CommonTreeAdaptor` y tan sólo deberemos redefinir el método `Create` para devolver un árbol de tipo `FkvmAST`. Veamos cómo quedaría nuestro adaptador:

```
class FKTreeAdaptor : CommonTreeAdaptor
{
    public override object Create(IToken payload)
    {
        return new FkvmAST(payload);
    }
}
```

Una vez creado el nuevo adaptador debemos indicar a ANTLR en el programa principal que debe hacer uso de éste para la construcción del árbol de sintaxis abstracta durante la fase de análisis sintáctico. Para ello asignaremos la propiedad `TreeAdaptor` de nuestro parser antes de comenzar el análisis:

```
CommonTokenStream tokens = new CommonTokenStream(lexer);
FKVMParser parser = new FKVMParser(tokens);

ITreeAdaptor adaptor = new FKTreeAdaptor();
parser.TreeAdaptor = adaptor;

FKVMParser.programa_return result = parser.programa();
```

3. Implementación del analizador en ANTLR v3

Como indicamos anteriormente, para la fase de análisis semántico vamos a utilizar un analizador de árboles AST, lo que en ANTLR se define como `tree grammar`. Las opciones de este analizador serán las mismas que las establecidas en la fase anterior, con la excepción de la opción `tokenVocab` que indicará a ANTLR que debe utilizar los mismos tokens que se generaron para la fase de análisis sintáctico. De esta forma, ANTLR importará estos tokens desde el fichero `FKVM.tokens` generado en la fase anterior.

```
tree grammar FKVMSem;

options {
    tokenVocab=FKVM;
    ASTLabelType=FkvmAST;
    language=CSharp;
}
```

Por otro lado, el analizador semántico recibirá como entrada, además del árbol AST, la tabla de símbolos generada anteriormente por lo que deberá declararse la variable que la contendrá. Como siempre haremos esto en las secciones `@header` y `@members`. Incluiremos también la variable `numErrors` para ir realizando el recuento de errores de forma análoga a la fase anterior.

```
@header {
using System.Collections.Generic;
}

@members {
public Dictionary<string,Symbol> symtable;
public int numErrors = 0;
}
```

El paso de parámetros a una regla en ANTLR se indica a continuación de la regla y entre corchetes, y la asignación de dicho parámetro a nuestra variable interna la podemos realizar en la sección `@init` de nuestra regla principal. Vemos cómo quedaría por la primera regla de nuestra gramática:

```
programa[Dictionary<string,Symbol> symtable]
@init {
    this.symtable = symtable;
}
: ^(PROGRAMA declaraciones_api principal) ;
```

En el último apartado de esta sección veremos cómo podemos pasar en el programa principal la tabla de símbolos construida durante el analisis sintáctico como parámetro del analizador semántico.

4. Cálculo y chequeo de tipos

El cálculo y chequeo de tipos de una expresión se realizará comenzando por las expresiones más simples de nuestra gramática, es decir, los literales e identificadores. Una vez calculado el tipo de una expresión se asiganará éste a su nodo del árbol y además se devolverá como valor de retorno para que pueda ser consultado por la reglas superiores a la actual. El valor de retorno se indica en ANTLR mediante la cláusula `returns` seguida de la declaración entre corchetes de la variable a devolver.

```
expresion returns [String type]
```

Esta variable se inicializará en la sección `@init` de la regla y se asignará convenientemente dentro de cada subregla. Además, dentro de esta sección también obtendremos una referencia al nodo principal del arbol de la expresión para poder asignarlo al fializar la regla. Este nodo lo obtendremos mediante el método `LT()` del objeto `input`, que representa la secuencia de nodos del árbol que estamos analizando. De esta forma, el siguiente nodo de la secuencia, lo obtendremos mediante la llamada `input.LT(1)`. Por último, en la sección `@after` de la regla asignaremos a este nodo el tipo calculado y que será devuelto como retorno. Veamos pues como queda la regla por el momento:

```
expresion returns [String type]
@init {
    $type="";
    FkvmAST root = (FkvmAST)input.LT(1);
}
@after {
    root.expType = $type;
}
```

Todo el cálculo de tipos se realizará en la regla que analiza las expresiones, donde están contenidos tanto los elementos más simples como los literales o identificadores, como las expresiones más complejas, donde habrá que calcular su tipo en función de los elementos que la componen. Empecemos por tanto por los elelemtos más sencillos.

Literales

El cálculo del tipo de un literal será tan sencillo como consultar el tipo de token devuelto por el analizador sintáctico y asignar directamente un tipo u otro en la subregla correspondiente. Como puede verse en el código siguiente, la técnica seguida en la regla `literal` para devolver el tipo calculado a la regla superior y asignarlo a su vez al nodo del árbol es la misma que la descrita para las expresiones.

```
expresion returns [String type]
...
    | literal {$type=$literal.type;}
    ;

literal returns [String type]
@init {
    $type="";
    FkvmAST root = (FkvmAST)input.LT(1);
}
@after {
    root.expType = $type;
}
    : LIT_ENTERO {$type="int";}
    | LIT_REAL {$type="float";}
    | LIT_CADENA {$type="string";}
    | LIT_LOGICO {$type="bool";}
    ;
```

Identificadores

El cálculo de tipo para un identificador es más sencillo aún que en el caso de los literales ya que nos basaremos directamente en el tipo almacenado en la tabla de símbolos para dicho identificador. Además, en caso de no encontrar en la tabla de símbolos algún identificador informaremos del error al usuario, de forma que no permitiremos en nuestro lenguaje el uso de variables que no hayan sido declaradas.

```
expresion returns [String type]
...
    | IDENT {
          if(symtable.ContainsKey($IDENT.text))
          {
               root.symbol = (Symbol)symtable[$IDENT.text];
$type=root.symbol.type;
          }
          else
          {
               registrarError(root.Line, "Identifier '" + $IDENT.text + "' has
not been declared.");
          }
    }
    | literal {$type=$literal.type;}
    ;
```

Llamadas a función externa

El tipo de una llamada a una función externa se calcula de la misma forma que el de los identificadores ya que el mecanismo que hemos utilizado para registrar su tipo durante el análisis sintáctico ha sido el mismo, la tabla de símbolos. Veamos por tanto cómo queda esta regla:

```
expresion returns [String type]
...
    | IDENT {root.symbol = (Symbol)symtable[$IDENT.text];
$type=root.symbol.type;}
    | literal {$type=$literal.type;}
    | llamada {$type=$literal.type;}
    ;

llamada returns [String type]
@init {
$type="";
FkvmAST root = (FkvmAST)input.LT(1);
}
@after {
root.expType = $type;
}
: ^(LLAMADA IDENT lista_expr) {
        if(symtable.ContainsKey($IDENT.text))
        {
             root.symbol = (Symbol)symtable[$IDENT.text];
$type=root.symbol.type;
        }
        else
        {
             registrarError(root.Line, "Api function '" + $IDENT.text + "' has
not been declared.");
        }
} ;
```

Expresiones complejas

Una vez hemos calculado el tipo de las expresiones más simples de nuestro lenguaje ya deberíamos ser capaces de calcular y chequear el de una expresión compuesta por estos elementos. Así, por ejemplo, para las expresiones lógicas consideraremos que su tipo es siempre `bool` y que ésta es correcta si los tipos de las dos subexpresiones son iguales o uno entero y otro real. Esto lo comprobaremos mediante el método `comprobarTipoExpComp(tipo1, tipo2)` que definiremos en la sección `@members` y si se determina que la combinación de tipos no es correcta se reportará al usuario el error correspondiente.

```
expresion returns [String type]
...
    | ^(opComparacion e1=expresion e2=expresion) {
            $type="bool";

            if(!comprobarTipoExpComp($e1.type, $e2.type))
            {
                 registrarError(root.Line, "Incorrect types in expression.");
            }
      }
    | IDENT {root.symbol = (Symbol)symtable[$IDENT.text];
$type=root.symbol.type;}
    | literal {$type=$literal.type;}
    | llamada {$type=$literal.type;}
    ;
```

```
@members {
public Dictionary symtable;
public int numErrors = 0;

public bool comprobarTipoExpComp(string t1, string t2)
{
```

```
    bool res = false;

    if(t1.Equals(t2) ||
      (t1.Equals("int") && t2.Equals("float")) ||
      (t1.Equals("float") && t2.Equals("int")) )
    {
        res = true;
    }

    return res;
}
}
```

El resto de expresiones incluidas en el analizador (expresiones aritméticas, operador menos-unario y operador no-lógico) las definiremos de forma análoga:

```
expresion returns [String type]
   : ^(opComparacion e1=expresion e2=expresion) {
           $type="bool";

           if(!comprobarTipoExpComp($e1.type, $e2.type))
           {
                registrarError(root.Line, "Incorrect types in expression.");
           }
      }
    | ^(opAritmetico e1=expresion e2=expresion) {
           $type=$e1.type;

           if(!comprobarTipoExpArit($e1.type, $e2.type))
           {
                registrarError(root.Line, "Incorrect types in expression.");
           }
      }
    | ^(MENOSUNARIO e1=expresion) {
           $type=$e1.type;

           if(!($e1.type.Equals("int") || $e1.type.Equals("float")))
           {
                registrarError(root.Line, "Incorrect types in expression.");
           }
      }
      | ^('!' e1=expresion) {
           $type=$e1.type;

           if(!$e1.type.Equals("bool"))
           {
                registrarError(root.Line, "Incorrect types in expression.");
           }
      }
    | IDENT {root.symbol = (Symbol)symtable[$IDENT.text];
$type=root.symbol.type;}
    | literal {$type=$literal.type;}
    | llamada {$type=$literal.type;}
    ;
```

