Lenguajes de programación y modelos de computación

Asignatura: Análisis Comparativo de Lenguajes
Responsable: Ariel Gonzalez
e-mail: agonzalez@dc.exa.unrc.edu.ar

Departamento de Computación
Facultad de Ciencias Exactas, Físico-Químicas y Naturales
Universidad Nacional de Río Cuarto - Argentina

2017

Abstract

Este libro es el resultado del dictado del curso *Análisis Comparativo de Lenguajes* para alumnos de pregrado en la Universidad Nacional de Río Cuarto.

Si bien existe una vasta bibliografía en el tema, es difícil encontrar un único libro que cubriese todos los temas y con el enfoque que es buscado en la asignatura.

Los principales objetivos de este trabajo es recopilar contenidos de varias fuentes bibliográficas y compilarlas desde un enfoque de las características de los lenguajes de programación a partir de un punto de vista de modelos de computación y paradigmas (o estilos) de programación, desarrollando los conceptos relevantes de los lenguajes de programación.

En cada capítulo se desarrollan los conceptos a partir de un lenguaje de programación básico, para luego compararlo con las construcciones similares encontradas en algunos lenguajes de programación seleccionados.

Los lenguajes de programación se han seleccionado por su difusión en la industria y por su importancia desde el punto de vista académico, los cuales se analizan en base a los conceptos básicos estudiados.

El enfoque es centrado en la elección de un lenguaje núcleo, para el cual se define su sintaxis y semántica (en base a su máquina abstracta correspondiente). El mismo, es extendido con adornos sintácticos y otras construcciones básicas en función de las características a analizar. La semántica formal permite realizar análisis de correctitud y su complejidad computacional.

Este material está dirigido a alumnos de segundo o tercer año de carreras de ciencias de la computación o ingeniería de software. Sus contenidos permiten desarrollar un curso en cuatro meses de duración con prácticas de aula y talleres. Al final de cada capítulo se proponen ejercicios correspondientes a cada tema.

Los paradigmas estudiados implican el modelo **imperativo**, **funcional**, **orientado a objetos**, **lógico** y el **concurrente**. Este último modelo es transversal a los demás modelos, por lo que se hace un análisis y consideraciones en cada contexto en particular.

El lenguaje *kernel* seleccionado es **Oz**, el cual es un lenguaje académico desarrollado específicamente para el estudio de los diferentes modelos de computación.

Contents

Capítulo 1

Introducción

Los lenguajes de programación son la herramienta de programación fundamental de los desarrolladores de software. Desde los comienzos de la computación, la programación fue evolucionando desde la simple configuración de interruptores, pasando por los primeros lenguajes **assembly**, los cuales permitan escribir las instrucciones de máquina en forma simbólica y la definición de *macros*, hasta llegar a los lenguajes de programación de alto nivel que permiten abstraer al programador de los detalles de la arquitectura y el desarrollo de programas *portables* entre diferentes sistemas de computación[1].

El objetivo de este material es estudiar los conceptos y principios que encontramos en los lenguajes de programación modernos.

Es importante conocer un poco la historia y la evolución de algunos conceptos para poder entender algunas características de algunos lenguajes.

En la actualidad se encuentran catalogados mas de 1500 lenguajes de programación, por lo cual una currícula en ciencias de la computación o de desarrollo de software no puede enfocarse en base al dictado de cursos sobre lenguajes concretos, sino que es necesario que se estudien lenguajes de programación desde el punto de vista de los diferentes modelos o estilos de computación en los cuales se basan.

Estos modelos o estilos permiten clasificar a los lenguajes de programación en familias que generalmente se conocen como *paradigmas*.

El estudio de los lenguajes en base al análisis de cada paradigma permite generalizar conceptos utilizados en grupos de lenguajes mas que en lenguajes particulares.

El enfoque utilizado permite realizar análisis de los conceptos utilizados en todos los lenguajes de programación existentes, permitiendo realizar comparaciones entre lenguajes o familias.

El estudio de los conceptos y principios generales, en lugar de estudiar la sintaxis de lenguajes específicos, permite que el desarrollador pueda estudiar y aprender por sí

[1] Un sistema de computación comprende el *hardware* y el software de base, es decir, sistema operativo, enlazador, compiladores, editores, etc.

mismo, a utilizar correctamente las facilidades provistas por un nuevo lenguaje (o uno desconocido).

Los paradigmas estudiados comprenden el *declarativo*, dentro del cual podemos encontrar el *funcional* y el *lógico*, el *imperativo*, en el cual podemos encontrar una gran cantidad de lenguajes ampliamente utilizados como Pascal, C, Basic, Ada, FORTRAN, COBOL, etc., con sus evoluciones en la *programación orientada a objetos (POO)* y los lenguajes basados en componentes.

