Compiladores

José Juan Palacios

Esta guía de lectura tiene como propósito servir como referencia básica para el curso de compiladores. Los conceptos se sintetizan, enfatizando la ilustración de los mismos mediante ejemplos sencillos. A su vez, fomentan la lectura a profundidad de la bibliografía propuesta. Se espera evolucionarlas para convertirlas en notas de curso.

Contents

1 Introducción **3**
1.1 Modelo de Análisis y Síntesis 4
1.1.1 Fases de un Compilador 5
1.2 Compilador sencillo (1 pasada) 6
1.3 Definición del Lenguaje Fuente 7
1.3.1 Sintaxis 10
1.4 Arquitectura de la Máquina Abstracta de Pila 32

2 Análisis Léxico **36**

3 Análisis Sintáctico **40**
3.1 Análisis Sintáctico Descendente 40
3.1.1 Transformaciones gramaticales 41
3.1.2 ASDP LL 42
3.2 Análisis Sintáctico Ascendente 43
3.2.1 Analizadores Sintácticos *LR(k)* 45
3.2.2 Construcción de Tablas A.S. SLR 47

Capítulo 1

Introducción

Compilador: *(definición intuitiva)* programa que *traduce* expresiones de un *programa* definidas en un *lenguaje fuente* (entrada) a expresiones correspondientes en un *lenguaje objeto* (salida) (ver fig. 1.1).

Podemos ver de manera abstracta a un compilador como una *función* que recibe como argumentos expresiones x (cadenas de *símbolos* definidas sobre un *alfabeto finito* Σ) y que regresa como resultado la correspondiente expresión *compilada* x^c siempre y cuando la expresión x **sea correcta en términos de la definición del lenguaje fuente**:

$$x \in \Sigma^* \qquad\qquad C : \Sigma^* \mapsto E^*$$
$$C(x) = \begin{cases} x^c & \text{si } x \in L(G) \\ \textbf{error} & \text{en caso contrario} \end{cases}$$

donde E es el conjunto de *instrucciones ejecutables* para alguna *arquitectura o modelo de computadora*. La expresión compilada x^c es construída fundamentalmente en base a un *análisis* de la *estructura* de x para poder *verificar* que x está *bien definida* (i.e. pertenece al lenguaje fuente).

Consideraciones: sin pérdida de generalidad, se considera que el lenguaje fuente puede dividirse en dos conjuntos (fundamentalmente disjuntos):

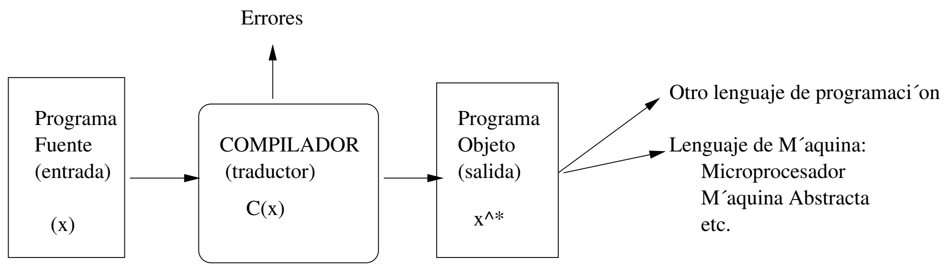


Figura 1.1: Descripción abstracta de un compilador

- **Átomos**: expresiones más sencillas con significado independiente, e.g: "8", "a", "3.1416", etc.(expresiones básicas)

- **Operadores**: expresiones que tienen asociadas *funciones constructoras* mediante las cuales se construyen *expresiones complejas*: "8 + 3.1426 - a"; etc. Evidentemente, para propósitos de su *ejecución* cada operador *denota una función* definida sobre cierto dominio de interpretación; por ejemplo, el operador "+" puede denotar a la función definida sobre los reales $Sum : R \times R \mapsto R$ que recibe como argumentos dos *valores* reales y regresa el valor real correspondiente a la suma.

Intuitivamente, un *tipo* es una especificación de comportamiento caracterizada a partir de una *relación de pertenencia*: "34" puede tener asociado como tipo $34 \in I$ (i.e. entero), o puede tener asociado el tipo $34.0 \in R$ (real). Dada la definición que cada operador denota, se puede verificar cúando una expresión tiene sentido en términos de esa definición: " 8 + 3.1416" tiene sentido si $8 \in R$ y $3.1416 \in R$; pero si $I \subset R$ y $8 \in I$, también se puede considerar como una expresión válida.

Las áreas involucradas en la escritura (diseño/ construcción) de compiladores son:

- Lenguajes de Programación
  Teoría Formal de Lenguajes y Autómatas

- Arquitectura de computadoras

- Algoritmia

- Ingeniería de Software.

por tanto, el análisis, diseño e implantación de compiladores resulta una disciplina fundamental, tanto en áreas pragmáticas como de desarollo puro (teórico). El énfasis en este curso es empírico, es decir, la experimentación pragmática es esencial para capturar y apreciar en todo su valor los conceptos teóricos que son discutidos.

1.1 Modelo de Análisis y Síntesis

Tradicionalmente[Aho86], el proceso de compilación se realiza en 2 partes bien diferenciadas:

Análisis: divide el programa fuente en sus componentes básicos y construye una representación interna: $\mapsto R$ estructura jerárquica **árbol de análisis sintáctico (AAS)**.

Síntesis: construye el programa objeto a partir de la representación interna.

1.1.1 Fases de un Compilador

Operacionalmente, un compilador se divide en las siguientes *fases*:

1. Análisis Lineal o **Léxico**: exploración (scánner)
   realiza una lectura secuencial de los símbolos del programa fuente (generalmente a partir de un *archivo (buffer)* y los *agrupa* en **componentes léxicos (tokens)** i.e. secuencias (cadenas) de símbolos que tienen *significado colectivo* (átomos). A cada componente le asocia *atributos*, es decir, toda información pertinente para su manejo (i.e. su *tipo*, cantidad de memoria necesaria para representarlo, valor, etc.)

   LEX es una herramienta para construcción de analizadores léxicos funciona por definiciones de **expresiones regulares**: *patrón* $\rightarrow$ *acción*

2. Análisis Jerárquico o **Sintáctico**: construcción (parser)
   los tokens son agrupados/organizados jerárquicamente en *frases gramaticales* (representadas en un AAS) que son utilizadas para *sintetizar* la salida.
   Tal estructura jerárquica es definida mediante *reglas recursivas* (gramaticales).

   De igual forma, esta fase *registra* en una estructura de datos denominada **Tabla de Símbolos** a aquellos tokens cuyos valores de atributos *dependan de otros*, i.e. *identificadores* que representen *variables* dentro del programa. Los *atributos* de un identificador representan toda aquella información útil para la traducción: su tipo, la cantidad de memoria que debe reservarse para almacenarlo, etc.

3. Análisis **Semántico**: se verifica concordancia (compatibilidad) entre los *operandos y operadores* en las expresiones, i.e. que se cumplan las definiciones de las operaciones permitidas.

4. Generación de **Código Intermedio**: (opcional) algunos compiladores construyen una representación intermedia explícita del programa fuente. Tal representación puede verse como un programa para una *máquina abstracta*. Debe ser fácil de producir y fácil de traducir al código objeto.

5. Optimación: proceso de mejora al código intermedio de tal forma que sea más rápido y eficiente (generalmente pequeño) en su ejecución.

6. Generación de Código objeto: construye la secuencia de instrucciones objeto y resuelve la disposición final de recursos (asignamiento de localidades/direcciones de memoria, referencias, etc.) listo para ser ejecutado en la máquina destino.

Todas las fases hacen uso de los siguientes módulos:

- Administración de la **Tabla de Símbolos**: permite mantener información sobre los identificadores (variables, constantes, rutinas, etc.). Las operaciones se pueden dividir en general en *consulta y actualización*. La tabla

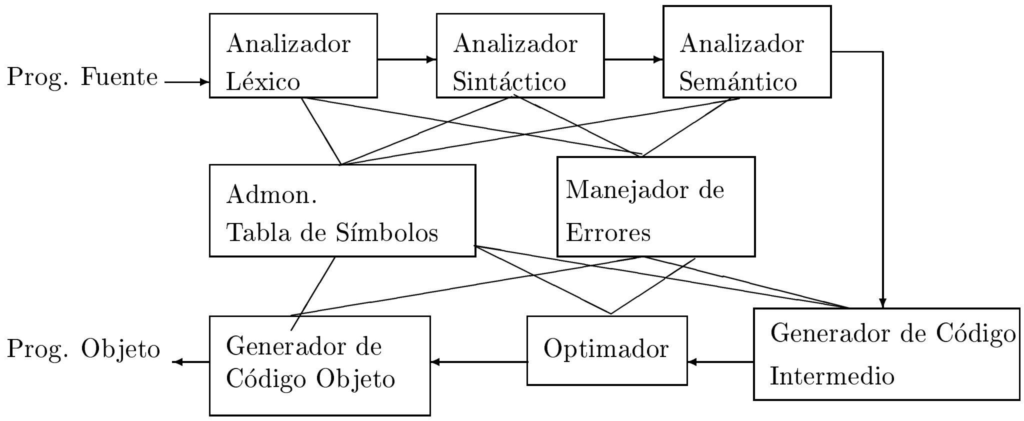


Figura 1.2: Fases típicas de un compilador.

es un conjunto de registros donde cada registro almacena la información (atributos) para cada identificador que lo requiera, tales atributos (almacenados en campos del registro) pueden ser: tipo, memoria, num. de argumentos (si los necesita), valor que representa como dato concreto, etc.

- Recuperación de **Errores**: cada fase puede encontrar errores, por lo que cada fase debe ser capaz de enfrentarlos adecuadamente para continuar *lo más posible* con la compilación, permitiendo la detección de más errores (pues un compilador que se detiene al primer error no resulta ser muy útil).

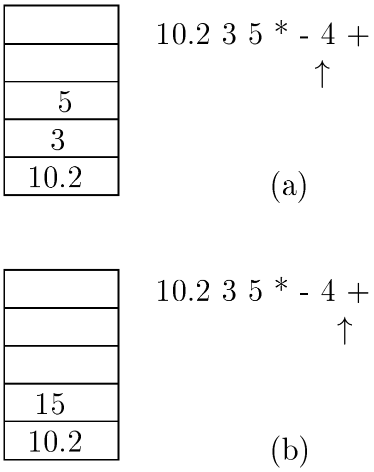
1.2 Compilador sencillo (1 pasada)

En este apartado, ilustraremos lo que realiza cada fase mediante un ejemplo concreto y sencillo.

Objetivo: definir un traductor que acepte expresiones aritméticas sencillas en *notación infija* y retorne como resultado un programa codificado mediante instrucciones para una máquina abstracta de pila. Este traductor utilizará como representación intermedia la denominada forma *sufija (postfija)*:

$$\underbrace{10.2 - 3 * 5 + 4}_{\text{infijo}} \quad \rightarrow \quad \underbrace{10.2\ 3\ 5\ *\ -\ 4\ +}_{\text{sufijo}}$$

A partir de esa representación en forma sufija, se procede a traducir en código de máquina abstracta de pila, el cual es un mecanismo de cómputo general y simple: la estructura de datos principal de esta máquina es una pila sobre la cual se realizan las operaciones. Dada la expresión sufija anterior, la recorre de izq. a der. cuando encuentra un dato (número) lo almacena en el tope de la



01:	empila 10.2
02:	empila 3
03:	empila 5
04:	multiplica
05:	resta
06:	empila 4
07:	suma

Figura 1.3: Evaluación de una expresión aritmética sencilla (en sufijo) por la máquina abstracta de pila: en la descripción instantánea del contenido del stack en la parte (a) se está por aplicar el operador de multiplicación, mientras que en (b) acaba de ser aplicado.

pila, cuando encuentra un operador desempila los n datos de la pila, les aplica el operador y empila el resultado; donde n es la *aridad* del operador (i.e. el número de argumentos que puede recibir), hasta terminar de examinar toda la expresión. En el tope de la pila estará el resultado final de *evaluar* la expresión. Por ejemplo, algunas acciones que realiza la máquina ante la expresión anterior se muestran en la fig. 1.3:

Para este ejemplo se describirán: Análisis Léxico, Sintáctico, construcción de código intermedio y Generación de Código objeto. Además, se ampliará modularmente el lenguaje fuente hasta obtener un lenguaje de programación imperativo típico (al estilo de la familia de lenguajes de Modula) enriqueciendo a su vez instrucciones correspondientes a la máquina abstracta.

Se asume que el lector conoce los fundamentos de la teoría de lenguajes y autómatas, por lo que se apelará a los conceptos de expresiones regulares, autómatas finitos y gramáticas libres de contexto en este apartado.

1.3 Definición del Lenguaje Fuente

Recordemos que un lenguaje es a final de cuentas un conjunto de cadenas o expresiones (constituídas por símbolos[1] tomados de un alfabeto finito Σ).