Instrucciones

El tipo de algunos de los elementos contenidos en instrucciones de nuestro lenguaje también debe ser comprobado durante la fase de análisis semántico. Así, por ejemplo, en las instrucciones de asignación deberemos comprobar que el tipo de la expresión derecha coincide con el de la variable que estamos asignando o que al menos es compatible. El tipo de la variable lo recuperaremos de la

tabla de símbolos y el de la expresión accediendo a su atributo `$type` que ya debería haber sido calculado en la regla `expresion`. Las combinaciones válidas de tipos las comprobaremos como en el caso de las expresiones mediante un método definido en la sección `@members`. Veamos cómo quedaría esta regla:

```
inst_asig
@init {
FkvmAST root = (FkvmAST)input.LT(1);
}
   : ^(ASIGNACION IDENT e1=expresion) {
          root.expType = $e1.type;

          if(symtable.ContainsKey($IDENT.text))
          {
               $IDENT.symbol = (Symbol)symtable[$IDENT.text];

               if(!comprobarTipoAsig(root.expType, $IDENT.symbol.type))
               {
                    registrarError(root.Line, "Incorrect type in assignment.");
               }
          }
          else
          {
               registrarError(root.Line, "Identifier '" + $IDENT.text +
                                 "' has not been declared.");
          }
};
```

Por su parte, para las instrucciones IF y WHILE deberemos comprobar que la condición viene expresada por una expresión de típo lógico. Para ello tan sólo deberemos comprobar el atributo `$type` de la expresión que hemos calculado en la regla `expresion`. Veamos por ejemplo la instrucción IF:

```
inst_if : ^(ins='if' e1=expresion lista_instrucciones lista_instrucciones) {
      if(!$e1.type.Equals("bool"))
      {
            registrarError($ins.Line, "Incorrect type in IF instruction.");
      }
};
```

5. Programa principal

Una vez finalizado nuestro analizador semántico podemos llamarlo desde el programa principal pasándole la tabla de símbolos calculada durante la fase anterior. Para ello, simplemente se creará el objeto FKVMSem con las estructuras necesarias procedentes del analisis sintáctico y se llamará a su método principal cuyo nombre debe coincidir con la regla principal de nuestra gramática, en este caso `programa()`. La tabla de símbolos se le pasará como parámetro de esta llamada ya que así lo definimos en la gramática.

```
//Análisis léxico semántico
FKVMLexer lexer = new FKVMLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
FKVMParser parser = new FKVMParser(tokens);
parser.TreeAdaptor = adaptor;
FKVMParser.programa_return result = parser.programa();

//Si no hay errores léxicos ni sintácticos ==> Análisis Semántico
if (lexer.numErrors + parser.numErrors == 0)
{
```

```
    //Analisis Semántico
    CommonTree t = ((CommonTree)result.Tree);
    CommonTreeNodeStream nodes2 = new CommonTreeNodeStream(t);
    nodes2.TokenStream = tokens;
    FKVMSem walker2 = new FKVMSem(nodes2);
    walker2.programa(parser.symtable);
}
```

6

Generación de Código de FKScript

En este último capítulo dedicado a la compilación de FKScript describiremos la fase de generación de código para nuestro lenguaje. Comenzaremos haciendo una bve reseña a la máquina virtual y posteriormente detallaremos la implementación sobre ANTLR v3.

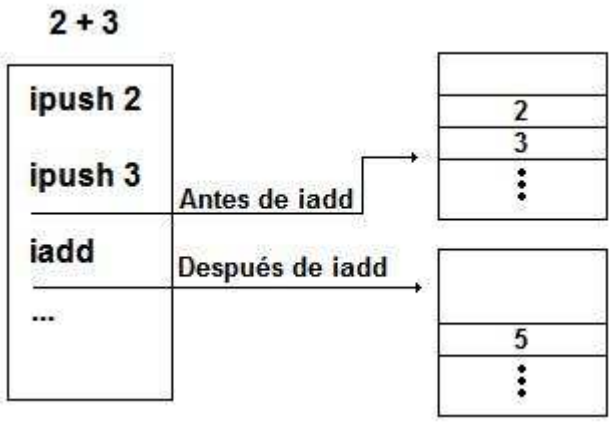
En esta sección abordaremos la última fase del proceso de compilación de nuestro lenguaje FKScript, la generación de código. Este paso se encargará de generar el código intermedio *FKIL* a partir del árbol de sintaxis abstracta generado en la fase de análisis sintáctico y enriquecido durante el análisis semántico.

1. Máquina Virtual FKVM

Antes de comenzar a ver cómo se va a realizar la generación de código para los distintos elementos de nuestro lenguaje debemos comentar algunos detalles acerca de la máquina virtual que va a ejecutar nuestro código y cuya implementación veremos más adelante.

Es importante indicar que nuestra máquina virtual (en adelante VM) estará basada en la pila, en contraposición a máquinas virtuales más clásicas basadas en registros. Con esto queremos decir que todas las operaciones ejecutadas por la VM se realizarán sobre datos que se encuentren con anterioridad en las posiciones superiores de la pila o bien que afectarán de una u otra forma al valor almacenado en la cima de la pila. Una vez ejectada la instrucción la VM se encargará de eliminar automáticamente sus parámetros de la pila si es necesario y de apilar el resultado obtenido si procede según el tipo de instrucción.

Esto se entiende más fácilmente con un ejemplo. Tomemos como muestra la instrucción IADD para realizar la suma de dos enteros. Para realizar esta suma nuestro programa deberá colocar en la cima de la pila sus dos parámetros, bien mediante la carga de un valor inmediato (PUSH) o bien recuperando el dato de alguna variable (LOAD). Una vez apilados los dos parámetros ya se podrá ejecutar la instrucción IADD para realizar la suma, cuyo resultado se almacenará en la cima de la pila tras haber eliminado en primer lugar los dos parámetros. Veámoslo gráficamente:



2. Primeros pasos

El paso de generación de código se implementará haciendo uso de ANTLR v3, mediante una gramática de tipo *tree grammar* al igual que la fase anterior, y utilizando la librería StringTemplate para generar los ficheros de salida a partir de plantillas predefinidas.

Empecemos como siempre por ver las opciones del analizador. En este caso, las opciones serán las mismas que las definidas en el analizador semántico más una adicional para indicar que la salida no será un árbol AST sino una plantilla de StringTemplate. Esta última opción se indicará con `output=template`.

```
tree grammar FKVMGen;

options {
    tokenVocab=FKVM;
    ASTLabelType=FkvmAST;
    output=template;
    language=CSharp;
}
```

En los siguientes apartados veremos cómo definir con ANTLR+StringTemplate la generación de código para cada uno de los elementos significativos de nuestro leguaje FKScript.

3. Generación de código para literales e identificadores

Vamos a empezar a comentar el proceso de análisis por las expresiones más simples del lenguaje como son los literales y los identificadores.

Veamos en primer lugar un ejemplo de cómo se traduciría a código intermedio una sencilla instrucción de asignación en la que tan sólo interviene un literal entero:

```
//Código en FKScript
int a;  //Variable número 1
a = 3;

#Código traducido a FKIL
ipush 3   #Almacena el valor 3 en la cima de la pila
istore 1  #Almacena la cima de la pila en la variable 1
```

Como puede observarse, para el literal la única instrucción de código FKIL que se genera es una instrucción de tipo PUSH con su tipo correspondiente, en este caso de tipo entero (IPUSH), seguida del literal correspondiente. La plantilla a definir para la generación de código de un literal entero será por tanto de la siguiente forma:

```
lit_entero(v) ::= "ipush <v>"
```

Para el resto de tipos de literales estas plantillas serán análogas, cambiando únicamente el tipo de la instrucción PUSH. Por su parte, en la gramática ANTLR tan sólo deberemos indicar la plantilla correspondiente dependiendo del tipo de literal leído y le pasaremos como argumento el texto del propio literal:

```
expresion
...
   |literal -> {$literal.st}
   ;

literal : LIT_ENTERO -> lit_entero(v={$LIT_ENTERO.text})
        | LIT_REAL -> lit_real(v={$LIT_REAL.text})
        | LIT_CADENA -> lit_cadena(v={$LIT_CADENA.text})
        | LIT_LOGICO -> lit_logico(v={$LIT_LOGICO.text})
        ;
```

Ampliemos ahora el ejemplo anterior para que también intervenga un identificador en la parte derecha de una asignación y veamos cómo quedaría su traducción a código FKIL:

```
//Código en FKScript
int a;  //Variable número 1
int b;  //Variable número 2
a = 3;
```

```
b = a;

#Código traducido a FKIL
ipush 3   #Almacena el valor 3 en la cima de la pila
istore 1  #Almacena la cima de la pila en la variable 1
iload 1   #Recupera el valor de la variable 1 a la cima de la pila
istore 2  #Almacena la cima de la pila en la variable 2
```

El código generado en este caso para la variable "a" en la asignación "b=a" ha sido tan sólo una instrucción de tipo LOAD (en este caso de tipo entero, ILOAD) seguida del número de orden de la variable dentro el programa. La plantilla a definir y la generación de código por tanto será tan sencilla como en el caso de los literales. Veamos en primer lugar la plantilla que hemos definido:

```
ident(op,nv) ::= <<
<op> <nv>
>>
```

Vemos que en este caso le pasaremos dos argumentos a la plantilla, el primero de ellos indicando el operador que utilizaremos para recuperar la variable (ILOAD, FLOAD, SLOAD o BLOAD), que dependerá del tipo del identificador, y el segundo que contendrá el número de orden de la variable en el programa. La gramática quedaría de la siguiente forma:

```
expresion
...
   |IDENT {oper=traducirTipo($IDENT.expType)+"load";}
          -> ident(op={oper},nv={$IDENT.symbol.numvar})
   |literal -> {$literal.st}
   ;
```

Para generar el operador hemos utilizado un método propio llamado `traducirTipo()` que devolverá el prefijo correcto del operador LOAD dependiendo del tipo del tipo del identificador pasado como parámetro, es decir, para un identificador de tipo entero devolverá "i", para uno de tipo real devolverá "f" y de forma análoga para el resto de tipos. Por otro lado, el número de la variable lo obtendremos directamente del atributo `symbol.numvar` del identificador, dato que generamos durante la fase de análisis sintáctico.