Los conceptos y principios de la *concurrencia* son aplicables a todos los demás paradigmas por lo que se estudia como un paradigma en particular analizándose su aplicación en cada modelo de computación en particular.

1.1 Lenguajes como herramientas de programación

Un lenguaje de programación permite al programador definir y usar *abstracciones*. El desarrollo de software se basa fundamentalmente en la utilización de los lenguajes de programación y los procesadores de lenguajes (compiladores, intérpretes y linkers).

Las demás herramientas son auxiliares (como los editores, entornos integrados de desarrollo, generadores de Código, etc.) y su objetivo es sólo hacer más cómoda, automatizable y rápida la tarea de producción de código.

Los métodos de desarrollo de software, los cuales incluyen lenguajes textuales o iconográficos, están basados en los mismos conceptos adoptados en los lenguajes de programación[2].

La afirmación anterior es fácilmente verificable ya que cualquier método de desarrollo deberá permitir la generación de código al menos para algún lenguaje de programación.

1.2 Abstracciones

En la sección anterior se afirma que un lenguaje de programación brinda mecanismos para la definición y utilización de abstracciones.

Estas abstracciones permiten que el programador tome distancia de las características de bajo nivel del hardware para resolver problemas de una manera mas *modular*, y contribuir así a un fácil *mantenimiento* a través de su vida útil.

Aceptando esta definición de lo que es un lenguaje de programación, es mas comprensible que los diseñadores de software a gran escala, generalmente son personas con amplios conocimientos sobre lenguajes (y su implementación), y muestra que es imposible que un (buen) diseñador de software no haya pasado por una etapa de verdadero desarrollo de software, es decir, la escritura de programas concretos en algún lenguaje de programación que incorpore conceptos modernos como abstracciones de

[2] En realidad las características que encontramos en los métodos de desarrollo se pueden encontrar en lenguajes de programación desarrollados con bastante anterioridad.

alto nivel.
Esto nos permite definir el término programación.

Definición 1.2.1 *La programación es la actividad que consiste en definir y usar abstracciones para resolver problemas algorítmicamente.*

Es importante comprender así a la programación, ya que esto muestra el porqué los mejores programadores o diseñadores son aquellos que tienen una buena base en contenidos, en los cuales el concepto de abstracción es indispensable en algunas áreas como la matemática, la lógica y el álgebra.

Un lenguaje de programación generalmente sugiere uno o más *estilos* de programación, por lo que su estudio permite su mejor aprovechamiento en el proceso de desarrollo de sotfware.

1.2.1 Abstracción procedural

Una abstracción procedural permite encapsular en una unidad sintáctica una computación parametrizada.

Es bien conocida la estrategia de solución de problemas conocido como *divide and conquer (divide y vencerás)*, la cual se basa en la descomposición del problema en un conjunto de subproblemas mas simples y una forma de composición de esos subproblemas para obtener la solución final.

La abstracción procedural es la base de la implementación de esta estrategia de resolución de problemas. A modo de ejemplo, la programación funcional se caracteriza por la definición de *funciones* y la composición funcional. En cambio la programación imperativa se caracteriza por definir la evolución de los estados de un sistema basándose en la composición secuencial y en operaciones de cambios de estado (asignación).

1.2.2 Abstracción de datos

Generalmente los programas operan sobre ciertos conjuntos de datos. Es bien conocido que los cambios mas frecuentes producidos en un sistema son los de representación de los datos que se manipulan. Por este motivo es importante poder *ocultar* los detalles de la representación (o implementación) de los datos para facilitar el mantenimiento y la utilización de subprogramas.

Los *tipos abstractos de datos (ADTs)* permiten definir tipos de datos cuyos valores están implícitos o denotados por sus operaciones. Es deseable que los lenguajes de programación permitan la especificación o implementación de ADTs ocultando los detalles de representación.

Es sabido que no todos los lenguajes lo permiten, pero las tendencias actuales han avanzado respecto a las capacidades de modularización y ocultamiento de información, otorgando un mayor control en el encapsulamiento de los componentes de las abstracciones.

1.3 Evaluación de un lenguaje de programación

Un lenguaje de programación puede ser evaluado desde diferentes puntos de vista. En particular, un lenguaje debería tener las siguientes propiedades:

- **Universal**: cada problema *computable* debería ser expresable en el lenguaje.

  Esto deja claro que en el contexto de este libro, a modo de ejemplo, un lenguaje como SQL[3] no es considerado un lenguaje de programación.

- **Natural**: con su dominio de su aplicación.

  Por ejemplo, un lenguaje orientado al procesamiento vectorial debería ser rico en tipos de datos de vectores, matrices y sus operaciones relacionadas.

- **Implementable**: debería ser posible escribir un intérprete o un compilador en algún sistema de computación.

