



UNIVERSIDAD
DE GRANADA

Procesadores de Lenguajes

Temario y Fundamentos de Compilación — Temas 1 al 6

Ismael Sallami Moreno

Recursos Ingeniería Informática y Ade

Licencia

Este trabajo está bajo una Licencia Creative Commons BY-NC-ND 4.0.

Permisos: Se permite compartir, copiar y redistribuir el material en cualquier medio o formato.

Condiciones: Es necesario dar crédito adecuado, proporcionar un enlace a la licencia e indicar si se han realizado cambios. No se permite usar el material con fines comerciales ni distribuir material modificado.



Procesadores de Lenguajes

Ismael Sallami Moreno

<https://elblogdeismael.github.io>

Índice general

1	Procesadores de Lenguajes y Arquitectura de la Traducción	7
1.1	1. Fundamentos y Definición Formal de la Traducción	7
1.2	2. Aplicaciones de los Procesadores de Lenguajes	8
1.3	3. Estructura General de un Traductor: Fases de Análisis y Síntesis . .	8
1.4	4. Compiladores frente a Intérpretes	9
1.5	5. Diagramas en T y Técnicas de Bootstrapping	10
2	Análisis Léxico y Generación Automática con LEX	11
2.1	1. Descripción Funcional y Conceptos Fundamentales	11
2.2	2. Fundamentos Formales: Álgebra de Lenguajes y Expresiones Regulares	11
2.3	3. Del Patrón al Autómata Finito Determinista	12
2.4	4. Tratamiento y Recuperación de Errores Léxicos	13
2.5	5. El Generador LEX: Arquitectura y Reglas Operativas	13
3	Fundamentos del Análisis Sintáctico	15
3.1	1. Misión y Descripción Funcional del Analizador Sintáctico	15
3.2	2. Gramáticas Libres del Contexto (GLC / Tipo 2)	15
3.3	3. Árboles Sintácticos y Ambigüedad Gramatical	16
3.4	4. Formas Normales de Gramáticas Libres del Contexto	16
3.5	5. Autómatas de Pila (AP)	17
3.6	6. Estrategias de Análisis Sintáctico	17
4	Métodos de Análisis Sintáctico Descendentes	19
4.1	1. El Método Intuitivo y el Problema de la Recursividad Izquierda . . .	19
4.2	2. Eliminación Sistemática de la Recursividad a la Izquierda	19
4.3	3. Factorización a la Izquierda para Analizadores Predictivos	20
4.4	4. Analizador Predictivo no Recursivo con Autómata de Pila	20

4.5	5. Cálculo Formal de los Conjuntos INICIALES y SEGUIDORES	21
4.6	6. Algoritmo de Construcción de la Tabla Predictiva M	22
4.7	7. Gramáticas $LL(1)$ y Condiciones de no Ambigüedad	22
4.8	8. Tratamiento y Recuperación de Errores Sintácticos	23
5	Métodos de Análisis Sintáctico Ascendentes y Precedencias	25
5.1	1. El Motor de Desplazamiento y Reducción (<i>Shift-Reduce</i>)	25
5.2	2. Los Conflictos Ascendentes Clásicos	25
5.3	3. Análisis de Precedencia de Operador	26
5.4	4. Análisis de Precedencia Simple (Wirth y Weber, 1966)	27
5.5	5. Detección y Tratamiento de Errores	28
6	Análisis Sintáctico Ascendente LR y Generación con YACC	29
6.1	1. La Familia de Analizadores LR(k): Fundamentos	29
6.2	2. Arquitectura y Dinámica de un Analizador LR	29
6.3	3. Construcción del Analizador SLR(1) (Simple LR)	30
6.4	4. Comparativa Teórica: SLR(1), LR(1) Canónico y LALR(1)	31
6.5	5. El Generador YACC (<i>Yet Another Compiler-Compiler</i>)	32

Procesadores de Lenguajes y Arquitectura de la Traducción

1.1 1. Fundamentos y Definición Formal de la Traducción

Un procesador de lenguajes es cualquier sistema software encargado de recibir un texto escrito en un lenguaje origen y transformarlo, estructurarlo o ejecutarlo en un entorno computacional determinado. Cuando el lenguaje de especificación de un programa se sitúa a un nivel de abstracción superior al código máquina nativo del procesador, la traducción se convierte en un requisito físico para hacer viable la computación. Todo lenguaje de programación de alto nivel define, en última instancia, una máquina virtual con su propio modelo de memoria y control de ejecución.