[1] Utilizaremos indistintamente los términos *caracter* y *componente léxico*.

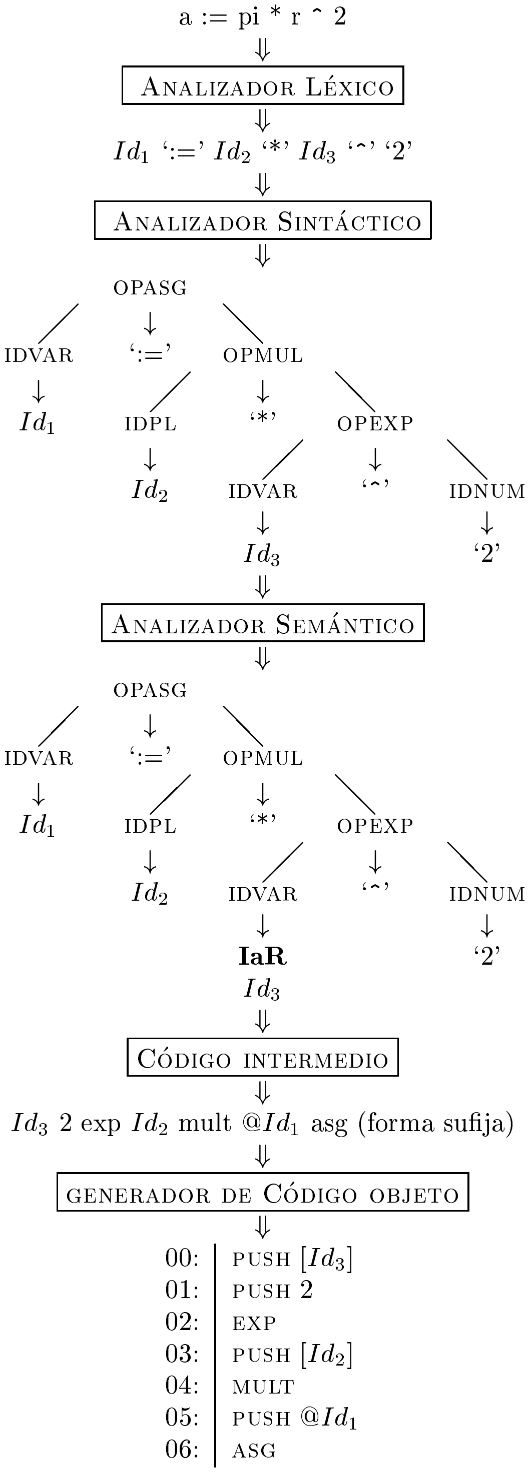


Figura 1.4: Ejemplo del funcionamiento de las fases de un compilador.

Tabla de Símbolos			
#	nombre	tipo	mem
1	**a**	R	8
2	**pi**	R	-
3	**c**	**I**	4
:	:		

Figura 1.5: Tabla de símbolos para el ejemplo anterior.

Caracterizaremos a un lenguaje de programación en dos aspectos fundamentales:

Sintaxis : indica *cómo* construír las expresiones del lenguage, i.e. el *aspecto de sus programas.* Se usará la notación *BNF (Backus-Naur)* fundamentada en la teoría de las *Gramáticas Libres de Contexto (CFG).* Además de servir para especificar la sintaxis de un lenguaje, la CFG permite describir técnicas para conducir la traducción de los programas, a estas técnicas se les denomina *Traducción Conducida por la Sintaxis (SDT)* y se utilizarán ampliamente en este apartado.

Semántica : expresa el significado de los programas[2]. Cabe citar a las siguientes enfoques:

Denotacional relaciona cada programa a una descripción de sus propiedades observables y comportamiento cuando es ejecutado (aplicable para capturar los requerimientos y el establecimiento de la correctez de las especificaciones y diseño).

Algebraica caracteriza a los programas mediante una colección de ecuaciones las cuales son postuladas y se cumplen en los programas mismos. Esta semántica puede utilizarse directamente para razonar acerca de las especificaciones, para transformar diseños y optimizar programas.

Axiomática establece significados a las construcciones de los programas asociándoles *reglas de inferencia* dentro de una lógica formal, enfatizando la correctez de los programas a traves de construcción de pruebas.

Operacional otorga significado a un programa mostrando cómo su ejecución puede ser dividida en una secuencia de pasos atómicos (*transiciones* ejecutados por un mecanismo abstracto simple. Ofrece aspectos fundamentales sobre la eficiencia y computabilidad del lenguaje. Este enfoque es el que utilizaremos, para lo cual será suficiente describir las *precondiciones* y *postcondiciones* al aplicar las *transiciones* (i.e. descripciones instantáneas del estado de la máquina).

[2]Aunque suena sencillo el propósito de la semántica, en general, con las notaciones disponibles es un problema mucho más difícil que la sintaxis. Aún existen problemas abiertos en éste ámbito y constituye un todo campo de investigación.

1.3.1 Sintaxis

Una *Gramática Libre de Contexto (CFG)* es una descripción estructural precisa de una lenguaje. Formalmente es una tupla $G =< Vn, Vt, P, S >$, donde:

Vn	conjunto finito de *símbolos no terminales*
Vt	conjunto finito de *símbolos terminales* ($Vn \cap Vt = \emptyset$)
P	es un conjunto finito de *producciones*
S	es el *símbolo inicial de la gramática*

Las producciones tienen la forma $A \to \alpha$ donde $A \in Vn$ y α es una expresión ya sea compuesta tanto por símbolos no terminales como terminales ($\alpha \in (Vt \cup Vn)^*$) ó la expresión nula (vacía) denotada por ϵ. Una producción puede verse como una *regla de reescritura* que indica cómo reemplazar símbolos no terminales por la expresión correspondiente en la parte derecha de la producción. Así, partiendo de algún $A \in Vn$, se pueden aplicar las reglas de P hasta alcanzar eventualmente a una expresión compuesta únicamente por símbolos terminales. A ésta expresión se le denomina *frase* ó *lexema*, mientras que a las expresiones formadas por símbolos terminales y no terminales se les denomina *formas de frase*.

Seguiremos las convenciones usuales[Dol79, Aho86]:

1. Letras romanas mayúsculas (A, B, C, D, E, S) representan símbolos no terminales (*variables*).

2. Letras romanas minúsculas (a, b, c, d, e), dígitos o expresiones encerradas entre comillas (‘,’) representan símbolos terminales.

3. X, Y, Z, W representan terminales o no terminales (metavariables).

4. u, v, w, x, y, z representan cadenas de terminales (frases).

5. Letras griegas minúsculas α, β, γ representan formas de frases.

Gracias a estas convenciones, es común describir una gramática enlistando únicamente sus producciones: si se tiene $A \to \alpha_1, A \to \alpha_2, \ldots, A \to \alpha_k$, se puede sintetizar mediante la expresión $A \to \alpha_1|\alpha_2|\ldots|\alpha_k$.

Derivaciones: Para cualquier CFG G, si $A \to \beta$ en P y $\alpha, \gamma \in (Vn \cup Vt)^*$, entonces $\alpha A\gamma \Rightarrow \alpha\beta\gamma$, lo cual se lee “la forma de frase $\alpha A\gamma$ *deriva directamente* a $\alpha\beta\gamma$”. Si se tiene que para $\alpha_1, \alpha_2, \ldots, \alpha_k \in (Vn \cup Vt)^*, (k \geq 1)$, $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_k$ se dice que “α_1 *deriva a* α_k, lo cual se escribe $\alpha_1 \stackrel{*}{\Rightarrow} \alpha_k$.

La relación $\Rightarrow$ entre formas de frase tiene las sigs. propiedades:

1. $\alpha \stackrel{*}{\Rightarrow} \alpha$ para cualquier α,

2. Si $\alpha \stackrel{*}{\Rightarrow} \beta$ y $\beta \stackrel{*}{\Rightarrow} \gamma$ entonces $\alpha \stackrel{*}{\Rightarrow} \gamma$.

(En otras palabras, $\stackrel{*}{\Rightarrow}$ es la cerradura reflexiva y transitiva de $\Rightarrow$ la cual es una relación sobre $\Sigma^* \times \Sigma^* \mapsto \Sigma^*$).

Se define al *lenguaje generado por* G, $L(G) = \{w | w \in Vt^* \text{ y } S \overset{*}{\Rightarrow} w\}$; (i.e. el conjunto de frases derivadas por G partiendo del símbolo inicial).

Un lenguaje *libre de contexto* es aquel generado por alguna CFG.

Para nuestro ejemplo, sea $G_0 =< Vn_0, Vt_0, P_0, S >$:

$$
\begin{array}{l}
Vn_0 = \{exp, dig\} \\
Vt_0 = \{0, 1, 2, \ldots, 9, +, -\} \\
S = exp \\
P_0 = \left\{ \begin{array}{lll}
exp \rightarrow & exp + dig & (1) \\
 & |exp - dig & (2) \\
 & |dig & (3) \\
dig \rightarrow & 0|1|2|3|4|5|6|7|8|9 & (4)
\end{array} \right\}
\end{array}
$$

A continuación se muestra la derivación de la frase "$4 - 3 + 1$ ",

$$
\begin{array}{lll}
exp \Rightarrow_1 & exp + dig & \Rightarrow_2 exp - dig + dig \\
 & \Rightarrow_3 dig - dig + dig & \Rightarrow_4 4 - dig + dig \\
 & \Rightarrow_4 4 - 3 + dig & \Rightarrow_2 4 - 3 + 1
\end{array}
\tag{1.1}
$$

(los subíndices de cada '$\Rightarrow$' indican el número de la producción utilizada en ese paso).

Por tanto "$4 - 3 + 1$"pertenece al lenguaje generado por G_0.

Podemos apreciar que este lenguaje genera expresiones formadas por dígitos separados por signos de '+' ó '-'. Por simplicidad, los elementos básicos (átomos) de este lenguaje son caracteres, más adelante extenderemos al lenguaje para que reconozca identificadores, números enteros y de punto flotante, etc.

En cada paso de una derivación se pueden aplicar las reglas (producciones) en cualquier orden, por lo cual se debe elegir:

1. Qué símbolo no terminal reducir,
2. Si la regla de producción para ese no terminal elegido tiene varias alternativas, cual de ellas optar.

Se puede elegir siempre el no terminal *más a la izquierda* en cada paso, la derivación así obtenida recibe el nombre de *derivación más a la izquierda*: $wA\gamma \Rightarrow w\beta\gamma$, para una $A \rightarrow \beta \in P$. Si α deriva a β mediante una derivación por la izquierda se escribe $\alpha \Rightarrow_{mi} \beta$.

En caso de elegir siempre el no terminal más a la derecha, la derivación obtenida recibe el nombre de *derivación canónica*: $\gamma Aw \Rightarrow_{md} \gamma\beta w$.

Por ejemplo, la derivación mostrada en 1.1 es una derivación más por la izquierda de "$4 - 3 + 1$ ", a continuación se muestra una derivación por la derecha para la misma expresión

$$\begin{array}{ll} exp \Rightarrow_1 exp + dig & \Rightarrow_3 exp + 1 \\ \Rightarrow_2 exp - dig + 1 & \Rightarrow_4 exp - 3 + 1 \\ \Rightarrow_3 dig - 3 + 1 & \Rightarrow_4 4 - 3 + 1 \end{array}$$

Cuando, para la misma expresión es posible encontrar más de una derivación más por la izquierda (o por la derecha), se dice que la gramática correspondiente es **ambigua**. Este aspecto tendrá repercusiones en términos de su implantación como podremos apreciar más adelante.

Por ejemplo, la gramática $G_1 =< Vn_0, Vt_1, P_1, exp >$ es ambigua:

$$Vt_1 = \{0, 1, 2, \ldots, 9, *, -\}$$
$$P_1 = \left\{ \begin{array}{rl} exp \rightarrow & exp * exp \\ & |exp - exp \\ & |dig \\ dig \rightarrow & 0|1|2|3|4|5|6|7|8|9 \end{array} \right\}$$

ya que podemos construír más de una derivación por la izq. para "$5 - 8 * 2$":

$$\begin{array}{lll} exp \Rightarrow_2 & exp - exp & \Rightarrow_3 dig - exp \\ & \Rightarrow_4 5 - exp & \Rightarrow_1 5 - exp * exp \\ & \Rightarrow_3 5 - dig * exp & \Rightarrow_4 5 - 8 * exp \\ & \Rightarrow_3 5 - 8 * dig & \Rightarrow_4 5 - 8 * 2 \end{array}$$

$$\begin{array}{lll} exp \Rightarrow_1 & exp * exp & \Rightarrow_2 exp - exp * exp \\ & \Rightarrow_3 dig - exp * exp & \Rightarrow_4 5 - exp * exp \\ & \Rightarrow_3 5 - dig * exp & \Rightarrow_4 5 - 8 * exp \\ & \Rightarrow_3 5 - 8 * dig & \Rightarrow_4 5 - 8 * 2 \end{array}$$

La ambigüedad, a final de cuentas, significa que una expresión del lenguaje pueda tener *más de una interpretación*, lo cual para nuestros propósitos no será permitido. En esta última gramática, tal ambigüedad está asociada con los operadores '*' y '-', por lo que debe estabecerse su *asociatividad* y *precedencia* para evitar tal ambigüedad. Detallaremos este aspecto para eliminar la ambigüedad más adelante.

