uc3m | Universidad Carlos III de Madrid

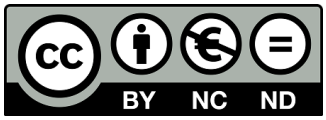
Grado en Ingeniería Informática

2020-2021

Apuntes

Procesadores de Lenguaje

Jorge Rodríguez Fraile[1]



[1]Universidad: 100405951@alumnos.uc3m.es | Personal: jrf1616@gmail.com

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1. INFORMACIÓN

1.1. Profesores

Teorías: Antonio Berlanga de Jesús y Jesús García Herrero

Prácticas: Juan Manuel Alonso Weber jmaw@ia.uc3m.es

1.2. Presentación

Prácticas semanales por parejas.

Hay clases en el cronograma de más, algunas se usarán para recuperar clases.

Las prácticas las haremos en Linux, se puede utilizar máquina virtual, guernika y WSL2, lo importante es tener un terminal.

Vamos a usar bison y flex, programaremos en C.

2. REPASO DE TALF

2.1. Definiciones

Forma sentencial: Aplicando reglas de producción llego a $x \in \sum^*$

Sentencia: Forma sentencial en el que x solo tiene Terminales.

Lenguaje asociado a una gramática: Todas las sentencias, secuencias de Terminales, de una gramática.

Recursividad: Aparece el terminal de la parte izquierda a la derecha.

- Recursividad a izquierda: $A \rightarrow Ay$
- Recursividad a derechas: $A \rightarrow xA$

Reglas compresoras: Aquellas que producen menos símbolos de los que hay a la izquierda. Como lo es $x ::= \lambda$

2.2. Gramática

$G = \{\sum_T, \sum_N, S, P\}$

- $\sum_T$: Conjunto de símbolos **terminales**, alfabeto del lenguaje.
- $\sum_N$: Conjunto de símbolos **no terminales**.
- S: **Axioma** $S \in \sum_N$
- P: Conjunto de **Reglas de producción**.
- Backus Normal Form: $P_i ::= P_d$
- $P_i \in (\sum_T \cup \sum_N)^+$ Al menos un símbolo
- $P_d \in (\sum_T \cup \sum_N)^*$ De 0 a infinitos símbolos
- λ Clausula vacía

2.2.1. Tipos de Gramáticas

Tipo 3: Autómata Finito Determinista y Autómata Finito No Determinista.

- Expresiones regulares o lineales.
- Un solo No Terminal para dar: $P_d ::= a|aN|Na; a \in \sum_T, N \in \sum_N$
- Lineal izquierda: $P_d ::= a|Na$
- Lineal derecha: $P_d ::= a|aN$

Tipo 2: Autómata a Pila.

- Lenguaje Independiente del Contexto.
- Un No Terminal para dar cualquier sentencia, excepto lambda que solo lo puede dar el axioma.

Tipo 1: Máquina Linealmente Acotada.

- Lenguaje de contexto libre.
- Cualquier sentencia para dar cualquier secuencia, pero que no sea una regla reductora(tampoco se puede No Terminal para dar lambda).

Tipo 0: Máquina de Turing.

- Lenguaje sin restricciones.
- Cualquier sentencia para dar cualquier secuencia de símbolos, incluso reglas reductoras.

2.3. Problemas de lectura de sentencias

Precedencia: Hay operadores con mayor prioridad. Se debe hacer desde lo último derivado hacia arriba. Esto provoca ambigüedad, que es nuestro enemigo.

Se hace desde lo último derivando hacia arriba

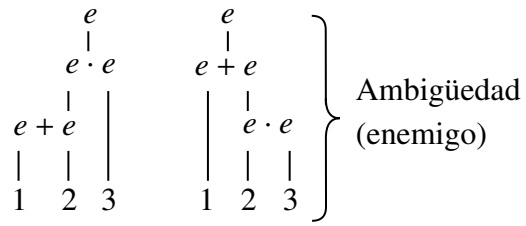


Fig. 2.1: Ejemplo Precedencia

Dangling Else: Sobre a quién pertenece el else, en C el else pertenece al último else.

```
if C1 then A else if C2
then B else C
```

Asociatividad: Entre operaciones con la misma precedencia. 1-1-1-1

2.4. Derivación

Aplicación de una producción a una forma sentencial.

Derivación más a la izquierda: Sustituye el símbolo no terminal más a la izquierda.

Derivación más a la derecha: Sustituye el símbolo no terminal más a la derecha.

2.5. Ambigüedad

Hay distintos niveles:

- **Sentencia**: Se puede obtener por dos derivaciones diferentes.
- **Gramática**: Si puede obtener una sentencia con dos derivaciones.
- **Lenguaje**: Todas las gramáticas que lo generan son ambiguas entonces es inherentemente ambiguo.

2.6. Bien formar Gramáticas

Limpiar:

1. Sin **reglas innecesarias**: Un terminal que se da a sí mismo. A→A
2. Sin **símbolos inaccesibles**: Poner T y NT, e ir tachando según aparecen, los que no aparecen regla fuera.
3. Sin **símbolos superfluos**: Empezamos con las reglas que solo producen terminales e ir subiendo por pasos.

Sin reglas **no generativas**: Las que dan lambda, quitarlas y sustituir sus apariciones por lambda(nada).

Sin reglas de **redenominación**: Las que llaman aun solo símbolo No Terminal, poner las reglas de ese símbolo en el que lo invoca.

2.7. Transformaciones útiles en compiladores

Estas transformaciones facilitan a los analizadores predictivos. Estos analizadores implican que nunca se va a hacer backtracking siempre sabemos que regla de producción hay que aplicar o no hay más, pero no se retrocede.

Factorización a izquierdas

- Cuando hay una estructura común en las reglas de producción.
- Se crea un nuevo No terminal para dar los símbolos que acompañan a esa parte común, en el original se pone la parte común seguida del nuevo símbolo.

P: $A \to \beta \cdot \alpha_1 | \beta \cdot \alpha_2$ Hay una estructura comun

1. $\Sigma_N = \Sigma_N \cup \{A'\}$ Nuevo no terminal
2. $B = (B - P)$ Se quitan las reglas
3. $B' = B \cup \{A \to \beta \cdot A', A' \to \alpha_1 | \alpha_2\}$

Eliminar recursividad por la izquierda

- Para evitar recursividades pendientes, ya que leemos de izquierda a derecha, así reconocemos mientras aplicamos reglas.
- A las reglas no recursivas a izquierda se le concatena un nuevo No Terminal (quitamos las que eran recursivas).
- Ese nuevo símbolo produce λ y lo que producía originalmente que era recursivo a izquierda, pero en recursividad a la derecha.

$\forall P \subseteq B | P = (A \to A \cdot \alpha | B)$

1. $\Sigma_N = \Sigma_N \cup \{A'\}$
2. $B = (B - P)$
3. $B' = B \cup \{A \to \beta \cdot A', A' \to \alpha \cdot A' | \lambda\}$

2.8. Autómata Finito Determinista

$AFD = (\sum, Q, f, q_o, F)$

Donde:

- $\sum$ Símbolos de entrada, Tokens.
- Q Estados.
- $q_o \in Q$ Estado inicial.
- $F \subseteq Q$ Estados finales.
- $f(q, a \cdot x)$ Función de transición. Con un estado y un símbolo, voy a un estado.

	a	**b**
→ $\boldsymbol{q_0}$	q_1	q_2
$\boldsymbol{q_1}$	q_1	q_0
$*\boldsymbol{q_2}$	q_1	q_2

Tabla 2.1: Tabla de transiciones

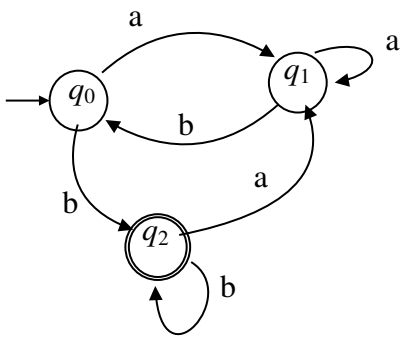


Fig. 2.2: Diagrama de transiciones

Extensión a palabras: Recibe una palabra y va cogiendo los símbolos de entrada, y aplicando reglas sucesivas. Palabras que partiendo del inicial nos permite llegar a uno final.

Lenguaje asociado: Aquellas palabras tales que aplicando extensión a palabra el q_o alcanza su estado de F, final.

Equivalencia de AFD: Si el lenguaje que reconocen es el mismo.

Minimización de AFD: Autómata finito mínimo equivalente es aquel con menor número de estado, pero mismo lenguaje.

Siempre se puede pasar de G3 a AFD, aunque puede que no directo, por medio de AFND.

2.9. Autómata Finito No Determinista

- Admite λ y más de una posibilidad por símbolo.
- $AFND = (\sum, Q, f, q_o, F)$ Los símbolos representan lo mismo que en AFD.

- En este caso se permite transicionar sin recibir símbolo, lambda, además, varias transiciones para un mismo símbolo en un mismo estado.

Para cualquier AFD existe uno no determinista equivalente.

Para toda gramática G3 existe un autómata determinista y no determinista.

2.10. Autómata a Pila (G2)

$AP = (\sum, \Gamma, Q, A_0, q_0, f, F)$

Donde:

- Γ Conjunto de símbolos de pila
- A_0 Símbolo de pila inicial
- $f(q, \lambda, A)$ Transita a uno o varios estados y puede escribir o no en pila, cuando escribe pueden ser más de 1.
- Para toda gramática G2 existe un AP.
- El AP puede ser no determinista y en general $APND \nrightarrow APD$.
- Lenguaje aceptado por un AP, palabras que reconoce, y el final lo determina:

Por estado final: Cuando llega a un estado final y la secuencia de entrada, está vacía, termina.

Por vaciado de pila: Entrada y pila vacía, termina.

2.11. Expresiones Regulares G3

Definición:

- es una ER.
- λ es una ER.
- a es una ER, siendo a un Tokens.
- $\alpha + \beta$ es una ER, la unión.
- $\alpha \cdot \beta$ es una ER, la concatenación.
- α^* es una ER, la clausura.

ER útiles:

- $\alpha^+ = \alpha \cdot \alpha^* = \alpha^* \cdot \alpha$ Al menos 1
- $\alpha? = \alpha|\lambda$ Puede ser alpha o lambda
- $[abc] = a|b|c$ Es a o b o c
- $[a - z] = a|b|...|z$

Dos EERR son equivalentes si describen el mismo lenguaje.

Lenguaje asociado a una ER

- si $\alpha = \emptyset, L(\alpha) = \emptyset$
- si $\alpha = \lambda, L(\alpha) = \lambda$
- si $\alpha = a | a \in \Sigma, L(\alpha) = \{a\}$
- si α y β son EERR $\Rightarrow L(\alpha + \beta) = L(\alpha) \cup L(\beta)$
- si α y β son EERR $\Rightarrow L(\alpha \cdot \beta) = L(\alpha) \cdot L(\beta)$
- si α^* es una ER $\Rightarrow L(\alpha^*) = L(\alpha)^*$

Propiedades

1. $(\alpha + \beta) + \sigma = \alpha + (\beta + \sigma)$
2. $\alpha + \beta = \beta + \alpha$
3. $(\alpha \cdot \beta) \cdot \sigma = \alpha \cdot (\beta \cdot \sigma)$
4. $\alpha \cdot (\beta + \sigma) = (\alpha \cdot \beta) + (\alpha \cdot \sigma)$

   $(\beta + \sigma) \cdot \alpha = (\beta \cdot \alpha) + (\sigma \cdot \alpha)$
5. $\alpha \cdot \lambda = \lambda \cdot \alpha = \alpha$
6. $\alpha + \phi = \phi + \alpha = \alpha$
7. $\lambda^* = \lambda$
8. $\alpha \cdot \phi = \phi \cdot \alpha = \phi$
9. $\phi^* = \lambda$
10. $\alpha^* \cdot \alpha^* = \alpha^*$
11. $\alpha \cdot \alpha^* = \alpha^* \cdot \alpha$
12. $(\alpha^*)^* = \alpha^*$
13. $\alpha^* = \lambda + \alpha + \alpha^2 + ... + \alpha^n + \alpha^{n+1} \cdot \alpha^*$
14. $\alpha^* = \lambda + \alpha \cdot \alpha^*$
15. $\alpha^* = (\lambda + \alpha)n - 1 + \alpha n \cdot \alpha^*$
16. Sea f una función, $f : E_{\Sigma^n} \rightarrow E_\Sigma$ se verifica:

    $f(\alpha, \beta, ..., \sigma) + (\alpha + \beta + ... + \sigma)^* = (\alpha + \beta + ... + \sigma)^*$
17. Sea f una función, $f : E_{\Sigma^n} \rightarrow E_\Sigma$ se verifica:

    $(f(\alpha^*, \beta^*, ..., \sigma^*))^* = (\alpha + \beta + ... + \sigma)^*$
18. $(\alpha^* + \beta^*)^* = (\alpha^* \cdot \beta^*)^* = (\alpha + \beta)^*$ (IMPORTANTE)
19. $(\alpha \cdot \beta)^* \cdot \alpha = \alpha \cdot (\beta \cdot \alpha)^*$
20. $(\alpha^* \cdot \beta)^* \cdot \alpha^* = (\alpha + \beta)^*$
21. $(\alpha^* \cdot \beta)^* \cdot \alpha^* = \lambda + (\alpha + \beta)^* \cdot \beta$
22. Regla de Inferencia:

    Si $X = AX + B$ entonces $X = A^*B$

3. TEMA 1: INTRODUCCIÓN

Compilador: Proceso de traducción que convierte un programa fuente escrito en un lenguaje de alto nivel a un programa objeto en código máquina y listo por tanto para ejecutarse en el ordenador. Solo se genera el programa objeto cuando no hay errores.