Desde una perspectiva teórica rigurosa, la traducción no se limita a un mero reemplazo de cadenas de caracteres, sino que articula un puente entre dos modelos semánticos formales. Siguiendo el marco matemático de la teoría de compiladores, un modelo semántico se define formalmente como la terna:

$$M = (L, D, I)$$

- **Lenguaje (L):** Conjunto estructurado de construcciones simbólicas formadas por secuencias válidas de símbolos pertenecientes a un alfabeto Σ . En informática se especifica formalmente mediante una gramática formal G , verificándose que $L = L(G)$.
- **Dominio semántico (D):** Conjunto de entidades, estados de memoria, valores elementales y estructurados sobre los que operan las instrucciones del programa.
- **Interpretación (I):** Función matemática $I : L(G) \rightarrow D$ que asigna a cada construcción sintáctica válida de L un significado operativo o valor exacto en el dominio semántico D .

1.1.1 Requisitos y Esquema de Traducción

Dados dos modelos semánticos $M_1 = (L_1, D_1, I_1)$ para el lenguaje fuente y $M_2 = (L_2, D_2, I_2)$ para el lenguaje objeto, la construcción de un traductor viable exige dos condiciones de partida:

1. **Inclusión de dominios semánticos:** El dominio destino debe ser capaz de representar todas las operaciones y valores del origen, lo que se formaliza algebraicamente como $D_1 \subseteq D_2$.
2. **Esquema de traducción (T):** Es una aplicación computable $T : L_1 \rightarrow L_2$ que debe preservar estrictamente el significado original. Para todo programa fuente $p \in L_1$, debe

satisfacerse la propiedad de equivalencia semántica:

$$I_2(T(p)) = I_1(p)$$

1.2 2. Aplicaciones de los Procesadores de Lenguajes

La tecnología de traducción trasciende los compiladores comerciales clásicos de sistemas como C, C++ o Rust. Los procesadores de lenguajes se aplican en diversos campos de la ingeniería informática:

- **Lenguajes orientados al dominio (*Domain-Specific Languages, DSL*):** Lenguajes concebidos para resolver problemas de un área de negocio concreta (por ejemplo, gestión de inventarios, automatización industrial o modelado financiero). Permiten formular instrucciones cercanas a la jerga del usuario que se traducen automáticamente a código de propósito general de tercera o cuarta generación.
- **Macroensambladores y preprocesadores:** Transforman secuencias de macros parametrizadas en instrucciones ensamblador canónicas antes del ensamblado final.
- **Motores de bases de datos y analizadores de consultas:** Convierten sentencias declarativas de alto nivel (como SQL) en árboles algebraicos relacionales optimizados que invocan rutinas directas del motor de almacenamiento.

1.3 3. Estructura General de un Traductor: Fases de Análisis y Síntesis

Para mantener la modularidad y simplificar el mantenimiento, la arquitectura clásica de un compilador se divide en dos grandes etapas: el extremo inicial (*Front-End* o fase de análisis) y el extremo final (*Back-End* o fase de síntesis).

1.3.1 Fase de Análisis (*Front-End*)

Depende exclusivamente de las reglas del lenguaje fuente y se descompone en tres verificaciones sucesivas:

1. **Análisis Léxico:** Escanea el flujo continuo de caracteres del fichero fuente y los agrupa en componentes léxicos (*tokens*). Su función consiste en abstraer los caracteres individuales, eliminar espacios en blanco o comentarios y detectar secuencias no conformes con el alfabeto del lenguaje.
2. **Análisis Sintáctico:** Verifica si la secuencia de tokens suministrada por el analizador léxico respeta las reglas jerárquicas de una gramática libre de contexto (tipo 2 de Chomsky). Genera la estructura en árbol que describe la composición de sentencias y expresiones.
3. **Análisis Semántico:** Inspecciona el árbol sintáctico para comprobar restricciones contextuales que escapan a la gramática formal: concordancia de tipos en asignaciones y operaciones aritméticas, comprobación de identificadores declarados previamente y gestión de visibilidad

de variables (*scopes*). Produce como resultado una representación en código intermedio.