4. Generación de código para expresiones aritméticas

Veamos en primer lugar un ejemplo de generación de código para una expresión de este tipo:

```
//Código en FKScript
int a;  //Variable número 1
int b;  //Variable número 2
a = 3;
b = a + 5;

#Código traducido a FKIL
ipush 3   #Almacena el valor 3 en la cima de la pila
istore 1  #Almacena la cima de la pila en la variable 1
iload 1   #Recupera el valor de la variable 1 a la cima de la pila
ipush 5   #Almacena el valor 3 en la cima de la pila
iadd      #Realiza la suma de los dos valores superiores de la pila
          #y almacena el resultado en la cima de la pila.
istore 2  #Almacena la cima de la pila en la variable 2
```

Como se observa en el ejemplo, para la expresiones aritméticas lo que se hará será generar en primer lugar el código de las subexpresiones, en este caso un identificador (ILOAD 1) y un literal

(IPUSH 5) y posteriormente llamar al operador correspondiente según el tipo de expresión (en nuestro caso IADD). El patrón para generar la plantilla parece por tanto claro:

```
op_aritmetico(op,e1,e2) ::= <<
<e1>
<e2>
<op>
>>
```

La plantilla recibirá tres argumentos: el operador aritmético a utilizar y el código de las dos subexpresiones de la expresión que estamos generando. En la gramática se seguirá un proceso muy parecido al de los identificadores:

```
expresion
...
   | ^(opa=opAritmetico e1=expresion e2=expresion)
      {oper=traducirTipo($opa.opType)+$opa.st.ToString();}
      -> op_aritmetico(op={oper}, e1={$e1.st}, e2={$e2.st})
...

opAritmetico returns [string opType]
          : op='+' -> {%{"add"}; $opType=$op.expType;}
          | op='-' -> {%{"sub"}; $opType=$op.expType;}
          | op='*' -> {%{"mul"}; $opType=$op.expType;}
          | op='/' -> {%{"div"}; $opType=$op.expType;}
          ;
```

El prefijo del operador lo obtendremos de la misma forma que para el caso de los identificadores, haciendo uso del método `traducirTipo()`, y el operador en concreto lo generaremos en la regla `opAritmetico`, donde también devolveremos el tipo de la expresión (que calculamos durante el análisis semántico) para que sirva como parámetro al método de traducción de tipos.

5. Generación de código para expresiones lógicas

El proceso de generación de código para una expresión lógica no será tan directo como para el resto de elementos que hemos comentado hasta el momento, y para tratar de explicar el por qué empecemos como siempre por ver un ejemplo de traducción de una expresión lógica a código FKIL:

```
Expresión lógica: a > 3

#Traducción a FKIL
ipush 1     #Valor por defecto de la expresión = 1 (true)
iload 1     #Se recupera el valor de la variable 1 a la cima de la pila
ipush 3     #Se coloca el valor 3 en la cima de la pila
ncmp        #Se comparan los dos valores superiores de la pila
ifgt etiq1  #Si el resultado de la comparación es >0 se salta a "etiq1"
pop         #Se desapila el valor por defecto
ipush 0     #Se apila el valor contrario =0 (false)
etiq1:      #Etiqueta de salida
```

Como vemos, la estrategia a seguir para calcular el resultado de este tipo de expresiones será siempre suponiendo que la expresión es verdadera, realizar la comparación, y en caso de ser falsa desapilar el resultado por defecto (true) y apilar el contrario (false).

En este patrón sin embargo hay mucho elementos variables que se deberán tener en cuenta a la hora de definir la plantilla a utilizar para la generación de expresiones lógicas. El primero de ellos es el

tipo de comparación. En el ejemplo hemos utilizado el operador `ncmp` (comparación numérica) debido a que las dos subexpresiones (la variable "a" y el literal "3") eran de tipo entero. Sin embargo, en el caso de comparaciones de cadenas debería utilizarse el operador `scmp` y para los valores lógicos el operador `bcmp`. El segundo parámetro a considerar será el operador utilizado para saltar a la etiqueta de salida. Así, para el operador ">" se utilizará `ifgt`, para "<" usaríamos `iflt` y de forma análoga para el resto de operadores lógicos. Por último, un factor importante será el nombre de la etiqueta de salida, ya que debemos asegurarnos de que en el programa resultante no se repita el nombre de ninguna etiqueta.

Teniendo todo esto en cuenta veamos cómo quedaría la gramática y la plantilla para estas expresiones:

```
@members {
    int nEtiqueta = 1;

    private string operadorComparacion(String t)
    {
        string op = "";

        if(t.Equals("int") || t.Equals("float"))
            op = "ncmp";
        else if(t.Equals("string"))
            op = "scmp";
        else if(t.Equals("bool"))
            op = "bcmp";

        return op;
     }
}

expresion
...
   : ^(opc=opComparacion e1=expresion e2=expresion) {operc =
operadorComparacion($opc.opSecType);}
     -> op_comparacion(opc={operc}, op={$opc.st}, e1={$e1.st}, e2={$e2.st},
et1={nEtiqueta++})
...

opComparacion returns [string opSecType]
          : op='==' -> {%{"ifeq"}; $opSecType=$op.expSecType;}
          | op='!=' -> {%{"ifne"}; $opSecType=$op.expSecType;}
          | op='>=' -> {%{"ifge"}; $opSecType=$op.expSecType;}
          | op='<=' -> {%{"ifle"}; $opSecType=$op.expSecType;}
          | op='>'  -> {%{"ifgt"}; $opSecType=$op.expSecType;}
          | op='<'  -> {%{"iflt"}; $opSecType=$op.expSecType;}
          ;
```

El primero de los problemas a resolver, la elección del operador de comparación, lo solventamos mediante el método propio `operadorComparacion()` (definido en la sección `@members`) al que le pasaremos el tipo de las subexpresiones de la expresión lógica que estamos generando. Este tipo lo almacenamos durante el análisis semántico en el atributo `expSecType` por lo que sólo tenemos que recuperarlo en la regla `opComparacion`. El operador IF a utilizar para el salto a la etiqueta de salida también lo decidimos en dicha regla y se asignará directamente dependiendo del tipo de expresión que hayamos leido de nuestro programa. Por ultimo, el nombre de la etiqueta se generará a partir de un secuencial que iremos incrementando durante la ejecución de nuestro analizador acada vez que haga falta generar una etiqueta. Eset secuencial lo declararemos una vez más en la sección `@members` y en nuestro caso se llamará `nEtiqueta`. La plantilla por su parte quedaría de la siguiente forma:

```
op_comparacion(opc,op,e1,e2,et1) ::= <<
ipush 1
<e1>
<e2>
<opc>
<op> etiq<et1>
pop
ipush 0
etiq<et1>:
>>
```

6. Generación de código para asignaciones

La generación de código para las asignaciones es muy similar al ya visto para los identificadores ya que la única dificultad a salvar será el operador STORE a utilizar para almacenar el valor de la variable, y éste se calculará a partir del tipo de la expresión derecha mediante el método ya comentado `traducirTipo()`.

```
inst_asig
@init {
string oper = "";
} : ^(ASIGNACION IDENT expresion) {oper =
traducirTipo($ASIGNACION.expType)+"store";}
    -> asignacion(op={oper}, nv={$IDENT.symbol.numvar}, val={$expresion.st});
```

El número de orden de la variable se recuperará del atributo `symbol.numvar` que calculamos durante el análisis sintáctico. La plantilla, muy sencilla en este caso, quedaría así:

```
asignacion(op, nv, val) ::= <<
<val>
<op> <nv>
>>
```

7. Generación de código para instrucciones condicionales y bucles

Vemos ahora la generación de código para las instrucciones `if` y `while` de FKScript. Ninguna de ellas añade ninguna dificultad a las ya comentadas por lo que las trataremos muy brevemente. Veamos en primer lugar un ejemplo de traducción de cada una de ellas:

```
//Instrucción IF
if(...expresion-logica...)
{
    //instrucciones-si
}
else
{
    //instrucciones-else
}

#Traducción a FKIL
...expresión-logica...
ifeq etiq1
...instrucciones-si...
goto etiq2
etiq1:
...instrucciones-else...
etiq2:
```

```
//Instrucción WHILE
```

```
while(...expresion-logica...)
{
    //instrucciones-while
}

#Traducción a FKIL
etiq1
...expresion-logica...
ifeq etiq2
...instrucciones-while...
goto etiq1
etiq2:
```

