Trabajo Práctico Integrador

Compiladores 2019

***Alumnos*:**

- Díaz, Pablo 67170
- Enríquez, Facundo 65804
- Selva, Ricardo 65363

Profesor:

- Ruidías, Héctor Javier

Informe

Resumen. En este documento se expondrá el proceso de desarrollo de un compilador que genera código PHP. Se describirán las herramientas Flex y Bison para lograr tal desarrollo, las etapas generales de la compilación, los componentes léxicos definidos y la gramática utilizada. también se desarrollarán programas escritos en lenguaje fuente para ejemplificar algunas funcionalidades.

Introducción

Un compilador es un programa informático que se encarga de traducir (compilar) el código fuente de cualquier aplicación que se esté desarrollando. En decir, es un software que se encarga de traducir el programa hecho en lenguaje de programación, a un lenguaje de máquina que pueda ser comprendido por el equipo y pueda ser procesado o ejecutado por este.

Objetivo

El objetivo del trabajo es desarrollar un compilador que traduce un lenguaje fuente en pseudocódigo (definido por el grupo) a lenguaje PHP . El código generado tendrá que permitir el uso de estructuras de control, expresiones, constantes, variables, funciones y arreglos.
Para comprobar el correcto funcionamiento del compilador se utilizarán como casos de prueba varios programas escritos en el lenguaje fuente.

Etapas de un compilador

La construcción de un compilador involucra la división del proceso en fases que varía de acuerdo a su complejidad. Generalmente estas fases se agrupan en dos tareas: el **análisis** del programa fuente y la **síntesis** del programa objeto.

- *Análisis*: En esta etapa se controla que el texto fuente sea correcto en todos los sentidos, se generan las estructuras necesarias para la generación de código y se crea una representación intermedia mediante un árbol sintáctico.
- *Síntesis*: En esta etapa el compilador ya se encuentra en disposición de generar el código máquina equivalente semánticamente al programa fuente. Para ello se parte de las estructuras generadas en dicha etapa anterior: árbol sintáctico y tabla de símbolos.

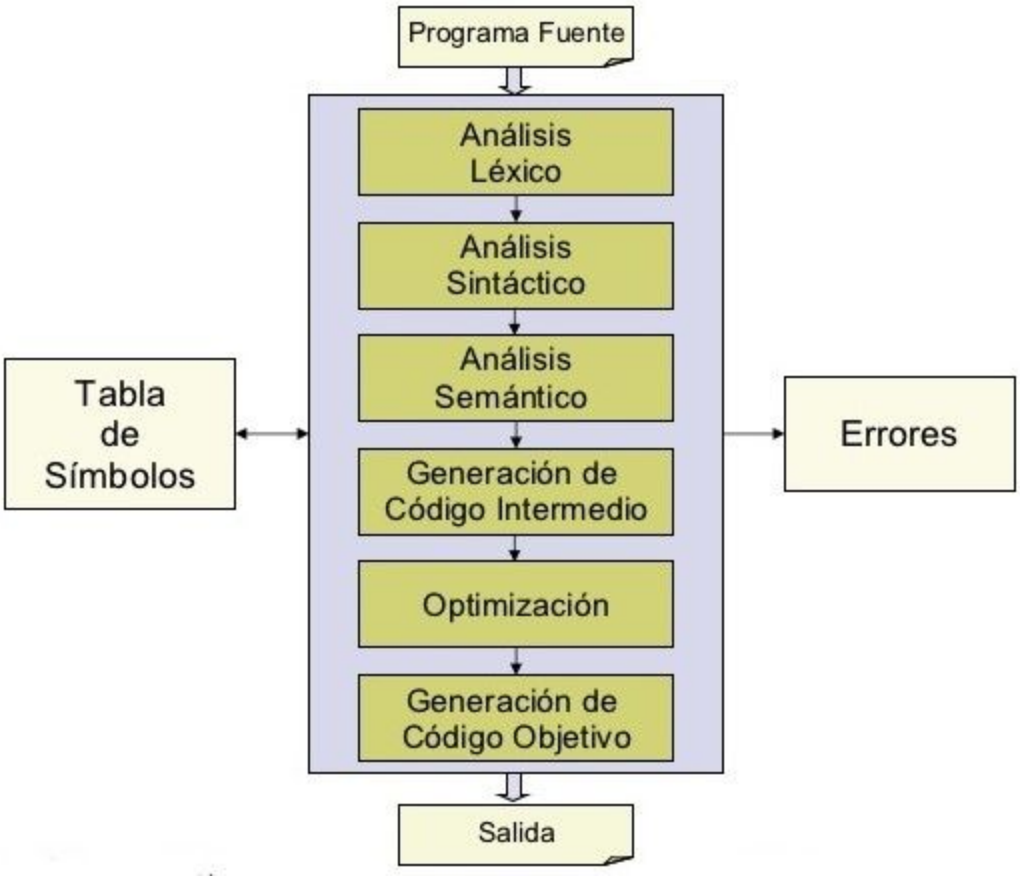

Fig 1. Fases de un compilador [1]