- **Eficiente**: cada característica del lenguaje debería poder implementarse utilizando la menor cantidad de recursos posibles, tanto en espacio (memoria) y número de computaciones (tiempo).

- **Simple**: en cuanto a la cantidad de conceptos en los cuales se basa. A modo de ejemplo, lenguajes como PLI y ADA han recibido muchas críticas por su falta de simplicidad.

- **Uniforme**: los conceptos básicos deberían aplicarse en forma consistente en el lenguaje. Como un contraejemplo, en C el símbolo * se utiliza tanto para las declaraciones de punteros como para los operadores de *referenciación* y multiplicación, lo que a menudo confunde y da lugar a la escritura de programas difíciles de entender.

- **Legible**: Los programas deberían ser fáciles de entender. Una crítica a los lenguajes derivados de C es que son fácilmente confundible los operadores == y =.

- **Seguro**: Los errores deberían ser detectables, preferentemente en forma estática (en tiempo de compilación).

Los lenguajes de programación son las herramientas básicas que el programador tiene en su *caja de herramientas*. El conocimiento de esas herramientas y cómo y en qué contexto debe usarse cada uno de ellos hace la diferencia entre un programador recién iniciado y un experimentado especialista.

Es fundamental que los conceptos sobre lenguajes de programación estén claros para poder aplicar (y entender) las otras áreas del desarrollo de software como lo son las estructuras de datos, el diseño de algoritmos y estructuración (diseño) de programas complejos. En definitiva estas tareas se basan siempre en un mismo concepto: *abstracciones*.

[3] En SQL no se pueden expresar *clausuras*.

1.4 Definición de un lenguaje de programación

Para describir un lenguaje de programación es necesario definir la forma de sus *frases* válidas del lenguaje y de la semántica o significado de cada una de ellas.

1.4.1 Sintaxis

Los mecanismos de definición de sintaxis han sido ampliamente estudiados desde los inicios de la computación. El desarrollo de la teoría de lenguajes y su clasificación[6] ha permitido que se definan formalismos de descripción de lenguajes formales e inclusive, el desarrollo de herramientas automáticas que permiten generar automáticamente programas reconocedores de lenguajes (parsers y lexers) a partir de su especificación[4].

La sintaxis de un lenguaje se especifica por medio de algún formalismo basado en *gramáticas libres de contexto*, las cuales permiten especificar la construcción (o derivación) de las frases de un lenguaje en forma modular.

Las gramáticas libres de contexto contienen un conjunto de *reglas de formación* de las diferentes frases o *categorías sintácticas de un lenguaje.*

Definición 1.4.1 *Una gramática libre de contexto (CFG) es una tupla* (V_N, V_T, S, P)*, donde* V_N *es el conjunto finito de símbolos no terminales,* V_T *es el conjunto finito de símbolos terminales,* $S \in V_N$ *es el símbolo de comienzo y* P *es un conjunto finito de producciones.*

Los conjuntos V_N *y* V_T *deben ser disjuntos (*$(V_N \cap V_T) = \emptyset$*) y denotaremos* $\Sigma = V_N \cup V_T$*.*

P *es un* conjunto de producciones*, donde una producción* $p \in P$ *tiene la forma* (L, R)*, donde* $L \in V_N$ *es la parte izquierda (lhs) de la producción y* $R \in (V_N \cup V_T)^*$ *es la parte derecha (rhs).*

Por claridad, en lugar de describir las producciones como pares, se denotará a una producción rotulada p: $(X_0, (X_1, \ldots, X_{n_p}))$, con $n_p \geq 0$ como:

$$p : X_0 \to X_1 \ldots X_{np} \tag{1.1}$$

y en el caso que $n_p = 0$, se escribirá como:

$$p : X_0 \to \lambda \tag{1.2}$$

De aquí en adelante se asumirá que el símbolo de comienzo S aparece en la parte izquierda de una única producción y no puede aparecer en la parte derecha de ninguna producción[5].

Es común que un conjunto de producciones de la forma $\{X \to \alpha, \ldots, X \to \beta\}$ se abrevie de la forma $X \to \alpha \mid \ldots \mid \beta$.

[4] Como las populares herramientas *lex* y *yacc*.
[5] Esta forma se denomina *gramática extendida*.

Definición 1.4.2 *Sean* $\alpha, \beta \in (V_N \cup V_T)^*$ *y sea* $q : X \to \varphi$ *una producción de* P, *entonces* $\alpha X \beta \underset{G}{\overset{q}{\Longrightarrow}} \alpha\varphi\beta$

La relación $\underset{G}{\Longrightarrow}$ se denomina *relación de derivación* y se dice que la cadena $\alpha X \beta$ deriva directamente (por aplicación de la producción q) a $\alpha\varphi\beta$.