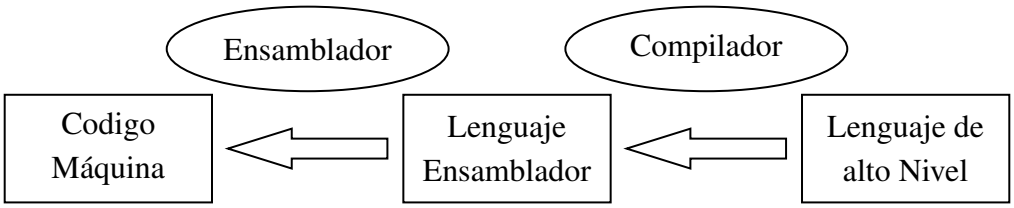


Fig. 3.1: Diagrama tipos de programación

Programa fuente → Compilador → Programa objeto/Mensajes de error.

Originalmente, cuando no existía se metía con 1's y 0's, código máquina.

Después, se creó el ensamblador que traduce un código nemotécnico (lenguaje ensamblador) a código máquina.

Por último, el compilador sobre el ensamblador, esto facilito mucho las cosas.

En los 50 se consideraban los programas más difíciles.

Con el tiempo se ha facilitado mucho con entornos de programación y herramientas, no hay una fecha exacta del primer compilador.

Los primeros traducían fórmulas aritméticas a código máquina.

Motivación:

- Saber cómo se obtiene un ejecutable para alcanzar mejor eficiencia y corrección.
- Para entender mejor los lenguajes de programación.
- Conocer la teoría que hace posible su funcionamiento y como se ha llegado hasta este punto.
- Aplicar la teoría y herramientas a otros campos:
  - Intérpretes de comandos y consultas
  - Formateadores de texto (LaTeX, HTML)
    - Gráficos, ecuaciones,... (PS, GIF, EQN, ...)
    - Lenguajes de simulación (GPSS)

Aplicación:

- Desarrollo de interfaces de texto.
- Tratamiento de ficheros de texto estructurado.
- Procesadores de texto.
- Formateo de texto y descripción gráfica.
- Procesamiento del lenguaje natural.
- ...

Diferencia entre un compilador y un intérprete: El compilador solo hace la traducción a algo que se puede ejecutar, sin embargo, el intérprete no solo lo traduce, sino que lo va ejecutando.

Ensamblador: Compilador sencillo, traduce a código máquina sentencias simples de lenguaje fuente.

Traductor: Normalmente se refiere a la traducción entre dos lenguajes al mismo nivel de abstracción.

Interprete: Es un programa que simultáneamente analiza y ejecuta un programa escrito en lenguaje fuente.

Compilador cruzado: Compilador que traduce un lenguaje fuente a objeto donde el objeto es para un ordenador distinto del que compila.

Compile-Link-Go frente a Compile-Go: Uno Fragmenta, Compila, Enlaza las partes y tras crearlo lo ejecuta, y el otro Compila en un módulo cargable y Ejecuta el módulo.

Compilador de una o varias pasadas.

Una pasada es un recorrido total de todo el código fuente con una misión específica.

Traductor o compilador incremental: Encontrados y corregidos los errores después solo se compilan estos.

Autocompilador: Compilador escrito en el propio lenguaje que compila. Esto facilita la portabilidad.

Metacompilador: Programa que recibe la definición de un lenguaje (Gramática) y genera un compilador para ese lenguaje.

Ejecutable: Código objeto enlazado con las librerías.

3.1. Compilación

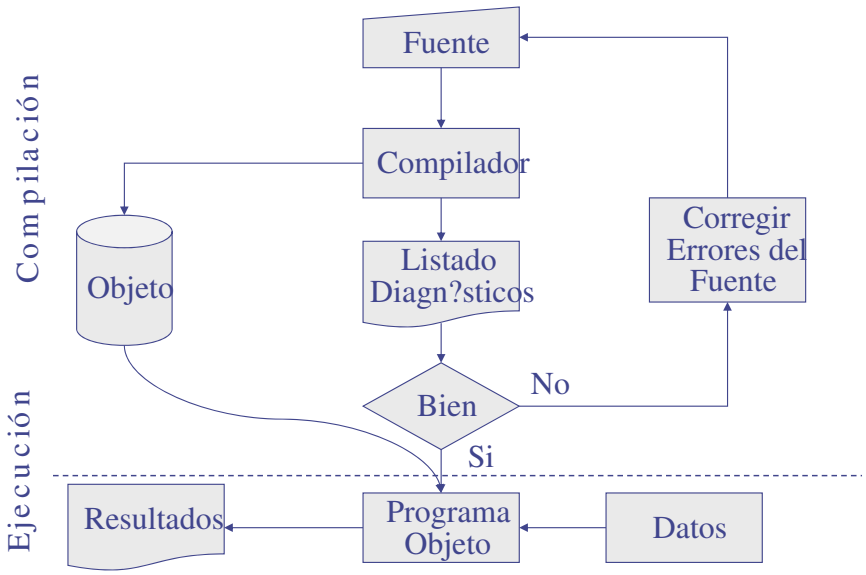


Fig. 3.2: Esquema de Compilación

Antes del pasar al compilador se pasa al **preprocesador que hace muchas funciones**, entre ellas, sustituir las constantes por su valor y quita los comentarios. Cuando termina pasa el programa fuente al compilador.

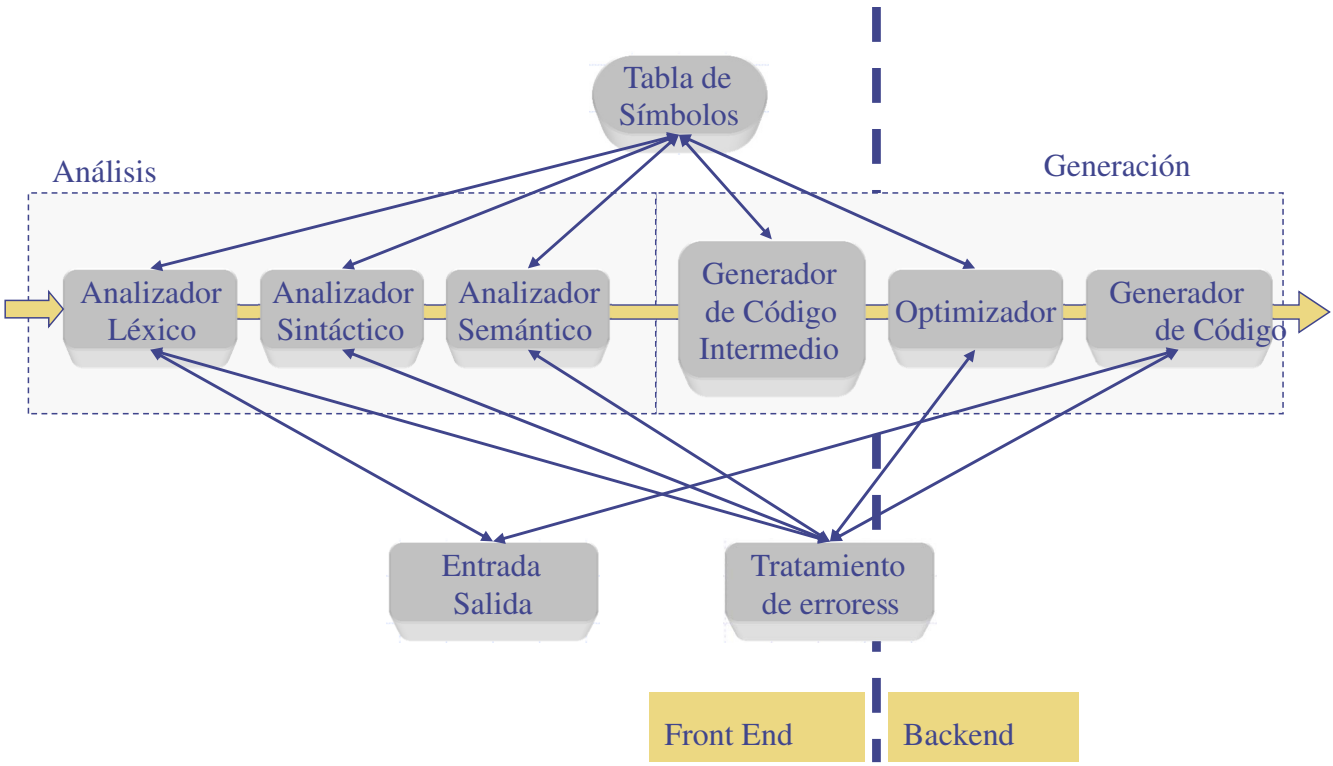


Fig. 3.3: Diagrama de Fases de un Compilador

Análisis: Parte que solo depende de la estructura del lenguaje, no de la arquitectura de la máquina.

- **Analizador léxico (scanner)**: Convierte la entrada (y::= 3+b*3) en Símbolos terminales (id op num op id op num) de una gramática. A este llegan directamente los datos, no al resto. Su salida va al siguiente.

  Análisis lineal: La cadena de entrada se lee de izquierda a derecha y se agrupa en componentes léxicos (tokens)

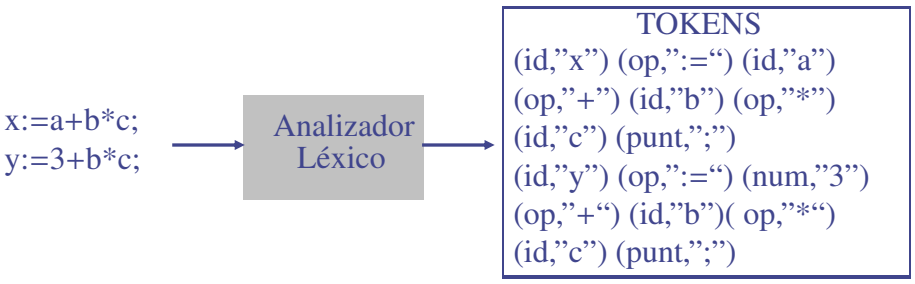


Fig. 3.4: Diagrama Analizador Léxico

- **Analizador sintáctico:** Agrupa los componentes léxicos en frases gramaticales que el compilador utiliza. A partir de los tokens produce un árbol sintáctico.

  - Recibe tokens → crea un árbol.
  - En las hojas tiene los terminales en la manera en la que identifica el token.

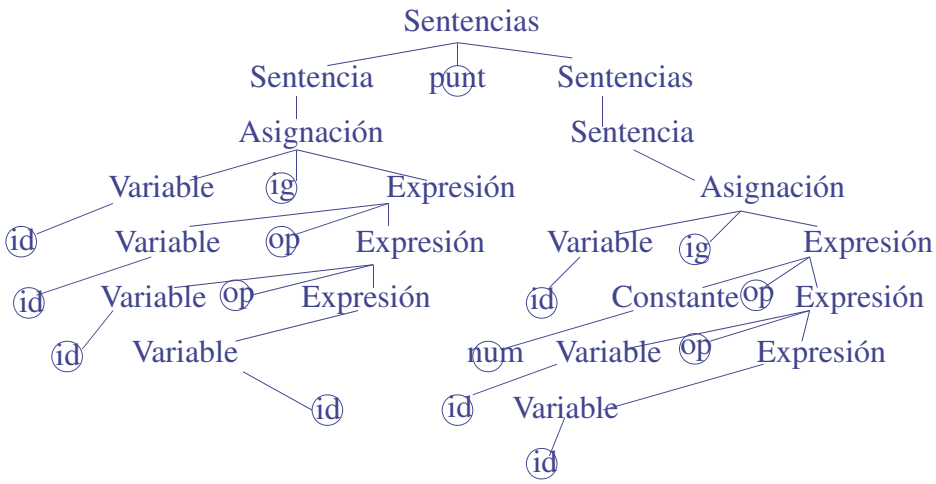


Fig. 3.5: Diagrama Analizador Sintáctico

- **Analizador semántico:** Busca errores semánticos, reúne información de tipos, identifica operadores y operandos.

    - Recibe el árbol sintáctico explícita o implícitamente, y trata de determinar si tiene sentido las distintas operaciones.

      Por ejemplo: Sumar un número y una cadena de caracteres.

Generación: Depende de la arquitectura, pero no del lenguaje.

- **Generador de código intermedio**: A veces se encuentra en análisis y otras en generación. Convierte el árbol en instrucciones, código intermedio (ensamblador). El recorrido es importante, ya que al generar código no se ve la jerarquía. Por ejemplo: Primero se tienen que hacer las operaciones antes de asignar el valor.

  El recorrido es importante ya que al generar codigo no se ve la jerarquia.

- **Optimizador:** Trata de darse cuenta de subárboles comunes para no repetir operaciones en las que no cambian variables. En vez de volver a poner esa estructura apunta a ella.

- **Generador de código:** El uso de código intermedio reduce la complejidad del desarrollo de compiladores. m front ends y n back-ends comparten un código intermedio común.

Front-end: Etapa inicial. Fases que dependen del lenguaje fuente y que son independientes de la máquina. Análisis léxico, sintáctico, semántico y generación de código intermedio, manejo de errores de cada parte.

Back-end: Etapa final. Fases que dependen de la máquina, dependen del lenguaje intermedio. Optimización de código, generación de código, operaciones con la tabla de símbolos.

3.2. Diagramas de Tombstone

Conjunto de piezas de puzzle útiles para razonar acerca de los procesadores de lenguaje y los programas.

Está formado por piezas que deben encajar, encajan según unas reglas de formación.

La máquina y el lenguaje deben coincidir.