A partir de estos patrones, las plantillas de StringTemplate a definir son prácticamente directas:

```
instif(cond,instsi,instelse,et1,et2) ::= <<
<cond>
ifeq etiq<et1>
<instsi>
goto etiq<et2>
etiq<et1>:
<instelse>
etiq<et2>:
>>

instwhile(cond,instrucciones,et1,et2) ::= <<
etiq<et1>
<cond>
ifeq etiq<et2>
<instrucciones>
goto etiq<et1>
etiq<et2>:
>>
```

La gramática por su parte tampoco añade ninguna particularidad extra a las ya comentadas en apartados anteriores por lo que no nos pararemos demasiado en comentarla:

```
inst_if : ^('if' expresion isi=lista_instrucciones ielse=lista_instrucciones)
        -> instif(cond={$expresion.st},instsi={$isi.st},instelse={$ielse.st},
                  et1={nEtiqueta++},et2={nEtiqueta++});

inst_while : ^('while' expresion li=lista_instrucciones)
           -> instwhile(cond={$expresion.st},instrucciones={$li.st},
                        et1={nEtiqueta++},et2={nEtiqueta++});
```

8. Generación de código para el programa principal FKIL

Una vez comentada la generación de código para cada uno de los elementos principales de nuestro lenguaje ya sólo nos queda ver cómo generar la estructura principal del programa. Esto lo haremos en la regla `principal` de la gramática.

```
principal[int locales] : ^('program' tipo IDENT li=lista_instrucciones)
                       -> principal(nom={$IDENT.text}, loc={locales},
instr={$li.st});
```

Como vemos, esta regla recibirá un parámetro externo llamado `locales` que contendrá el número de variables locales utilizadas por el programa. Este dato lo usaremos para generar la directiva `.locals` de FKIL. El parámetro lo deberemos pasar convenientemente desde el programa principal a la hora de llamar al analizador que acabamos de crear. Por lo demás, el único dato adicional

necesario será el nombre del programa que lo obtendremos directamente del texto del identificador correspondiente en la regla. La plantilla nos quedaría de la siguiente forma:

```
principal(nom, loc,instr) ::= <<
.program <nom>
.locals <loc>
<instr>
>>
```

9. Programa principal

En este punto ya hemos finalizado nuestra gramática y nuestro fichero de plantillas por lo que estamos en condiciones de completar nuestro programa principal para llamar al generador de codigo al final del proceso.

```
//Análisis léxico semántico
FKVMLexer lexer = new FKVMLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
FKVMParser parser = new FKVMParser(tokens);
parser.TreeAdaptor = adaptor;
FKVMParser.programa_return result = parser.programa();

//Si no hay errores léxicos ni sintácticos ==> Análisis Semántico
if (lexer.numErrors + parser.numErrors == 0)
{
    //Analisis Semántico
    CommonTree t = ((CommonTree)result.Tree);
    CommonTreeNodeStream nodes2 = new CommonTreeNodeStream(t);
    nodes2.TokenStream = tokens;
    FKVMSem walker2 = new FKVMSem(nodes2);
    walker2.programa(parser.symtable);

    //Si no hay errores en el análisis semántico ==> Generación de código
    if (walker2.numErrors == 0)
    {
         //Plantillas
         TextReader groupFileR = new StreamReader("FkvmIL.stg");
         StringTemplateGroup templates = new StringTemplateGroup(groupFileR);
         groupFileR.Close();

         //Generación de Código
         CommonTreeNodeStream nodes = new CommonTreeNodeStream(t);
         nodes.TokenStream = tokens;
         FKVMGen walker = new FKVMGen(nodes);
         walker.TemplateLib = templates;
         FKVMGen.programa_return r2 = walker.programa(parser.numVars);
    }
}
```

En primer lugar leeremos el fichero de plantillas `FkvmIL.stg` para generar el objeto `templates`. Este objeto se pasará como entrada a nuestro analizador a traves del atributo `TemplateLib` para indicar las plantillas a utilizar durante la generación de código. Por último, no deberemos olvidar pasar como parámetro de entrada el número de variables locales del programa, dato que obtendremos directamente del analizador sintáctico donde como ya comentamos definimos un atributo llamado `numVars` que contenía precisamente ese dato.

7

Ensamblador de código FKIL (FKASM)

En este capítulo describiremos la fase posterior al proceso de compilación, el módulo ensamblador. Como siempre, comenzaremos enumerando las tareas a realizar por este proceso y posteriormente detallaremos la implementación del módulo en C#.

Una vez hemos construido el compilador para el lenguaje FKScript, que se encargará de transformar el lenguaje de script de alto nivel en código intermedio FKIL, necesitamos un nuevo módulo para transformar éste último en el código binario final que será interpretado por la máquina virtual. En esta sección nos pararemos a detallar la implementación en C# del módulo ensamblador para FKIL.

1. Tareas del ensamblador

Las tareas a realizar por el ensamblador serán las siguientes:

1. Comprobar la validez del código FKIL, es decir, comprobar que la estructura del programa es correcta, comprobar la validez de las directivas e instrucciones utilizadas y validar sus parámetros.
2. Comprobar y traducir las etiquetas utilizadas en el programa.
3. Construir la tabla de literales y resolver sus referencias.
4. Generar el fichero binario ejecutable del programa.

Para llevar a cabo todas estas tareas se realizarán dos pasadas al fichero de entrada. En la primera de ellas (método `generacionP1()`) se verificará el código FKIL y se generarán las tablas de traducción de etiquetas y literales. En la segunda pasada (método `generacionP2()`) se generará el código binario del programa resolviendo convenientemente todas las referencias a etiquetas y literales a partir de las tablas de traducción ya construidas.

```
public void ensamblar(string pathEntrada, string pathSalida)
{
    ensambladoOK = true;

    //Primera pasada
    generacionP1(pathEntrada);

    //Segunda pasada (si la primera no ha producido errores)
    if(ensambladoOK)
        generacionP2(pathEntrada, pathSalida);
}
```

2. Estructuras de datos

Para la realización de todas las tareas comentadas el ensamblador necesitará disponer de varias estructuras de datos para almacenar toda la información necesaria durante el ensamblado:

- Ficheros de entrada (FKIL) y salida (Binario).
- Contador de programa.
- Tabla de instrucciones.
- Tabla de literales.
- Tabla de cadenas.
- Cabecera del fichero de salida.

```
//Tabla de Instrucciones
private Dictionary<string,Instruccion> instSet = null;

//Fichero de entrada
private StreamReader fe = null;
```

```
//Fichero de salida
private BinaryWriter fsal = null;

//Tabla de literales
private List<string> cadenas = new List<string>();

//Tabla de etiquetas
private Dictionary<string, int> etiquetas = new Dictionary<string, int>();

//Cabecera
private Cabecera cab = new Cabecera();

//Contador de programa
private int progCounter = 0;
```

3. Inicialización del ensamblador

Como primer paso del ensamblado vamos a inicializar la tabla de intrucciones. Esta tabla se utilizará tanto para la validación de las intrucciones (operadores y parámetros) del programa como para la posterior generación del fichero de salida.

Esta tabla consistirá en una colección de objetos que nos indiquen, para cada instrucción válida, su nombre, su número de parámetros y su código binario. Para ello, definiremos en primer lugar una clase para almacenar esta información que llamaremos `Instruccion`:

```
class Instruccion
{
    public string nombre;   //Nombre de la instrucción
    public int opcode;      //Código de la instrucción
    public int numpar;      //Número de parámetros

    public Instruccion(string nombre, int opcode, int numpar)
    {
        this.nombre = nombre;
        this.opcode = opcode;
        this.numpar = numpar;
    }
}
```

Una vez contamos con esta clase, la inicialización de la tabla de instrucciones se limitará a añadir a la colección uno de estos objetos por cada instrucción válida de nuestro lenguaje intermedio FKIL:

```
private void inicializarInstSet()
{
    instSet = new Dictionary<string, Instruccion>();

    instSet.Add("ipush", new Instruccion("ipush", 1, 1));
    instSet.Add("fpush", new Instruccion("fpush", 2, 1));
    instSet.Add("spush", new Instruccion("spush", 3, 1));
    instSet.Add("bpush", new Instruccion("bpush", 4, 1));
    instSet.Add("iload", new Instruccion("iload", 5, 1));
    instSet.Add("fload", new Instruccion("fload", 6, 1));
    instSet.Add("sload", new Instruccion("sload", 7, 1));
    //....
}
```

4. Primera pasada del ensamblador

Como ya comentamos anteriormente, durante la primera pasada del ensamblador se realizará la validación de las instrucciones del programa y la generación de las tablas de literales y etiquetas que se utilizarán posteriormente para generar el fichero final ejecutable.

En primer lugar deberemos identificar aquellas líneas del programa que no debemos procesar, como son los comentarios y las líneas en blanco. Posteriormente, para el resto de líneas decidiremos si se trata de una *directiva*, una *etiqueta* o una *instrucción* y actuaremos en consecuencia llamando a métodos independientes.

```
private void generacionP1(string pathEntrada)
{
    string linea;

    //Se abre el fichero de entrada
    fe = File.OpenText(pathEntrada);

    linea = fe.ReadLine();

    //Se procesan todas las lineas que no sean comentarios ni lineas en blanco
    while (linea != null)
    {
        if (linea != null)
        {
            linea = linea.Trim();

            if (!linea.StartsWith("#") && !linea.Equals(""))
            {
                procesarLineaP1(linea);
            }
        }

        linea = fe.ReadLine();
    }

    fe.Close();
}

private void procesarLineaP1(string linea)
{
    if (linea.Trim().EndsWith(":"))         //Etiqueta
    {
        procesarEtiquetaP1(linea);
    }
    else if (linea.Trim().StartsWith("."))  //Directiva
    {
        procesarDirectivaP1(linea);
    }
    else                                    //Instrucción
    {
        procesarInstruccionP1(linea);
    }
}
```

4.1. Procesamiento de directivas

Las directivas soportadas por FKIL no darán como resultado ninguna instrucción ejecutable en el programa final, y tan sólo se utilizarán para completar la información incluida en la cabecera del

fichero ejecutable, que contendrá datos generales sobre el formato del fichero, el programa, y el entorno de ejecución que creará la máquina virtual para albergarlo.