Cuando se desee hacer explícita la producción usada en un paso de derivación se denotará como $\underset{G}{\overset{q}{\Longrightarrow}}$.

Se escribirá $\underset{G}{\overset{*}{\Longrightarrow}}$ a la clausura reflexo-transitiva de la relación de derivación.

Definición 1.4.3 *Sea* $G = (V_N, V_T, S, P)$ *una gramática libre de contexto. Una cadena* α, *obtenida por* $S \underset{G}{\overset{*}{\Longrightarrow}} \alpha$ *que contiene sólo símbolos terminales* ($\alpha \in V_T^*$), *se denomina una sentencia de* G. *Si la cadena* $\alpha \in (V_T \cup V_N)^*$ *(contiene no terminales) se denomina forma sentencial.*

Definición 1.4.4 *El lenguaje generado por* G, *denotado como*

$$L(G) = \{w | w \in V_T^* \mid S \underset{G}{\overset{*}{\Longrightarrow}} w\}$$

Definición 1.4.5 *Sea el grafo dirigido* $ST = (K, D)$ *un árbol, donde* K *es un conjunto de* nodos *y* D *es una relación no simétrica, con* k_0 *como raíz, una función de rotulación* $l : K \to V_T \cup \epsilon$ *y sean* $k_1, \ldots, K_n$, $(n > 0)$, *los sucesores inmediatos de* k_0.

El árbol $ST = (K, D)$ *es un árbol de derivación (o parse tree) correspondiente a* $G =< V_N, V_T, P, S >$ *si cumple con las siguientes propiedades:*

1. $K \subseteq (V_N \cup V_T \cup \epsilon)$
2. $l(K_0) = S$
3. $S \to l(k_1) \ldots l(k_n)$
4. *Si* $l(k_i) \in V_T$, $(1 \leq i \leq n)$, *o si* $n = 1$ *y* $l(k_1) = \epsilon$, *entonces* K_i *es una hoja de* ST.
5. *Si* $l(k_i) \in V_N$, $(1 \leq i \leq n)$, *entonces* k_i *es la raíz del árbol sintáctico para la gramática libre de contexto* $< V_N, V_T, P, l(k_i) >$.

Definición 1.4.6 *Sea ST(G) un árbol de derivación para* $G =< V_N, V_T, S, P >$. *La frontera de ST(G) es la cadena* $l(k_1) \ldots l(k_n)$ *tal que* $k_1 \ldots k_n$ *es la secuencia formada por las hojas de ST(G) visitadas en un recorrido preorden.*

Teorema 1.4.1 *Sea* $G =< V_N, V_T, S, P >$ *una gramática libre de contexto,* $S \underset{G}{\overset{*}{\Longrightarrow}} \alpha$ *si y sólo si existe un árbol de derivación para* G *cuya frontera es* α.

La figura 1.1 muestra una gramática libre de contexto y un árbol de derivación para la cadena *"a + b * c"*.

La gramática dada en la figura 1.1 es *ambigüa* ya que para una misma cadena existen dos (o más) árboles de derivación diferentes. Una gramática puede desambig'uarse introduciendo producciones que definan la *precedencia* entre los diferentes no terminales.

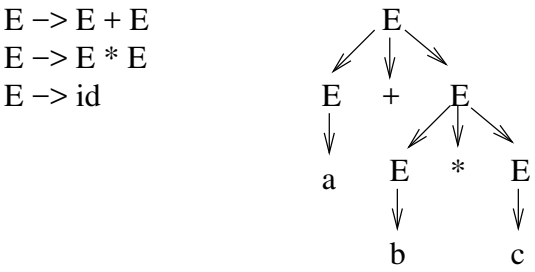

Fig. 1.1: Una CFG y un árbol de derivación.

Definición 1.4.7 *Dos gramáticas g_1 y g_2 son* equivalentes *si generan el mismo lenguaje, es decir que $L(g_1) = L(g_2)$*[6].

Hay gramáticas *inherentemente ambigüas* para las cuales no existe una gramática equivalente no ambigüa.

1.4.1.1 Lenguajes regulares

Las *palabras* que se pueden formar en un lenguaje generalmente se describen con formalismos que no requieren describir estructuras de las frases. Estos formalismos se conocen como las *gramáticas regulares*. Existen otros formalismos equivalentes ampliamente utilizadas, como las *expresiones regulares*.