Ejercicios 1 • *Para cada una de las siguientes CFGs:*

- *Describa cuáles son los símbolos terminales, no terminales, etc.*
- *Describa en lenguaje natural cual es el lenguaje que generan.*
- *Construya, para alguna expresión del lenguaje, una derivación más por la izq. y otra por la derecha.*
- *Concluya si la gramática es o no ambigua.*

1. $S \rightarrow aSb|ab|\epsilon$
2. $S \rightarrow SS + |SS * |A, A \rightarrow 0|1.$

3. $S \to `('S`)'S|\epsilon$.

4. $S \to Aa|b, A \to Ac|Sd|\epsilon$.

5. $S \to aSbS|bSaS|\epsilon$.

6. $S \to 0|1|S+S|S*|SS|`('S`)'$.

7. $S \to a|`('L`)',\ L \to SL|\epsilon$.

- *Construya una CFG que genere cada uno de los siguientes lenguajes:*

  1. *Listas de dígitos separados por comas.*
  2. *Cadenas que representen números en punto flotante al estilo de Pascal ó C.*
  3. *Identificadores (i.e. secuencias de letras ó dígitos que comienzan siempre por un a letra) en lenguajes tipo C o Pascal.*
  4. *Palabras palíndromas (que se leen de igual forma en ambos sentidos) sobre el alfabeto $\{a, b\}$.*
  5. *El conjunto de todas las palabras sobre el alfabeto $\{a, b\}$ que tienen 2 veces más a's que b's.*
  6. *Números impares en binario.*

Ante una CFG G, la primer pregunta es ¿cual es el lenguaje generado por G?. Se puede formular una descripción en lenguaje natural al respecto, por ejemplo, para $L(G_0)$ podemos afirmar:
El lenguaje de expresiones aritméticas en notación infija sencillas $L(G_0)$ es el conjunto de cadenas compuestas por dígitos separados por operadores '+' y '-' que sigue las siguientes reglas:

1. Un dígito es una expresión correcta
2. Si E_1 y E_2 son expresiones correctas, también lo son $E_1 + E_2$ y $E_1 - E_2$.
3. Ninguna otra expresión construída del alfabeto es una expresión correcta de este lenguaje.

Sin embargo, esta descripción *informal* no resulta suficiente, sino que debe demostrarse formalmente que efectivamente se está *construyendo un conjunto de expresiones*: para ello, se puede construír una *prueba por inducción*. Normalmente deben demostrarse por inducción dos *contenciones*[3]:

1. Mostrar que toda frase derivable apartir de S es una expresión del lenguaje.
2. Mostrar que toda cadena del lenguaje puede ser derivable de S.

[3]A final de cuentas se trata de mostrar la igualdad entre dos conjuntos: el de las expresiones derivables de S y el conjunto de expresiones que se supone consiste el lenguaje.

Mostraremos a continuación para el caso de nuestra gramática G_0, para ello refinaremos convenientemente la descripción informal: $L(G_0)$ describe el conjunto de expresiones de longitud impar, formadas por parejas de dígitos separados por un *operador* '+' ó '-', es decir, si la cadena tiene n dígitos entonces debe tener $n-1$ operadores (o equivalentemente, si tiene n operadores entonces tiene $n+1$ dígitos).

1. Para el primer caso, aplicamos inducción sobre el número de pasos n de una derivación:

   (a) Caso base: $n = 2$, $exp \Rightarrow dig \Rightarrow k$, donde k es cualquier dígito $(0, \ldots, 9)$.

   (b) Suponiendo que todas las cadenas derivadas con menos de i pasos $(i \geq 2)$ producen expresiones correctas, por demostrar que una expresión construída a partir de una derivación con exactamente i pasos también es una expresión correcta.
   Consideremos una derivación por la izq. con exactamente i pasos:

   $$exp \Rightarrow exp\ op\ dig \stackrel{*}{\Rightarrow} w\ op\ dig$$

   donde $op \in \{+, -\}$ y w es una frase que cumple con la hipótesis de inducción (i.e. tiene n dígitos y $n-1$ operadores). Claramente, la expresión $w\ op\ dig$ genera una frase que tiene $n+1$ dígitos y n operadores, terminando así esta parte de la prueba.

2. Ahora, para el segundo aspecto, aplicamos inducción con respecto a la longitud de las expresiones (impar):

   (a) Caso base: $n = 1$. Claramente podemos derivar todas las expresiones de long. 1 de este lenguaje, pues son propiamente dígitos mediante derivaciones de 2 pasos: $exp \Rightarrow dig \Rightarrow$ '$0'$, '$1'$, etc.

   (b) Ahora nuestra hipótesis de inducción es que toda expresión correcta de longitud menor a $2n+1$ (para $n \geq 1$) es derivable a partir de *exp*. Sea w una cadena de long. exactamente $2n+1$ ($|w| = 2n+1$), probaremos que w es derivable apartir de *exp*:

   $$exp \Rightarrow exp\ op\ dig \stackrel{*}{\Rightarrow} v\ op\ dig \Rightarrow w$$

   donde claramente v es de longitud $2n-1$: $|v| = 2(n-1)+1 = 2n-1$.

Ejercicios 2 • *Demuestre (aplicando el principio de inducción) cuál es el lenguaje que generan las gramáticas del ejercicio anterior.*

Árboles de Análisis Sintáctico (AAS)

Un AAS es una representación gráfica de una derivación, no muestra la elección relativa al orden de substitución; por lo tanto ofrece una descripción estructural de las expresiones. Dada una CFG G:

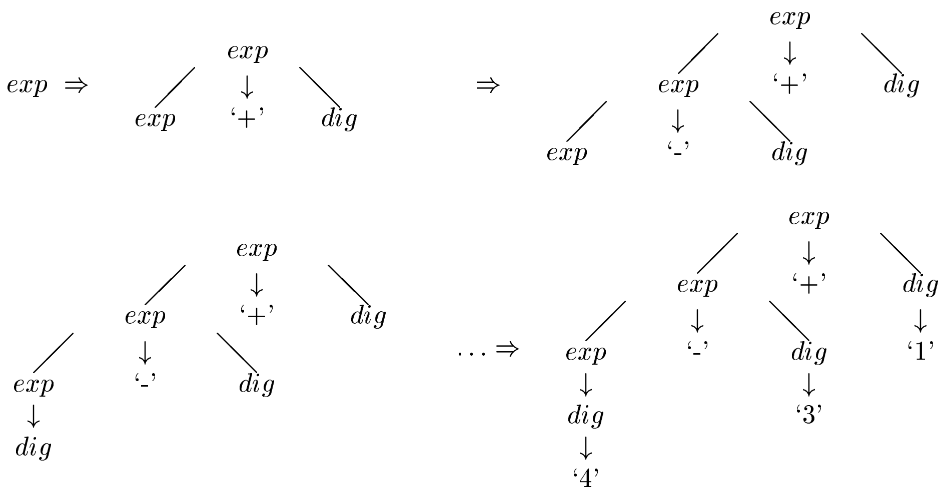
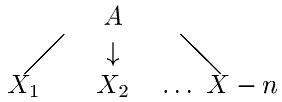


Figura 1.6: Construcción del AAS a partir de la derivación 1.1.

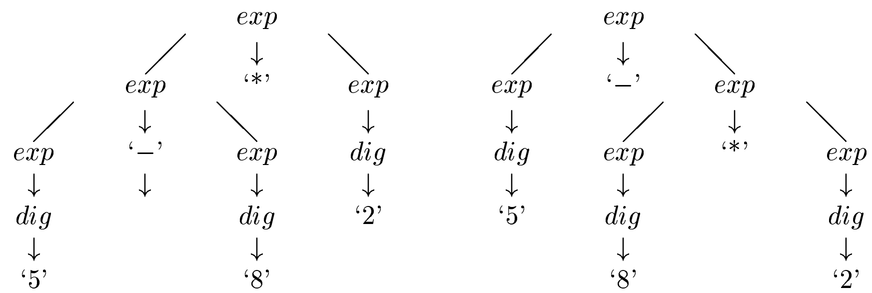
1. La raíz está etiquetada con S.
2. Cada hoja está etiquetada con un componente léxico (terminal) o con ϵ (cadena nula).
3. Cada nodo interior está etiquetada con un símbolo no-terminal.
4. Si A es un no-terminal que etiqueta a un nodo interior y $X_1 X_2 \ldots X_n$ son las etiquetas de los hijos de ese nodo, (de izq. a der.) entonces $A \rightarrow X_2 \ldots X_n$ es una producción en P,



A partir de esta definición se puede inferir un *algoritmo* para construír un AAS para una expresión dada y una CFG G, partiendo del nodo raíz (etiq. por S) y llegando hasta los nodos hojas (etiquetados por los terminales que concatenados en orden izq-der forman la frase original). Un aspecto sumamente importante y que profundizaremos más adelante es que el **recorrido** de un AAS permite realizar la **evaluación** de la expresión correspondiente: para este ejemplo, la evaluación de los nodos hojas resulta el valor asociado con los dígitos, y la evaluación de un nodo interior resulta la aplicación de la *función semántica* (es decir, en este caso la operación aritmética correspondiente, suma y resta) a los valores recién obtenidos.

Para la gramática G_0 y derivación de (1.1), el AAS que genera la expresión correspondiente se muestra en la fig. 1.6.

Alternativamente, $L(G)$ se define como el conj. de cadenas generadas por el AAS. Correspondientemente, una CFG G es ambigua cuando una expresión puede ser generada por más de un AAS: se muestran 2 AAS's para la expresión "$5 - 8 * 2$" en G_1:



(a) (b)

En (a) el árbol crece por la izq. mientras que en (b) por la derecha, lo cual significa que en algún punto para la derivación a partir de un nodo se tiene *más de una producción a elegir* (más de un camino) que permite reconocer la *misma subexpresión* (en este caso, la frase completa); en (a) se eligió la prod. 1 al inicio para reducir el primer *exp* (raíz), mientras que en (b) se eligió la prod.2 para reducir el mismo nodo.

Ejercicios 3

- *Para cada una de las gramáticas del ejercicio anterior, construya al menos un AAS para alguna expresión dada, muestre en cada caso si la gramática es ambigua.*

Asociatividad y Precedencia de operadores

Los 2 AAS anteriores nos establecen 2 formas de agrupamiento ó asociatividad distintas para con los operadores, intuitivamente, si un operador relaciona dentro de un subárbol en mayor profundidad entonces se considera que asocia más fuerte:

1. El AAS (a) indica que el operador '$-$' asocia primero (más fuerte) que '*', mientras que el AAS (b) indica lo contrario. Como bien sabemos, la expresión "$5 - 8 * 2$" se evalúa como $5 - (8 * 2) = 5 - 16 = -9$. Sin reglas para especificar la precedencia relativa de los operadores, los *agrupadores* (paréntesis) serían indispensables para indicar los operandos de un operador. Habitualmente, al operador de multiplicación se le establece *mayor prioridad o precedencia* (considera a sus operandos antes) que al operador de resta. A su vez, los operadores $*, /$ tienen la misma prioridad, y en seguida los operadores $+, -$.

2. Sin embargo, además de prioridad se debe establecer las *asociatividad* ó convenciones de agrupamiento de los operadores, i.e. si asocian por la izquierda o por la derecha: cuando se tienen expresiones que involucran operadores de misma precedencia, por ejemplo $9 + 3 - 1$, ¿cual de las

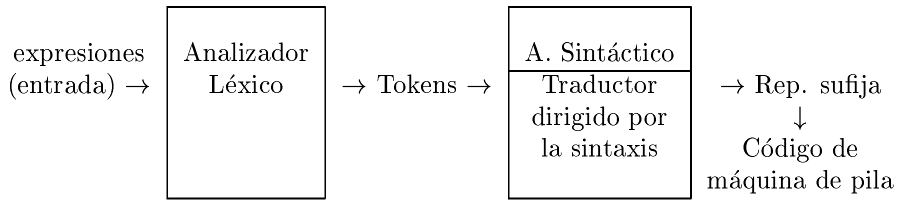


Figura 1.7: Estructura general del traductor para expresiones aritméticas a cd́igo de máquina de pila.

siguientes convenciones es la correcta: $9+(3-1)$ ó $(9+3)-1$? Como es usual, en este caso se considera la segunda, dando como resultado 11.

Un operador es *asociativo por la izquierda* si las subexpresiones que contienen apariciones múltiples del operador (o algun otro de la misma precedencia) se agrupan de izquierda a derecha: los operadores $+,-,*,/$ son asociativos por la izquierda.

