

En este proyecto vamos a tratar el desarrollo de un compilador de lenguajes. El lenguaje que va a interpretar el compilador es orientado a objetos, y con una sintaxis similar a Java, aunque el idioma del lenguaje va a ser el castellano. Este lenguaje es conocido como L-2 y será descrito en profundidad en secciones posteriores.

El desarrollo de un compilador completo de lenguaje es un trabajo enorme, en el que podría intervenir cientos de programadores trabajando durante años. Por eso, en este proyecto sólo vamos a cubrir algunas etapas del proceso de compilación. Un compilador real tiene las siguientes etapas:

1. Análisis léxico

En esta etapa se lee el archivo de texto que contiene el código fuente y las cadenas se van agrupando en componentes léxicos o tokens. Estos componentes léxicos son los operadores, identificadores y palabras reservadas del lenguaje. Por lo tanto, transforma en flujo de caracteres en un flujo de objetos tokens, ignorando los comentarios y los espacios en blanco y tabulaciones. Esta primera fase es capaz de detectar errores de tipo léxico. Por ejemplo, si en lugar de poner `if` pusieramos `ifi`, el analizador léxico no usaría el token predefinido `IF` para esa secuencia de caracteres, sino que entendería que es un identificador cualquiera (como pudiera ser el nombre de una variable).

2. Análisis sintáctico

Esta etapa se encarga de verificar que el documento tiene una disposición sintáctica correcta, agrupando el código en frases gramaticales con sentido. En otras palabras, que el flujo de tokens tenga el orden correcto. Esta etapa detecta errores sintácticos. Siguiendo el ejemplo anterior, si hubiesemos puesto `ifi` en lugar de `if`, esta etapa detectaría que la estructura de un bloque `if` no es así, detectando así el error en el código fuente. A la siguiente fase le manda árboles de sintaxis abstracta. Por ejemplo, un árbol cuyo nodo raíz es `PROGRAMA`, que se compone de una lista de clases. Cada una de estas serían árboles que describen los distintos atributos y métodos de la misma, y así sucesivamente. Esta fase no puede detectar los errores.

3. Analizador semántico

Esta es una de las fases más complejas del proceso de compilación. Tiene que realizar varias tareas: la resolución de nombres, el tipo de las expresiones y la evaluación L/R. La resolución de nombres consiste en comprobar que cada identificador que usemos (ya sea una variable local, parámetro, atributo o método) ha sido previamente declarado y además, si es visible desde el ámbito actual. En cuanto a los tipos, hay que comprobar que en las expresiones donde intervengan 2 identificadores (asignación, suma, producto, concatenación, etc.) comprobar que ambos identificadores aceptan dicha operación, que son de tipos compatibles. La evaluación L/R consiste en que para cada asignación, cada componente puede aparecer o no en esa posición. Por ejemplo, el resultado de una serie de operaciones no puede aparecer a la izquierda, sólo a la derecha, ya que no se puede asignar un valor al resultado de una serie de operaciones.

4. Optimización del código

Esta es, sin lugar a dudas, la fase más compleja de todo el desarrollo. Esta etapa puede ir antes o después de la generación de código, o realizarla en dos pasos. Dichas optimizaciones pueden ser de dos tipos principalmente: dependientes de la máquina (una compilación específica del compilador) o independientes de la máquina (desenrollando bucles, eliminando redundancias, etc.).

5. Generacion de código.

Es una fase relativamente sencilla. Obtenemos de la etapa anterior unos arboles que sabemos que son perfectamente válidos. En esta etapa, para cada arbol generamos el codigo (ya sea alguno existente, como ensamblador, bytewcodes de java, il de .net, o uno que inventemos y que interprete una maquina virtual).

Este proyecto cubre las dos primeras fases del análisis, asi como la resolución de nombres de la tercera etapa.

1. Objetivos

Los objetivos de nuestra aplicacion es validar y mostrar los errores de un trozo de codigo que tiene que cumplir que:

Use una sintaxis adecuada.

Para ello construiremos el analizador lexico y el analizador sintactico.

Sea posible usar los identificadores que aparezcan (resolución de nombres).