Definición 1.4.8 *Una* ***gramática regular*** *es una gramática cuyas producciones tienen la forma: $X \to Ya$ y $X \to a$, donde $X, Y \in N$ y $a \in T$.*

Estas gramáticas sólo permiten describir la conformación de las *palabras o tokens* de un lenguaje lenguaje, pero no es posible describir la estructura de frases. A modo de ejemplo se muestra una gramática regular que describe la formación de un valor entero positivo:

```
N --> N '0' | N '1' | ... | N '9' | '0' | '1' | ... | '9'
```

El ejemplo anterior muestra que es extenso definir la forma de construcción de símbolos de un lenguaje por medio de una gramática regular, por lo que es común que se definan por medio de un formalismo, las *expresiones regulares*, cuya expresividad es equivalente y permiten definiciones mas compactas y legibles.

A continuación se da una gramática libre de contexto que describe la sintaxis de una expresión regular:

```
E --> t
| E E            -- secuencia
```

[6] La determinación si dos gramáticas libres de contexto son equivalentes es indecidible, es decir, no existe un algoritmo que lo determine.

```
| (E '|' E)       -- alternativa (choice)
| (E)?            -- opcional (cero o una vez)
| (E)*            -- cero o m\'as veces
```

donde t es un símbolo terminal.

Las *gramáticas regulares extendidas* introducen otras construcciones más cómodas en la práctica como las siguientes:

```
E --> [ E ... E ]      -- set: equivalente a (E | ... | E)
    | (E)+             -- una o m\'as veces: equivalente a (E(E)*)
```

1.4.1.2 EBNFs y diagramas de sintaxis

Una *Extended Backus Naur Form* es una extensión de las gramáticas libres de contexto que permite la descripción de un lenguaje en forma mas compacta.

Informalmente, se puede decir que permiten escribir expresiones regulares extendidas en la parte derecha de las producciones. Las notaciones mas comunmente mas utilizadas son:

- (S): S ocurre una o mas veces.
- $\{S\}$: S ocurre cero o mas veces.
- $[S]$: S es opcional (cero o una vez).

A continuación de muestra un ejemplo de una EBNF.

```
...
var-decl --> var id {',' id} ':' type ';'
type     --> integer | real | ...
...
if-stmt  --> if condition then stmt [ else stmt ]
...
```

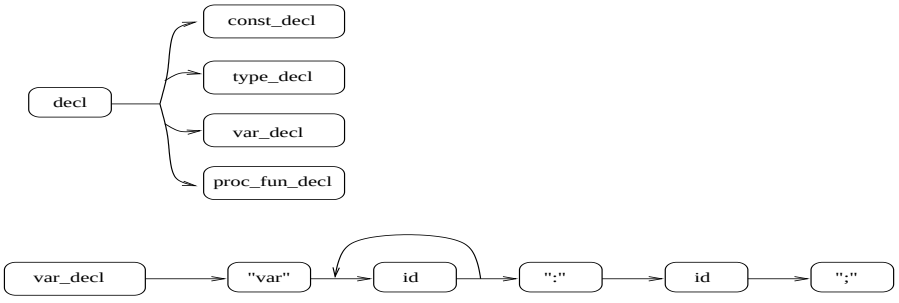

Fig. 1.2: Ejemplo de diagramas de sintaxis.

Los diagramas de sintaxis son una representación gráfica por medio de un grafo dirigido el cual muestra el flujo de aparición de los componentes sintácticos. Los nodos del grafo corresponden a los símbolos terminales y no terminales y los arcos indican el símbolo que puede seguir en una frase. Es común que los nodos correspondientes a los símbolos terminales se denoten con círculos y los nodos que corresponden a no terminales se denoten como óvalos.

La figura 1.4.1.2 muestra un ejemplo de diagramas de sintaxis.

1.4.2 Semántica

La semántica de un lenguaje de programación describe el significado, comportamiento o efectos de las diferentes frases del lenguaje.

Es muy común que en los manuales de los lenguajes de programación la semántica de cada una de las frases de describa de manera informal.

Esta informalidad ha llevado muchas veces a confusiones en los programadores o los implementadores de herramientas como compiladores e intérpretes, causando que los resultados de un programa en una implementación no sean los mismos que en otra[7].

Para dar una definición precisa de la semántica de un lenguaje es necesario utilizar algún formalismo que describa en forma clara y no ambigüa el significado de las frases.

Se han utilizado diferentes estilos de formalismos para dar semántica:

- **Denotacional**: cada construcción del lenguaje se relaciona con alguna entidad matemáticas (ej: conjuntos, funciones, etc) que representa el significado de cada estructura.

  Esta forma de dar semántica es útil desde el punto de vista teórico, pero en general no es cómodo para los implementadores de lenguajes y los desarrolladores.

- **Operacional**: descripción del *efecto o ejecución* de cada construcción del lenguaje en una *máquina abstracta* dada. Una máquina abstracta está basada en algún modelo de computación.

  Esta forma es útil tanto para los implementadores del lenguaje como para los desarrolladores de programas, ya que tienen una visión mas concreta (operacional) del lenguaje.