Correspondientemente, un operador es *asociativo por la derecha* si las subexpresiones involucrando operadores de igual o menor precedencia se agrupan de derecha a izquierda: el operador de potencia ó exponenciación ˆ es asociativo por la derecha $2^{3^4} = 2^{(3^4)} = 2^{81}$.

Ejercicios 4 • *Defina una CFG (no ambigua) que reconozca expresiones aritméticas involucrando los 5 operadores aritméticos principales* $(+,-,*,/,$ *ˆ), agrupadores ('(', ')') y que siga las convenciones de asociatividad y prioridad mencionadas. Para 5 expresiones, construya su AAS correspondiente.*

Estamos listos para describir operacionalmente la traducción de expresiones a una forma intermedia (*sufija*) a partir de la cual se puede construír código de máquina de pila.

Análisis Sintáctico

Podemos definir al **Análisis Sintáctico** como el proceso de *determinar (decidir)* si una cadena X de componentes léxicos pertenece a un lenguaje generado por una gramática, lo cual se reduce a construír un AAS para la cadena X. Hasta el momento sólo hemos descrito la construcción de AAS de forma *descendente* (partiendo de la raíz y llegando hasta las hojas); pero también se pueden construír en orden *ascendente* (de las hojas hacia la raíz). Justamente, estas estrategias definen las clases de analizadores sintácticos en descendentes y ascendentes respectivamente. Los fundamentos en la construcción de herramientas (programas) que apartir de una descripción de CFG se obtenga un analizador sintáctico serán discutidos en el cap. 3. En lo que resta de este capítulo describiremos en términos generales el funcionamiento de una herramienta que a partir de una CFG G permitirá obtener un analizador sintáctico que reconozca expresiones del lenguaje que genera G conocida como *yacc*.

Como mencionamos anteriormente, un **atributo** correspondiente a un componente léxico es aquella información útil para la traducción,

Componente léxico	$\mapsto$	*atributos*
(token)		registro en T.S. (campos)

Los atributos son almacenados en la tabla de símbolos, un registro por cada *identificador* y cada campo dentro del registro contiene un atributo en particular. El valor de un atributo puede representar un tipo, una cadena, una localidad en memoria, etc. cualquier información útil para sintetizar la salida.

En este apartado presentaremos un formalismo llamado **Definición Dirigida por la Sintaxis (SDT)** para especificar la traducción (i.e. cómo sintetizar la salida). Este formalismo especifica la traducción de una construcción sintáctica *en función* de los atributos asociados con sus componentes sintácticos. Para el ejemplo que hemos presentado al inicio de este capítulo, recordemos que una etapa intermedia antes de generar el código objeto es traducir la expresión de entrada, que está en forma infija (EXP_{infix}), a su correspondiente forma sufija, la cual podemos definir inductivamente de la siguiente manera:

Notación sufija:
$sufix : EXP_{infix} \mapsto EXP_{sufix}$
Dada una expresión $E \in EXP_{infix}$:

1. $E \in Var$ ó $E \in Cons$, $sufix(E) = E$
2. E tiene la forma $E_1 op E_2$, $(op \in \{+, *, -, /\})$,
   $sufix(E) = sufix(E_1)sufix(E_2)op$
3. E es (E_1), $sufix(E) = sufix(E_1)$

Var y $Cons$ son los conjuntos que contienen las *variables* y *constantes* (i.e. átomos) respectivamente[4].

Esta función toma como argumentos expresiones (cadenas) y nos regresa a su vez cadenas, implícitamente está involucrada la operación de *concatenación* entre las subexpresiones. Por ejemplo, para las expresiones que construye $L(G_0)$, la traducción de una cadena a su correspondiente forma sufija se muestra abajo

$$\begin{aligned} \textit{sufix}(\text{“}4-3+1\text{ ”}) &= \textit{sufix}(\text{ “}4-3\text{”})\ \textit{sufix}(\text{ “}1\text{”})\ \text{“}+\text{”} \\ &= \textit{sufix}(\text{ “}4\text{”})\ \textit{sufix}(\text{ “}3\text{”})\ \text{“}-\ 1\ +\text{”} \\ &= \text{“}4\ 3\ -\ 1\ +\text{”}. \end{aligned}$$

Podemos apreciar que en este ejemplo, los valores de los atributos asociados a los símbolos gramaticales (en este caso, dígitos y operadores) son *ellos mismos*, i.e. el caracter (ascii) que representan. La función *sufix* fundamentalmente recolecta y concatena los componentes léxicos siguiendo *cierto orden*. Esto quedará más claro cuando coloquemos los atributos explícitamente en los AAS asociados a la expresión correspondiente.

Una definición dirigida por la sintaxis (SDT) utiliza una CFG y a cada símbolo de la CFG le asocia un conjunto de atributos y a cada producción le

[4] Hasta el momento no hemos considerado la presencia de identificadores (variables) dentro de las expresiones aritméticas simples, eso lo haremos más adelante.

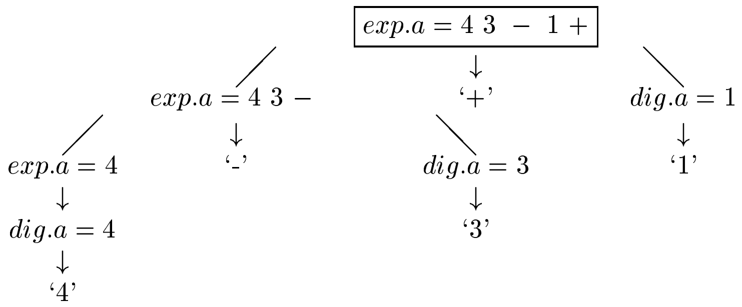


Figura 1.8: AAS con anotaciones y cálculo de los valores de los atributos en los nodos.

asocia un conjunto de *reglas semánticas* para calcular los valores de los atributos involucrados con los símbolos que ocurren en esa producción. La traducción es la transformación de una cadena de entrada x a una cadena de salida x^c, la cual podemos especificar de la siguiente forma:

1. Se construye un AAS para x,
2. Sea n un nodo del AAS para x, etiq. con $X \in (Vn \cup Vt)$: $X.a$ indica el valor del atrib. a de X en n.
3. El valor de $X.a$ se calcula mediante la regla semántica para a asociado con la producción de X en n.

Un AAS que muestra los valores de los atributos en cada nodo lo denominaremos AAS con anotaciones. A continuación mostramos una SDT para nuestra gramática G_0 y para una expresión de entrada indicamos un ASS con anotaciones, la salida (traducción) será obtenida aplicando las reglas semánticas

Producción	Regla Semántica
$exp \to exp_1\ +\ dig$	$exp.a := exp_1.a \mid\mid dig.a \mid\mid$ '+'
$exp \to exp_1\ -\ dig$	$exp.a := exp_1.a \mid\mid dig.a \mid\mid$ '-'
$exp \to dig$	$exp.a := dig.a$
$dig \to 0$	$dig.a :=$ '0'
...	...
$dig \to 9$	$dig.a :=$ '9'

Nótese que aquí estamos indicando la operación de concatenación mediante el símbolo '$\mid\mid$'. Los subíndices para algunos símbolos gramaticales (*exp* en el ejemplo) permiten indicar su *posición* y distinguirlos en la parte derecha (regla semántica) si es que ocurren varios. El atributo corespondiente es indicado por $.a$.

A partir de la figura 1.8 es inmediato afirmar que, *un recorrido en orden posterior* (en profundidad) del AAS con anotaciones nos permite obtener los valores de los atributos. Recordemos que el recorrido de un árbol en orden posterior para un nodo n, primero se visitan los nodos hijos en orden posterior

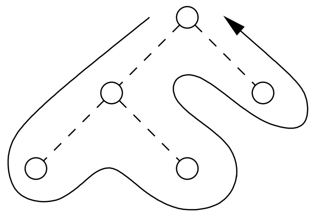


Figura 1.9: Recorrido en profundidad de un árbol

(de izq. a der.) y se visita en nodo padre n. Las reglas semánticas, por tanto, se evalúan en un nodo dado cuando todos los nodos hijos hayan sido visitados. Esto lo sintetizamos en la fig.1.9.

Es importante indicar las relaciones de dependencia que existen entre los atributos para calcular sus valores. Decimos que un atributo es *sintetizado* para un nodo n si su valor se determina a partir de los valores de los atributos asociados a sus nodos hijos. Generalmente los atributos sintetizados se pueden calcular mediante un sólo recorrido. De nuestro ejemplo anterior, fácilmente podemos observar que todos los atributos son sintetizados. Se define un atributo como *heredado* si su valor depende de un nodo ancestro ó un nodo hermano, por ejemplo, supongáse que extendemos la gramática G_0 de la siguiente forma:

$G_3 =< Vn_0 \cup \{Prog, L, decl, Asg, tipo, id, f\}, Vt_0 \cup \{int, char, :, `,', a, b, \ldots, z\},$
$P_3, S >$
$S = Prog$

$$P_3 = \left\{ \begin{array}{rll} Prog \rightarrow & decl : L; Asg & (1) \\ decl \rightarrow & int|char & (2) \\ L \rightarrow & L, id|id & (3) \\ Asg \rightarrow & id := exp & (4) \\ exp \rightarrow & exp + f & (5) \\ & |exp - f & (6) \\ & |f & (7) \\ f \rightarrow & dig|id & (8) \\ dig \rightarrow & 0|1|2|3|4|5|6|7|8|9 & (9) \\ id \rightarrow & a|b|\ldots|z & (10) \end{array} \right\}$$

(Obviamente, los conjuntos indicados por el subíndice 0 son los mismos de G_0, y por tanto la numeración de las producciones se recorre).

Estamos incorporando algunas producciones nuevas: se deja al lector verificar que el lenguaje que genera G_3 son sentencias compuestas por una parte de declaración de identificadores (variables) seguida por una expresión de asignación; la cual se compone de un nombre de identificador seguida por el símbolo de asignación (':=') y una expresión aritmética sencilla de las que genera G_0.

Dado el atributo $.t$ que denota el tipo para los componentes sintácticos, se muestra a continuación como se *hereda* para los correspondientes nodos en el AAS con anotaciones

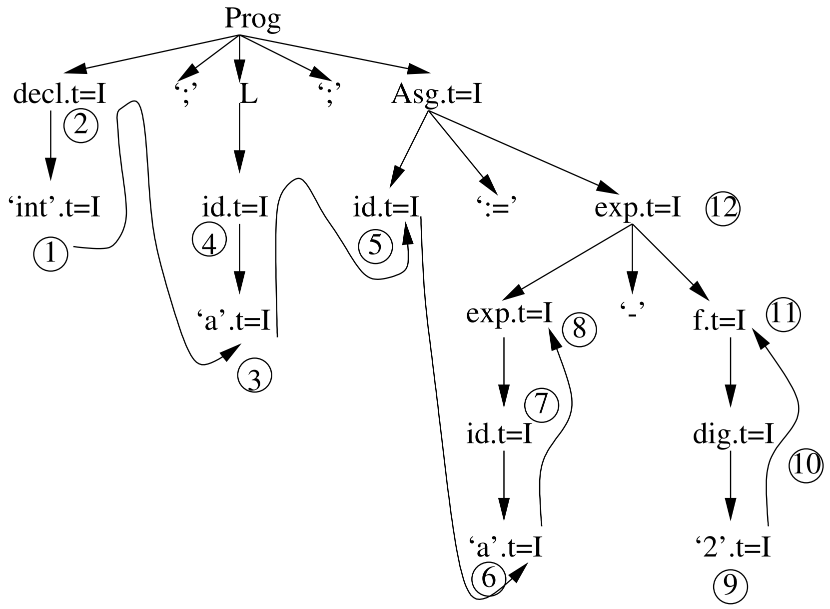


Figura 1.10: Atributos heredados para una expresión de $L(G_3)$, las flechas indican la dependencia y los números encerrados en un círculo indican el orden que debe seguirse para evaluar los valores de los atributos.

Ejercicios 5 *Describa cuáles deben ser las reglas semánticas correspondientes en una SDT para G_3.*

Un **Esquema de Traducción** es una CFG con acciones semánticas en la parte derecha de las producciones pero difiere de una def. dirigida por la sintaxis en que el *orden de evaluación de las reglas semánticas* se muestra explícitamente. Las reglas semánticas se expresan generalmente con código del lenguaje de implantación (subrutinas, funciones, etc.), i.e. está orientado a describir los detalles de implantación. Un esquema de traducción es, por tanto, la especificación del traductor.

A continuación se muestra un esquema de traducción para G_0:

Producción	Acción Semántica
$exp \rightarrow exp + dig$	{ *print('+')*}
$exp \rightarrow exp - dig$	{ *print('-')*}
$exp \rightarrow dig$	
$dig \rightarrow 0$	{ *print('0')*}
...	...
$dig \rightarrow 9$	{ *print('9')*}

La herramienta conocida como YACC (*Yet Another Compiler Compiler*) permite especificar esquemas de traducción, las reglas semánticas son código de lenguaje C y como salida ofrece un archivo `.c` que contiene la codificación del analizador sintáctico que reconoce las expresiones del lenguaje generado por la CFG correspondiente. A continuación se presenta una introducción al uso de esta herramienta.