Para almacenar y tratar esta información construiremos una nueva clase llamada `Cabecera`:

```
public class Cabecera
{
    public int magic          = 8080;  //Identificación del formato de fichero
    public int version        = 1;     //Versión del formato de fichero
    public int revision       = 0;     //Revisión del formato de fichero
    public string programName = "";    //Nombre del programa
    public int stackSize      = 1024;  //Tamaño de la pila
    public int heapSize       = 1024;  //Tamaño de la memoria dinámica
    public int nConst         = 0;     //Num. elementos en la tabla de literales
    public int nLocals        = 0;     //Num. variables locales utilizadas

    public Cabecera(){}
}
```

Como vimos en la sección sobre la especificación del lenguaje FKIL, son cuatro las directivas que podemos incluir en un programa escrito en este lenguaje: `.program`, `.stack`, `.heap` y `.locals` , que se corresponden directamente con cuatro de los datos almacenados en la cabecera por lo que su procesamiento por parte del ensamblador será directo. Nos limitaremos a leer cada una de las directivas y trasladar su información asociada a la cabecera:

```
private void procesarDirectivaP1(string linea)
{
    //Se separa la directiva de su parámetro asociado
    string[] tokens = linea.Split(new char[] { ' ' });

    //Se rellena su atributo correspondiente de la cabecera
    if (tokens[0].StartsWith(".program"))
    {
        cab.programName = tokens[1];
    }
    else if (tokens[0].StartsWith(".stack"))
    {
        cab.stackSize = Convert.ToInt32(tokens[1]);
    }
    else if (tokens[0].StartsWith(".heap"))
    {
        cab.heapSize = Convert.ToInt32(tokens[1]);
    }
    else if (tokens[0].StartsWith(".locals"))
    {
        cab.nLocals = Convert.ToInt32(tokens[1]);
    }
}
```

4.2. Procesamiento de etiquetas

El procesamiento de las etiquetas utilizadas en el programa será aún más sencillo que el de las directivas, ya que tan sólo deberemos almacenar las etiquetas encontradas en la tabla de etiquetas junto a la posición que ocupan dentro del programa. Esta posición se obtiene a partir de la variable `progCounter`, que se ira actualizando, como veremos después, a medida que se procesan y validan instrucciones ejecutables.

```
private void procesarEtiquetaP1(string linea)
{
```

```
    etiquetas.Add(linea.Substring(0, linea.Length - 1), progCounter);
}
```

4.3. Procesamiento de instrucciones

Durante el procesamiento de cada instrucción deberemos realizar las siguientes tareas:

1. Validación de la instrucción: el operador tendrá que ser uno de los soportados por el lenguaje.
2. Validación de los parámetros de la instrucción: el número de parámetros de la instrucción deberá coincidir con la información contenida en la tabla de instrucciones.
3. Actualización del contador de programa: se actualizará convenientemente la variable `progCounter` según la instrucción procesada y su número de parámetros.
4. Actualización de la tabla de literales: se añadirá el literal correspondiente a la tabla de literales cada vez que se procese una instrucción que acepte este tipo de parámetros (SPUSH y CALLAPI).

```
private void procesarInstruccionP1(string linea)
{
    //Separamos la instrucción de sus parámetros
    string[] tokens = linea.Split(new char[] { ' ' });

    //Buscamos la instrucción en la tabla de instrucciones
    Instruccion inst = instSet[tokens[0]];

    //Si la instrucción existe
    if (inst != null)
    {
        //Si el número de parámetros de la instrucción es correcto
        if ((tokens.Length - 1) == inst.numpar)
        {
            //Se actualiza el contador de programa
            progCounter += inst.numpar + 1;

            //Si es una instrucción SPUSH o CALLAPI almacenamos
            //el literal en la tabla de cadenas
            if (tokens[0].Equals("spush"))
            {
                cadenas.Add(tokens[1].Substring(1, tokens[1].Length - 2));
            }
            else if (tokens[0].Equals("callapi"))
            {
                cadenas.Add(tokens[1]);
            }
        }
        else
        {
            Console.WriteLine("Número de parámetros incorrecto:");
            Console.WriteLine(linea);
            ensambladoOK = false;
        }
    }
    else
    {
        Console.WriteLine("Instrucción inexistente:");
        Console.WriteLine(linea);
        ensambladoOK = false;
    }
}
```

5. Segunda pasada del ensamblador

En la primera pasada del módulo ensamblador nos hemos preocupado de validar el programa FKIL y de recopilar toda la información necesaria para la verdadera generación del programa ejecutable final. En esta segunda pasada nos preocuparemos tan sólo de las instrucciones ya que como dijimos son los únicos elementos que generarán el código ejecutable del programa, el resto de elementos tan sólo servirán de apoyo para generar correctamente el fichero de salida.

En primer lugar abriremos el fichero de salida y escribiremos toda la información de la cabecera que hemos generado durante la primera pasada. Esto lo haremos mediante la llamada al método `escribirCabecera()` cuya implementación es directa:

```
private void escribirCabecera()
{
    //Cabecera
    fsal.Write(cab.magic);
    fsal.Write(cab.version);
    fsal.Write(cab.revision);
    fsal.Write(cab.programName);
    fsal.Write(cab.stackSize);
    fsal.Write(cab.heapSize);
    fsal.Write(cab.nLocals);
    fsal.Write(cadenas.Count);

    //Tabla de literales
    foreach (string c in cadenas)
        fsal.Write(c);
}
```

Tras escribir la cabecera, recorreremos de nuevo el fichero de entrada pero esta vez procesando tan solo las lineas que contienen instrucciones ejecutables:

```
private void generacionP2(string pathEntrada, string pathSalida)
{
    string linea;

    //Abrimos el fichero de entrada
    fe = File.OpenText(pathEntrada);

    //Abrimos el fichero de salida
    fsal = new BinaryWriter(new FileStream(pathSalida, FileMode.Create));

    //Generación de la Cabecera
    escribirCabecera();

    //Generación del Código
    linea = fe.ReadLine();

    while (linea != null)
    {
        if (linea != null)
        {
            //Si no es un comentario o una linea en blanco se procesa
            if (!linea.Trim().StartsWith("#") && !linea.Trim().Equals(""))
            {
                procesarLineaP2(linea);
            }
        }

        linea = fe.ReadLine();
```

```
    }

    fe.Close();

    fsal.Flush();
    fsal.Close();
}

private void procesarLineaP2(string linea)
{
    //Procesamos tan sólo la líneas que contengan instrucciones
    if(!linea.Trim().StartsWith(".") && !linea.Trim().EndsWith(":"))
    {
        procesarInstruccionP2(linea);
    }
}
```

Por último, el procesamiento de cada instrucción será una tarea sencilla, debiéndonos preocupar tan sólo de escribir al fichero de salida el código de la instrucción leída junto a sus parámetros en caso de existir. Para ello, nos basaremos una vez más en la información de la tabla de instrucciones.

```
private void procesarInstruccionP2(string linea)
{
    //Separamos la instrucción de sus posibles parámetros
    string[] tokens = linea.Split(new char[] { ' ' });

    //Buscamos la instrucción en la tabla de instrucciones
    Instruccion inst = instSet[tokens[0]];

    //Escribimos el código de la instrucción al fichero de salida
    fsal.Write((float)inst.opcode);

    //Si la instrucción tiene ún parámetro simple
    if (inst.nombre.Equals("ipush") ||
       inst.nombre.Equals("iload")  ||
       inst.nombre.Equals("istore") ||
       inst.nombre.Equals("bpush")  ||
       inst.nombre.Equals("bload")  ||
       inst.nombre.Equals("bstore") ||
       inst.nombre.Equals("sload")  ||
       inst.nombre.Equals("sstore") ||
       inst.nombre.Equals("fpush")  ||
       inst.nombre.Equals("fload")  ||
       inst.nombre.Equals("fstore"))
    {
        fsal.Write(Convert.ToSingle(tokens[1]));
    }
    //Si la instrucción tiene asociada una etiqueta
    else if (inst.nombre.Equals("goto") ||
       inst.nombre.Equals("ifeq") ||
       inst.nombre.Equals("ifne") ||
       inst.nombre.Equals("ifgt") ||
       inst.nombre.Equals("ifge") ||
       inst.nombre.Equals("iflt") ||
       inst.nombre.Equals("ifle"))
    {
        fsal.Write((float)etiquetas[tokens[1]]);
    }
    //Si la instrucción tiene asociada un literal cadena
    else if (inst.nombre.Equals("spush"))
    {
        fsal.Write((float)cadenas.IndexOf(
                    tokens[1].Substring(1, tokens[1].Length - 2)));
```

```
    }
    //Si la instrucción tiene asociada un nombre de función
    else if(inst.nombre.Equals("callapi"))
    {
        fsal.Write((float)cadenas.IndexOf(tokens[1]));
    }
}
```

Como podemos ver en el código anterior, el procesador comenzará escribiendo el código de la instrucción al fichero de salida. Posteriormente decidirá si debe escribir algún dato más asociado a dicha instrucción, como puede ser un parámetro numérico, una referencia a etiqueta o una referencia a un literal.