Las fases que se muestran en la figura 1, se pueden agrupar en una *etapa inicial* y una *etapa final*:

- ***Etapa inicial*** *(Front-End)*: abarca las fases que dependen del lenguaje fuente y son independientes de la máquina objeto. (análisis léxicos, sintácticos y semántico, tabla de símbolos, y generación de código intermedio y el manejo de errores de cada fase.
- ***Etapa final*** *(Back-End)*: abarca las fases que dependen de la máquina objeto. abarca aspectos de optimización y generación de código, manejo de errores y las operaciones con la tabla de símbolos.

3.1 Fases de un compilador

1. ***Análisis léxico*** *(scanner):* Lee la secuencia de caracteres de izquierda a derecha del

programa fuente y agrupa las secuencias de caracteres en unidades con significado propio (componentes léxicos o “tokens” en inglés). Los tokens pueden ser constantes, identificadores, palabras reservadas, etc.

2. ***Análisis sintáctico*** *(parser)*: Determina si la secuencia de componentes léxicos sigue la sintaxis del lenguaje y obtiene la estructura jerárquica del programa en forma de árbol, donde los nodos son las construcciones de alto nivel del lenguaje.
3. ***Análisis semántico***: Realiza las comprobaciones necesarias sobre el árbol sintáctico para determinar el correcto significado del programa.Las tareas básicas a realizar son: La verificación e inferencia de tipos en asignaciones y expresiones, la declaración del tipo de variables y funciones antes de su uso, el correcto uso de operadores, el ámbito de las variables y la correcta llamada a funciones.
4. ***Generación de código intermedio***: Se traduce la representación del árbol en una secuencia ordenada de instrucciones de un lenguaje abstracto (corresponde a una máquina abstracta, no es un lenguaje de programación). Se debe definir lo más general posible para traducirlo a cualquier máquina real.
5. ***Optimización de código intermedio***: Se mejora el código intermedio para que sea más rápido de ejecutar, y así reducir la cantidad de recursos. Algunas optimizaciones pueden consistir en evaluación de expresiones constantes, el uso de la propiedad asociativa o conmutativa de algunos operadores, reducción de expresiones comunes, etc.
6. ***Generación de código***: Toma como entrada la representación intermedia y genera el código objeto. La optimización depende de la máquina, es necesario conocer el conjunto de instrucciones, la representación de los datos (número de bytes), modos de direccionamiento, número y propósito de registros, jerarquía de memoria, encauzamientos, etc.

Fases auxiliares. Interactúan con las seis fases mencionadas anteriormente durante el proceso de compilación:

1. *Manejo de errores:* los errores encontrados en las fases de análisis se envían a un módulo de manejo de errores. Símbolos incorrectos, violación de reglas de estructura, operaciones significativas, etc.
2. *Tabla de símbolos*: es una estructura de datos que contiene toda la información relativa a cada identificador que aparece en el programa fuente. Cada elemento de la tabla se compone de al menos del identificador y sus atributos.

Los atributos son informaciones relativas a cada identificador, necesarias para realizar el análisis semántico o para la traducción. Cualquiera de los tres analizadores puede rellenar algún atributo, pero el nombre del identificador es misión del analizador léxico.

Herramientas Flex y Bison

Flex es una herramienta para generar escáneres: programas que reconocen patrones léxicos en un texto. flex lee los ficheros de entrada dados, o la entrada estándar si no se le ha indicado ningún nombre de fichero, con la descripción de un escáner a generar. La descripción se encuentra en forma de parejas de expresiones regulares y código C, denominadas reglas.

flex genera como salida un fichero fuente en C, `lex.yy.c', que define una rutina `yylex()'. Este fichero se compila y se enlaza con la librería `-lfl' para producir un ejecutable. Cuando se arranca el fichero ejecutable, este analiza su entrada en busca de casos de las expresiones regulares. Siempre que encuentra uno, ejecuta el código C correspondiente.

GNU bison es un programa generador de analizadores sintácticos de propósito general perteneciente al proyecto GNU disponible para prácticamente todos los sistemas operativos.