Una especificación YACC[5] consta de 3 partes bien diferenciadas:

Parte de iniciación (declaraciones)
%%
Reglas de Traducción
%%
Código auxiliar

El funcionamiento para construír un analizador se esquematiza abajo:

especificación de YACC (*parser*.y) $\longrightarrow$ [compilador de YACC] $\longrightarrow$ *parser*`.tab.c` *parser*`.h`

parser`.tab.c` $\rightarrow$ [gcc] $\rightarrow$ ejecutable!

parser.y es el nombre del archivo de entrada que contiene una especificación YACC, esta herramienta da como resultado (si no hay errores) los archivos `.tab.c`

[5] Utiizaremos indistintamente los nombres bison y yacc, ya que bison es el nombre de la versión GNU para yacc.

y .h que son el analizador sintáctico (el programa generado como tal) y un archivo de encabezado de definiciones, respectivamente[6].

Por ejemplo, una especificación YACC para G_0 se muestra a continuación:

```
%{
  /* inxcalc1.y: esta parte encerrada por `%{' y `%}' puede contener
  declaraciones globales, que ser'an insertadas en el prog. final*/
#include <ctype.h>
#include <stdio.h>
%}
%token DIGITO

%%

linea : exp '\n' {printf("%d\n", $1); }
      ;
exp   : exp '+' dig {$$ = $1 + $3; }
      | exp '-' dig {$$ = $1 - $3; }
      | dig
      ;
dig   : DIGITO ;

%%

/* funci'on punto de entrada del analizador l'exico */
yylex() {
  int c;
  c = getchar();
  if (isdigit(c)) {
    yylval = c - '0'; /* <-- IMPORTANTE ! */
    return DIGITO;
  }
  return c;
}

yyerror (char *s) { /* invocado por yyparse ante un error */
   printf ("%s\n", s);
}

main() {
  yyparse(); /* punto de entrada del analizador sint'actico */
}
```

[6]Este va a contener todo lo que se haya colocado dentro de '%{' y '%}' así como las definiciones de **tokens**, los cules son muy importantes pues permiten la comunicación entre el a. léxico y el a. sintáctico.

Primero se define un token para indicar el símbolo gramatical que permite derivar dígitos, a continuación, la primer regla de producción indica una `linea` compuesta por una expresión aritmética y finalizada por ';', la lectura de las producciones que siguen es directa, las reglas semánticas permiten evaluar el valor de la expresión, por lo tanto los atributos serán el valor (entero, por default) asociado a cada componente. '`$$`' denota el valor de atributo para el símbolo gramatical a la izq. (que nombra a la producción, el nodo padre en el AAS), mientras que cada `$`acompañado de un índice indica el valor de atributo para cada símbolo gramatical a la derecha de ':' y de izq. a derecha: por ejemplo, para la 2da regla gramatical, `$1` indica a `exp`, `$2` a '-', `$3` a `dig`; y así sucesivamente.

El analizador léxico en este caso es una función que establece como valor del atributo[7] el valor entero del componente léxico (caracter en este caso) siempre y cuando represente un dígito y devuelve el token (para poder ubicarlo dentro de los símbolos gramaticales que ocurren en las producciones. En otro caso, simplemente devuelve el caracter y como tal dentro de la gramática correspondería a un operador + ó -. Note el uso de la variable global `yylval` predefinida por bison, sirve para asignar el valor del atributo al componente léxico actual.

A continuación está la función para indicar la presencia de errores, la cual es invocada por el analizador cuando alguna expresión no pertence al lenguaje derivado por G_0.

La función `main()` simplemente invoca al punto de entrada del analizador, nombrada `yyparse`.

El siguiente ejemplo enriquece la gramática para ahora evaluar expresiones aritméticas más complejas.

```
/* inxcalc2.y : Infix notation calculator */

%{
#define YYSTYPE double
#include <math.h>
%}

/* BISON Declarations */
%token NUM

%left '-' '+'  /* lower priority */
%left '*' '/'  /higher priority */
%left NEG     /* negation--unary minus */
%right '^'    /* exponentiation - the highest priority for this set */

/* Grammar follows */
%%
input:    /* empty string */
```

[7] A menos que se indique explícitamente otro tipo, por default bison asume que los valores de atributos para los símbolos gramaticales son enteros.

```
     | input line
     ;

line:     '\n'
     | exp '\n'  { printf ("\t%.10g\n", $1); }
     | error '\n' { yyerrok; }
     ;

exp:      NUM                { $$ = $1;         }
      | exp '+' exp        { $$ = $1 + $3;    }
      | exp '-' exp        { $$ = $1 - $3;    }
      | exp '*' exp        { $$ = $1 * $3;    }
      | exp '/' exp        { $$ = $1 / $3;    }
      | '-' exp  %prec NEG { $$ = -$2;        }
      | exp '^' exp        { $$ = pow ($1, $3); }
      | '(' exp ')'        { $$ = $2;         }
     ;
%%
#include <ctype.h>
#include <stdio.h>

      /* error handling function */
int yyerror (char *s) {
          printf("%s\n", s);
          return 0;
     }

yylex ()     {
       int c;

       /* skip white space  */
       while ((c = getchar ()) == ' ' || c == '\t')
         ;
       /* process numbers   */
       if (c == '.' || isdigit (c))
         {
           ungetc (c, stdin);
           scanf ("%lf", &yylval);
           return NUM;
         }
       /* return end-of-file  */
       if (c == EOF) /* a CNTRL-D from keyboard */
         return 0;
       /* return single chars */
       return c;
 }
```

```
main() {
   yyparse();
}
```

La parte de declaraciones llama la atención: mediante la macro `YYSTYPE` se especifica el tipo de valores para los atributos, en este caso números de punto flotante de precisión doble. A continuación, se declara el token principal para representar números. Ese token nombra al componente gramatical que será emitido por el analizador léxico una vez que empate en el archivo de entrada un lexema que cumpla con el patrón de número: secuencia de dígitos (parte entera) a continuación un '.' y otra secuencia de dígitos (parte fraccionaria). La asociatividad y prioridad de los operadores aritméticos son declaradas en seguida: el orden de presentación indica de menor a mayor prioridad. Nótese que para la asociatividad de el operador '-' unario se hace mediante el nombre `NEG`. Las reglas gramaticales y sus acciones semántica vienen a continuación: el primer conjunto de producciones es importante, ahora `input` es el símbolo inicial, el cual puede derivar una expresión nula (ϵ) ó una secuencia de `lines`; quien por su parte puede tener la forma de una expresión vacía (note la diferencia con respecto a la producción ϵ anterior) ó una expresión aritmética finalizada por un retorno de carro (en cuyo caso la regla semántica despliega el valor de la expresión) ó la presencia de un *error*, lo cual es indicado por el token especial (reservado por bison) `error`. Este caso es ilustrativo para la recuperación de errores, esta producción se aplicará cuando falle al intentar reducir un símbolo gramatical (en este caso, `exp`), i.e. cuando una expresión no esté sintácticamente bien construída. Esta producción puede leerse como "cualquier otra cosa que no sea una expresión y que termina con retorno de carro es un error"; su acción semántica `yyerrok` significa recuperarse del error encendiendo una bandera.

El conjunto de casos asociados a la producción `exp` se explican por sí mismos, al igual que sus reglas semánticas, por lo que ya hemos mencionado para el anterior ejemplo.

Podemos apreciar que la función que realiza el análisis léxico es engorrosa de escribir, lo que nos permite introducir el manejo de una herramienta para construír analizadores léxicos a partir de la definición de *expresiones regulares*. Aún cuando revisaremos con suficiente rigor esta parte en el sig. capítulo, conviene dar una definición breve:

Dada $a, b \in \Sigma$, una expresión regular (e.r.) se construye de acuerdo a las siguientes cláusulas:

1. ϵ es una e.r. y *designa* al conj. $\{\epsilon\}$.
2. a es una e.r. y denota al lexema 'a'.
3. Si R, R_1 son e.r., también lo son:
   (a) RR_1 concatenación, ej. si $R = \{a\}, R_1 = \{b\}$ entonces $RR_1 = \{ab\}$.
   (b) $R|R_1$ alternativa, ej. $R|R_1 = \{a, b\}$.

(c) R^+ cerradura positiva, todas las cadenas que se pueden construír concatenando Rs con al menos una R: ej, $R^+ = \{a, aa, aa\ldots a, \ldots\}$.

(d) R^* cerradura Kleene, $R^+ \cup \epsilon$, $R^* = \{\epsilon, a, aa, aa\ldots a, \ldots\}$.

(e) (R) una e.r. entre paréntesis denota el conj. de cadenas obtenida a partir de R (para agrupación y mejorar legibilidad).

4. Ninguna otra expresión es una e.r.

Se dice que el conjunto de cadenas denotadas por una e.r. es un lenguaje regular. Nuestra definición, aunque muy informal, nos permite por ejemplo reconocer que toda cadena que representa un número de punto flotante, por ejemplo "43.0567", la podemos describir en general mediante una e.r. Esto mismo para los *identificadores* (i.e. secuencias de letras y dígitos que siempre comienzan por una letra) los cuales describen el conjunto de variables, palabras reservadas, etc. del lenguaje fuente.

Incorporemos un a. léxico a nuestro ejemplo anterior definido en flex, el cual, a partir de una especificación en términos de reglas patrón (e.r) acción nos construye un a. léxico (código fuente c) listo para ser compilado y enlazado. Una definición flex, al igual que su primo bison, tiene 3 partes: declaraciones, reglas y código auxiliar. Se pueden crear programas independientes (i.e. en la parte de código auxiliar puede ir la función `main()`) ó integrarlo como módulo en un programa más complejo (que es lo que ilustraremos a continuación).

```
/* scanner2.l: analizador l'exico para inxcalc2.y */
%option noyywrap
%option yylineno

/* definiciones */
num   [0-9]+\.? | [0-9]*\.[0-9]+

%%

[ ]   {  /* come los espacios en blanco */
{num} { sscanf(yytext, ``%lf'', &yylval);
        return(NUM); }
\n|.  { return yytext[0]; }

%%
```

Esta especificación es muy simple: la opción `%noyywrap` indica lo que debe hacer el a.léxico por omisión al encontrar el final del archivo de entrada: simplemente regresa. A continuación se define primero una e.r. de nombre `num` cuya definición permite reconocer números: secuencia de al menos un dígito seguida de un '.' opcional ó una secuencia de 0 ó más dígitos y a continuación el '.' decimal y una secuencia de al menos un dígito. Nótese las acciones que se realizan: se establece como **valor de atributo** para el componente léxico actual el número real que éste denota asignándolo a la variable `yylval`, (la cual es una variable global volátil reservada por bison con éste propósito) y a continuación, regresa (al invocador, en éste caso al parser) qué **token** acaba de ser aceptado. Este nombre de token debe corresponder con los declarados en la parte superior en el módulo bison. El lexema que conforma el último token examinado está en la variable (volátil reservada por flex) `yytext`[8].

El caso por default (el final de la línea ó cualquier otro caracter) es indicado por la segunda regla, la cual simplemente regresa el caracter leído.

Para consultar una tabla detallada de las posibles e.r.'s en la sintaxis permitida por flex puede ver 2.1, en la pág 39.

Para que los módulos de a.léxico construído con flex y el correspondiente a.sintáctico producido a su vez mediante bison puedan interactuar, se tienen que cumplir algunos detalles:

1. Se debe compilar primero el módulo bison, para que produzca el archivo ".h" el cual contiene la definición de los tokens como constantes simbólicas, en nuestro ejemplo

   ```
   % bison -d incalc2.y
   ```

   producirá los archivos `incalc2.tab.c` e `incalc2.h`.

2. Incluír en la parte de encabezado en el módulo para flex (en nuestro ejemplo `scanner2.l`) la siguiente línea:

   ```
   #include ``incalc2.h''
   ```

   dentro de los delimitadores '`%{` ' y '`%}` '. Compilar ahora el módulo flex:

   ```
   % flex scanner2.l
   ```

3. Ahora ya se puede producir el archivo ejecutable.

(Obviamente, se elimina la función `yylex()` que estaba en el módulo `incalc2.y` ya que será reemplazada por la que produzca el módulo para flex).