En el primer caso, parámetro numérico, se escribirá el parámetro directamente al fichero de salida. En caso de referencias a etiquetas se traducirá el nombre de la etiqueta por su posición en el código, información que teníamos ya almacenada en la tabla de etiquetas. Por último, para instrucciones que hacen referencia a cadenas de caracteres (SPUSH) o nombres de función (CALLAPI) resolveremos dicha referencia consultando la tabla de literales que hemos construido durante la primera pasada del ensamblador.

8

Máquina Virtual de FKScript (FKVM)

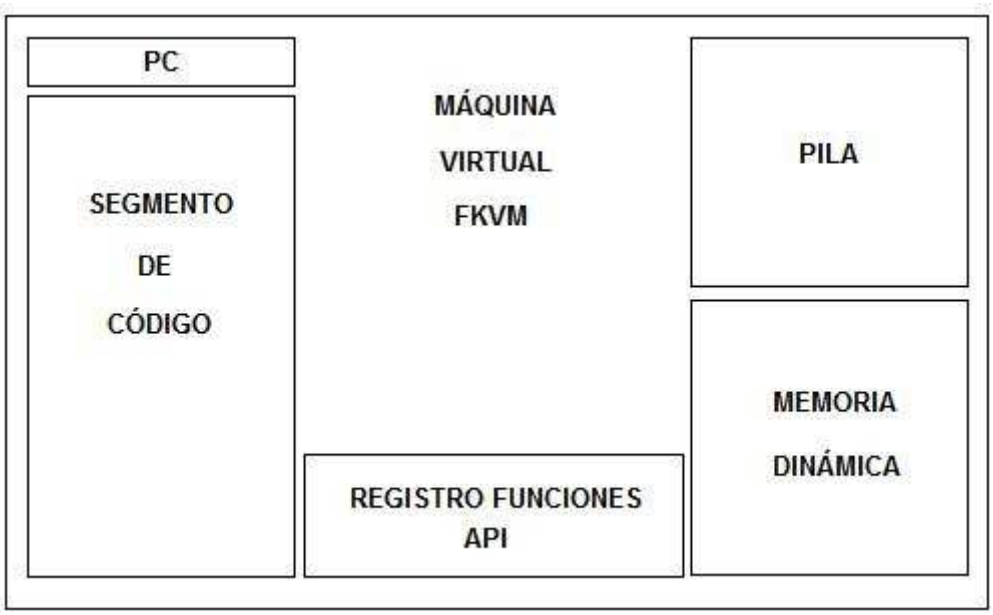
En este último capítulo nos centraremos en el módulo encargado de la ejecución de un programa escrito en FKScript, la máquina virtual. Veremos su implementación en C# y la forma en que podremos integrarla con otras aplicaciones para trabajar de forma conjunta.

El último módulo necesario en nuestro sistema será la máquina virtual, que será la encargada de interpretar el fichero generado por el ensamblador y ejecutar las instrucciones contenidas en él.

En los próximos apartados veremos la estructura de esta máquina virtual, los procesos de carga de y ejecución del programa y el mecanismo definido para integrar la máquina virtual con otras aplicaciones y permitir la interacción entre ambas.

1. Estructura de la máquina virtual

La estructura de la máquina virtual FKVM se representa en la siguiente figura:



A continuación se realiza una breve descripción de cada uno de estos elementos.

1.1. Segmento de código

En esta estructura se almacenará todo el código del programa a ejecutar, sin incluir los datos de la cabecera del programa ni los literales iniciales contenidos en el fichero del programa.

1.2. Registro contador de programa

Este registro contendrá en todo momento la posición, dentro del segmeto de código, de la próxima instrucción a ejecutar por la máquina virtual, o en su defecto, el próximo parámetro a leer durante la ejecución de una instrucción determinada.

1.3. Pila

La pila constituye el elemento más importante de la máquina virtual y aque en ella se almacenarán las variables locales del programa y albergará todos los resultados intermedios producidos por el programa en ejecución. Sobre esta estructura la máquina virtual realizará todas las operaciones del programa almacenando en ella todos los valores necesarios para su ejecución. Los valores numéricos y booleanos se almacenarán directamente en la pila, y para los valores de tipo cadena se almacenará una referencia a la memoria dinámica.

1.4. Memoria dinámica

En esta estructura de la máquina virtual se almacenarán todos los valores que no sean numéricos o booleanos. En nuestro caso, tan sólo contendrá los valores de tipo cadena de caracteres, ya sean literales utilizados en operaciones del programa o nombres de funciones externas.

1.5. Tabla de funciones API

En esta tabla se registrarán todas las funciones externas disponibles para ser llamadas durante la ejecución de un programa FKScript. Contendrá la relación entre el nombre de éstas y una referencia a la función en sí.

Todas estas estructuras, a excepción de las relacionadas con las funciones API que se comentarán más adelante, estarán representadas en nuestra implementación mediante las siguientes colecciones de C#:

```
//Pila
private Pila pila = null;

//Memoria dinámica
private List<string> heap = null;

//Segmento de código
private List<float> codigo = null;

//Contador de programa (PC)
private int pc = 0;
```

2. Carga de un programa

Durante el proceso de carga de un programa en la máquina virtual se deberán realizar las siguientes operaciones:

1. Apertura y lectura del fichero de entrada.
2. Lectura y registro de la cabecera del fichero.
3. Inicialización del segmento de código, pila y memoria dinámica.
4. Lectura y almacenamiento en la memoria dinámica de los literales iniciales del programa.
5. Lectura y almacenamiento en el segmento de código de todo el código del programa.

La cabecera del fichero se almacenará en un objeto de tipo `Cabecera` como el que ya utilizamos durante el ensamblado del programa.

```
//Apertura del fichero de entrada
BinaryReader fent = new BinaryReader(
                        new FileStream(path,
                                FileMode.Open, FileAccess.Read));

//Lectura de la cabecera
cab.magic = fent.ReadInt32();
cab.version = fent.ReadInt32();
cab.revision = fent.ReadInt32();
cab.programName = fent.ReadString();
cab.stackSize = fent.ReadInt32();
cab.heapSize = fent.ReadInt32();
cab.nLocals = fent.ReadInt32();
cab.nConst = fent.ReadInt32();
```

La inicialización de estructuras es también sencilla y se utilizará parte de la información leida de la cabecera. Así, la pila se inicializará con un tamaño inicial igual al número de variables locales del programa, `cab.nLocals`, y la memoria dinámica se inicializará al tamaño máximo permitido `cab.heapSize`.

```
//Inicialización de estructuras
pc = 0;                                 //Contador de programa
codigo = new List<float>();             //Segmento de código
pila = new Pila(cab.nLocals);           //Pila
heap = new List<string>(cab.heapSize);  //Memoria dinámica
```

Por último, insertaremos en la memoria dinámica los literales iniciales del programa contenidos en la tabla de literales del fichero de entrada y cargaremos el segmento de código con todo el código del programa.

```
//Literales iniciales
for (int i = 0; i < cab.nConst; i++)
{
    heap.Add(fent.ReadString());
}

//Código del programa
while (fent.PeekChar() != -1)
{
    codigo.Add(fent.ReadSingle());
}

fent.Close();
```

3. Ejecución del programa

El bucle principal de ejecución del programa, una vez inicializadas y cargadas todas las estructuras de la máquina virtual, será muy sencillo. Nos limitaremos a ejecutar todas las instrucciones del programa mientras el contador de programa no haya alcanzado el final del fichero.

Se ha añadido una nueva inicialización previa de la máquina virtual para permitir varias ejecuciones de un mismo programa sin tener que cargarlo de nuevo.

```
public void ejecutar()
{
    //Inicialización de registros y estructuras
    inicializarEstado();

    //Bucle principal
    while (pc < codigo.Count)
    {
        ejecutarInstruccion();
    }
}
```

4. Ejecución de instrucciones

El proceso de ejecución de una instrucción dependerá obviamente de cada instrucción leída del fichero de entrada. Definiremos un método para cada una de las instrucciones soportadas por FKIL e insertaremos un paso inicial donde se decida qué método ejecutar según la instrucción leida.

```
private void ejecutarInstruccion()
{
```

```
    int opcode = (int)codigo[pc++];

    switch (opcode)
    {
        case IPUSH:
        case FPUSH:
        case SPUSH:
        case BPUSH:
            ejecutarPUSH();
            break;
        case POP:
            ejecutarPOP();
            break;
        case IADD:
        case FADD:
            ejecutarADD();
            break;
        case ISUB:
        case FSUB:
            ejecutarSUB();
            break;
        ...
}
```

Como puede observarse en el código anterior, algunas de las instrucciones de FKIL podrán agruparse en un sólo método de ejecución ya que su efecto sobre la máquina virtual será exactamente el mismo. Así, por ejemplo, todas las operaciones de tipo *PUSH* se ejecutarán mediante el método único `ejecutarPUSH()`.

A continuación veremos la implementación de los métodos de ejecución de algunas instrucciones de FKIL. En el código fuente proporcionado pordrán consultarse el resto de instrucciones.