- **Axiomática**: descripción de cada construcción del lenguaje en términos de cambios de estado. Un ejemplo es la lógica de Hoare, que es muy útil para el desarrollo y verificación formal de programas imperativos.

  Esta técnica es útil para los desarrolladores pero no demasiado buena para los implementadores del lenguaje.

En este libro se utilizará la semántica operacional para dar el significado al lenguaje que se irá desarrollando en cada capítulo, siguiendo la idea de *lenguaje núcleo (kernel)* el cual permite dar una sintaxis y semántica de manera sencilla para luego *adornar* el lenguaje con mejoras sintácticas (syntactic sugars) y abstracciones sintácticas o lingüísticas prácticas, las cuales tendrán un patrón de traducción al lenguaje núcleo.

[7] Esto ha sucedido en C, C++, FORTRAN, y hasta en los lenguajes de reciente aparición.

1.5 Herramientas para la construcción de programas

El programador cuando utiliza un lenguaje de programación, utiliza herramientas que implementan el lenguaje. Estas herramientas son programas que permiten ejecutar en la plataforma de hardware utilizada las construcciones del lenguaje de alto nivel. En general se disponen de las siguientes herramientas:

- **Compilador**: traduce un programa fuente a un programa *assembly* u *objeto* (archivo binario enlazable).

- **Intérprete**: programa que toma como entrada programas fuentes, genera una representación interna adecuada para su ejecución y evalúa esa representación emulando la semántica de las construcciones del programa dado.

  Es posible encontrar intérpretes de bajo nivel, también conocidos como *ejecutores* de programas. Estos ejecutores interpretan lenguajes de bajo nivel (assembly real o hipotético).

  Es común que una implementación de un lenguaje venga acompañado por un compilador a un assembly de una máquina abstracta y un intérprete de ese lenguaje de alto nivel. Ejemplos de esto son algunos compiladores de COBOL, Pascal (se traducía a P-code).

  Actualmente uno de los casos mas conocidos sea Java. Es común que un compilador de Java traduzca los módulos a un assembly sobre una máquina abstracta conocida como la *Java Virtual Machine (JVM)*.

  Este último enfoque permite obtener *portabilidad* binaria, ya que es posible ejecutar un programa en cualquier plataforma que tenga una implementación (intérprete) de la máquina abstracta.

- **Enlazador (linker)**: un archivo objeto puede hacer referencia a símbolos (variables, rutinas, etc) de otros archivos objetos. Estas referencias se denominan *referencias externas*. El linker toma un conjunto de archivos objetos[8], arma una imagen en memoria, resuelve las referencias externas de cada uno (asigna direcciones de memoria concretas a cada referencia externa no resuelta) y genera un archivo binario ejecutable (*programa*).

  En forma más rigurosa, un linker básicamente implementa una función que toma una referencia a un símbolo externo y retorna la dirección de memoria de su definición.

  Generalmente cada archivo objeto se corresponde con un *módulo* del programa fuente. La modularización es útil para dividir grandes programas en unidades lógicas reusables.

[8] Generalmente llamados módulos binarios.

Además, los ambientes de desarrollo generalmente vienen acompañados por módulos básicos para hacerlo mas útil en la práctica (módulos para hacer entrada-salida, funciones matemáticas, implementación de estructuras de datos, etc) lo que comúnmente se conoce como la *biblioteca estándar* del lenguaje.

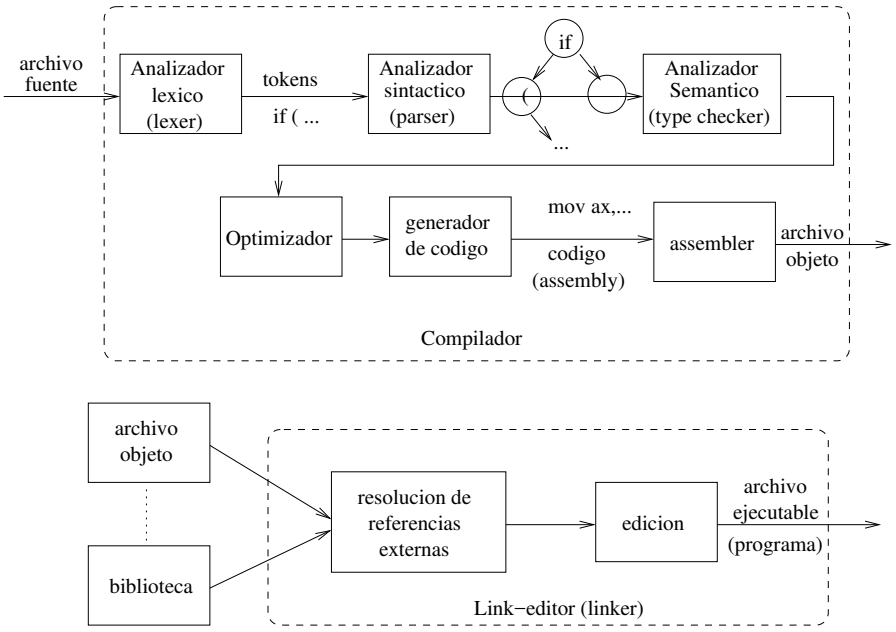

Fig. 1.3: Esquema de compilación de un programa.