Como se ha descrito, este ejemplo sencillo funciona teniendo como entrada el teclado. Recordemos que todo programa en unix tiene asociados 3 archivos por default: para entrada estándar (*stdin*), salida estándar (*stdout*) y salida estándar de errores (*stderr*), asociados respectivamente al teclado, la terminal actual y la consola. Se pueden sin embargo *redireccionar* o asociar a archivos de texto en disco, para lo cual se recomienda lo siguiente:

- Flex tiene predefinidos los archivos `yyin, yyout, yyerr` para entrada, salida y salida de errores, respectivamente. Suponiendo por simplicidad que el módulo parser contiene el punto de entrada del programa principal (i.e. la función `main()`) entonces debe incorporarse la sig. línea en el encabezado:

[8] Use siempre `strcpy()` si necesita copiar el lexema actual a una variable auxiliar, como es en el caso de almacenarlo en la tabla de símbolos

```
extern FILE *yyin, *yyout, *yyerr;
```

la cual permite al enlazador evitar reportar la presencia de referencias ambigua ante la presencia de tales variables. Una vez esto, la función `main()` puede abrir los archivos, cuyo nombre pueda ser indicado por el usuario desde la línea de órdenes:

```
:
main(char **argv, int argc) {

  if (argc > 1) {
    yyin = fopen(argv[1],''r'');
    yyout = fopen(argv[2], ``w'');
  }
   else {
     printf(``Debe dar el nombre de archivo de entrada\n'');
     exit 1;
   }
   ini_tablasim(); /* iniciar la tabla de simbolos */
   yyparse();  /* punto de entrada del compilador */

   close(yyin);
   close(yyout);
}
```

El código mostrado es tan sólo el esqueleto y no está protegido: se debe verificar adecuadamente el número de los argumentos recibidos en la línea de órdenes al ejecutar el programa, así como preveer el resultado de invocar las funciones de biblioteca para manejo de archivos (`fopen()`).

En general, la interacción entre los módulos de análisis léxico y sintáctico construídos mediante *flex* y *bison* respectivamente funciona de la siguiente manera (vea la fig. 1.11):

1. El a. sintáctico (`yyparse()`) intenta reducir la producción inicial para cada símbolo gramatical en la parte derecha de la producción si es terminal invoca al analizador léxico solicitándole el siguiente token del archivo de entrada. Si es no terminal, se invoca sucesivamente a la regla de producción correspondiente.

2. El a.léxico (`yylex()`) lee a partir del archivo de entrada la siguiente palabra que cumpla con algún patrón especificado en sus reglas (definiciones de expresiones regulares). Al encontrar concordancia con el primer patrón (si es que podrían existir varios) ejecuta el código especificado en su parte de *acción*, lo cual en general debe de establecer el(los) valor(es) de atributos asociados con ese lexema, e indicar de qué token se trata, para que el a.sintáctico pueda ubicarlo dentro de sus reglas de producción. Por ello,

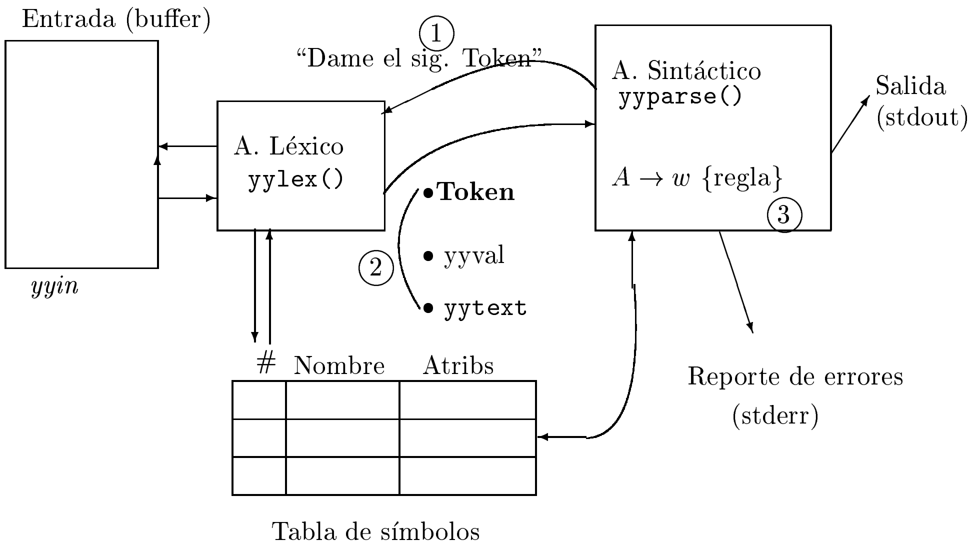


Figura 1.11: Interacción entre los módulos de análisis léxico y sintáctico construídos mediante *flex* y *bison* respectivamente.

los nombres de token deben concordar entre ambos módulos. Típicamente, ante la presencia de *identificadores* en el archivo de entrada (correspondiendo a nombres de variables, constantes, funciones, procedimientos etc. en el prog. fuente) se actualiza la tabla de símbolos, generalmente almacenando el nombre de identificador y sus valores de atributos correspondientes que puedan ser inferidos a partir de su descripción léxica (reglas de convención de escritura).

3. Al reducir con éxito todos los símbolos en la parte derecha de la producción actual, se procede a ejecutar la regla semántica correspondiente, la cual puede inclusive estar definida por partes y cada una de las cuales estar directamente incrustada entre los símbolos gramaticales de la producción actual, indicando así el orden de evaluación.

Ahora podemos definir un pequeño pero ilustrativo lenguaje de programación imperativa dentro de la familia pascal, fundamentalmente extendiendo a nuestra gramática G_3. A partir de este lenguaje se describirá un esquema de traducción para construír un compilador sencillo que produzca código de máquina de pila. Describiremos primero el lenguaje que estamos interesados informalmente:

- Un programa del lenguaje consta de dos partes: una parte de *declaración de variables* y un *bloque de proposiciones* (que constituye el programa propiamente).
- La parte de declaración está formada por *listas de declaraciones* separadas por ';', cada una de las cuales comienza por una palabra reservada que

indica el *tipo* y a continuación una lista de nombres de variables separadas por ‘,’. El tipo puede ser ‘int’, ‘char’ ó ‘real’ para indicar enteros, caracteres y números de punto flotante, respectivamente.

- Una *proposición* (enunciado ejecutable en el programa) puede ser de los siguientes casos:

    1. Un enunciado de *asignación*, el cual está compuesto por un nombre de variable (previamente definido), el operador ‘:=’ y a continuación una *expresión aritmética*.
    2. Un enunciado de *decisión*, el cual está formado por la palabra reservada ‘if’, a continuación una *expresión booleana*, en seguida la palabra ‘then’ seguida por una proposición (caso verdadero) y la palabra ‘else’ seguida por otra proposición (caso falso).
    3. Una proposición de *iteración*, formada por la palabra ‘while’ seguida de una expresión booleana y a continuación ‘do’ seguida por una proposición (cuerpo del bucle).
    4. Un *bloque*, el cual está formado por una lista de proposiciones (de todas las anteriores e inclusive también de bloque) separadas por ‘;’ y encerradas entre ‘begin’ y ‘end’.

- Una expresión booleana está constituida por expresiones aritméticas relacionadas por los operadores infijos ‘<’, ‘>’, ‘=’, ‘>=’, ‘<=’, ‘<>’, ‘@’ (conjunción), ‘#’ (disyunción) y ‘~’ (negación); encerrada entre paréntesis.

- Una expresión aritmética puede ser un átomo (variable, constante) ó puede estar constituída a su vez por subexpresiones relacionadas por los operadores aritméticos ‘+,*,-,/,div, mod, exp’, etc. posiblemente encerradas entre paréntesis.

Ejercicios 6 *Defina la CFG correspondiente para el lenguaje de programación descrito.*

Recordemos que el propósito del traductor que se desea construír para este lenguaje es producir código ejecutable para un mecanismo de cómputo abstracto (el modelo de máquina de pila). Recordemos también que el proceso de traducción lo podemos describir mediante la evaluación de reglas semánticas para calcular los valores de atributos asociados a los símbolos gramaticales en cada producción de la gramática dada. Por tanto, el problema se reduce a resolver:

1. ¿Que atributos deben establecerse para conducir la traducción? ¿Que tipos para los valores de atributos son convenientes?

2. ¿Cuál es la traducción (codificación) para las expresiones más simples (atómicas) y que relación tienen con la traducción correspondiente a las expresiones más complejas?

La primer pregunta la podemos resolver de acuerdo a los tipos de valores que maneje la máquina abstracta, por simplicidad podemos asumir una representación directa (aunque puedan ser codificadas de manera automática e interna por la máquina), reduciendo así a discutir por ejemplo para tipos de valores de atributos enteros.

Para la segunda pregunta, pudimos apreciar en base a la estructura *inductiva* de las producciones en las CFG, la *interpretación* de expresiones compuestas no ambiguas depende de la interpretación asociada a las expresiones básicas ó atómicas; de tal suerte que las reglas sem
anticas en los esquemas de traducción que se discutirán en este curso son *isomorfas* a la estructura sintáctica del lenguaje generado por la CFG. Este aspecto quedará claro en el siguiente apartado.

1.4 Arquitectura de la Máquina Abstracta de Pila

Hemos descrito informalmente el funcionamiento de una arquitectura de computadora en la cual se ejecutarán los programas objeto producidos por el traductor de expresiones. En este apartado describiremos a mayor detalle el funcionamiento de la máquina abstracta, también extenderemos el lenguaje fuente para soportar las construcciones del lenguaje de programación descrito al final del apartado anterior.

1. Las estructuras para almacenamiento principales de la máquina son:

   (a) **Stack:** donde se almacenan los datos para ser evaluados; el resultado de toda *operación primitiva* queda en el tope del stack.

   (b) *Lista de instrucciones:* el programa está constituído por una lista secuencial *indexada* (i.e. cada entrada tiene asociada una localida/dirección) de instrucciones.

   (c) *Área de Datos:* area de memoria (una tabla) que mantiene localidades asociadas al contenido de los identificadores (variables, constantes, etc.). La dirección ó localidad de alguna entrada de la tabla (i.e. dato) puede colocarse en el stack de tal forma que se permita *asignar* valores a dichos datos (referenciados por la dirección que ocupan en la tabla).

2. La máquina soporta un conjunto de instrucciones básicas asociadas a las operaciones aritméticas (las cuales toman sus argumentos del tope del stack), también tiene definidas operaciones para manipulación del stack (colocar referencias, depositar datos, desempilar, asignar, copiar) y operaciones para flujo de control (con respecto a las instrucciones: saltos condicionales o incondicionales, detención de la ejecución, etc.).

Primitivas Aritméticas:
Consideremos por simplicidad (pero sin pérdida de generalidad) que los datos son de tipo entero. Cada operación aritmética tiene una instrucción correspondiente en la máquina: ADD adición, SUB diferencia, MUL producto, DIV división, MOD residuo de división entera, y EXP potencia; por mencionar las más importantes. Todas estas instrucciones toman sus k argumentos a partir del tope de la pila, los consumen y empilan el resultado (evidentemente k indica la aridad ó número de argumentos que espera la nstrucción). Ocurre una **excepción** (o error de ejecución) si no hay suficientes argumentos en la pila, ó si se intenta evaluar algo como $0/0$, o cuando el resultado de la evaluación excede la capacidad discreta de representación.

Manipulación del Stack:
Recordemos que existe una diferencia entre el significado de los identificadores en el lado izquierdo y el lado derecho en una expresión de asignación:

$$\underbrace{id}_{\text{parte izquierda}} \quad := \quad \underbrace{id * 56}_{\text{parte derecha}}$$

el lado derecho especifica un **valor**, mientras que el lado izquierdo indica **dónde** debe **almacenarse** dicho valor.

Se tienen por tanto las siguientes primitivas para almacenar datos (valores) ó *referencias* (localidades, posiciones) en el stack:

- PUSH *val* - inserta un valor (entero) constante en el tope del stack,
- PUSHD *addr* - empila el *contenido* de la dirección ó localidad de memoria del área de datos indicada por *addr*.
- PUSHI *addr* - empila la dirección (referencia) de la localidad indicada por *addr*.
- CPY - duplica el tope del stack.

La primitiva ASG codifica la operación de asignación: asume que en la pila se encuentran un valor (lado derecho) en el tope y una referencia (lado izquierdo) inmediatamente debajo, y coloca el valor en la dirección apuntada por la referencia, desempilando ambos.

Recordemos de la CFG G_3 la producción (4) para reconocer expresiones de asignación:

$$Asg \rightarrow id`{:=}\text{'}\ exp$$

recordemos también que la forma sufija de las expresiones aritméticas nos conducen la traducción a código de máquina de pila: las expresiones atómicas tienen una interpretación inmediata (mediante instrucciones asociadas a empilar los datos que éstos átomos representan), y las instrucciones de máquina correspondientes a los operandos son colocadas al final ,(i.e. la traducción de una expresión es la *concatenación* en orden inverso de la traducción de sus subexpresiones). Consideremos que la emisión de código compilado se realiza sobre un archivo secuencial (texto) mediante la función `emite()`, la cual fundamentalmente escribe sobre el archivo de salida las instrucciones de máquina correspondientes a la subexpresión que está siendo analizada. Por ejemplo, podemos sintetizar la traducción para las expresiones de asignación de la sig. forma:

$Asg \rightarrow$	Id ':='	{ emite("PUSHI", \$1); }
	exp	{ emite("ASG"); }

Nótese que primero se emite la instrucción que empilará una referencia al dato Id, indicado por su valor de atributo. El lector puede inferir que en este caso un valor de atributo conveniente es la posición que el identificador ocupa en la tabla de símbolos, de tal suerte que la traducción de identificadores (variables) es una instrucción de empilamiento, ya sea de la referencia ó el contenido de la posición del identificador en la tabla de símbolos. Puede considerarse por tanto, que la tabla de símbolos es *vaciada* adecuadamente en el área de datos de memoria en la máquina.