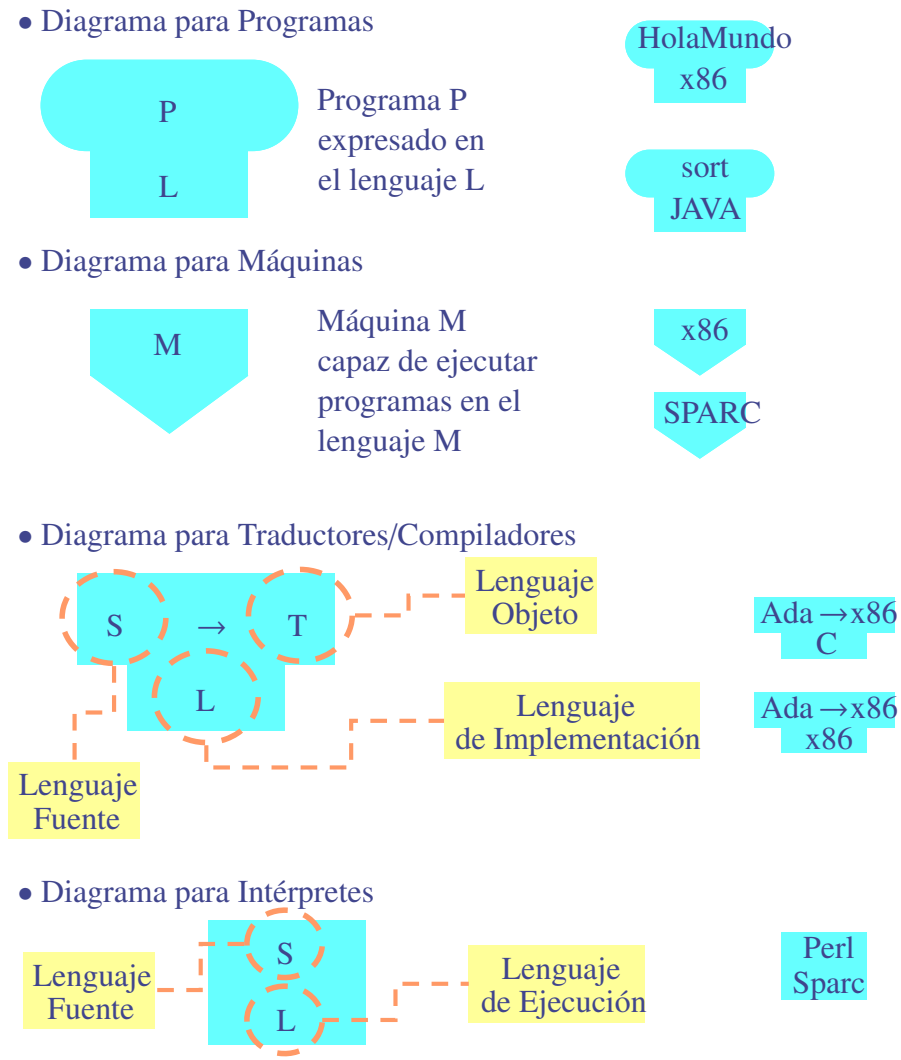


Fig. 3.6: Diagrama Tombstone Programas y Máquinas

4. TEMA 2: ANÁLISIS LÉXICO

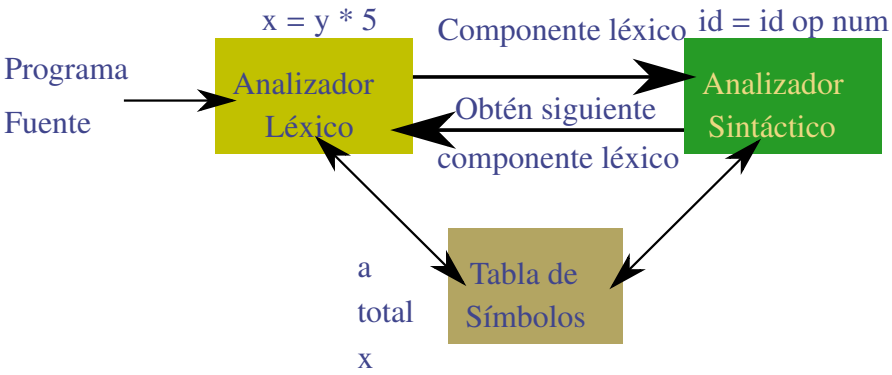


Fig. 4.1: Diagrama AL

4.1. Funciones del Análisis Léxico

Manejar el fichero fuente: Leer los caracteres de la entrada. Eliminar comentarios y delimitadores. Relacionar los mensajes de error con las líneas del programa fuente.

Interfaz con las otras fases de análisis: Generar la secuencia de componentes léxicos (tokens). Introducir los identificadores en la **tabla de símbolos**. Manejar macros. pi→3,14

Controlar si es de formato libre o no: Libre: PASCAL. No libre: FORTRAN.

4.2. Aspectos del Análisis Léxico

Diseño más sencillo: Los símbolos que trata el scanner se describen con una gramática más simple que la del parser.

Mejora la eficiencia: Gran parte del tiempo de compilación se consume en la lectura y exploración.

Mejora la portabilidad: Se pueden tener varias versiones del scanner una para distintos códigos.

Tokens: Unidades sintácticas de un lenguaje. Símbolos terminales de una gramática, identificadores, palabras reservadas.

- num es el token.

Categoría:

- **Palabras reservadas**: IF, THEN, DO...

- **Identificadores**: main, suma, identificadores...
- **Símbolos especiales**: Como <, >...
- **Constantes y literales.**

Lexema: Secuencias válidas para un token. Secuencia de caracteres del código fuente que son identificados como un token específico.

- El lexema de num son los valores que puede tomar: 12, 34, 480

Atributos: Información adicional que tiene el token.

- id.lexema="x" id.tipo id.ámbito

Patrón: Define la secuencia de caracteres válidos para un tokens, formato de los lexemas.

- $[0-9]^+$ Naturales $-?[0-9]^+$ Enteros de -9 a 9
- El + al final, indica que hay 1 o más ocurrencias. Para poner determinados símbolos se ponen

Especificar tokens: Con expresiones regulares, patrones, o Autómatas Finitos.

- Los elementos básicos en un lenguaje deben ser tokens.
- Se pueden usar Expresiones regulares o Autómatas Finitos.

Tabla de símbolos: Almacenan la correspondencia entre tokens y lexemas. En muchos lenguajes hay palabras claves (reservadas) para las que no se puede declarar una variable, y estas se almacenan en la tabla.

- Los atributos de los identificadores se pueden guardar en la tabla de símbolos. Los otros en otra tabla.

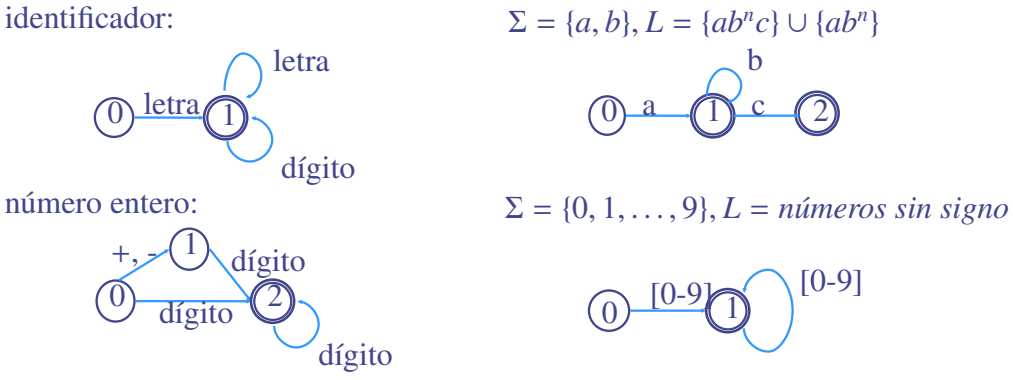


Fig. 4.2: Diagramas A. Lexico

Un AF no devuelve el token, se usa otra cosa para que nos dé el token cuando reconoce símbolos.

Nosotros vamos a usar DT para representar.

4.3. Diagrama de Transiciones - DT

Definir formalmente un AL es hacer los DT de los tokens (ojo con los separadores que llevan a finales, si no hay nada, que no salga)

Se usa para especificar el funcionamiento de un Analizador Léxico mediante un AFD. **UNO POR TOKEN**

Se diferencian porque el DT reconoce un token cuando reconoce un delimitador y devuelve entonces el token. **Lee caracteres hasta completar un token entonces: Devuelve el token leído** y Deja el buffer listo para la siguiente llamada.

No tiene estados de error. De los estados de aceptación no salen transiciones, necesita una transición más que no pertenece al token, que le hace salir y devolver el token.

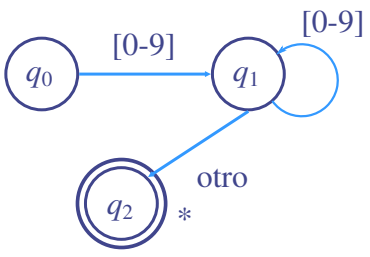
Esto quiere decir que aquellos estados que son terminales y pueden transitar a otro estado tengan una transición para otro tipo de carácter que hace que salga como final.

4.4. Construcción de un AFD para AL

Normalmente se parte de representación de las reglas de tokens con expresiones regulares.

1. Toda expresión regular tiene un AFND asociado.
2. Construir un AFD para el mismo lenguaje del AFND
3. Optimización del AFD.

4.5. Tabla de Transiciones de DT



	Entradas			
estado	**0-9**	**Otro**	***token***	**Retroceso**
q_0	q_1	Error	-	-
q_1	q_1	q_2	-	-
q_2	-	-	Num_entero	1

Fig. 4.3: Diagrama Tabla de Transiciones

4.6. Autómata programado

Es la manera de programar un Analizador Léxico.

Representa directamente con un programa al DT en cuestión.

```
estado := 1 # El inicial es el estado 1
while estado<>6 do
  Leecar {devuelve en car el siguiente carater leido}
  case estado of
    1: if car="a" then estado:=2 else
       if car="d" then esatdo:=5 else error
    2: if car="c" then estado:=3 else
       if car="b" then estado:=2 else error
    3: if car="a" then estado:=4 else error
    4: if car="$" then estado:=6 else error
    5: if car="d" then estado:=3 else error
    3: error
  end case
end while
```

4.7. Implementación de un AL

4.7.1. Utilizando un generador de Analizadores Léxicos (es lo que usaremos)

- **Ventajas**: Comodidad y Rapidez de desarrollo.
- **Inconvenientes**: Ineficiencia.
- **Recomendación**: Ordenar las reglas de acuerdo con la frecuencia de utilización.

4.7.2. Utilizando ensamblador

- **Ventajas**: Más eficiente y compacto.
- **Inconvenientes**: Más difícil de desarrollar y Dificultad de mantenimiento de código generado.

4.7.3. Utilizando un lenguaje de alto nivel

- **Ventajas**: Eficiente y Compacto.
- **Inconveniente**: Realizar todo a mano.
- Técnicas:

- **Programación**: Muy eficiente.
- **Controlado por tablas**: Código pequeño, general y manejable.

4.8. Programación de un AL

Dos punteros de lectura:

- **Puntero actual**: El último carácter aceptado.
- **Puntero de búsqueda**: El último carácter leído.

	m	a	t	1	[	i	n	d	]		=		3	+	2	
	pA			pB												

Tabla 4.1: Diagrama punteros AL

Funciones de lectura: GetChar (avanza uno del avanzado y da el valor), Fail (Mueve el puntero avanzado de vuelta al aceptado), Retract (mueve el puntero avanzado uno atrás) y Accept (mueve el puntero actual hasta el avanzado).

Primitivas: IsLetter (letra), IsDigit (número) e IsDelimiter (separadores).

Acciones: InstallName (introduce un nombre en la tabla de símbolos).

4.9. Errores Léxicos

Hay pocos detectables por el analizador léxico.

Detectables, según el diseñador:

- Número de caracteres de los identificadores.
- Caracteres ilegales.
- Otros, como admitir números sin la parte entera.
- Cadena que no concuerda con ningún token.

En caso de error:

1. Anotar el error, indicarlos.
2. Recuperarme del error para continuar con el análisis. Ignorar, Borrar, Insertar o Corregir.
3. Seguir.

4.10. Identificación de palabras reservadas

Mediante **resolución implícita**, todas identificadas en el comienzo de la tabla de símbolos, o **resolución explícita**, usando patrones para las palabras reservadas.

4.11. Prioridad de los tokens

Criterio: Se utilizan heurística para dar prioridad al token que concuerda con **el lexema más largo**.

En el caso de que varios tokens tengan la misma longitud se asocia al que esté **en primer lugar**. Se definen por orden de prioridad.

4.12. AL y Lenguajes de Programación

El **AL agrupa caracteres para formar tokens**, por tanto, es importante definir el «delimitador».

- Carácter que **delimita el token sin pertenecer** a él.

Otro concepto importante es el de «**palabra reservada**»

- El lenguaje **prohíbe el uso libre al programador de determinadas palabras** que tienen un significado específico y único en el lenguaje

Lenguajes según uso de delimitadores y palabras reservadas:

- **Delimitadores blancos; con palabras reservadas**
  - Caso más sencillo de lenguaje (PASCAL, COBOL)
- **Delimitadores blancos; sin palabras reservadas**
  - PL/I
- **Blancos se ignoran; sin palabras reservadas**
  - El tipo más difícil de lenguaje, aparecen ambigüedades (FORTRAN)
  - El espacio en blanco se elimina con un preprocesador.
  - El AL tiene que retroceder y comenzar de nuevo.