La figura 1.3 muestra un esquema del proceso de compilación de un programa.

1.5.1 Bibliotecas estáticas y dinámicas

Una *biblioteca* es un archivo que contiene archivos objeto.

Generalmente un programa de usuario se enlaza con al menos unas cuantas rutinas básicas que comprenden el sistema de tiempo de ejecución (*runtime system*). El runtime system generalmente incluye rutinas de inicio (start-up) de programas[9], y la implementación de otras rutinas básicas del lenguaje.

Cuando en el programa obtenido se incluye el código (y posiblemente datos) de las rutinas de biblioteca utilizadas se denomina enlazado estático (static linking).

Un programa enlazado estáticamente tiene la ventaja que cuando se lo transporta a otra computadora tiene todas sus dependencias resueltas, es decir que todas sus

[9] Una rutina de startup generalmente abre archivos de entrada-salida estándar e invoca a la rutina principal del programa.

referencias (a datos y código) están resueltas y todo está contenido en un único archivo binario.

Los primeros sistemas de computación generalmente soportaban este único tipo de enlazado. De aquí el nombre a estos linkers conocidos como *link-editores*.

A medida que el tamaño de los programas crece, el uso de bibliotecas generales es común. Más aún, en los sistemas multitarea (o multiprogramación), comienzan a aparecer varias desventajas y el mecanismo de enlazado estático se torna prácticamente inviable.

Las principales desventajas son:

- El tamaño de los programas se hace muy grande.
- En un sistema multitarea hay grandes cantidades del mismo código replicado en la memoria y en el sistema de archivos.
- No tiene en cuenta la evolución de las bibliotecas, cuyas nuevas versiones pueden corregir errores o mejorar su implementación.

Por este motivo aparece el enfoque de las *bibliotecas de enlace dinámico*[10] (DLLs).

Este enfoque requiere que el sistema operativo contenga un linker dinámico, es decir que resuelva las referencias externas de un módulo (archivo objeto) en tiempo de ejecución.

Cuando un proceso (instancia de programa en ejecución) hace referencia a una entidad cuya dirección de memoria no haya sido resuelta (referencia externa), ocurre una trampa (trap) o excepción generada por el sistema operativo. Esta trampa dispara una rutina que es la encargada de realizar el enlace dinámico.

Posiblemente se requiera que el código (o al menos la parte requerida) de la biblioteca sea cargada en la memoria (si es que no lo estaba).

Cabe hacer notar que los archivos objetos deben acarrear mas información de utilidad por el linker dinámico. Un programa debe acarrear la lista de bibliotecas requeridas y cada archivo objeto de cada bibliotecas debe contener al menos el conjunto de símbolos que exporta.

Las principales ventajas que tiene este mecanismo son:

- El código de las rutinas de las bibliotecas se encuentra presente una sola vez (no hay múltiples copias).
- El código se carga baja demanda. Es decir que no se cargará el código de una biblioteca que no haya sido utilizada en una instancia de ejecución.

Como desventaja tiene que la ejecución de los programas tiene una sobrecarga adicional (overhead) que es el tiempo insumido por la resolución de referencias externas

[10] En el mundo UNIX son conocidas como *shared libraries*.

y la carga dinámica de código.

Un linker con capacidades de generar bibliotecas dinámicas deberá generar archivos objetos con la información adicional que mencionamos arriba y el sistema operativo deberá permitir ejecutar código reubicable, es decir independiente de su ubicación en la memoria[11].

Una biblioteca compartida no debería tener estado propio, ya que puede ser utilizada por múltiples procesos en forma simultánea, es decir que es un recurso compartido por varios procesos. Por ejemplo, un programador de una biblioteca que pueda utilizarse en forma compartida no podrá utilizar variables globales.

Lo anterior es muy importante a la hora de diseñar bibliotecas. Es bien conocido el caso de la biblioteca estándar de C, la cual define una variable global (`errno`), la cual contiene el código de error de la última llamada al sistema realizada.
Al querer hacer la biblioteca de C compartida, los desarrolladores tuvieron que implementar un atajo para solucionar este problema.