Una vez que se ha emitido la traducción de la parte izquierda de la expresión de asignación, se espera que la regla semántica asociada a exp emita su código correspondiente; entonces se emite la instrucción que codifica la operación, en este caso de asignación. El lector puede inferir las reglas semánticas asociadas a la traducción de expresiones aritméticas.

Flujo de control:

La máquina ejecuta las instrucciones en secuencia consecutiva a menos que se le indique que ejecute una instrucción no consecutiva mediante un salto condicional o incondicional. Se permite especificar el destino del salto de manera simbólica, i.e. la máquina maneja *etiquetas*.

- LABL *etiq* - establece un destino(*punto de entrada*) a ser argumento de instrucciones de salto, no tiene otro efecto
- JMP *etiq* - salto incondicional a la instrucción etiquetada por *etiq*
- IFALSE *etiq* - si el valor en el tope del stack se evalúa a **falso** (i.e., 0) cambia el flujo de control a la instrucción etiquetada por *etiq* desempilando el valor, de lo contrario continúa con la siguiente instrucción
- IFTRUE *etiq* - salta a *etiq* si el valor (desempilado) en el tope del stack es distinto de 0
- HALT - detiene la ejecución (permite indicar la finalización del programa).

Describimos a continuación unos esqueletos útiles para la traducción de proposiciones de decisión e iteración, respectivamente.

IF:
⋮
exp
IFALSE $etiq_1$
$prop_1$
JMP $etiq_2$
LABL $etiq_1$
$prop_2$
LABL $etiq_2$
⋮

prop →	'if' *exp*	{ `etiq1 = nuevaetiq();` `emite('IFALSE', etiq1);` }
	'then' $prop_1$	{ `etiq2 = nuevaetiq();` `emite('JMP', etiq2);` `emite('LABL', etiq1);` }
	'else' $prop_2$	{ `emite('LABL', etiq2);` }

Nótese que el esquema de traducción está capturando la *forma sufija* correspondiente para cada proposición. A su vez, puede verificarse fácilmente que un recorrido en orden del AAS nos permite generar el código correspondiente. Es importante indicar la presencia de las reglas semánticas incrustadas dentro de las producciones, indicando justamente lo que deba hacerse al reducir cada símbolo gramatical.

WHILE:
⋮
LABL $etiq_1$
exp
IFALSE $etiq_2$
$prop_1$
JMP $etiq_1$
LABL $etiq_2$
⋮

prop →	'while ('	{ `etiq1 = nuevaetiq();` `emite('LABL', etiq1);` }
	exp ')'	{ `etiq2 = nuevaetiq();` `emite('IFALSE', etiq2);` }
	$prop_1$	{ `emite('JMP', etiq1);` `emite('LABL', etiq2);` }

Ejercicios 7 Integración de las técnicas: *construya un compilador de una pasada que traduzca programas cuyo lenguaje fuente sea generado por G_p (o una CFG equivalente) y produzca como salida código para la máquina de pila descrita. Debe utilizar manejo de tabla de símbolos y verificación semántica de la concordancia en la construcción de las expresiones que involcran operadores (aritméticos). Utilice flex y bison.*

Capítulo 2

Análisis Léxico

El propósito del analizador léxico es obtener componentes léxicos ó lexemas (palabras con significado colectivo) a partir de una lectura secuencial del programa fuente.

($\rightarrow$ diagrama que ilustre la estructura de los componentes léxicos *(tokens)* del lenguaje).

Prog. dirigida por patrones: ejecutar las acciones activadas por patrones dentro de las cadenas (LEX *patrón - acción*)

Expresiones Regulares $\rightarrow$ *Autómata Finito*

• Un **patrón** es una *regla* que describe el *conjunto de lexemas* que pueden representar a un determinado componente léxico.
$Token \in S^*$ *(clase de equivalencia)*
Un patrón *empata* con cada cadena de S.

Token	Patrón	Ejemplo
const	*const*	`const`
rel	$< \mid > \mid <= \mid >= \mid <>$	`<, >, <=, >=, <>`
id	*letra(letra$\mid$dig)**	`pi, count, D2`
NUM	*(cualq. constante numérica)*	`3.14159, 0, 5.01E34`
literal	*(cualq. caracter entre " y ")*	`"hola"`

Componente léxico $\rightarrow$ atributos (traducción tokens).
$e := m * c^2$:

- $<$ **id**, $\rightarrow$ a la tabla de simb. para $e>$
- $<$ **opasgn**, $>$
- $<$ **id**, $\rightarrow$ a la tabla de simb. para $m>$
- $<$ **opmul**, $>$
- $<$ **id**, $\rightarrow$ a la tabla de simb. para $c>$
- $<$ **opexp**, $>$

- < **num**, 2>

Errores: *"panic-mode"* → elim. caracteres sucesivos hasta que el scanner pueda hallar un compo. léxico bien formado.
Acciones del reconocedor de errores léxicos:

1. eliminar un char extraño
2. insertar el char que falta
3. reemplazar un char incorrecto por uno correcto
4. intercambio chars

Métodos generales en la *implantación* de un a. léxico:

1. LEC: (generador de analizadores léxicos) a partir de una especificación dada en *expresiones regulares* (lectura de *buffer*). $\ddot\smile$
2. Escribir el scanner *a mano* (en un leng. prog. convencional). $\ddot{-}$
3. Escribirlo en ensamblador. $\ddot\frown$

Especificación de los Componentes léxicos:
Σ - alfabeto, Σ^* - todas las cadenas, ϵ - cadena nula. $s \in \Sigma^*, |s|$ - long. cadena (num. elems.) $|\epsilon| = 0$. $(s\epsilon = \epsilon s = s)$.
$s^0 = \epsilon, s = s^{i-1}s$

Partes de una cadena:

TÉRMINO	DEFINICIÓN
prefix s	cad. obtenida elim. $n \geq 0$ simbs. desde la derecha de s
sufix s	cad. obtenida elim. $n \geq 0$ simbs. desde la izq. de s
substr s	cad. obtenida suprim. un prefix y un sufix de s
" proper s	subcad. *no vacía* x de s, $s \neq x$
subseq s	cad. obtenida elim. $n \geq 0$ simbs. no necesariam. contiguos de s

Operaciones sobre lenguajes:

OPERACIÓN	DEFINICIÓN
$L \cup M =$	$\{s \mid s \in L \vee s \in M\}$
$LM =$	$LM = \{sm \mid s \in L \wedge m \in M\}$
cerr. Kleene: L^*	$\bigcup_{i=0}^{\infty} L^i$
cerr. positiva: L^+	$\bigcup_{i=1}^{\infty} L^i$

Ejemplo: $l = \{A, \ldots, Z, a, \ldots, z\}, D = \{0, 1, \ldots, 9\}$

1. $L \cup D$
2. $LD =$ conj. cadenas que constan de una letra y un dígito
3. $L^4 =$ conj. cadenas 4 letras
4. $L*$ todas las cadenas, incluyendo a ϵ
5. $L(L \cup D)^*$ - conj. cadenas (letras y díg.) que comienzan con una letra.

6. D^+ - conj. de las cadenas de 1 o más díg.

Def. Expresiones Regulares sobre Σ:

1. ϵ es E.R. y *designa* al conj. $\{\epsilon\}$
2. Si $\mathbf{a} \in \Sigma$, entonces $\mathbf{a}$ es una E.R. y designa $\{a\}$ (alas!: cadena, símbolo o E.R.)
3. Si $\mathbf{r}$ y s son E.R. representadas por $L(r)$ y $L(s)$, entonces,
   (a) $(r)|(s)$ es una E.R. repres. por $L(r) \cup L(s)$
   (b) $(r)(s)$ es E.R. repres. por $L(r)L(s)$
   (c) $(r)^*$ es E.R. repres. por $(L(r)^*)$
   (d) (r) es una E.R. repres. por $L(r)$ ('(' opcionales)

convenciones: $*$ mayor precedencia, asoc. izq.; la concatenación tiene siguiente precedencia y también asoc. izq.; | menor preced. y asoc. izq.

Def. Regulares:

$d_1 \rightarrow r_1$
$d_2 \rightarrow r_2$
$\ldots$
$d_n \rightarrow r_n$

d_i es un nombre distinto, cada r_i es una E.R. sobre $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$

LEX:

```
%{
 /* Def. constantes manifiestas
    LESS, LESQ, EQ, IF, THEN, ELSE, ID, NUM, OPREL */
%}
/* definiciones regulares */
delim       [ \t\n]
ws          {delim}+
letra       [A-Za-z]
dig         [0-9]
id          {letra}({letra} | {dig})*
num         {dig}+(\.{dig}+)?(E[+\-]?{dig}+)?

%%

{ws}        {/*ninguna acci'on, no rse regresa algo*/}
if          {return(IF);}
then        {return(THEN);}
else        {return(ELSE);}
{id}        {yyval = inst_id(); return(ID);}
{num}       {yyval = inst_num(); return(NUM);}
"<"         {yyval = LESS; return(OPREL);}
"<="        {yyval = LESQ; return(OPREL);}
```

EXPRESIÓN	EMPATA CON	EJEMPLO
c	any char c no operador	`a`
$\backslash c$	el char c literalmente	
"s"	la cadena s literalmente	`"**"`
.	any char excepto newline	`a.*b`
^	el comienzo de línea	`^1z3`
$	fin de línea	`0zv$`
[s]	any char en s	`[abc], [0-9]`
[^s]	any char NO en s	`[^abc], ^[0-5]`
*r**	0 o más r	`a*`
r+	1 o más r	`a+`
r?	0 o 1 r	`a?`
$r\{m,n\}$	m a n casos de r	`a{1,5}`
$r_1 r_2$	r_1 y entonces r_2	`ab`
r_1 r_2	r_1 ó r_2	`a|b`
(r)	r	`(a|b)`
r_1/r_2	r_1 cuando va seguida de r_2	`xyz/abc`

Figura 2.1: Principales definiciones regulares en flex.

```
"="         {yyval = EQ;   return(OPREL);}

%%

int inst_id() {
  /* colocar el lexema en la TS, el primer char est'a siendo apuntado
  por yytext, con longitud yylong, regresar la referencia (num
  localidad) donde queda*/

}


int ints_num() {
    /* TS[i++] = atoi(yytext); return(i); */
}
```

Capítulo 3

Análisis Sintáctico

Hemos descrito de manera general el proceso del análisis sintáctico como:

- el intento por obtener una derivación más por la izquierda (ó equivalentemente más por la derecha) para la expresión de entrada;
- el proceso de construcción de un AAS para la expresión de entrada.

Recordemos que la construcción del AAS correspondiente puede realizarse en dos formas:

- Partiendo de la raíz y alcanzando los nodos hojas, el cual es un proceso *descendente*.
- Comenzando por los nodos hojas y alcanzando eventualmente la raíz, i.e. en dirección *ascendente*.

Estas 2 formas para construír un AAS nos permitirán clasificar a los analizadores sintácticos en justamente ascendentes y descendentes. El propósito por tanto de éste capítulo es describir algoritmos eficientes para construír analizadores en cada clase y apreciar las gramáticas que puedan ser abordadas por cada clase, pudiendo comparar la capacidad expresiva de las mismas. En orden de expresividad (y complejidad), estudiaremos primero a la clase de analizadores descendentes.

3.1 Análisis Sintáctico Descendente

El propósito de esta clase de analizadores lo podemos describir como:

- intento por encontrar una *derivación más por la izq.* para la expresión de entrada, ó equivalentemente
- costruír el AAS partiendo de la raíz y alcanzando las hojas.

Ya hemos discutido ambos aspectos anteriormente. Considérese la descripción siguiente de un analizador sintáctico que denominaremos *descendente recursivo*:

- Sigue un orden fijo en la aplicación de las producciones, pudiendo sin embargo *retroceder* en la elección de otro caso[1]

- en cada paso de la derivación siempre intenta reducir el siguiente no terminal más por la izquierda.

Resulta importante mencionar que desde el punto de vista operacional un programa para análisis sintáctico que siga estas restricciones, ante una gramática **recursiva por la izquierda** irremediablemente caerá en un *ciclo infinito*:
ejemplo

Una CFG es *recursiva por la izquierda* si tiene al menos un no terminal A tal que existe una derivación $A \stackrel{+}{\Rightarrow} A\alpha$ para alguna forma de frase α.

Los tipos de Analizadores Sintácticos Descendentes (ASD) a mencionar son:

- **A.S. Por Descenso Recursivo (ASPDR)**: forma general, admite retroceso.