4.1. Instrucciones PUSH

Para instrucciones de tipo PUSH leeremos el dato a apilar desde el segmento de código (posición actual del contador de programa) y lo añadiremos a la cima de la pila.

```
private void ejecutarPUSH()
{
    float op1 = codigo[pc++];
    pila.Add(op1);
}
```

4.2. Instrucciones LOAD

Para instrucciones de tipo LOAD leeremos el número de la variable a recuperar desde el segmento de código (posición actual del contador de programa) y lo añadiremos a la cima de la pila. Recordemos que las variables locales también se encuentran almacenadas en la pila.

```
private void ejecutarLOAD()
{
    int op1 = (int)codigo[pc++];
    pila.Add(pila[op1]);
}
```

4.3. Instrucciones STORE

Para instrucciones de tipo STORE leeremos el número de la variable a actualizar desde el segmento de código (posición actual del contador de programa), actualizaremos la variable y desapilaremos el valor almacenado.

```
private void ejecutarSTORE()
{
    int op1 = (int)codigo[pc++];
    pila[op1] = pila.pop();
}
```

4.4. Instrucciones aritméticas

Las instrucciones aritmética sobre valores enteros o reales se realizarán también de forma conjunta mediante un sólo método. Éste leerá y desapilará los dos valores sobre los que actuará la instrucción, ejecutará la operación y volverá a apilar en la cima de la pila el resultado obtenido. Como ejemplo, veamos cómo quedaría la ejecución de una instrucción de tipo ADD:

```
private void ejecutarADD()
{
    float op1, op2;

    op1 = pila.pop();
    op2 = pila.pop();
    pila.Add(op1 + op2);
}
```

4.5. Instrucciones de comparación

Las comparaciones numéricas o de booleanos serán igual de sencillas que las ya vistas hasta el momento. Se leerán y desapilarán los valores a comparar de la pila, se realizará la comparación y se generará el resultado según el convenio establecido de valores de retorno. En nuestro caso indicamos este convenio en los cometarios al principio del método de ejecución. Así, por ejemplo, la comparación numérica quedaría de la siguiente forma:

```
private void ejecutarNCMP()
{
    //Si OP1 > OP2 --> -1
    //Si OP1 = OP2 -->  0
    //Si OP1 < OP2 --> +1

    float op1, op2, res = 0F;

    op1 = pila.pop();
    op2 = pila.pop();

    if (op1 > op2)
        res = -1.0F;
    else if (op1 < op2)
        res = +1.0F;

    pila.Add(res);
}
```

4.6. Instrucciones de salto condicional

Veamos ahora algún ejemplo de instrucción de salto de tipo IF. Estas instrucciones leerán y desapilarán el valor a comparar desde la pila, realizarán la comparación correspondiente según el tipo concreto de instrucción y actualizarán el contador de programa con el valor correspondiente

según se haya cumplido la comparación o no. Así, si la comparación es verdadera se actualizará el contador de programa con el valor almacenado en la cima de la pila (segundo parámetro de la instrucción de salto), y en caso contrario se incrementará el contador en una unidad como ocurría con el resto de instrucciones. Veamos como ejemplo la instrucción IFEQ:

```
private void ejecutarIFEQ()
{
    float op1;

    op1 = pila.pop();

    if (op1 == 0F)
        pc = (int)codigo[pc];
    else
        pc++;
}
```

4.7. Instrucción de llamada a función externa

Por último veamos como implementar la ejecución de llamadas a funciones externas. En primer lugar recuperaremos el nombre de la función a partir del parámetro almacenado en la cima de la pila y los literales almacenados en la memoria dinámica de la máquina virtual. Con este valor, accederemos a la tabla de funciones API y en caso de tratarse de una llamada válida llamaremos a la función a través de su delegado.

```
private void ejecutarCALLAPI()
{
    string fun = heap[(int)codigo[pc++]];

    if (registroApi.ContainsKey(fun))
        registroApi[fun]();
}
```

5. Integración con otras aplicaciones

Uno de los requisitos que nos pusimos al comienzo de este desarrollo fue que nuestra máquina virtual pudiera integrarse con otras aplicaciones, siempre que éstas expusieran una API con el formato correcto, de forma que pudieran comunicarse entre sí. Dicho de otra forma, queremos que nuestra máquina virtual pueda acoplarse fácilmente como módulo a cualquier otra aplicación y que podamos utilizar parte de la funcionalidad de dicha aplicación desde nuestro lenguaje de script.

Si consultamos la especificación de FKScript, podemos ver que la forma de llamar a funciones de la API de una aplicación externa será declarar dichas funciones al comienzo del programa mediante la palabra clave `api` e insertar llamadas en el programa (utilizando la sintaxis tradicional de C# o Java) como si de cualquier otra expresión se tratara. Veamos un ejemplo:

```
api float calcularRadio();
api void dibujarCirculo(int x, int y, float radio);

program void Prueba
{
   float r;

   r = 10 + calcularRadio();

   dibujarCirculo(50, 100, r);
}
```

En el programa anterior se utilizan dos funciones de la API de una aplicación externa. Ambas funciones están convenientemente declaradas al comienzo del script, indicando su nombre, parámetros y tipo de salida. En el cuerpo del programa se utiliza la primera de ellas, sin parámetros, en el interior de una expresión y la segunda como llamada aislada sin devolver ningún resultado, ambas formas serán válidas en FKScript.

Cuando compilamos este programa a código intermedio, el resultado que obtendremos será el siguiente:

```
.program Prueba
.locals 1
ipush 10
callapi calcularRadio   //Llamada a la función de la API
iadd
istore 0
ipush 50                //Primer parámetro de la llamada = 50
ipush 100               //Segundo parámetro de la llamada = 100
fload 0      //Tercer parámetro de la llamada = Variable 'r' (nº variable = 0)
callapi dibujarCirculo  //Llamada a la función de la API
```

Vemos que las llamadas a funciones de la API se transforman en llamadas a la instrucción `callapi` de FKIL. Además, como se observa, los parámetros que reciba dicha llamada serán apilados previamente a la ejecución de dicha instrucción.

Una vez visto cómo funcionan a alto nivel las llamadas a la API debemos plantearnos varias cuestiones. En primer lugar habrá que implementar algún mecanismo para que la aplicación que hospedará a la máquina virtual comunique a ésta las funciones de su API que estarán disponibles, y en segundo lugar habrá que definir la forma en que deberán estar definidas estas funciones y la forma de comunicación entre ambas aplicaciones.

5.1. Registro de funciones API

Tanto para poder validar las llamadas realizadas a funciones externas (algo que se hará por tanto en tiempo de ejecución y no durante la compilación) como para disponer de las referencias necesarias a dichas funciones la máquina virtual deberá contar con un *índice* donde se almacene de alguna forma la colección de funciones externas disponibles.

En nuestro caso esto lo conseguiremos utilizando una colección de *delegados*. Un *delegado* podría definirse de forma sencilla como una referencia a una función (sé que alguien me caneará por explicarlo de esta manera :) con un prototipo concreto, es decir, un determinado tipo de salida y unos parámetros definidos.

Dado que no sabemos a priori el prototipo de todas las posibles funciones que pueden formar parte de una API determinada, nosotros vamos a optar por obligar a que todas las funciones de la API de una aplicación que quiera hacer uso de FKScript como lenguaje integrado tengan la siguiente forma:

```
void NombreFuncionAPI()
```

Es decir, que ninguna función externa podrá recibir directamente parámetros ni devolver ningún resultado. Aunque esto pueda parecer un gran inconveniente no es tal, ya que en la práctica estas restricciones no serán reales sino sólo una cuestión de forma. En el apartado siguiente veremos cómo solventar esto para que nuestras funciones puedan recibir parámetros y devolver resultados.

Una vez decidida la forma que tendrán las funciones externas ya podemos definir nuestro delegado y la colección que hará las funciones de índice:

```
public delegate void ApiCall();
private Dictionary<string, ApiCall> registroApi = null;
```

La colección `registroApi` contendrá una relación entre los nombres de las funciones disponibles (primer parámetro: `string`) y una referencia a la función propiamente dicha (segundo parámetro: `ApiCall`) de forma que ésta pueda ser llamada directamente desde la colección.

El mantenimiento de este ínidce deberá realizarlo la aplicación host, es decir, que será la aplicación contenedora la que añada o elimine las funciones que estarán disponibles para su uso desde FKScript. Por lo tanto, tendremos que proporcionar a ésta una forma de realizar estas operaciones. Definiremos para ello dos métodos con los que poder registrar y eliminar funciones del índice de forma sencilla:

```
public void registrarFuncionApi(string nombre, ApiCall ac)
{
    registroApi.Add(nombre, ac);
}

public void deregistrarFuncionApi(string nombre)
{
    registroApi.Remove(nombre);
}
```

5.2. Definición de la API de la aplicación externa

Como dijimos en el apartado anterior, las funciones definidas como API de la aplicación externa deberán tener todas el mismo prototipo: no podrár recibir parámetros ni devolver resultados. Sin embargo ya indicamos que esto no era más que una cuestión de forma y que por tanto nuestras funciones sí que podrían realizar dichas operaciones en la práctica. ¿Cómo conseguiremos esto?

Tanto la recepción de parámetros como la devolución de resultados se realizarán de forma explícita mediante operaciones de la propia función de la API, es decir, será cada función la encargada de leer desde la máquina virtual los parámetros que le sean necesarios y de devolver a la máquina virtual el resultado generado en caso de ser así.

Para ello, implementaremos en nuestra máquina virtual una serie de métodos públicos que puedan ser llamados desde las funciones API para realizar todas las operaciones descritas.