Para ello usaremos estructuras que nos permitan conocer la localizacion de los atributos que se usen en el programa.

Ser facilmente escalable, de modo que este preparado para convertirse, sin cambiar lo que esté ya hecho, en un compilador real.

Cualquier proyecto que se precie debe contemplar la posibilidad de futuras ampliaciones, y más aun cuando no es un proyecto finalizado completamente. Si terceros programadores, o yo mismo, en un futuro pretende abordar este problema de nuevo, que pudiese basarse en lo que este ya hecho para facilitarle el trabajo.

2. Justificacion

Este proyecto tiene como base una practica de la asignatura "Procesadores de Lenguajes 2". Antes de comenzar la practica, existia la posibilidad de que llegase a convertirse en un proyecto fin de carrera, por lo que tras hablar con el profesor que tutela mi proyecto, seguimos todos los pasos necesarios para realizar un avance seguro durante el desarrollo del trabajo, identificando previamente los requisitos necesarios para el proyecto y generando toda la documentacion necesaria.

3. Analisis de lo existente

El primer compilador nació allá por 1957, que permitia traducir el lenguaje FORTRAN a codigo maquina. Desde entonces, el estado actual de los compiladores es muy avanzado. Incorporan tecnicas de optimizacion con heurísticas muy avanzadas, fruto de años de investigacion. El proceso de compilacion con todas las optimizaciones activadas puede generar un codigo final practicamente ininteligible para cualquier persona, aunque muchisimo mas optimo. No obstante, aun queda mucho por investigar en este campo, sobre todo en el campo de la computacion paralela. Hoy en dia, cuando programamos para maquinas multiprocesadores es necesario incorporar al codigo demasiado 'overhead', es decir, codigo para indicar al compilador en que procesador ejecutar el proceso, que variables tiene que dejar en memoria (ya que son compartidas por varios procesos) y otros parametros que dificultan bastante la programacion paralela.

En cuanto al lenguaje, hoy en día no existe ningún lenguaje oficial en castellano. Implementar un compilador de este tipo puede tener interesantes fines académicos, con el fin de introducir a los novatos en el campo de la programación de una manera mucho más sencilla y cómoda, así como explicar los algoritmos en un lenguaje un poco más sencillo que se abstraiga de los detalles técnicos de los lenguajes actuales.

4. Propuesta detallada

4.1. Introducción

En este proyecto vamos a tratar los siguientes problemas:

1. Especificación de un lenguaje de programación, que va a ser el que interprete nuestro compilador. Será un lenguaje orientado a objetos, y en castellano, con la posibilidad de usar referencias adelantadas de identificadores (poder usar variables o métodos cuya declaración aun no se ha realizado cuando se ha usado).
2. Construir un analizador léxico para dicho lenguaje.
3. Construir un analizador sintáctico para dicho lenguaje.
4. Realizar la etapa de resolución de nombres sobre un código que lea de un archivo de texto. Notificará al usuario los posibles errores que encuentre durante el análisis del mismo.

4.2. Participantes en el Proyecto

Miguel Ángel Lobato Brenes.

Actualmente curso el 5º curso de Ingeniería Superior Informática por la Universidad de Sevilla. Yo me encargare del desarrollo completo del proyecto.

Don Francisco J. Galán Morillo, profesor titular de la Universidad de Sevilla, para el departamento de Lenguajes y Sistemas Informáticos, quien se encargara de tutelar el avance del proyecto durante todas las fases de su desarrollo.

4.3. Requisitos del Sistema

Requisitos funcionales:

Son las etapas ya anteriormente comentadas:

Análisis léxico, sintáctico y resolución de nombres.

Reportar al usuario los errores que encuentre durante el análisis del código.

Requisitos no funcionales del sistema

El sistema debe de correr tanto en entornos Linux como en entornos Windows. Dado que Mono aun no está plenamente desarrollado, se ha elegido Java como lenguaje para programar toda la aplicación. Java, al ser un lenguaje interpretado, permite que, instalando la correspondiente máquina virtual en el sistema, ejecutar la misma aplicación en distintos sistemas operativos. Otras alternativas son la plataforma .NET, que en parte nos permite mudar la aplicación a Linux usando la plataforma MONO. Este sistema aun no es muy estable ni está desarrollado al 100%. También podríamos haber usado python, pero no domino este lenguaje con la suficiente soltura como para iniciar un proyecto de tal envergadura bajo este lenguaje.