- **A.S. Predictivo** (ASP): ASPDR sin retroceso,
  $\rightarrow$ se debe reconocer, dado el símbolo inicial a de entrada y el no terminal A a expandir, cuál de las alternativas de producción

  $$A \rightarrow \alpha_1|\alpha_1|\alpha_2|\ldots|\alpha_n$$

  es la <u>única</u> que da lugar a una cadena que comience con a.

Por tanto, los requerimientos para que una CFG G sea abordada por un ASD son:

1. no debe ser ambigua

2. no debe ser recursiva por la izquierda

3. sus producciones no deben tener factores comunes por la izquierda

3.1.1 Transformaciones gramaticales

Eliminación de Recursión (inmediata) por la Izquierda De:

$$A \rightarrow A\alpha_1|A\alpha_2|\ldots|A\alpha_n|\beta_1|\beta_2|\ldots|\beta_m$$

A:

$$\begin{array}{c} A \rightarrow \beta_1 A'|\beta_2 A'|\ldots|\beta_m A' \\ A' \rightarrow \alpha_1 A'|\alpha_2 A'|\ldots|\alpha_n A'|\epsilon \end{array}$$

Factorización por la Izquierda De:

$$A \rightarrow \alpha\beta_1|\alpha\beta_2|\ldots|\alpha\beta_n|\gamma$$

[1] En el sentido que para un no terminal se tengan varias alternativas (partes derechas).

Por:

$$A \to \alpha A'|\gamma$$
$$A' \to \beta_1|\beta_2|\ldots|\beta_n$$

Algunas consideraciones para evitar/eliminar la ambigüedad
Diagramas de Transiciones para ASP:

- diagrama para cada no terminal
- etiquetas de aristas: componentes léxicos (terminales) y no terminales
- una transición etiq. con un terminal supone que se debe tomar dicha transición si ese componente léxico es el sig. símbolo de entrada.
- una transición etiq. con un no-terminal A es una invocación a la rutina para A

Construcción de diagramas:

- Eliminar la recursión por la izq. de la gramática G,
- Factorizar G por la izq.
- Para cada no terminal A:
  1. Crear un edo. inicial y uno final (de retorno)
  2. Para cada producción $A \to X_1X_2\ldots X_n$ crear un camino desde el edo. inicial al final con aristas etiquetadas $X_1, X_2, \ldots, X_n$. (ϵ sin avanzar la entrada)

Indeterminismo (ambigüedad) $\to$ retroceso.

3.1.2 ASDP LL

Implantación del ASDP no recursivo:

uso de un *stack* para almacenar los estados S cuando hay una transición con un no terminal saliendo de S e eliminando el topo de la pila al alcanzar el edo. final de un no terminal.

$\star$ determinar la producción que debe aplicarse a un no-terminal.

Comportamiento del programa (X, a):

1. Si $X = a = \$$: halt - success
2. Si $X = a \neq \$$: pop X, mover el apuntador de entrada al sig. elemento del buffer (se consume - una prod. acepta a a)
3. Si $X \in Vn$, se consulta $M[X, a]$ - prod. de X o una *prod. de error*:, $M[X, a] = \{X \to UVW\}$
   • sustituír X por UVW (U en el tope) - (efectuar acción semántica asociada)
   • si $M[X, a] =$**error**, invocar la rutina para recuperación de error asociada

Algoritmo ASP (no recursivo):
Input: $w \in \Sigma^*$, M tabla de análisis sint'actico para G
Output: si $w \in L(G)$ una derivación por la izq. de w, de lo contrario una indicación de error

1. Configuración inicial = stack: $S\$$, buffer: $w\$$
2. apuntar $ip \uparrow first(w\$)$
3. repetir hasta que $X = \$$, i.e vaciar la pila:
   - $top(stack) = X, ip = `a'$
   - Si $X \in Vt$ o $X = \$$ entonces
     Si $X = a$, extraer X del stack y avanzar ip, de lo contrario *error()*
     de lo contrario: $(X \in Vn)$
     Si $M[X, a] = X \to Y_1 Y_2 \dots Y_k$ entonces:
     - extraer X del stack,
     - empilar $Y_1 Y_2 \dots Y_k$ (Y_1 en el tope);
     - emitir la prod. y realizar la acción semántica asociada

     de lo contrario, indicar *error()*

NO	SÍMBOLO DE ENTRADA					
TERMINAL	**id**	$+$	$*$	(	)	\$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$			$F \to (E)$		

Algoritmo para la construcción de Tablas de ASDP-LL
Conjuntos FIRST y FOLLOW:
Mostrar secuencias de pasos

3.2 Análisis Sintáctico Ascendente

Para una frase de entrada construye el AAS por las hojas y avanza hacia la raíz, i.e intenta "reducir" una cadena al símbolo inicial de la gramática.

$\star$**Desplazamiento y Reducción** en cada paso de la reducción se *sustituye* una subcadena determinada que concuerde con el lado der. de una producción por (gracias al) símbolo del lado izq. de dicha prod. (si se elige correctamente se traza una *derivación por la derecha en sentido inverso*).

Handle: un "mango" de una forma de frase derecha γ es una producción $A \rightarrow \beta$ y una posición de γ donde β podría encontrarse y sustituírse por A para producir la forma de frase derecha de γ.

$$S \Rightarrow^*_{md} \alpha A w \Rightarrow^*_{md} \alpha \beta w$$

Analizadores Sintácticos por Desplazamiento y Reducción *"Shift-Reduce"*: (poda de handles)

1. Situar la subcadena a reducir en forma de frase derecha
2. Determinar que producción elegir en caso de haber más de una prod. con dicha subcadena en el lado derecho.

$\rightarrow$ *Stack & Buffer*
Operaciones principales:

1. *Shift*: el sig. símbolo de la entrada se empila
2. *Reduce*: el analizador sabe que el extremo derecho del handle está en el tope del stack; entonces debe localizar el extremo izq. del handle dentro de la pila y *decidir* el no terminal con el cual substituír el handle.
3. *Aceptar*: termina con éxito.
4. *Error*: invoca la rutina para recuperación de errores.

$\rightarrow$ El handle siempre aparecerá en el tope del stack.
Las formas posibles de 2 pasos sucesivos en cualquier derivación por la derecha:

1. $S \Rightarrow^*_{md} \alpha A z \Rightarrow^*_{md} \alpha \beta B y z \Rightarrow^*_{md} \alpha \beta \gamma y z$
2. $S \Rightarrow^*_{md} \alpha B x A z \Rightarrow^*_{md} \alpha \beta B x y z \Rightarrow^*_{md} \alpha \gamma x y z$

$\rightarrow$ ¿Cómo elegir las acciones?
Prefijos viables: prefijos en forma de frase derecha que pueden aparecer en el stack de un A.S.S-R:

- prefijo de una forma de frase derecha que no continúa más allá del extremo derecho del handle situado más a la derecha de esta forma de frase
- $\rightarrow$ Siempre es posible añadir símbolos terminales al final de un prefijo viable para obtener una forma de frase derecha.

Conflictos:

- no se puede decidir si reducir o desplazar *(shift-reduce)*
- no se puede decidir que tipo de reducción efectuar *(shift/shift)*

$\star$ Las gramátricas ambigüas NO pueden ser LR(k)

A.S. por *Precedencia de Operadores*
A.S. LR(k) : *SLR* , *LALR*

3.2.1 Analizadores Sintácticos *LR(k)*

Analizadores Sintácticos *LR(k)*

- *Left to right scanning, Right-most derivation; k lookahead*
- útil para prácticamente toda construcción de los lengs. de programación (CFG)
- método de análisis*Shift-Reduce* sin retroceso ḿas general que se conoce (permite implantación eficiente)
- la clase de gramáticas que se pueden analizar con los métodos LR es un supra conjunto de la clase de gramáticas que se pueden analizar con los A.S. predictivos
- un A.S. LR puede detectar un error sintáctico tan pronto como sea posible hacerlo en un exámen de izq. a der.

Tres técnicas para la construcción de una *Tabla de A.S.* LR :

1. *SLR*
2. *LALR*
3. *canonical LR*

$$s_0 X_1 s_1 X_2 \ldots X_m s_m$$

Controlador del analizador:

- Se determina s_m, el *estado* en el tope del stack, y a_i el símbolo en curso de la entrada,
- Se consulta la tabla, en *acción [s_m, a_i]*:
    1. shift s, (s es un estado)
    2. reducir por $A \to \beta$
    3. aceptar
    4. error
- la *función goto*(s, α) toma un estado y un símbolo gramatical y produce un estado (función de transición de un DFA que reconoce los *prefijos viables de G*)
- Una *configuración* es un par:

$$(\underbrace{s_0 X_1 s_1 X_2 \ldots X_m s_m}, \underbrace{a_i a_{i+1} \ldots a_n \$})$$

representa la *forma de frase derecha:*

$$X_1 X_2 \ldots X_m a_i a_{i+1} \ldots a_n$$

el sig. movimiento se determina leyendo a_i y s_m y consultando la entrada $accion[s_m, a_i]$.

Las configuraciones sigs son (para cada tipo de movimiento):

1. si $accion[s_m, a_i] =$*shift* s,

$$(s_0 X_1 s_1 X_2 \ldots X_m s_m \underbrace{a_i s}, a_{i+1} \ldots a_n \$)$$

2. si $accion[s_m, a_i] =$*reduce* $A \to \beta$,

$$(s_0 X_1 s_1 X_2 \ldots \underbrace{X_{m-r} s_{m-r} A s}, a_i a_{i+1} \ldots a_n \$)$$

   donde $s = goto(s_{m-r}, A)$, y r es la longitud de β. El analizador *extrajo primero* $2r$ símbolos del stack (estados y de la gramática) luego empiló A y s (no se modificó el símbolo de entrada en curso). Se genera la salida ejecutando la *acción semántica* asociada a la prod.

3. si $accion[s_m, a_i] =$ *aceptar*, termina
4. si $accion[s_m, a_i] =$ *error*, invoca a la rutina de recuperación de errores

Aspectos Importantes:

- Intuitivamente: para que una G sea LR basta con que un A.S. S-R que opere de izq. a der. pueda *reconocer los handles cuando aparezcan en el tope del stack.*

- El tope tiene toda la información necesaria.

- $\to$ Si se puede reconocer un handle conociendo sólo los símbolos gramaticales del stack entonces existe un AF el cual, leyendo tales símbolos de arriba hacia abajo, determina el handle (si es que éste existe) que está en el tope del stack.

- El último símbolo almacenado en el tope del stack es el estado en el que estaría el AF reconocedor de los handles si hubiera leído los símbolos gramaticales del stack desde abajo hacia arriba.

- Se toman k símbolos gramaticales siguientes para tomar *decisiones shift-reduce* ($k \leq 1$).

- Diferencia entre las gramática $LL(k)$ y $LR(k)$:
  para que G sea $LR(k)$ hay que reconocer ls presencia del lado derecho de una prod. habiendo visto todo lo que deriva de dicho lado der. con k símbolos de examen por anticipado.
  para que G sea $LL(k)$ hay que ser capaz de reconocer el uso de una prod. *viendo sólo los primeros k símbolos que se derivan de su lado der.*

3.2.2 Construcción de Tablas A.S. SLR

• **Elemento** $LR(0)$ de G es una prod. con un punto ('.') en alguna posición del lado derecho:

$A \to XYZ$	$A \to \epsilon$
$A \to X.YZ$	$A \to .$
$A \to XY.Z$	
$A \to XYZ.$	

$\to$ un elemento indica *hasta dónde se ha visto de una prod. en un momento dado*: por ejemplo, $A \to X.YZ$ indica que se espera ver en la entrada una cadena derivada de X y a continuación una cadena derivable de YZ.

Idea Central (SLR): construír a partir de G un AFD para reconocer los *prefijos viables*. Los *elementos* se agrupan en conjuntos (*estados* del A.S. SLR).

• Base de la construcción: *colección canónica* $LR(0)$:

1. Gramática aumentada: G, $S' \to S \in G'$.
   propósito: indicar al analizador cuándo se debe detener y aceptar w.

2. *cerradura(I)*: dado un conj. de *elementos* de I, se define:

   (a) $\forall x \in I$ implica $x \in cerradura(I)$

   (b) Si $A \to \alpha.B\beta \in cerradura(I)$ y $B \to \gamma \in G$, incorporar $B \to .\gamma$ a $cerradura(I)$. Se aplica esta regla hasta que ya no se puedan incorporar más elementos.

3. *ira(I,X)*: es la *cerradura* del conjunto de todos los elementos $\{[A \to \alpha X.B\beta]\}$ tales que $[A \to \alpha.XB\beta] \in I$.
   Si I es el conj. de los elementos válidos para algún prefijo viable entonces $ira(I, X)$ es el conj. de elementos válidos para el prefijo viable γX.

Bibliografía

[Aho86] *Compiladores: principios, técnicas y herramientas.* Adisson-Wesley Iberoamericana, 1986.

[Dol79] *Introduction to Automata Theory. Languages and Computation.* Addison-Wesley, 1979.