4.13. Ejemplo de crear un AL y ASI

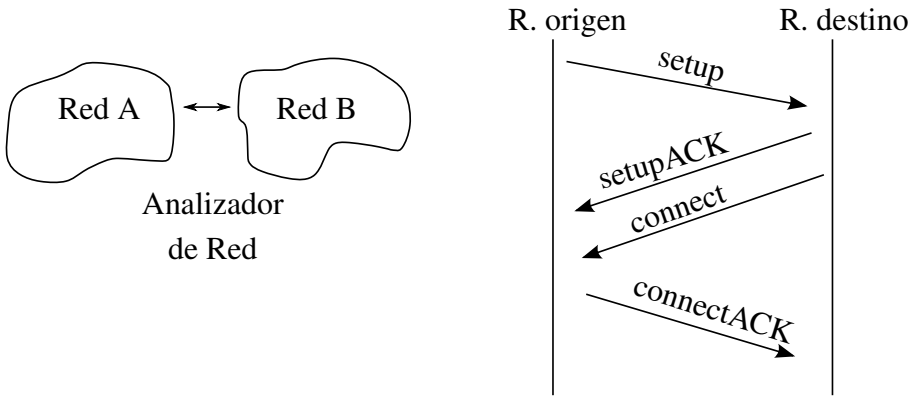


Fig. 4.4: Ejemplo de crear un AL y ASI I

Mensaje	Elementos Información	Obl.
Setup	EIsetup	X
	EIsentido	X
	EIorigen	
	EIdestino	X
SetupACK	EIsetupACK	X
	EIsentido	X
Connect	EIconnect	X
	EIsentido	X
ConnectACK	EIconnectACK	X
	EIsentido	X

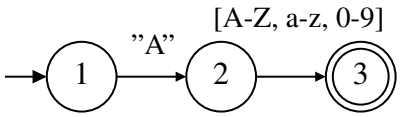
	Campo 1	Campo 2
EIsetup	”A”	[A-Z, a-z, 0-9]
EIsetupACK	”B”	[A-Z, a-z, 0-9]
EIconnect	”C”	[A-Z, a-z, 0-9]
EIconnectACK	”D”	[A-Z, a-z, 0-9]
EIsentido	”E”	$0\ A \rightarrow B$ $1\ B \rightarrow A$
EIorigen	”F”	$[0\text{-}9]^3$
EIdestino	”G”	$[0\text{-}9]^3$

4.13.1. Preguntas

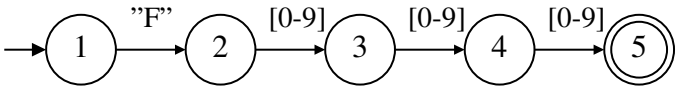
1. Definir formalmente el Analizador lexico cuyos tokens era los elementos de información.

   Σ_T = {EIsetup, EIsetupACK, EIconnect, EIconnectACK, EIorigen, EIdestino, EIsentido}

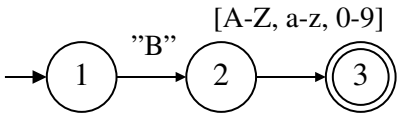
EIsetup



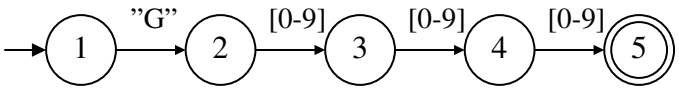
EIorigen



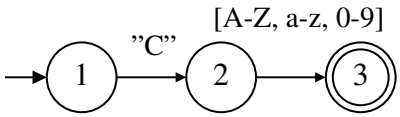
EIsetupACK



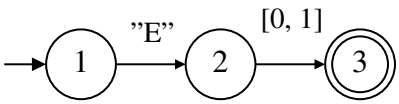
EIdestino



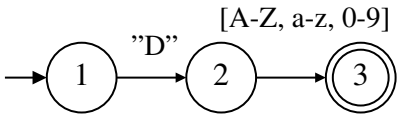
EIconnect



EIsentido



EIconnectACK



2. Gramatica que utiliza el analizador de red para analizar los mensajes validos

   M ::= SetupM | SetupACKM | ConnectM | ConnectACKM | λ

   Setup ::= EIsetup EIsentido EIorigen EIdestino | EIsetup EIsentido EIdestino

   SetupACK ::= EIsetupACK EIsentido

   Connect ::= EIconnect EIsentido

   ConnectACK ::= EIconnectACK EIsentido

5. TEMA 3: ANÁLISIS SINTÁCTICO

5.1. Introducción

Su función es comprobar que la secuencia de componentes léxicos es una secuencia del lenguaje y generar el árbol sintáctico (explicito o no)

Los lenguajes de programación que debe reconocer normalmente son independientes del contexto, por lo que se representan con gramáticas de tipo 2, diagramas de sintaxis o autómatas a pila.

Ventajas de utilizar gramáticas:

- Son especificaciones sintácticas y precisas de lenguajes.
- Se puede generar automáticamente un analizador.
- Mientras se construye se pueden descubrir inconsistencias, ambigüedades,...
- Da estructura al lenguaje de programación lo que hace que sea más fácil de generar código y detectar errores.
- Fácil de ampliar y modificar el lenguaje.

Nos interesan los analizadores deterministas, que son los que de un estado con un símbolo solo puede ir a 1 estado. También admitiremos las transiciones con λ.

Tipos de analizadores:

- **Descendente**: Va desde la raíz (axioma) hasta las hojas (tokens, símbolos terminales). Nosotros trataremos los predictivos solo, que son capaces de elegir el token correcto en cada momento, no los que tienen retroceso.

  LL(k), LL(1) Tengo que leer k tokens para identificar que regla de producción usar.

- **Ascendente**: Va de los nodos hojas (secuencia de tokens) hasta la raíz (axioma). Es predictivo.

  LR(k), que puede ser LR(0), SLR(1), LALR(1) y LR(1)

LR significa que lee de izquierda a derecha (LX) y elige la producción más a la derecha (XR, right).

5.2. Análisis Sintáctico Descendiente

Tenemos una secuencia de tokens x y el objetivo es **determinar si la secuencia es una secuencia del lenguaje** definido por la gramática G. El proceso que seguimos es:

1. **Partir del axioma** la forma sentencial.
2. **Coger un token de izquierda a derecha.**
3. **Seleccionar una regla de producción**. Se debe saber
   según el token actual la regla que usar.
4. **Si coinciden** token y símbolo de la forma sentencial, se **lee el siguiente token**.
5. **Sustituir el símbolo no terminal** de la forma sentencial por la parte derecha de la regla elegida.
6. **Repetir** el proceso hasta que la entrada haya sido procesada.

5.2.1. Con Retroceso

Lo que cambia es la manera de seleccionar la regla, este **lo hace por búsqueda en profundidad con retroceso** lo que hace que tenga una **complejidad de** $O(k^n)$.

5.2.2. Predictivo

Analizador que solo necesita conocer k **tokens de la cadena de entrada, para determinar la regla de producción** que debe aplicarse. El número de tokens, k, necesarios para tomar la decisión de que regla de producción aplicar, define el nombre del analizador, $LL(k)$, $LR(k)$

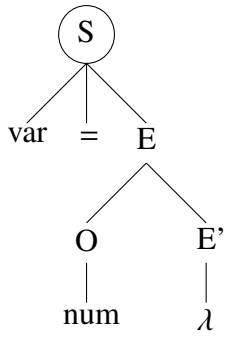
Características:

- Son autómatas a pila deterministas por vaciado.
- No hay retroceso, usa autómatas a pila deterministas predictivos.
- La **complejidad es lineal** $O(n)$, tal que n es el número de tokens que hay en la sentencia (preferible al exponencial del con retroceso)

Para evitar problemas a la hora de seleccionar una regla de selección **se debe eliminar la Recursividad a izquierdas y Factorizar a izquierdas**. La recursividad a izquierdas hay que evitarla desde el principio, pero la factorización si se produce no es un gran problema es fácil de resolverla.

$S \rightarrow var = E$
$E \rightarrow OE'$
$E' \rightarrow \lambda | opE$
$O \rightarrow var | num$

Tabla 5.1: Gramatica para Análisis Sintáctico



$$S \rightarrow var = E \rightarrow var = OE' \rightarrow var = numE' \rightarrow var = num$$

En los casos que se tienen varias reglas de producción entre las que elegir usará el conjunto primero del terminar más a la izquierda.

Conjunto Primero

$PRIMERO(\alpha)$ Todos los Terminales que aparecen más a la izquierda que se derivan de esa forma sentencial. **Son los posibles terminales que pueden aparecer más a la izquierda de cualquier secuencia de producciones de α.**

$$PRIMERO(\alpha) = \{x | (\alpha \rightarrow_* x \cdot \beta), (x \in \Sigma_T \cup \Sigma^*)\}$$

Fig. 5.1: Conjunto Primero

Si $X = \lambda \Rightarrow PRIMERO(X) = \{\lambda\}$

Si $X \in \Sigma_T \Rightarrow PRIMERO(X) = \{X\}$ El conjunto primero de un terminal es él mismo.

Si X tiene alguna regla que empieza por un No Terminal es como calcular el conjunto primero de este. Ojo con lambda.

Conjunto Siguiente

Conjunto primero de todo lo que aparece concatenado de un símbolo No terminal en todas las formas senténciales que se derivan del axioma. Son todos los símbolos Terminales que se producen al derivar desde el axioma.

El conjunto siguiente nunca va a contener a λ. \$ es el token que indica fin de sentencia.

Se aplica sobre los No Terminales de la parte de la derecha de todas las producciones.

$$SIGUIENTE(A) = \{x | (S \rightarrow_* \alpha A \beta), (A \in \Sigma_N), (\alpha \in \Sigma^*), (\beta \in \Sigma^+), (x \in PRIMERO(\beta) - \{\lambda\})\}$$

Fig. 5.2: Conjunto Siguiente

Fases:

- Obtener las reglas de formación de los conjuntos siguientes. Se aplica sobre todos los NT de la derecha de la regla de producción. Tiene la forma de $A \rightarrow \alpha B \beta$, se asocian las partes de la regla con los símbolos, donde B es el NT sobre del que queremos el S(B).

    1. El conjunto siguiente del axioma es $ siempre, pero hay que aplicar el conjunto siguiente sobre sus reglas.
    2. Se aplica $S(B) = S(B) \cup PRIMERO(\beta) - \{\lambda\}$, que si no hay β hace que esta regla no haga nada y se haga la siguiente.
    3. Si $PRIMERO(\beta)$ contiene a λ también se aplica, $S(B) = S(B) \cup S(A)$

- Ir aplicando las reglas, escribiendo los T en una tabla, hasta que el conjunto siguiente no cambie en una iteración.

Conjunto de Predicción

Es el criterio para determinar que regla expandir para un símbolo no terminal, y se aplica sobre las reglas de producción completa.

Dado $A \rightarrow \alpha$:

$$
\begin{aligned}
PREDICCION(A \rightarrow \alpha) &= PRIMERO(\alpha) / \lambda \notin PRIMERO(\alpha) \\
PREDICCION(A \rightarrow \alpha) &= [PRIMERO(\alpha) - \{\lambda\}] \cup S(A) / \lambda \in PRIMERO(\alpha)
\end{aligned}
$$

Fig. 5.3: Conjunto de Predicción

Condiciones LL

Condiciones que debe tener para hacer Análisis Sintáctico Descendente Predictivo. Estas condiciones se aplican sobre NT que tienen más de una regla de producción.

1. **Condición PRIMERO-PRIMERO:** Los conjuntos primeros de las reglas de producción del NT deben ser disjuntos (no tengan símbolos en común).

   $A \rightarrow \alpha_1|...|\alpha_n, P(\alpha_1) \cap ... \cap P(\alpha_n) =$

2. **Condición PRIMERO-SIGUIENTE**: Esta se aplica cuando alguna de las reglas tienen en su conjunto primero λ, entonces el conjunto primero del resto de las reglas (sin incluir la de lambda) deben ser disjuntos con el conjunto siguiente del NT de la izquierda.

Análisis Descendente Recursivo

Estrategia Para construirlo se crea un procedimiento por cada NT que sea capaz de reconocer las distintas reglas de producción que tiene asociadas, para ello va leyendo símbolo a símbolo con token_actual, siguiendo las siguientes reglas:

- **Si se lee un Terminal**: Si coincide el token_actual con lo esperado lee el siguiente token, si no coincide da error sintáctico.
- **Si se lee un NT**: Se llama al procedimiento del NT que se encargara de reconocerlo.
- **Si se lee una lambda**: No se hace nada.

Si la pila se vacía y la entrada ha sido procesada entonces la secuencia de tokens es reconocida, entonces iría al analizador semántico.

Ventajas

- Requieren formalizar una gramática.
- Son fáciles de escribir e interpretar.
- Adecuados para analizadores simples.

Inconvenientes

- Difíciles de ampliar y mantener.
- Coste computacional asociado a la recursividad.
- Construcción específica para el lenguaje reconocido.
- Conjunto reducido de lenguajes independientes de contexto.