El sistema debe estar preparado para, por un lado continuar con el trabajo hasta conseguir un compilador completo, y por otro lado, ser facilmente adaptable a posibles cambios.

El sistema debe leer el codigo de entrada de un archivo de texto, cuyo nombre se le pasa como parámetro a la aplicacion.

5. Descripción del lenguaje L-2

En esta seccion vamos a describir detalladamente el lenguaje sobre el que va a trabajar nuestro compilador. L-2 es un lenguaje de programacion imperativo, en castellano, y orientado a objetos, con la posibilidad de incluir asertos. Usa una sintaxis similar a cualquier lenguaje de programacion orientado a objetos, por lo que cualquier usuario conocedor de los mismos, no tendra problemas en usar esta sintaxis.

Tipos en L-2

1. Tipos Predefinidos Simples

Entero

Valores: ..., -2, -1, 0, 1, 2 ...,

Funciones: +, -, *, /: entero, entero -> entero

<, >, <=, >=, =: entero, entero -> lógico

Prioridades (de mayor a menor): {*, /}, {+, -}, {<, >, <=, >=, =}

Real

Valores: ..., 1.2, -2.34, ...,

Funciones: +, -, *, /: real, real -> real

<, >, <=, >=, =: real, real -> lógico

Prioridades (de mayor a menor): {*, /}, {+, -}, {<, >, <=, >=, =}

Lógico

Valores: cierto, falso,

Funciones: y, o: lógico, lógico -> lógico

no: lógico -> lógico

Prioridades (de mayor a menor): no, {y, o}.

2. Clases. Las clases encapsulan datos y procedimientos. Se distinguen dos tipos de clases en el lenguaje L-2, clases instanciables (admiten creación de objetos) y clases no instanciables (no admiten creación de objetos).

Las clases instanciables se califican con el lexema inst y predefinen un procedimiento para crear objetos llamado crear(Nombre_clase_instanciable).

L-2 propone el lexema objeto para referirse un objeto a sí mismo.

Las clases no instanciables no se califican y no poseen la capacidad de crear objetos.

Toda clase en un programa es visible a las restantes clases de dicho programa.

No se admite definiciones anidadas de clases.

Ejemplo de clase instanciable:

```
inst clase C
{
entero cont; //atributo
iniciar(entero e) { //método
cont:=e;
}
incrementar(entero cantidad) { //método
cont := cont + cantidad;
}
decrementar(entero cantidad) { //método
cont := cont - cantidad;
}
consultar( ) dev entero { //método
dev cont;
}
}
```

Ejemplo de clase no instanciable:

```
clase D
{
p( ) {
C obj;
obj := crear(C);
obj.incrementar(2);
}
}
```

La creación de objetos se realiza a través de la llamada al procedimiento predefinido crear(Nombre_clase_instanciable) (el programador no programa dicho método, sólo lo usa).

No hay destrucción explícita de objetos en L-2. Se supone la existencia, en tiempo de ejecución, de un recolector de memoria para objetos inaccesibles.

Los atributos pueden ser de tipos predefinidos simples, compuestos o de tipo clase.

Los métodos son abstracciones procedimentales.

Los métodos en L-2 sólo usan argumento por valor.

Las identificaciones de los objetos quedan ocultas al programador permitiéndose sólo el uso del identificador nulo.

No se admiten definiciones anidadas de métodos.

3. Secuencias finitas. Se trata de un constructor de tipos anónimo (tipos sin nombre). Los elementos de la secuencia deben ser de tipos predefinidos simples o clases.