Bison convierte la descripción formal de un lenguaje, escrita como una gramática libre de contexto LALR, en un programa en C, C++, o Java que realiza análisis sintáctico. Es utilizado para crear analizadores para muchos lenguajes, desde simples calculadoras hasta lenguajes complejos. Para utilizar Bison, es necesaria experiencia con el la sintaxis usada para describir gramáticas.

Un fuente de Bison (normalmente un fichero con extensión .y) describe una gramática. El ejecutable que se genera indica si un fichero de entrada dado pertenece o no al lenguaje generado por esa gramática.

Desarrollo

Como se mencionó en la introducción, el trabajo consistió en la generación de código en lenguaje PHP a partir de un pseudocódigo modificado por el grupo utilizando las herramientas Flex y Bison en un framework que ayuda a la generación de nodos y el mantenimiento del arbol sintactico. Para llevar a cabo esta tarea se realizaron dos actividades: construir un *analizador léxico* a partir del cual se define la tabla de símbolos o componentes léxicos, y construir un *analizador sintáctico* mediante el cual se define la estructura gramatical del lenguaje. Al analizador sitactico se le agrego la tarea de controlar algunos aspectos de la semantica.

Construcción del analizador léxico

La especificación léxica se codificó en archivo '*lexer.l*'. El objetivo fue especificar un conjunto de patrones definidos mediante expresiones regulares junto a acciones asociadas a dichos patrones.

Los patrones especifican los componentes léxicos o tokens que se definieron como unidades indivisibles que pueden ser reconocidas en el lenguaje fuente. Los tokens que se definieron son constantes, identificadores, palabras reservadas, operadores, delimitadores, entre otros.

A medida que se definían los tokens se fue especificando la gramática que define el lenguaje o conjunto de componentes léxicos que son derivados a partir del axioma.

El analizador léxico reconoce las palabras en función de dicha gramática, donde los componentes léxicos se corresponden con sus terminales. De esta forma se logra una consistencia con el analizador sintáctico, quien solicita dichos tokens durante el proceso de compilación.

Los tokens son introducidos en una tabla de símbolos que es accedida y modificada por fases posteriores.

A continuación, se especifican los componentes léxicos de la gramática:

Componente Léxico	Patrón	Lexema
`T_COMMENT_LINE`	`\!\!.*\n`	**cualquier cadena que empiece con dos signos de admiración y no contenga saltos de línea**
`T_COMMENT_BLOCK`	`\!\+(.|\n)*\+\!`	**Cualquier cadena que empiece con un signo de admiración y un signo más, y termine con un signo más y un signo de admiración**

T_STRING	(\'.*\'|\".*\")	Cualquier cadena de texto entre comillas simples o dobles
T_INTEGER	\-?[0-9]+	Cualquier número entero
T_NUMBER	\-?[0-9]*\.[0-9]+	Cualquier número con decimales
T_IF	si	si
T_ELSE	sino	sino
T_WHILE	mientras	mientras
T_FOR	para	para
T_FROM	desde	desde
T_TO	hasta	hasta
T_ECHO	mostrar	mostrar
T_INPUT	pedir	pedir
T_LOAD_ARRAY	cargardesdearchivo	cargardesdearchivo
T_FOREACH	cada	cada
T_IN	en	en
T_EXIT	terminar	terminar
T_FUNCTION	funcion	funcion
T_RETURN	retornar	retornar

T_LLA_I	\{	{
T_LLA_D	\}	}
T_PAR_I	\(	(
T_PAR_D	\)	)
T_BRA_I	\[	[
T_BRA_D	\]	]
T_COMMA	,	,
T_OP_ASSIGN	\<=	<=
T_OP_ACUMULATE_PLUS	\<\+	<+
T_OP_ACUMULATE_MINUS	\<\-	<-
T_OP_CONCAT	\>\<	><
T_OP_EQUAL	igual	igual
T_OP_DISTINCT	distinto	distinto
T_OP_LESSER	menor	menor
T_OP_GREATER	mayor	mayor
T_OP_LESSER_EQ	menoroigual	menoroigual
T_OP_GREATER_EQ	mayoroigual	mayoroigual

T_OP_AND	y	y
T_OP_OR	o	o
T_AOP_PLUS	\+	+
T_AOP_MINUS	\-	-
T_AOP_MUL	*	*
T_AOP_DIV	\/	/
T_AOP_POW	potencia	potencia
T_SEMICOLON	;	;
T_ID	[a-zA-Z][a-zA-Z0-9]*	**Cualquier identificador que empiece con un carácter alfabético, seguido de cualquier cantidad de caracteres alfanuméricos**

Construcción del analizador sintáctico

La especificación sintáctica se codificó el archivo '*parser.y*'. El objetivo fue especificar reglas de producción para definir la gramática o estructura sintáctica del programa. A partir de esta gramática se define un árbol de análisis sintáctico requerido por etapas posteriores.