Ejemplo:

	Predicción
$S \rightarrow AS'$	$\{id\}$
$S' \rightarrow S$	$\{id\}$
$S' \rightarrow \lambda$	$\{\$\}$
$A \rightarrow id = E$	$\{id\}$
$E \rightarrow OE'$	$\{num, id, fun\}$
$E' \rightarrow opE$	$\{op\}$
$E' \rightarrow \lambda$	$\{id, \$,)\}$
$O \rightarrow num$	$\{num\}$
$O \rightarrow id$	$\{id\}$
$O \rightarrow fun(E)$	$\{fun\}$

Tabla 5.2: Descendente Recursivo: Gramática y Procedimiento Axioma

```
main {
  $ al final de la entrada
  leer_tok()
  S()
  si tok_act = "$"
    ACEPTAR
  sino
    ERROR
}
```

```
S {
  A()
  S'()
}
```

```
S' {
  si tok_act = "id"
    S()
  sino
    si tok_act = "$"
    sino
      ERROR
}
```

```
A {
  si tok_act = "id"
    leer_tok()
    si tok_act = "="
      leer_tok()
      E()
    sino
      ERROR
  sino
    ERROR
}
```

```
E {
  O()
  E'()
}
```

```
E' {
  si tok_act = "op"
    leer_tok()
    E()
  sino
    si tok_act = "id" | "$" | ")"
    sino
      ERROR
}
```

```
O {
  si tok_act = "num"
    leer_tok()
  sino
    si tok_act = "id"
      leer_tok()
    sino
      si tok_act = "fun"
        leer_tok()
        si tok_act = "("
          leer_tok()
          E()
          si tok_act() = ")"
            leer_tok()
          sino
            ERROR
        sino
          ERROR
}
```

id = fun (num op id) $	S
= fun (num op id) $	A S
fun (num op id) $	E A S
(num op id) $	O E A S
num op id) $	E O E A S
op id) $	O E O E A S
op id) $	E O E A S
id) $	E' E O E A S
id) $	E E' E O E A S
) $	O E E' E O E A S
) $	E E' E O E A S
) $	E' E E'E O E A S
) $	E E'E O E A S
) $	E' E E' E O E A S
) $	E E' E O E A S
$	O E A S
$	E A S
$	S
$	S' S
$	S
	ACEPTAR

Tabla 5.3: Descendente Recursivo: Reconocimiento secuencia

Análisis Sintáctico LL(1)

Es un analizador predictivo que procesa los tokens de izquierda a derecha, con derivaciones más a la izquierda.

Necesita solo 1 símbolo de preanálisis para determinar qué producción aplicar.

Análisis Descendente Dirigido por tablas

Características:

- Se construye en árbol de derivación desde el axioma.
- La cima de la pila determina la operación a realizar.
- La sustitución de los símbolos no terminales por producciones está definida en una tabla, en la que las filas son los NT y las columnas los T.
- Debe cumplir las condiciones LL para analizarse con LL(1)

Construcción de la tabla: Se construye con el Conjunto de Predicción, filas NT y columnas Terminales y $.

- Dentro se ponen las reglas de producción, que se ponen en la fila del NT de la izquierda de la regla y en las columnas de todos los T del conjunto de predicción de la regla.

La idea general es:

- Si el símbolo de preanálisis y la cima de la pila son $ es que hemos terminado con éxito.
- Si el símbolo de preanálisis es en un T y la cima de pila es el mismo, pero no $ (sería final), se saca de la pila y se lee token (lo que lo saca de la secuencia a reconocer)
- Si el símbolo de cima de pila es un NT, se consulta la fila del NT y la columna del símbolo de preanálisis, si esta está vacía, error, sino se meta en la cima de la pila.

Ejemplo completo

$S \rightarrow AS'$	$E' \rightarrow opE$
$S' \rightarrow S$	$E' \rightarrow \lambda$
$S' \rightarrow \lambda$	$O \rightarrow num$
$A \rightarrow id = E$	$O \rightarrow id$
$E \rightarrow OE'$	$O \rightarrow fun(E)$

Tabla 5.4: Gramatica ejemplo

1. Calcular Conjuntos Primeros.

	Primero
S	$\{id\}$
S'	$\{id, \lambda\}$
A	$\{id\}$
E	$\{num, id, fun\}$
E'	$\{op, \lambda\}$
O	$\{num, id, fun\}$

Tabla 5.5: Ejem. Conjuntos Primeros

2. Calcular Conjuntos Siguientes.

	Siguiente
S	$\{\$\}$
S'	$\{\$\}$
A	$\{id, \$\}$
E	$\{id, \$,)\}$
E'	$\{id, \$,)\}$
O	$\{op, id, \$\}$

Tabla 5.6: Ejem. Conjuntos Siguientes

$A \rightarrow \alpha B \beta$

1) $S(B) = S(B) \cup P(\beta) - \{\lambda\}$

2) Si $\lambda \in P(\beta) \Rightarrow$

$S(B) = S(B) \cup S(A)$

$\boldsymbol{S \rightarrow AS'}$

$S(A) = S(A) \cup P(S') - \{\lambda\}$

$S(A) = S(A) \cup S(S)$

$S(S') = S(S') \cup S(S)$

$\boldsymbol{S' \rightarrow S}$

$S(S) = S(S) \cup S(S')$

$\boldsymbol{A \rightarrow id = E}$

$S(E) = S(E) \cup S(A)$

$\boldsymbol{E \rightarrow OE'}$

$S(O) = S(O) \cup P(E') - \{\lambda\}$

$S(O) = S(O) \cup S(E)$

$S(E') = S(E') \cup S(E)$

$\boldsymbol{E' \rightarrow opE}$

$S(E) = S(E) \cup S(E')$

$\boldsymbol{O \rightarrow fun(E)}$

$S(E) = S(E) \cup P(")") - \{\lambda\}$

3. Calcular Conjuntos de Predicción.

	Predicción
$S \rightarrow AS'$	$\{id\}$
$S' \rightarrow S$	$\{id\}$
$S' \rightarrow \lambda$	$\{\$\}$
$A \rightarrow id = E$	$\{id\}$
$E \rightarrow OE'$	$\{num, id, fun\}$
$E' \rightarrow opE$	$\{op\}$
$E' \rightarrow \lambda$	$\{id, \$,)\}$
$O \rightarrow num$	$\{num\}$
$O \rightarrow id$	$\{id\}$
$O \rightarrow fun(E)$	$\{fun\}$

Tabla 5.7: Ejem. Conjuntos de Predicción

$$A \rightarrow \alpha$$

$$Si\ \lambda \notin P(\alpha) \Rightarrow Pred = P(\alpha)$$

$$Si\ \lambda \in P(\alpha) \Rightarrow Pred = P(\alpha) - \{\alpha\} \cup S(A)$$

4. Comprobación de las Condiciones de LL(1)

   Condiciones de LL(1)

$$PP \left| \begin{array}{ll} S' \begin{cases} S' \rightarrow S & \{id\} \\ S' \rightarrow \lambda & \{\lambda\} \end{cases} \\ E' \begin{cases} E' \rightarrow opE & \{op\} \\ E' \rightarrow \lambda & \{\lambda\} \end{cases} \end{array} \right. \quad PS \left| \begin{array}{ll} S' \begin{cases} S' \rightarrow S & \{id\} \\ S' \rightarrow \lambda & \{\$\} \end{cases} \\ E' \begin{cases} E' \rightarrow opE & \{op\} \\ E' \rightarrow \lambda & \{id, \$,)\} \end{cases} \end{array} \right.$$

5. Tabla de Conjuntos de Predicción

	id	**=**	**op**	**num**	**fun**	**(**	**)**	**$**
S	1							
S'	2							3
A	4							
E	5			5	5			
E'	7		6				7	7
O	9			8	10			

Tabla 5.8: Ejem. Tabla de Conjuntos de Predicción

6. Prueba de lectura de secuencia de entrada (falta que se vaya) y los NT que quedan se van con el $)

id = fun (num op id) $	S $
id = fun (num op id) $	A S' $
id = fun (num op id) $	id = E S' $
= fun (num op id) $	= E S' $
fun (num op id) $	E S' $
fun (num op id) $	O E' S' $
fun (num op id) $	fun (E) E' S' $
(num op id) $	(E) E' S' $
num op id) $	E) E' S' $
num op id) $	O E') E' S' $
num op id) $	num E') E' S' $
op id) $	E') E' S' $
op id) $	op E) E' S' $
id) $	E) E' S' $
id) $	O E') E' S' $
id) $	id E') E' S' $
) $	E') E' S' $
) $	) E' S' $
$	E' S' $
$	S' $
$	$

Tabla 5.9: Prueba de lectura de secuencias ASI

5.3. Análisis Sintáctico Ascendente

Una secuencia de tokens x pertenece al lenguaje definido por G si genera una secuencia válida del lenguaje. Se van aplicando reglas de producción que generan un árbol sintáctico.

1. Coger tokens de izquierda a derecha.
2. Buscar el pivote de reducción (que reduciremos)
3. Aplicar derivaciones más a la derecha, sustituyen por NT.
4. Repetir el proceso hasta que la entrada ha sido procesada y reducida al axioma.

Pivote de reducción: Subcadena w de una forma sentencial x, $x \in \sum^+$, que es la parte derecha de una regla de producción y se sustituye a la inversa, es decir se mete el NT del lado izquierdo.

La decisión es si añadir otro terminal al pivote (desplazar) o aplicar la producción en orden inverso.

- $x = uwv, A \rightarrow w | S \rightarrow uAv$
- Reduce el pivote más a la izquierda.
- A la derecha del pivote solo hay terminales o λ.
- El pivote se puede buscar utilizando una pila.

5.3.1. Prefijo

Terminales y No Terminales que se pueden encontrar en pila, que nos permite llegar al axioma.

5.3.2. Prefijo viable

Terminales y No Terminales que pueden encontrarse en la pila y pueden formar parte de un pivote que permita reducir hasta el axioma.

Un prefijo de un prefijo viable también es un prefijo viable.

La estrategia es construir un autómata que reconozca los prefijos viables de la gramática, el autómata LR(0) los reconoce.

$$\begin{cases} E \rightarrow E + M | M \\ M \rightarrow M * T | T \\ T \rightarrow int \end{cases} \quad \textit{Respuesta: int, T, M*, M*T, E, E+, E+M}$$

5.3.3. Análisis LR

Para algunas gramáticas independientes de contexto, LR(k), se puede construir un autómata a pila determinista.

- k es la cantidad de símbolos de preanálisis necesarios para tomar una decisión.

Tipos de analizadores:

- **SLR(1)** es un LR simple. La clase más pequeña de gramáticas, número de estados más pequeño, simple y rápida.
- **LR(1)**. La clase más grande de gramáticas, el mayor número de estados, construcción larga y penosa.
- **LALR(1)**. La clase intermedia de gramáticas, número de estados como SLR(1), construcción larga y penosa.

En cuanto a que gramáticas pueden representas: $G2 > LR > LALR > SLR$

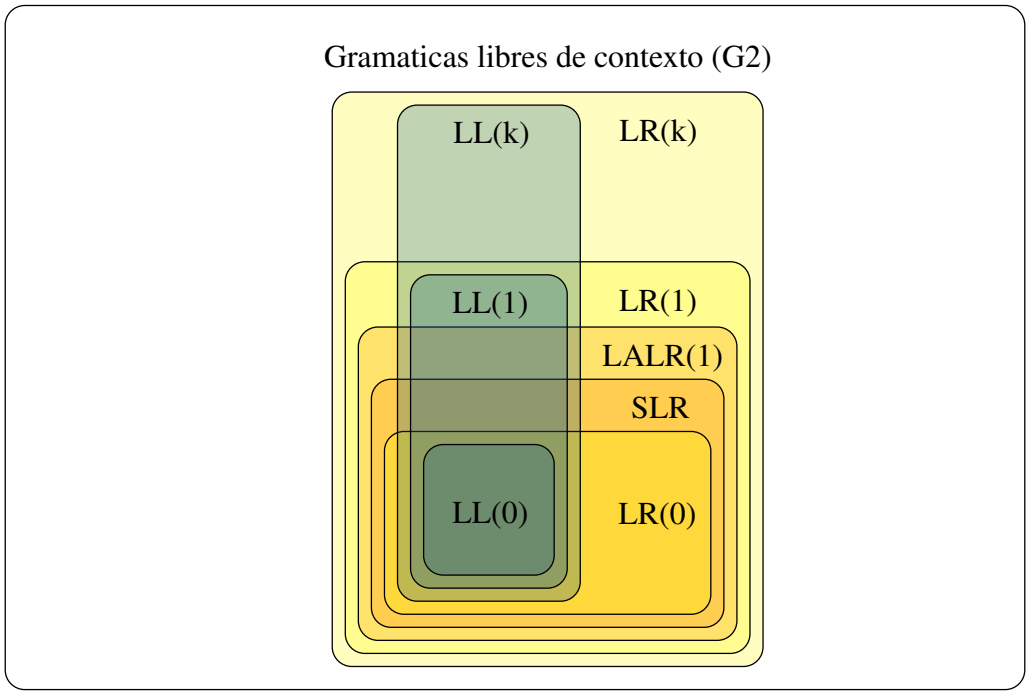


Fig. 5.4: Comparación de grámaticas

5.3.4. Análisis Sintáctico LR(1)

Esquema del Análisis Ascendente dirigido por tabla

En este caso las filas son el Conjunto LR(0), también llamado **conjunto LR canónico**, que son todos los estados resultados de hacer el cierre del axioma y los sucesivos Ir_a.

Las columnas son **Acciones**, Terminales y $, y por otro lado **Ir_a**, No Terminales.

Item LR(0)

Son reglas de producción con ·, que indica que lo de la izquierda está en la pila.

- $A \rightarrow \alpha\beta \cdot \gamma$, α, β en pila.
- Los símbolos a la derecha todavía no se han procesado. γ sin procesar.
- Cuando se produce que a la derecha de · no hay símbolos es posible que haya encontrado un pivote de reducción.

Analizador LR(0)

A partir de los ítems se puede construir un AFD que realice el análisis de prefijos viables.

Los estados de ese AFD se les llama **Conjunto canónico LR(0)**, que es lo que buscamos para la tabla, cada uno de esos estados está formado por un conjunto de ítems.