Secuencia(T) (T es un tipo predefinido simple o clase)

Valores: (opaco, es decir, no se pueden usar explícitamente en un programa)

Funciones: primero : Secuencia(T) -> T

resto: Secuencia(T) -> Secuencia(T)

vacía: Secuencia(T) -> Lógico

añadir_al_principio: Secuencia(T), T -> Secuencia(T)

añadir_al_final: Secuencia(T), T -> Secuencia(T)

Inferencia de Tipo en L-2

Toda expresión (con sintaxis correcta) en L-2 tiene un único tipo (tipado fuerte). L-2 no acepta coerciones de tipos.

Las reglas para inferir el tipo de una expresión en L-2 son:

- a. (Base) Si v es una variable/constante de tipo T entonces v es una expresión bien formada de tipo T .
- b. (Base) Si atr es un atributo de tipo T declarado en una clase C no instanciable entonces la expresión $C.atr$ es una expresión bien formada de tipo T .
- c. (Base) Si atr es un atributo de tipo T declarado en una clase C instanciable entonces la expresión $o.atr$ es una expresión bien formada de tipo T siendo o una instancia de C .
- d. (Inducción) Sean exp_1, \dots, exp_k un conjunto de expresiones bien formada de tipo T T_1, \dots, T_k respectivamente y f un método con dominio T_1, \dots, T_k y rango T declarado en una clase C no instanciable entonces $C.f(exp_1, \dots, exp_k)$ es una expresión bien formada de tipo T .

e. (Inducción) Sean exp_1, \dots, exp_k un conjunto de expresiones bien formada de tipo T T_1, \dots, T_k respectivamente y f un método con dominio T_1, \dots, T_k y rango T declarado en una clase C instanciable entonces $o.f(exp_1, \dots, exp_k)$ es una expresión bien formada de tipo T siendo o una instancia de C .

f. (Inducción) Sean exp_1, \dots, exp_k un conjunto de expresiones bien formada de tipo T_1, \dots, T_k respectivamente y f una función definida en un tipo predefinido de L-2 con dominio T_1, \dots, T_k y rango T entonces la expresión $f(exp_1, \dots, exp_k)$ es una expresión bien formada de tipo T .

La inferencia (o asignación) de tipo para una expresión L-2 se realiza en tiempo de compilación (tipado estático).

Instrucciones

En esta sección presentamos el conjunto de instrucciones básicas de L-2.

Asignación

Elemento de programación básico. Presenta la forma general $x := e$, donde x es una variable o atributo de un objeto y e es una expresión del mismo tipo que x .

Dado que L-2 define sólo argumentos por valor, éstos no pueden aparecer en la parte izquierda de las asignaciones.

Ejemplo:

```
cont := cont - cantidad; (siendo cont y cantidad dos variables de tipo entero)
```

Devolución de valores

Todo método f con comportamiento funcional devuelve valores mediante la instrucción `dev e`, donde e es una expresión coincidente en tipo con el rango de f .

Ejemplo:

```
dev cantidad; (cantidad es una variable local de tipo entero)
```

Activación de método

Existen dos localizaciones desde la que se puede activar un método en L-2:

1. Activación desde un objeto (patrón `objeto.método(parametros reales)`)
2. Activación desde una clase no instanciable (patrón `clase.método(parametros reales)`).

Ejemplo de activación desde un objeto:

```
obj.incrementar(2);
```

Ejemplo de activación desde una clase no instanciable:

D.p();

Instrucción de Salida

L-1 posee el procedimiento predefinido escribir(expresión) para escribir en la salida estándar un valor entero, real o lógico.

Ejemplo:

```
escribir(cont);
```

Condicional

Presenta la forma si (condicion) entonces ... [sino ...] finsi.

Ejemplo:

```
si (cont >= cant) entonces
```

```
cont := cont - cantidad;
```

```
finsi
```

Iteración

Presenta la forma mientras (condicion) hacer ... finmientras.

Ejemplo:

```
mientras (cont < 100) hacer
```

```
cont := cont - 1;
```

```
finmientras
```

Ámbitos y Resolución de nombres.

El significado de cualquier programa depende, en gran medida, del significado de los nombres presentes en él. La declaración es el mecanismo que permite vincular significado a un nombre en un programa. Formalmente, la declaración es un par (N, def(N)) donde N es un nombre y def(N) la definición según la cual se interpreta N.