Para la lectura de parámetros la máquina virtual devolverá el dato apilado en la cima de la pila, siempre teniendo en cuenta su tipo:

```
public int obtenerParametroInt()
{
    return (int)pila.pop();
}

public float obtenerParametroFloat()
{
    return pila.pop();
}

public bool obtenerParametroBool()
{
```

```
    return pila.pop() == 0F ? false : true;
}

public string obtenerParametroString()
{
    return heap[(int)pila.pop()];
}
```

Y para la devolución de resultados la máquina virtual se limitará a apilar el dato proporcionado por la función API, teniendo en cuenta su tipo para el correcto almacenamiento y acciones adicionales (como por ejemplo el registro de una cadena en la tabla de literales):

```
public void devolverRetornoInt(int ret)
{
    pila.push((float)ret);
}

public void devolverRetornoFloat(float ret)
{
    pila.push(ret);
}

public void devolverRetornoInt(bool ret)
{
    pila.push(ret == true ? 1F : 0F);
}

public void devolverRetornoString(string ret)
{
    heap.Add(ret);

    pila.push((float)heap.Count-1);
}
```

Como ejemplo, imaginemos que tenemos que implementar una función externa que indique si un determinado número entero es par. Nuestra función recibirá como parámetro un número entero y devolverá como resultado un booleano. La implementación quedaría de la siguiente forma:

```
public void esPar()   //Prototipo real:  bool esPar(int num)
{
    bool res = false;

    //Leemos el parámetro desde la máquina virtual
    int num = vm.obtenerParametroInt();

    if(num % 2 == 0)
        res = true;

    //Devolvemos el resultado a la máquina virtual
    vm.devolverRetornoBool(res);
}
```

ANEXO I

Especificación del lenguaje FKScript

Indice

1. Programa FKScript

Un programa en FKScript estará compuesto por una serie de declaraciones de funciones API y una única función expresada mediante la sintaxis siguiente:

```
declaraciones-api

program tipo nombrePrograma
{
    ...CódigoPrograma...
}
```

El *tipo* podrá ser cualquiera de los indicados en el apartado siguiente e indica el tipo de dato del valor devuelto por el programa.

Por su parte, las declaraciones de la API seguirán la notación siguiente:

```
api tipo nombreFuncion ( lista_parámetros );
```

La lita de parámetros estará formada por una serie de declaraciones de variable separadas por comas. Un ejemplo de delaración de función sería el siguiente:

```
api int sumaEnteros ( int entero1, int entero2 );
```

2. Tipos de datos

Los tipos de datos permitidos serán:

`int`	Valor numérico entero
`float`	Valor numérico real
`bool`	Valor lógico

`string`	Cadena de caracteres

3. Declaración de variables

La declaración de variables en FKScript se realizará de la misma forma que en los lenguajes C# o Java, con la diferencia de no poder inicializarse dicha variable en la propia declaración. La sintaxis será la siguiente:

```
Tipo NombreVariable;
```

El tipo de dato debe ser uno de los indicados en el apartado anterior y el nombre de la variable debe cumplir las siguientes reglas:

1. Comenzar por una letra o por el caracter de subrayado '_'.
2. El resto de caracteres deben ser dígitos, letras o caracteres de subrayado '_'.

El lenguaje será sensible a mayúsculas y minúsculas, por lo que identificadores como por ejemplo `NumLinea` y `numlinea` se considerarán distintos.

Como ejemplo, para declarar una variable de tipo entero llamada numeroFila utilizaremos la siguiente sentencia:

```
int numeroFila;
```

4. Instrucciones

4.1. Asignaciones

Las instrucciones de asignación serán idénticas a las de C# o Java, siguiéndose la siguiente sintaxis:

```
NombreVariable = expresión;
```

La expresión asignada podrá ser un literal, un identificador o cualquier operación entre ellos. Así por ejemplo, serán asignaciones válidas:

```
miVariable = 3;
miOtraVariable = miVariable;
otraVariableMas = 3 + (miVariable * 5);
```

4.2. Instrucción condicional `if`

La instrucción `if` seguirá la siguiente sintaxis:

```
if ( expresionLogica )
{
    Código ejecutado cuando 'expresionLogica' es cierta
}
else
{
```

```
    Código ejecutado cuando 'expresionLogica' es falsa
}
```

El bloque `else` será opcional en esta instrucción.

4.3. Instrucción iterativa `while`

La instrucción `while` seguirá la siguiente sintaxis:

```
while ( expresionLogica )
{
    Código ejecutado mientras 'expresionLogica' es cierta
}
```

4.4. Instrucción `return`

La instrucción `return` se utilizará para indicar la expresión devuelta por el programa y seguirá la siguiente sintaxis:

```
return expresion
```

5. Expresiones

5.1. Expresiones Aritméticas

Las expresiones aritméticas permitidas para los tipos `int` y `float` serán las siguientes:

+	Suma
-	Resta / Menos unario
*	Producto
/	División

5.2. Expresiones Lógicas

Las expresiones lógicas permitidas serán las siguientes:

==	Igual
!=	Distinto
>	Mayor
>=	Mayor o igual
<	Menor
<=	Menor o igual
!	Negación lógica

5.3. Expresiones de cadenas

Las expresiones entre cadenas permitidas serán las siguientes:

+	Concatenación de cadenas

6. Ejemplo programa FKScript

A continuación se muestra un programa simple escrito en lenguaje FKScript:

```
api int sumaEnteros(int e1, int e2);

program Prueba
{
    int c;

    c = sumaEnteros(5,2);

    if(c > 2)
    {
        c = 1;
    }

    return c;
}
```

ANEXO II

Especificación del lenguaje FKIL

Índice

1. Programa FKIL

Un programa en FKIL estará compuesto por un unico módulo con la siguiente estructura:

```
directiva_1
directiva_2
...
instrucción_1
instrucción_2
...
```

Nota: Todas las directivas incluidas en el programa deberán aparecer antes de cualquier instrucción.

2. Comentarios

Se podrán incluir comentarios en el código precediendo la una linea con el caracter '#'.

3. Directivas

Las directivas permitidas en FKIL son las siguientes:

Directiva	Descripción	Valor por defecto
`.program`	Indica el nombre del programa.	
`.stack`	Indica el tamaño máximo de la pila.	1024
`.heap`	Indica el tamaño máximo de la memoria dinámica.	1024
`.locals`	Indica el número de variables locales utilizadas.	0

Ejemplos:

```
.program Prueba
.stack 1500
.heap 1000
.locals 4
```

4. Juego de instrucciones

Las instrucciones permitidas en FKIL son las siguientes:

Instrucción	Descripción	Nº Par.	Parámetro
ipush	Coloca una constante entera en la pila	1	Constante
fpush	Coloca una constante real en la pila	1	Constante
spush	Coloca una constante cadena en la pila	1	Constante
bpush	Coloca una constante booleana en la pila	1	Constante
iload	Carga el valor de una variable entera en la pila	1	Nº de variable
fload	Carga el valor de una variable real en la pila	1	Nº de variable
sload	Carga el valor de una variable cadena en la pila	1	Nº de variable
bload	Carga el valor de una variable booleana en la pila	1	Nº de variable
istore	Almacena en una variable entera el primer elemento de la pila	1	Nº de variable
fstore	Almacena en una variable real el primer elemento de la pila	1	Nº de variable
sstore	Almacena en una variable cadena el primer elemento de la pila	1	Nº de variable
bstore	Almacena en una variable booleana el primer elemento de la pila	1	Nº de variable
pop	Elimina el primer elemento de la pila	0	
iadd	Suma de enteros	0	
fadd	Suma de reales	0	
isub	Resta de enteros	0	
fsub	Resta de reales	0	
imul	Producto de enteros	0	
fmul	Producto de reales	0	
idiv	División de enteros	0	
fdiv	División de reales	0	
nneg	Negación numérica	0	
bneg	NO lógico	0	
ncmp	Comparación numérica	0	
bcmp	Comparación booleana	0	
goto	Salto incondicional	1	Etiqueta
ifeq	Salto si igual a 0	1	Etiqueta
ifne	Salto si distinto de 0	1	Etiqueta
iflt	Salto si menor que 0	1	Etiqueta
ifgt	Salto si mayor que 0	1	Etiqueta
ifge	Salto si mayor o igual que 0	1	Etiqueta
ifle	Salto si menor o igual que 0	1	Etiqueta
scmp	Comparación de cadenas	1	Etiqueta
sadd	Concatenación de cadenas	1	Etiqueta
iret	Retorno de entero	0	
fret	Retorno de real	0	

sret	Retorno de cadena	0	
bret	Retorno de booleano	0	
callapi	Llamada a función de la API externa	1	Nombre Función

La máquina virtual se basará únicamente en la pila, por lo que los parámetros de la mayoría de las instrucciones deberán apilarse antes de ejecutar la instrucción.

5. Etiquetas

Las etiquetas incluidas en el código, a las cuales podrán hacer referencia todas las instrucciones condicionales se indicarán con un identificadro seguido del caracter ':'

Ejemplo:

```
ifeq etiqueta1
ipsuh 1

etiqueta1:
ipush 2
```

6. Ejemplo programa FKScript

A continuación se muestra un programa simple escrito en lenguaje FKScript:

```
#Programa de prueba

.program Prueba
.locals 1

ipush 5
istore 0
ipush 1
fload 0
ipush 2
ncmp
ifgt etiq1
pop
ipush 0
etiq1:
ifeq
etiq2
ipush 1
istore 0
goto etiq3
etiq3:
fload 0
ipush 3
fdiv
fret
```