Cuando el axioma tenga varias producciones, se amplía la gramática, haciendo que un nuevo NT sea el axioma y produzca el axioma anterior. $S' \rightarrow S$

Para crear los estados del Conjunto canónico LR(0) utilizaremos:

- **Cierre(ítems)** Determina los ítems que pertenecen a un estado.

  Lo que hace es añadir a todas las reglas de producción del NT izquierdo, el punto · a la izquierda. Además, si el punto está justo a la izquierda de un NT se hace también para sus reglas.

- **Ir_a(ítems, Símbolo)** Genera un nuevo estado a partir de los ítems de un estado y símbolo gramaticales.

  Hace el cierre de pasar el punto al otro lado de Símbolo, que se pasa como parámetro. Es vacío cuando no hay ningún item que tenga el punto a su izquierda.

El analizador LR(0) es un Autómata a Pila Determinista, los estados del analizador son conjuntos de ítems.

Algoritmo de creación del analizador LR(0)

1. Crear el conjunto canónico LR(0)

   a) Ampliar la gramática $S' \rightarrow S$, cuando axioma tiene más de una regla.

   b) $E_0 = Cierre(S' \rightarrow S)$.

c) Repetir mientras son creados nuevos estados, $E_i = Ir\_a(I_k, \alpha), \forall \alpha \in \sum^+$, hacer para aquellos símbolos que estén a la derecha de un punto, si no darán vacío y no nos son de utilidad. Que crea tantos estados como posibles aplicaciones de la función Ir_a en todos los estados, mientras no genere estados exactamente iguales, pero puede haber reglas repetidas.

2. Asociar acciones a los ítems de los estados, que son: Desplazar, Reducir, Ir a otro estado, Aceptar y Error.

Ejemplo:

$$
\begin{aligned}
&S' \to S \\
&S \to T \\
&S \to S\,opT \\
&T \to num \\
&T \to (S)
\end{aligned}
$$

1. Ampliamos: $S' \to S$

$$
\begin{aligned}
&S' \to S \\
&S' \to S \\
&S \to T \\
&S \to S\,opT \\
&T \to num \\
&T \to (S)
\end{aligned}
$$

2. Hacer el cierre de la ampliación del axioma

$$
\begin{aligned}
E_0 :&S' \to \cdot S \\
&S \to \cdot T \\
&S \to \cdot S\,opT \\
&T\cdot \to num \\
&T\cdot \to (S)
\end{aligned}
$$

3. Hacer Ir_a para $\Sigma = \{S, T, op, num, (,)\}$, no hace falta hacer todos, solo tiene sentido para aquellos que tienen un punto a la izquierda. Un estado por cada Ir_a que se puede hacer desde E_0

$$
\begin{aligned}
E_1 :&S' \to \cdot S \\
&S \to S \cdot opT \\
E_2 :&S \to T\cdot \\
E_3 :&T \to num\cdot \\
E_4 :&T \to (\cdot S)
\end{aligned}
$$

$$
\begin{aligned}
E_5 :& S \rightarrow S\,op \cdot T \\
& T \rightarrow \cdot num \\
& T \rightarrow \cdot(S) \\
E_6 :& T \rightarrow (S \cdot) \\
& S \rightarrow S \cdot opT \\
E_7 :& S \rightarrow S\,opT \cdot \\
E_8 :& T \rightarrow (S) \cdot
\end{aligned}
$$

Asociar acciones a los ítems de los estados:

- $\forall I_k = \{A \rightarrow \alpha \cdot x\beta\} I_k \in E_j y x \in \sum_T, Accion(E_j, x) = DesplazarE_n, E_n = Ir\_a(E_j, x)$
- $\forall I_k = \{A \rightarrow \alpha \cdot X\beta\} I_k \in E_j y X \in \sum_N, Ir\_a(E_j, X) = E_n$
- $\forall I_k = \{A \rightarrow \alpha \cdot\} I_k \in E_j, Accion(E_j, x) = ReducirA \rightarrow \alpha, \forall x \in \{\sum_T, \$\}$
- $I_k = \{S' \rightarrow S \cdot\} I_k \in E_j, Accion(E_j, \$) = Aceptar$

Ejemplo:

	Acción	**Entrada**					**Ir_a**	
		(	**num**	**)**	**op**	**$**	**S**	**T**
0	Desplazar	4	3				1	2
1	Desplazar				5	ACP		
2	Reducir	$S \rightarrow T$						
3	Reducir	$T \rightarrow (S)$						
4	Desplazar	4	3				6	2
5	Desplazar	4	3					7
6	Desplazar							
7	Reducir	$S \rightarrow SopT$						
8	Reducir	$T \rightarrow (S)$						

Tabla 5.10: Analizador LR(0)

LR(0) toma la decisión de desplazar o reducir SIN depender del token de entrada, solo del estado, pero si del estado al que transita.

Se puede producir **Conflicto de Reducción-Desplazamiento**, cuando para un mismo estado y símbolo hay dos acciones. Para solventar esto se amplía el LR(0) a SLR(1)

Ventajas

- Fáciles de construir.
- Las reducciones están asociadas a estados.

Inconvenientes

1. Son muy pocas las que se pueden utilizar.
2. Estrategias de recuperación de error.

5.3.5. Análisis SLR(1)

Es un Autómata a Pila Determinista.

- Los estados del analizador son conjuntos de ítems.

Se construye igual que el analizar LR(0), pero la asignación de la acción Reducir cambia.

Algoritmo de creación del analizador SLR(1)

- Crear el conjunto canónico LR(0), los estados.
- Asociar acciones (Desplazar, Reducir, Ir_a, Aceptar, Error) a los ítems de los estados.

Asociar acciones a los ítems de los estados:

- $\forall I_k = \{A \rightarrow \alpha \cdot x\beta\} I_k \in E_j y x \in \sum_T, Accion(E_j, x) = DesplazarE_n, E_n = Ir\_a(E_j, x)$
- $\forall I_k = \{A \rightarrow \alpha \cdot X\beta\} I_k \in E_j y X \in \sum_N, Ir\_a(E_j, X) = E_n$
- $\forall I_k = \{A \rightarrow \alpha \cdot\} I_k \in E_j, Accion(E_j, x) = ReducirA \rightarrow \alpha, \forall x \in Siguiente(A), \alpha \in \sum^*$
  Se pone reducción solo para los T de $S(A)$
- $I_k = \{S' \rightarrow S \cdot\} I_k \in E_j, Accion(E_j, \$) = Aceptar$

Cuando se indica R2, indica que la regla que se reduce es la numerada como 2.

Condiciones SLR(1): Tienen que cumplirse dos condiciones para poder hace un análisis SLR(1).

No es necesario eliminar la recursividad a derechas e izquierda y ni es necesario factorizar.

- Sin conflictos **Desplazamiento-Reducción**, debe darse que en los estados que haya reducciones no haya otro item que tenga desplazar para un símbolo del conjunto siguiente de la reducción.

  $A \rightarrow \alpha \cdot x\beta$

  $B \rightarrow \alpha \cdot$

  Entonces $x \notin Siguiente(B)$

- Sin conflictos **Reducción-Reducción**, debe cumplir que cuando haya más de una regla de reducción en un estado, sus conjuntos siguientes sean disjuntos.

  $A \rightarrow \alpha\cdot$

  $B \rightarrow \gamma\cdot$

  Entonces $Siguiente(A) \cap Siguiente(B)$

0) S' → S
1) S' → var = E
2) E → O
3) E → O op E
4) O → var
5) O → num

$E_0 : S' \rightarrow \cdot S$
$S \rightarrow \cdot var = E$
$E_1 : S' \rightarrow S\cdot$
$E_2 : S \rightarrow var\cdot = E$
$E_3 : S \rightarrow var = \cdot E$
$E \rightarrow \cdot O$
$E \rightarrow \cdot OopE$
$O \rightarrow \cdot var$
$O \rightarrow \cdot num$
$E_4 : S \rightarrow var = E\cdot$
$E_5 : E \rightarrow O\cdot$
$E \rightarrow O \cdot opE$
$E_6 : O \rightarrow var\cdot$
$E_7 : O \rightarrow num\cdot$
$E_8 : E \rightarrow Oop \cdot E$
$E \rightarrow \cdot O$
$E \rightarrow \cdot OopE$
$O \rightarrow \cdot var$
$O \rightarrow \cdot num$
$E_9 : E \rightarrow OopE\cdot$

	var	**=**	**op**	**num**	**$**	**S**	**E**	**O**
E_0	DE_2					E_1		
E_1					ACP			
E_2		DE_3						
E_3	DE_6	E_7					E_4	E_5
E_4					R1			
E_5			EE_8		R2			
E_6			R4		R4			
E_7			R5		R5			
E_8	EE_6			DE_7			E_9	E_5
E_9					R3			

var = var op num $	E_0 $
= var op num $	E_2 var E_0 $
var op num $	E_3 = E_2 var E_0 $
op num $	E_6 var E_3 = E_2 var E_0 $
op num $	E_5 O E_3 = E_2 var E_0 $
num $	E_8 op E_5 O E_3 = E_2 var E_0 $
$	E_7 num E_8 op E_5 O E_3 = E_2 var E_0 $
$	E_5 O E_8 op E_5 O E_3 = E_2 var E_0 $
$	E_9 E E_8 op E_5 O E_3 = E_2 var E_0 $
$	E_4 E E_3 = E_2 var E_0 $
$	E_1 S E_0 $
	ACEPTAR

Tabla 5.11: Ejemplo analizador SLR(1)

5.3.6. Comparación LL vs LR

Simplicidad: LL es más fácil de construir y completar.

Tamaño: LL es más pequeño.

Generalidad: Todas las LL son LR, pero no al contrario.

Detección de error: LR requiere menos acciones para detectar el error.

Recuperación de errores: LL es más sencillo de implementar.

6. TEMA 4: TRATAMIENTO DE AMBIGUEDAD Y ERROR

Acceso en Drive a las diapositivas

7. TEMA 5: ANÁLISIS SEMÁNTICO

7.1. Introducción

El análisis semántico es una **extensión del análisis sintáctico para la compresión del programa**, para comprobar que tiene sentido y que está conectado. Destaca por la verificación de tipos.

Es un paso **previo a su traducción, aunque hay partes que se pueden ir haciendo**. También modifica la tabla de símbolos.

Las gramáticas independientes del contexto **(G2) no son suficientes para este análisis**, se necesita realizar comprobaciones de larga distancia en el árbol sintáctico (que no se pueden representar con reglas de producción) y no solo comprobar que se deriva de la gramática del lenguaje.

Para cumplir lo anterior se utilizan Gramáticas de atributos (gramáticas atribuidas), que son más ricas.

Algunas de las comprobaciones que se realizan durante el Análisis Semántico:

- Comprobación de **tipos**, los operadores y operandos deben ser compatibles.
- Comprobaciones de **unicidad**, en aquellas situaciones en las que un objeto solo puede definirse una vez exclusivamente.
- Comprobaciones **relacionadas con nombres**, cuando el mismo nombre debe aparecer dos o más veces.
- Comprobación del **flujo de control**, controlar las proposiciones para abandonar el flujo de control, que tiene finalización.

7.2. Gramáticas atribuidas

Son gramáticas G2 a las que se añaden atributos y reglas de evaluación de atributos (reglas semánticas).

- Cada **atributo** es una variable que representa una propiedad del símbolo terminal o no terminal, puede ser una cadena, número, tipo, posición de memoria, etc.
- **Reglas semánticas** (Reglas para calcular/Reglas para comprobar), se asocian a las producciones sintácticas (reglas de producción). Se definen en función de los atributos de los símbolos en la producción.
- Sobre los atributos y reglas hay unas **condiciones semánticas**.

Se pueden construir Árboles adornados o decorados, donde se muestra la relación entre atributos de una misma regla de producción.

Gramáticas atribuida $GA = \{G, A, R, B\}$

- G - **Gramática** libre de contexto (G2), $G = \{\Sigma_N, \Sigma_T, S, P\}$.
- A - Conjunto de **atributos** asociados a cada símbolo X. Son disjuntos.
- R - **Reglas semánticas** asociadas a cada regla en P.
- B - **Condiciones** asociadas a cada regla de P.

Existen diferentes Especificaciones:

- **DDS - Definición dirigida por la sintaxis**, acciones tras cada producción (sin orden específico).

producción	**Acciones semánticas**
S ::= T id L ;	S.cod = ”var” || id.cod || L.cod || ”:” || T.cod
L ::= , id L	L_0.cod = ”,” || id.cod || L_1.cod
L ::= λ	L.cod = ””
T ::= float	T.cod = ”real”
T ::= int	F.cod = ”int”

Tabla 7.1: Ejemplo DDS

- **EDT - Esquema de Traducción**, las acciones semánticas se intercalan con los símbolos de la producción, indicando el orden de ejecución de las acciones semánticas. Es una notación para hacer un traductor. Tiene dos tipos:

- EDT **solo con atributos sintetizados**, las acciones se ponen al final de la producción.
- EDT con **atributos sintetizados y heredados**, la posición de las acciones debe tener en cuenta los cálculos y atributos que contribuyen.
  - Un atributo heredado (en la parte derecha de la producción) DEBE calculará con una acción antes de dicho símbolo.
  - Una acción NO debe referirse a un atributo sintetizado de un símbolo que esté a la derecha de la acción.
  - Para calcular el No Terminal izquierdo DEBE haberse calculado previamente todos los atributos de los símbolos que participen.

Una gramática atribuida GA es **completa** si, para todo atributo de toda regla hay una acción semántica asociada para calcularlo.