Sin embargo, es teóricamente posible usar nombre iguales para referentes distintos provocando ambigüedad en la interpretación de los nombres implicados. Por ejemplo, supongamos que x es una variable local declarada en un método de una clase C que posee además un atributo x. La ocurrencia de x en el cuerpo del método podría interpretarse como variable local o como atributo (ambigüedad).

Para solucionar estos problemas se asocia el concepto de ámbito o contexto de interpretación a

distintos elementos de programación (programa, clase y método) con la intención de que en un ámbito dado, distintas ocurrencias de un mismo nombre se interpreten de manera única.

Concepto de Programa

Desde un punto de vista sintáctico, el programa L-2 está formado por un conjunto de clases.

Destaca una clase no instanciable llamada Programa con un único método llamado inicio() desde el que se inicia la ejecución del programa.

```
programa p
```

```
inst clase Elemento
```

```
{
```

```
oculto entero e;
```

```
consultar() dev entero
```

```
{
```

```
dev e;
```

```
}
```

```
modificar(entero n)
```

```
{
```

```
objeto.e:=n;
```

```
}
```

```
}
```

```
inst clase Pila
```

```
{
```

```
oculto secuencia entero almacen;
```

```
apilar(Elemento elem)
```

```
{
```

```
anadir_al_principio(almacen,elem);
```

```
}
```

```
desapilar() dev Elemento
```

```
{  
Elemento e;  
e := primero(almacen);  
almacen := resto(almacen);  
dev e;  
}
```

cima() dev Elemento

```
{  
dev primero(almacen);  
}
```

clase Programa

```
{  
inicio()  
{  
Elemento e;  
Pila p;  
entero i;  
i:=1;  
mientras (i<=10) hacer  
e.modificar(i);  
p.apilar(e);  
i:=i+1;  
finmientras  
}
```

Desde el punto de vista físico, todo programa se localiza en un fichero con el mismo nombre del programa y extensión cfg

Por ejemplo, el programa anterior se ubica en un fichero llamado (p.cfg)

Asertos

Un aserto es una expresión lógica localizada en un punto específico de la zona de instrucciones de un programa en forma de anotación. Las aserciones pueden evaluarse en tiempo de ejecución. Cuando el flujo de control del programa alcanza un aserto, éste es evaluado y si el

resultado de la evaluación es igual a falso se emite un informe indicando dicha situación.

Para entender el concepto de aserto es pertinente definir el concepto de estado de un programa. Podemos distinguir dos tipos de variables en L-2: variables objeto y variables dato. Las primeras almacenan referencias a objetos y las segundas almacenan valores pertenecientes a algún tipo predefinido del lenguaje L-2. El estado de una variable objeto queda determinado por el valor de los atributos del objeto y el estado de una variable dato queda determinado por el dato almacenado. Se define el estado de un programa P en tiempo de ejecución en un punto p de su código al estado de las variables de P visibles en p.

Un aserto ass localizado en un punto p del programa P define, en tiempo de ejecución, el conjunto de testados aceptables para P en el punto p. Todo estado de P que haga cierta la evaluación de ass se dice ser un estado aceptable para P en p. Si la evaluación fuera falsa entonces se dice que P viola la aserción ass (lo que equivale a decir que se ha detectado en p un error de programación).

Es importante destacar que la evaluación de un aserto nunca debe cambiar el estado del programa. Se dice que un método tiene comportamiento funcional puro si su ejecución no altera el estado del programa (p.e. método consultar en clase C). Por lo tanto, sólo los métodos funcionales puros pueden aparecer en los asertos.

El lenguaje de asertos de L-2 se define de la siguiente forma:

- (a) cierto es un aserto.
- (b) falso es un aserto.
- (c) la llamada a un método funcional puro que devuelve un tipo lógico es un aserto.
- (d) todo(elemento, secuencia, expresión lógica) es un aserto.
- (e) existe(elemento, secuencia, expresión lógica) es un aserto.
- (f) Si F y G son asertos entonces también lo son F equivale G, F implica G, F y G, F o G.
- (g) Si F es un aserto entonces también lo es no F.