Por lo tanto, en esta fase el analizador o parser generado por Bison verifica el texto de entrada en función de una gramática dada, y en caso de que el programa de entrada sea válido, genera el árbol sintáctico que lo reconoce y convierte las descripciones gramaticales en un programa en PHP.

Algunas reglas están ligadas a comprobaciones semánticas, como comprobar si existe una función antes de llamarla o declararla.

La gramática definida está compuesta por un lado izquierdo y un lado derecho. En el lado izquierdo hay componentes auxiliares (no terminales) y del lado derecho hay como mínimo una operación compuesta por tokens (terminales) y/o auxiliares.

Aspectos importantes del lenguaje

El lenguaje es debilmente tipado, para realizar la gramática se tuvieron en cuenta varios aspectos o reglas respecto a su implementación para evitar errores y ambigüedades en la misma:

- Palabras reservadas del lenguaje:

Palabras como **si**, **sino**, **mientras**, etc que se utilizan en las estructuras de control y funciones del lenguaje

- Los comentarios pueden ser de una linea, empezando por **!!**, o bien de bloque, rodeados por los signos **!+** Y **+!**

- Variables:

El *identificador* es el nombre de la variable, debe comenzar con una letra minúscula y le siguen caracteres alfabéticos o números. Para arreglos se expresa en indice entre corhetes

Ejemplos: x, contador, parametro1.

- *Sentencias*:

Permite la escritura de las instrucciones del programa. Cada sentencia debe terminar con un punto y coma.

- Operadores:
  - Asignacion : <=
  - Acumular mas: <+
  - Acumular menos: <-
  - Logicos: igual distinto mayor menor mayoroigual menoroigual
  - Matematicos : + - * /

- Estructuras condicionales:
  - si(condicion){bloque} sino {bloque}

- Bucles:
  - mientras(condicion){bloque}
  - para(variable “desde” numero o variable “hasta” numero o variable){bloque}
  - cada(elemento “en” iterable){bloque}

- Funciones:

- pedir(variable): Solicita ingreso por entrada estandar de texto y lo guarda en la variable
- mostrar(variable): muestra por salida estandar una variable o constante
- cargardesdearchivo(arreglo, ruta): carga en un arreglo las lineas de un archivo de texto

❖ Funciones de usuario:

- funcion nombre(parametros,con,comas){bloque} declara una funcion, se verifica que la funcion no exista
- nombre(argumentos); llama a una funcion, la funcion debe existir.

5.2.2 Gramática utilizada (Backus-Naur Form)

```
< sigma > ::= < lineas >

< lineas > ::= < linea > < lineas >
            | ""

< linea > ::= < if >
            | < while >
            | < for >
            | < foreach >
            | < operation > T_SEMICOLON
            | < loadArray > T_SEMICOLON
            | < exit > T_SEMICOLON
            | < function >_< decl >
            | < function >_< call > T_SEMICOLON
            | < return > T_SEMICOLON
            | < comment >

< if > ::= T_IF < cond > < then > < else >

< cond > ::= T_PAR_I < operation > T_PAR_D

< operation > ::= < operand > < operator > < operand >
            | < echo >
            | < input >
            | < id >

< operand > ::= < id >
            | T_STRING
            | T_NUMBER
            | T_INTEGER
            | < cond >
            | < function_call >

< operator > ::= T_OP_EQUAL
            | T_OP_DISTINCT
            | T_OP_LESSER
            | T_OP_GREATER
            | T_OP_LESSER_EQ
```

```
            | T_OP_GREATER_EQ
            | T_OP_AND
            | T_OP_OR
            | T_OP_ASSIGN
            | T_OP_ACUMULATE_PLUS
            | T_OP_ACUMULATE_MINUS
            | T_OP_CONCAT
            | T_AOP_PLUS
            | T_AOP_MINUS
            | T_AOP_MUL
            | T_AOP_DIV

< then > ::= T_LLA_I < lineas > T_LLA_D

< else > ::= T_ELSE T_LLA_I < lineas > T_LLA_D
            | ""

< while > ::= T_WHILE < cond > < then >

< for > ::= T_FOR T_PAR_I < id > T_FROM < operand > T_TO < operand > T_PAR_D < then >

< foreach > ::= T_FOREACH T_PAR_I T_ID T_IN < id > T_PAR_D < then >

< echo > ::= T_ECHO T_PAR_I < operand > T_PAR_D

< input > ::= T_INPUT T_PAR_I < id > T_PAR_D

< id > ::= T_ID T_BRA_I < operand > T_BRA_D
            | T_ID

< loadArray > ::= T_LOAD_ARRAY T_PAR_I < id > T_COMMA < operand > T_PAR_D

< function >_< decl > ::= T_FUNCTION T_ID T_PAR_I < vars > T_PAR_D < then >

< comment > ::= T_COMMENT_LINE
            | T_COMMENT_BLOCK

< vars > ::= T_ID T_COMMA < vars >
            | T_ID
            | ""

< function >_< call > ::= T_ID T_PAR_I < params > T_PAR_D

< params > ::= < id > T_COMMA < vars >
            | < id >
            | ""

< exit > ::= T_EXIT T_PAR_I T_INTEGER T_PAR_D

< return > ::= T_RETURN T_PAR_I < operand > T_PAR_D
```