- Para toda regla P: $X \rightarrow \alpha$ en P, AS(X) está en AC(P).
- Para toda regla Q: $Y \rightarrow \alpha X \beta$ en Q, AH(X) está en AC(Q).
- $AS(X) \cup AH(X) = A(X)$, $AS(X) \cap AH(X) = \emptyset$

Una gramática atribuida GA está **bien definida** si, para cualquier sentencia en L(GA), todos los atributos pueden ser calculados (este proceso se llama evaluación de la gramática). Además, una sentencia de su lenguaje tiene atributos correctos si se verifican las condiciones en B(P) para cada regla P de la gramática.

- **Gramática Atribuida bien definida ⇒ Gramática Atribuida completa**

En el caso de buscar que esté bien definido es más fácil buscar un fallo (para decir que no lo es), que comprobar que se cumple para todas las sentencias del lenguaje (que es inabarcable). Si no se encuentra el contraejemplo se dice que no se puede demostrar que no lo es.

7.3. Tipos de atributos y gramáticas

7.3.1. Atributos heredados y sintetizados

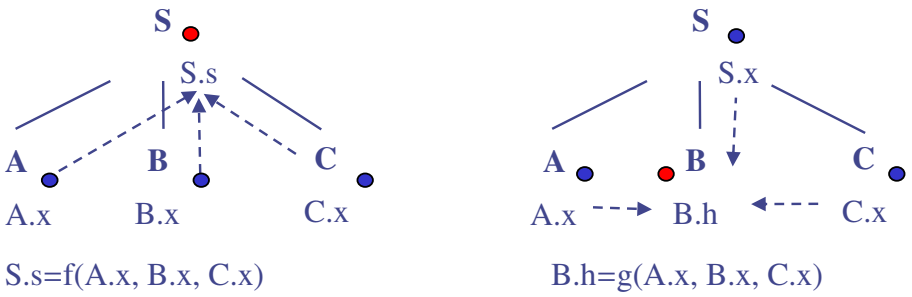


Fig. 7.1: Tablas para registros

Dos tipos de atributos:

- **Sintetizados:** El valor de un nodo (parte izquierda) depende solo de los valores de los nodos hijos (parte derecha). Se hace de abajo hacia arriba.
- **Heredados:** Los valores se pasan a niveles inferiores del árbol, el valor depende de los hermanos y del padre (como un sumidero de datos).

Los atributos mantienen su tipo en toda la gramática, y los tokens (símbolos terminales) solo tienen atributos sintetizados.

Los atributos heredados se tratan utilizando estructuras auxiliares para pasar los datos, como puede ser una tabla o una pila.

Atributos calculados AC(P): Atributos calculados con producción P, que exista una regla de cálculo asociada a la obtención de su valor.

- **Atributos sintetizados AS(X):** Cuando lo que se calcula es el símbolo de la parte izquierda de la regla.
- **Atributos heredados AH(X):** Cuando el valor que se calcula es el producido en una regla de producción.

7.3.2. Evaluación de las reglas semánticas

Hay dos métodos:

- Métodos con **árbol de análisis sintáctico**: Se realizan en el momento de compilación y el orden que se sigue viene dado por el grafo de dependencias (ordenamiento topológico) de cada nodo construido según el árbol de análisis sintáctico. No funciona para gramáticas con ciclos.
- Métodos de evaluación **durante el análisis sintáctico**: La evaluación se hace en una sola pasada y no es necesario construir un grafo de dependencias de forma explícita, lo que lo hace más eficiente.
  - Hay métodos **basados en reglas**, en el que las acciones semánticas asociadas a las producciones se analizan a mano o con alguna herramienta.
  - Hay métodos **sin recuerdo**, el orden de evaluación no tiene en cuenta las reglas semánticas.

7.3.3. Evaluación durante el análisis sintáctico

- **S-A Grammars - Gramáticas de Atributos Sintetizados:** Gramática donde solo existen atributos sintetizados.

  Se puede evaluar con un analizador ascendente, mediante una pila, y descendente por medio de rutinas y valores devueltos.

- **L-A Grammars - Gramáticas de Atributos por la Izquierda:** Gramática donde todos los atributos heredados de cualquier producción dependen solo de atributos de símbolos a su izquierda o del símbolo de la parte izquierda de la producción.

  La información debe fluir de izquierda a derecha.

  Las L-A de tipo LL(1) pueden evaluarse con analizadores descendentes y ascendentes.

  Solo algunas de las LR(1) que NO son LL(1) pueden evaluarse con L-A mediante un analizador ascendente.

7.4. Evaluación de Atributos

7.4.1. Evaluación Ascendente con Atributos Sintetizados

Los atributos sintetizados se pueden evaluar durante el análisis de la entrada con un analizador sintáctico ascendente.

La pila de estados LR tiene asociados los símbolos gramaticales. Basta con crear otra pila con los valores de los atributos sintetizados asociados.

Con cada reducción se calculan los nuevos atributos sintetizados accediendo en la pila a los atributos de símbolos gramaticales del lado derecho de la producción

7.4.2. Grafos de dependencia

La manera de resolver las gramáticas en el peor caso se consigue construyendo uno de estos grafos.

Los atributos no pueden evaluarse en cualquier orden, tras tenerlo construido se pasa un algoritmo que lo determina.

Nos permite representar interdependencias entre atributos heredados y sintetizados de un árbol de análisis sintáctico.

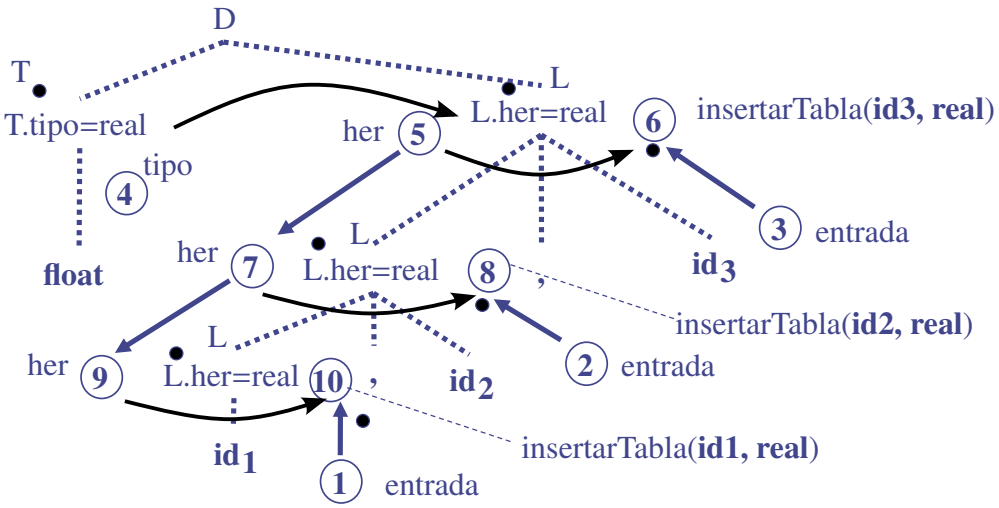


Fig. 7.2: Grafo de dependencia

Creación: Cada una de las reglas de producción $A \rightarrow X_1...X_n$ representan una parte del grafo, de tal manera que la unión de todas es el grafo completo.

1. Se crea un nodo por cada atributo $X_i.a_j$ de cada símbolo de la producción.
    - Atributos heredados - Izquierda.

- Atributos sintetizados - Derecha.

2. Para cada regla semántica $X_i.a_j = f(..., X_k.a_l, ...)$ se hacen arcos desde cada nodo $X_k.a_l$ hacia el nodo $X_i.a_j$, esto se repite para cada k y l afectada por la regla.

7.4.3. Árbol de Sintaxis abstracta

Es una alternativa más sencilla, el grafo de dependencias es más completo. Se utiliza cuando no es una Gramática de atributos por la izquierda mediante esquema de traducción con atributos sintetizados (siempre).

Se usa para los casos en los que la gramática no se puede evaluar durante el análisis sintáctico.

Evaluación en varias pasadas: Se construye el árbol explícitamente y se evalúa en función del orden de recorrido.

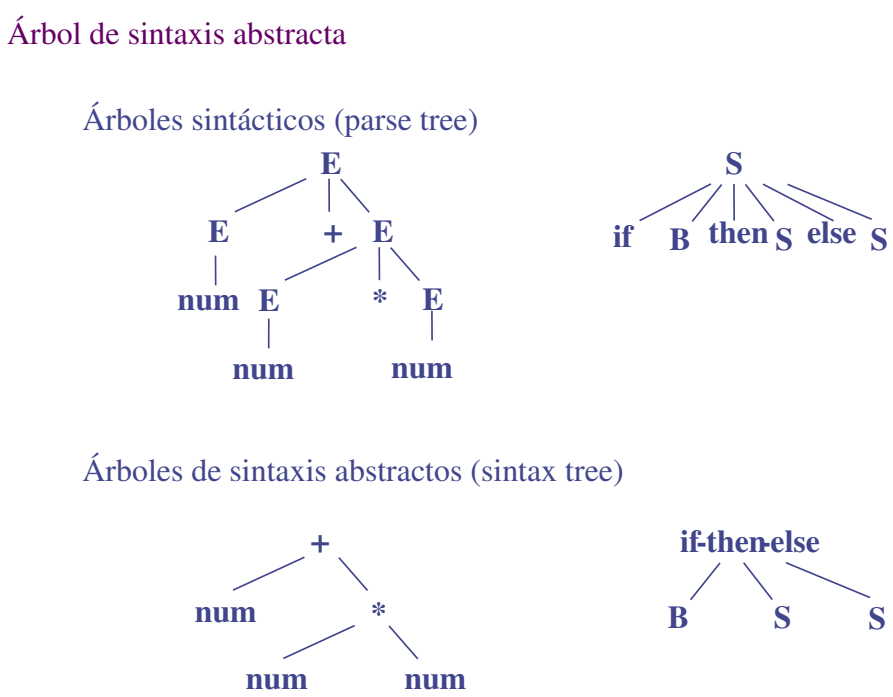


Fig. 7.3: Árbol Sintáctico Abstracto

El Árbol sintáctico abstracto es una estructura de datos que condensa un árbol de análisis sintáctico.

- Las operaciones y las palabras claves no forman parte de las hojas, sino que son absorbidas por el padre.
- Los elementos que tienen información semántica representan las hojas.

Construcción en la programación:

- Un nodo para cada operador y cada operando.
- Un nodo almacena su tipo y su valor si es un operando, en caso de ser un operador en vez de valor almacena el puntero a los operadores. Los punteros y tipo son atributos sintetizados

- hazNodo (operador, izquierda, derecha)
- hazHoja (id, entrada)
- hazHoja (núm, val)

7.5. Orden de recorrido del árbol

Los atributos heredados los calculamos antes y los sintetizados después.

Nos vamos moviendo de izquierda a derecha.

Una vez calculado el heredado se puede pasar al sintetizado.

Complejidad de la evaluación

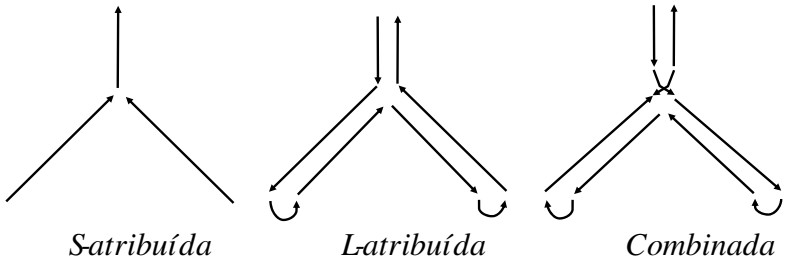


Fig. 7.4: Complejidad de la evaluación

- S-atribuida, Gramáticas atribuidas, va hacia arriba. Calculamos lo de los hijos y lo pasamos hacia arriba.
- L-atribuida, Gramáticas por la izquierda. De forma sistemática, se hace una sola visita, recorriendo el árbol.
- Combinada, hay que ver el orden de dependencia entre atributos para construir el orden.

Orden dinámico: Para aquellos que hasta que no lo construimos no sabemos el orden, cuando no está fijada la relación entre atributos.

Para estos casos es necesario construir el grafo de dependencias, no nos vale el Árbol Sintáctico Abstracto.

Orden estático: Analizando la gramática y las dependencias se establece el orden a priori.

- **Caso que no necesita un árbol de atributos:** Gramáticas S-A y L-A, las de una pasada.
- **Sobre el árbol de atributos:** Los 3 primeros se puede hacer de una sola pasada por el árbol en las que se calculan los atributos.

- **Recorrido post-orden**, depende solo de atributos sintetizados.
- **Recorrido pre-orden**, aparecen atributos heredados, el caso de las Gramáticas L-A.
- **Recorrido combinado**, atributos sintetizados dependen de heredados.
- **Recorrido de varias pasadas, Recorrido combinado**, atributos heredados dependen de sintetizados.