1.6 Ejercicios

Nota: los ejercicios están planteados para ser desarrollados en un sistema que disponga de las herramientas de desarrollo comúnmente encontrados en sistemas tipo UNIX. El práctico se puede desarrollar en cualquier plataforma que tenga instaladas las herramientas básicas de desarrollo del proyecto GNU (software libre) instaladas.

Herramientas necesarias: gcc (GNU Compiler Collection), gpc (GNU Pascal Compiler), ld, grep y wc.

1. Definir una expresión regular que denote un identificador en Pascal.
2. Definir un autómata finito que acepte el lenguaje denotado por la expresión regular del ejercicio anterior.
3. Definir un autómata finito que acepte cadenas de numeros binarios con cantidad par de 0's y cantidad par de 1's.
4. Definir un autómata finito que acepte cadenas de numeros binarios con cantidad par de 0's y cantidad impar de 1'.
5. Usar el comando **grep**[12] que seleccione las líneas del archivo fuente Pascal del ej. 7 los siguientes patrones:
   (a) Las líneas que contengan *Var*
   (b) Las líneas con comentarios

[11] Esto se logra utilizando algún mecanismo de *memoria virtual* (segmentación o paginado)
[12] Uso: grep expresión-regular [file]. Para mas información hacer "man grep".

(c) Comparar la cantidad de begin y la cantidad de end en un programa Pascal. Ayuda: usar grep y wc.

6. Dar una EBNF que defina las sentencias de Pascal.

7. Dado el siguiente programa Pascal y el siguiente fragmento de código C. El programa CallToC declara una variable *externa*, le asigna un valor e invoca a un procedimiento *externo*, el cual está implementado en C (en el módulo *inc.c*),

```
Program CallToC;

Var x:integer; external name 'y';
Procedure inc_x; external name 'inc_y';

begin { programa principal }
  x := 1;
  inc_x;
  writeln('x=',x)
end.
```

```
/* file inc.c */
int y;          /* global integer y */

void inc_y(void)
{
    y++;
}
```

   (a) compilar el programa Pascal (usando gpc). En caso de error describir su origen y quién lo genera (compilador o linker).

   (b) compilar el fragmento de programa C para obtener el archivo objeto correspondiente[13] analizando los pasos realizados. Usar el comando *objdump -t inc.o* para ver los símbolos definidos en el archivo objeto.

   (c) generar un archivo ejecutable en base a los dos módulos.

   (d) describir qué pasos se realizaron (compilación, assembly, linking) en el punto anterior.

8. Generar una biblioteca estática (llamada *libmylib.a*) que contenga el archivo objeto *inc.o* (del ejercicio anterior) con la utilidad *ar*.

   Usar el siguiente programa C (el cual invoca a *inc_y*()) para compilarlo enlazarlo con la biblioteca *mylib*.

```
int main(void)
{
  inc_y();
}
```

[13] Usar el comando *gcc -v -c inc.c*.

9. Recompilar el programa Pascal definido arriba usando la biblioteca creada en el ejercicio anterior.

10. El siguiente programa C muestra la carga de una biblioteca dinámica (math), la resolución de una referencia (externa) a la función *cos* (definida en math) y la invocación a *cos(2.0)*.

```
/* File: foo.c */
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

int main()
{
    void *handle;
    double (*cosine)(double); /* Pointer to a function */

    /* Load the math library */
    handle = dlopen("libm.so", RTLD_LAZY);

    /* Get (link) the "cos" function: we get a function pointer */
    cosine = (double (*)(double)) dlsym(handle, "cos");
    printf("%f\n", cosine(2.0));
    dlclose(handle);
    exit(EXIT_SUCCESS);
}
```

Compilar el programa (con el comando *gcc -rdynamic -o foo foo.c -ldl*) y ejecutarlo.

Ejercicios Adicionales

11. Implementar un programa que reconozca frases según la siguiente EBNF:

$E \rightarrow T['+'E]$
$T \rightarrow F['*'T]$
$F \rightarrow V \mid '('E')'$
$V \rightarrow ('0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9')+$

Ayuda: Por cada regla de la gramática de la forma $X \rightarrow' a'Y'b'$ se puede definir un procedimiento con la forma:

```
Procedure X;
begin
  if current_token = 'a' then begin
```

```
        next_token;
        Y
    end
    else
        error;
    if current_token = 'b' then
        next_token
    else
        error
end;
```

donde *next_token* es un procedimiento que obtiene el próximo (símbolo) token de la entrada.

Para ésta gramática *next_token* debería reconocer (y obtener) valores numéricos y los símbolos + y * (e ignorar espacios, tabs y new-lines).

Generar patrones de código para reglas que contengan componentes opcionales (0 o una vez) y repeticiones (0 o mas y 1 o mas).

12. Extender el programa anterior para que evalúe la expresión.
Ayuda: utilizar una pila de operandos y una pila de operadores.