Códigos de ejemplo

A continuación, se presentan varios escenarios de prueba para ejemplificar algunas funcionalidades previstas por el lenguaje.

Los escenarios incluyen operaciones básicas como declaración y asignación de variables, comparación de variables, utilización de operadores, lectura e

impresión en pantalla, lectura de archivos, etc.

Ejemplo 1. En este ejemplo se calcula el factorial de un numero, sirve para probar condiciones y bucles, también se puede apreciar la concatenación.

```
mostrar("Ingrese un numero para calcular su factorial\n");
pedir(num);
si(num menoroigual 0) {
   mostrar("El factorial de un numero negativo no existe!");
   terminar(0);
}

factorial <= num;
num <- 1;
mientras(num mayor 0) {
   factorial <= (factorial * num);
   num <- 1;
}

mostrar(("El factorial es igual a " >< factorial));
```

Ejemplo 2. En este ejemplo se comprueban las condiciones complejas, también el sino y el uso de arreglos.

```
i <= 0;
nota <= 15;

mientras(nota mayor 0) {
   mostrar("Ingrese una nota del 1 al 10, para terminar ingrese 0\n");
   pedir(nota);
   si((nota mayor 0) y (nota menoroigual 10)) {
      notas[i] <= nota;
      i <+ 1;
   }sino{
      mostrar("NUMERO NO VALIDO!\n");
   }
}
acu <= 0;
cada(nota en notas)   {
   acu <+ nota;
}

mostrar(("El promedio es " >< (acu / i)));
```

Ejemplo 3. Este ejemplo posee una funcionalidad similar al anterior, pero en este caso se lee los datos desde un archivo, el usuario ingresa la ruta al archivo. Otro aspecto importante del ejemplo es el uso de funciones.

```
funcion promedio(notas) {
   acu <= 0;
   i <= 0;
   cada(nota en notas)    {
      acu <+ nota;
      i <+ 1;
   }

   mostrar(("El promedio es " >< (acu / i)));
}

mostrar("Ingrese la ruta al archivo de notas
(compilado/ejemplo.txt)\n");
pedir(ruta);

cargardesdearchivo(notas,ruta);

promedio(notas);
```

Compilación y ejecución de los ejemplos

Para ejecutar los ejemplos se puede correr el archivo run.sh en la raíz del proyecto, este script comprueba si se encuentran instaladas las dependencias, en caso de no estarlo, se ofrece instalarlas, en caso de querer instalar manualmente se muestra el comando a ejecutar por consola.

Una vez instaladas las dependencias, se compila el compilador. Por ultimo se pasa a un menu con las opciones de ejecutar los ejemplos y también una pequeña ayuda para ejecutar manualmente.

Para ejecutar manualmente basta con ejecutar el archivo comp, que se encuentra en la carpeta bin, como primer parametro se pasa la ubicacion del archivo fuente, y como segundo parametro el archivo destino. En caso de no ofrecer un archivo destino, la salida se imprime por pantalla.

Conclusiones

Como conclusión se entiende la dificultad de desarrollar un compilador, se comprendieron las fases de la compilación y los problemas que pueden aparecer al desarrollar un lenguaje. Varias veces se optó por modificar la sintaxis en favor de simplificar los procesos de traducción.

Referencias y bibliografía

[1]. Figura extraída del libro “Compiladores: Principios, Técnicas y Herramientas”, Aho, Sethi, Ullman.

Bibliografía de consulta:

• GNU Bison https://es.wikipedia.org/wiki/GNU_Bison
• Material teórico de cátedra de Compiladores y Autómatas.
• Libro “Compiladores: Principios, Técnicas y Herramientas”, Aho, Sethi, Ullman.
• Manual de Flex 2.5 ftp://ftp.gnu.org/old-gnu/Manuals/flex-2.5.4/html_mono/flex.html