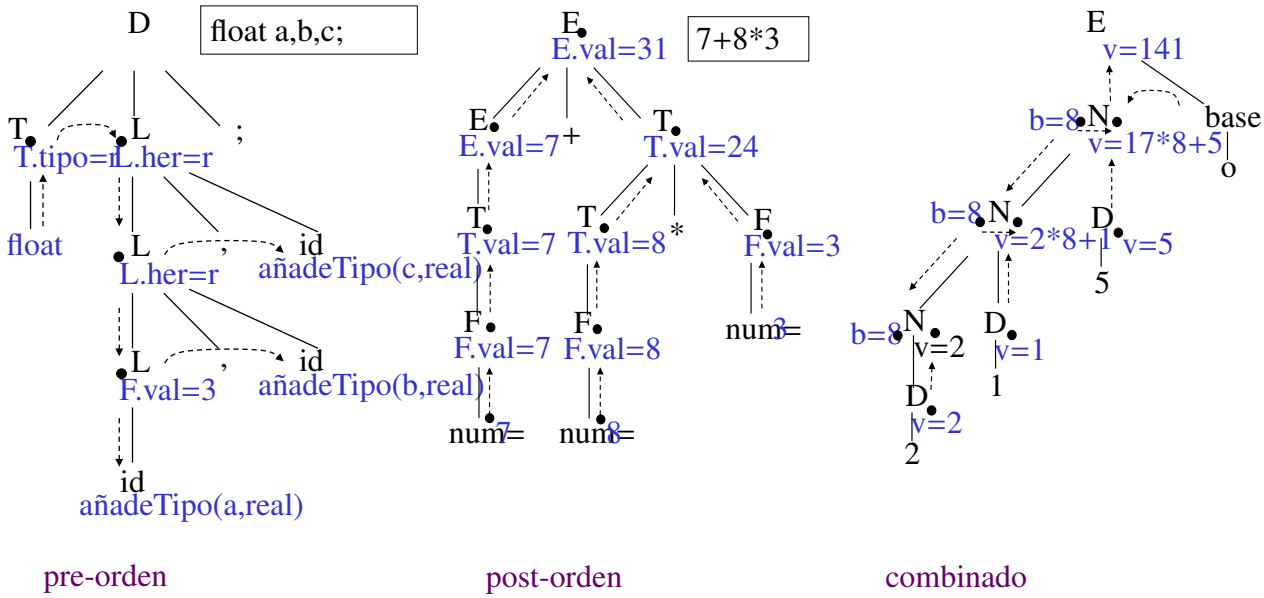


Fig. 7.5: Recorridos grados de dependencias

8. TEMA 6: COMPROBACIÓN DE TIPOS

8.1. Introducción

Su función es representar y mantener la información de tipos (inferencia).

Comprueba que el tipo de una construcción tenga sentido en su contexto según el lenguaje.

Se encuentra tras la construcción del TSA (árbol sintáctico abstracto). ??

En la gramática atribuida es sencillo, pero no siempre es así y para los tipos más complejos se usan tablas.

8.1.1. Ejemplo Comprobación de Tipos

Es una aplicación de evaluación de gramática atribuida

No siempre es tan sencillo: f(*v[i+j]);

- Las variables i y j han de estar declaradas, tener definas en su tipo la operación suma
- y calcular (inferir) el tipo de dicha operación
- Respecto al tipo calculado, deberá ser entero o convertible implícitamente (promocionable) a entero, puesto que en C los índices de un array han de ser enteros
- La variable v deberá estar declarada y poseer como tipo un array o puntero (en C el operador [] se puede aplicar a punteros) de punteros a otro tipo T
- La función f tendrá que haber sido declarada previamente y poseer un único parámetro de tipo T o promocionable a un tipo T

Un lenguaje **especifica que operaciones son válidas para cada tipo**, mediante la formalización de reglas semánticas de verificación.

Se **detectan errores** como: Acceso incorrecto a memoria, Límites de abstracción, mal uso de estructuras, etc.

Tipos de lenguajes:

- **Estáticamente tipificados:** La mayoría de las comprobaciones se realizan en tiempo de **compilación** (C, Java). Menos libres, pero más seguros.
- **Dinámicamente tipificados:** La mayoría de las comprobaciones en **ejecución** (Scheme, LISP)
- **No tipificados:** Ninguna comprobación (código ensamblador)

8.2. Sistemas de tipos

8.2.1. Expresiones de tipos

Representan el tipo de las construcciones los valores posibles, todos los tipos de datos y operaciones aplicables.

- Tipo de Expresión resultante de aplicar operadores aritméticos.
- Resultado de aplicar operador &.
- Tipo de llamada a función.
- ...

Pueden ser:

- **Tipos básicos:** El nombre de un tipo es una expresión de tipo. Su valor es una constante que lo representa.
  - boolean, char, integer, real, vacío, error_tipo,...
- **Constructor de tipos y expresiones de tipos:** Son tipos más complejos formados de tipos básicos.
  - **Array:** Es una estructura formada por una serie de elementos de un tipo, se especifica el número de elementos y el tipo de estos. $array[1..10]$
  - **Productos cartesianos:** Asociación de tipos en una estructura.
  - **Registros:** Es el producto de diferentes tipos, es decir usar como tipo productos cartesianos de otros tipos definidos.
    $record((\text{"}direccion\text{"} : integer) \times (\text{"}lexema\text{"} : array(1.,15, Char)))$
    $array(1..10, record((\text{"}direccion\text{"} : integer) \times (\text{"}lexema\text{"} : array(1..15, char)))$
  - **Punteros:** Dado un tipo T, el $Pointer(T)$ es el tipo del puntero hacia elementos de tipo T.
  - **Funciones:** Es una relación entre un tipo origen y un tipo destino. El origen suele ser un producto cartesiano.
    Una función del tipo $f(a, b : char) : \tilde{i}nteger$ tiene asociado $(char \times char) \rightarrow Pointer(integer)$

Representación de tipos: Las representaciones de expresiones de tipo se pueden hacer en forma de Árbol o con un Grafo dirigido acíclico.

Tipos anónimos: No se indica el tipo explícitamente.

Tipos declarados: Se declara el tipo (typedef) para luego crear variables.

8.2.2. Comprobación estática y dinámica

Estática en los lenguajes que los tipos no cambian.

Dinámica para los lenguajes más complejos, que pueden variar el tipo directamente.

Sistemas de tipos seguro: protege la integridad de abstracciones creadas por el programador, aunque algunas comprobaciones solo puede ser dinámicas.

8.3. Especificación de un comprobador de tipos sencillo

Ejemplo de lenguaje, con todos los identificadores declarados antes de uso.

8.3.1. Tipos de lenguaje

- Tipos básicos
  - Char
  - Integer
  - Error_Tipo
- Tipos complejos
  - Array[n] of T
  - $\hat{T}$

8.3.2. Gramatica del lenguaje

P → Ds E
Ds → D ; Ds | λ
D → id : T
T → char | integer | array [num] of T |^T
E → literal | num | id | E mod E | E [E] | E^

Tabla 8.1: Ej. Gramatica

8.3.3. Acciones semanticas constructor de tipos

P → Ds E	
Ds → D ; Ds	
Ds → λ	
D → id : T	{ añadeTipo (id.entrada, T.tipo) }
T → char	{ T.tipo ::= char }
T → integer	{ T.tipo ::= integer }
T → ^T	{ T_0.tipo ::= pointer (T_1.tipo) }
T → array [num] of T	{ T_0.tipo ::= array(1..num.val, T_1.tipo) }

Tabla 8.2: Ej. Acciones semanticas constructor de tipos

8.3.4. Acciones semanticas verificación de tipos

Constantes

E → literal	{ E.tipo ::= Char }
E → num	{ E.tipo ::= Integer }

Identificadores

E → id { E.tipo ::= buscaTipo(id.entrada) }

Operadores

E → E mod E

```
1 { if (E1.tipo = Integer) and
2       (E2.tipo = Integer)
3          E0.tipo = Integer
4   else E0.tipo = Error_Tipo }
```

Array

E → E[E]

```
1 { if (E2.tipo = Integer) and
2       (E1.tipo = array(s, t) ->)
3          E0.tipo = t
4   else E0.tipo = Error_Tipo }
```

Punteros

E → E^

```
1 { if (E1.tipo = pointer(t))
2          E0.tipo = t
3   else E0.tipo = Error_Tipo }
```

8.3.5. Tipos de sentencias

Ampliar G para permitir declaración de funciones

$$T \rightarrow T \text{ '}\rightarrow\text{' } T \quad \{ T_0.tipo = T_1.tipo \rightarrow T_2.tipo \}$$

Llamada a la función, con parámetros

$E \rightarrow E_1 (E_2)$

```
{ if (E2.tipo = s) and (E1.tipo = s->t)
        then E0.tipo = t
  else E0.tipo = Error_Tipo }
```

8.3.6. Conversiones de tipos

Algunas operaciones pueden aplicarse a operandos de distintos tipos, a un tipo más grande (coerción)

8.4. Tablas de Símbolos

Recogen las diferentes declaraciones del programa.

- Constantes
- Variables
- Funciones
- Tipos

Tiene una estructura para buscar por identificador, hash. Almacena identificadores definidos en el programa.

- Se usa en diferentes fases del análisis y síntesis.
- Funciones de inserción, búsqueda y eliminación.

NOMBRE	TIPO	TAMAÑO	DIRECCIÓN
a	Entero	4	100
b	Entero	4	104
c	Real	8	112
d	Real	8	120
e	Carácter	1	121
f	Carácter	1	122

Dos usos en la verificación semántica

- Que en la declaración de una variable no haya colisiones.
- Al usar una variable que sea del tipo esperado.

Tablas para estructuras registro, se almacena en la tabla de tipos el registro y en la tabla de registros se definen sus componentes con sus tipos y el registro que pertenece.

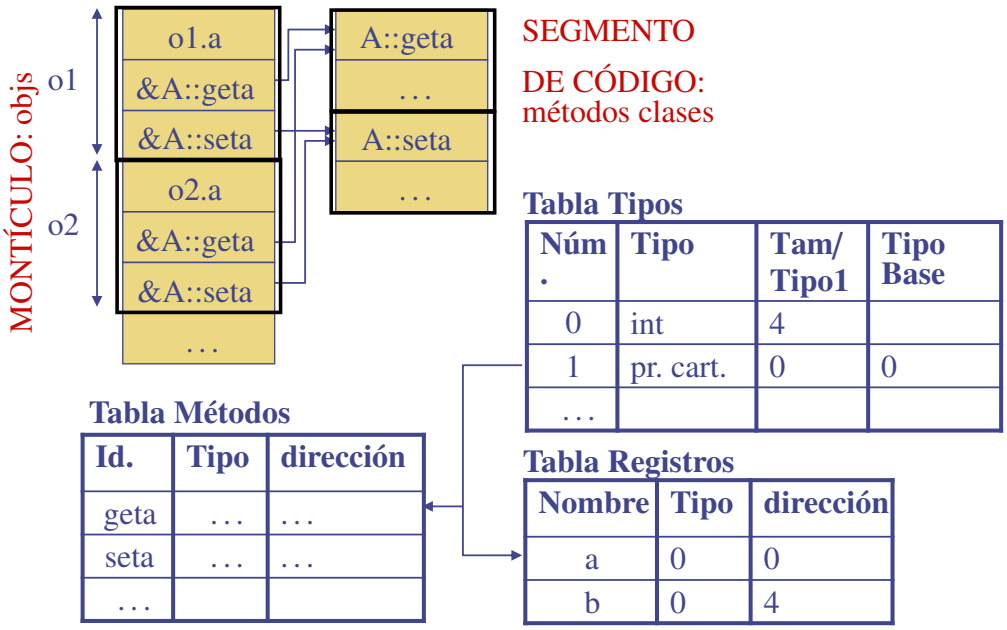


Fig. 8.1: Tablas para registros

8.5. Implementación de Clases

Las clases son tipos complejos que agrupan datos con operaciones (métodos) que lo usan y modifican.

a	b	setA	getA

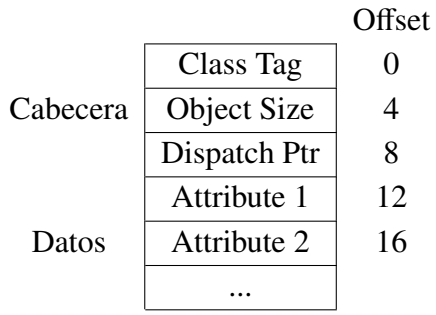
Esta clase de registro (tipo class) guarda los valores y el puntero a los procedimientos en la sección de código.

- a, b son datos enteros
- setA, getA son punteros al código de la clase

8.5.1. Esquema general de objetos

Similar a un registro, con campos específicos en la cabecera:

Los métodos se declaran antes y después los valores de atributos



8.5.2. Acceso a miembros de clase

Para referirse a un atributo f del objeto como this.f o simplemente f.

Se genera el mismo código, que ponga el this. o no se usa una referencia implícita al objeto para cualquier método.

8.5.3. Herencia

El desplazamiento (offset) de un atributo es el mismo en una clase y en las heredadas. Cualquier método de una clase A1 se pueden usar una subclase A2.

Polimorfismos: Que el mismo código se pueda comportar de diferentes maneras. Si llama una hija y la madre no hacen lo mismo. Para esto se usa una tabla de despacho dinámico. Los métodos de las clases que heredan usan los de la original, se amplía manteniendo lo que siga igual.

8.5.4. Tablas de Despacho

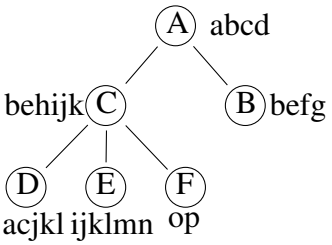
Cada clase tiene un conjunto fijo de métodos (incluidos los heredados)

Esta tabla indexa los métodos

- Array de puntos de entrada a métodos
- Cada método f a una distancia fija en la tabla de despacho, para una clase y sus heredadas.

	a	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**j**	**k**	**l**	**m**	**n**	**o**	**p**
A	01	02	03	04	0	0	0	0	0	0	0	0	0	0	0	0
B	01	09	03	04	06	07	08	0	0	0	0	0	0	0	0	0
C	01	09	03	04	10	0	0	11	12	13	14	0	0	0	0	0
D	15	09	26	04	10	0	0	11	12	17	18	19	0	0	0	0
E	01	09	03	04	10	0	0	11	20	21	22	23	24	25	0	0
F	01	09	03	04	10	0	0	11	12	13	14	0	0	0	26	27

Tabla 8.3: Tabla de despacho



9. TEMA 7: GENERACIÓN DE CÓDIGO INTERMEDIO

Acceso en Drive a las diapositivas