1.3.2 Fase de Síntesis (*Back-End*)

Toma el código intermedio generado en el análisis y lo adapta a la arquitectura física objetivo:

1. **Optimización de Código:** Transforma el código intermedio y las secuencias de instrucciones máquina para mejorar la velocidad de cómputo, reducir accesos a memoria principal o minimizar el tamaño del binario resultante, garantizando siempre la equivalencia semántica.
2. **Generación de Código Objeto:** Traduce la representación intermedia optimizada a instrucciones ensamblador o binario nativo, gestionando la asignación eficiente de registros del procesador y el direccionamiento en memoria.

1.3.3 Estructuras Transversales

Existen dos componentes que interactúan de forma permanente con todas las fases del procesador:

- **Tabla de Símbolos:** Estructura de datos central que almacena el nombre, tipo, ámbito, dirección de memoria y atributos asociados a cada identificador detectado en el programa. Se crea en el análisis léxico, se puebla en el sintáctico y semántico, y se consulta en la generación de código.
- **Manejador de Errores:** Registra, localiza y reporta cualquier anomalía sintáctica o semántica, coordinando mecanismos de sincronización para que el compilador no aborte en el primer fallo y continúe procesando el resto del archivo.

1.4 4. Compiladores frente a Intérpretes

La ejecución de un lenguaje de alto nivel puede abordarse mediante dos estrategias de procesamiento divergentes:

- **Compilador:** Traduce la totalidad del código fuente antes de comenzar la ejecución, produciendo un archivo binario autónomo para la máquina destino.
 - *Rendimiento:* La ejecución posterior es directa a velocidad de hardware nativo.
 - *Protección:* Distribuye binarios cerrados sin exponer el código fuente original.
 - *Inconvenientes:* Proceso de compilación costoso en tiempo y fuerte dependencia de la arquitectura física.
- **Intérprete:** Lee, traduce y ejecuta las instrucciones del código fuente paso a paso en tiempo real.
 - *Flexibilidad:* Máxima portabilidad y ciclo de depuración inmediato, idóneo para desarrollo interactivo.
 - *Inconvenientes:* Rendimiento notablemente inferior, ya que las sentencias contenidas en bucles deben ser analizadas y traducidas en cada iteración.
- **Modelos Mixtos y Máquinas Virtuales:** Plataformas como Java o .NET combinan ambos mundos. Un compilador traduce el código fuente a un lenguaje intermedio portable (*bytecode*), que posteriormente es ejecutado por una máquina virtual encargada de interpretarlo o de compilarlo en caliente a código nativo mediante técnicas *Just-In-Time* (JIT).

1.5 5. Diagramas en T y Técnicas de Bootstrapping

Para formalizar las dependencias de implementación y soporte de los compiladores, D. A. Watt propuso en 1993 la representación mediante diagramas en T (*Tombstone Diagrams*).

1.5.1 Componentes de un Diagrama en T

Un compilador involucra tres lenguajes distintos: - Lenguaje fuente (S , *Source*), situado en el brazo izquierdo de la T. - Lenguaje objeto (T , *Target*), situado en el brazo derecho de la T. - Lenguaje de implementación (L , *Language*), situado en la base de la T.

Para que un compilador pueda ejecutarse, el lenguaje de su base L debe coincidir exactamente con el código máquina que ejecuta la máquina anfitriona M .

1.5.2 Compilación Cruzada y Half-Bootstrap

La compilación cruzada (*Cross-Compilation*) se define como el proceso en el que un compilador se ejecuta sobre una máquina M_1 para generar código ejecutable destinado a una máquina distinta M_2 . Resulta indispensable cuando se programa para hardware embebido o arquitecturas con recursos limitados.

Cuando se dispone de un compilador de un lenguaje L para una máquina M_1 y se desea portarlo a M_2 , se aplica la técnica de *Half-Bootstrap*:

1. Se reescribe únicamente el generador de código del compilador para que emita instrucciones de la máquina destino M_2 .
2. Se compila este nuevo compilador en la máquina original M_1 . El resultado es un compilador cruzado que corre en M_1 y produce ejecutables para M_2 .
3. Utilizando este compilador cruzado, se compila el código fuente del compilador a sí mismo. El binario resultante corre de forma autónoma en la nueva máquina M_2 .

Análisis Léxico y Generación Automática con LEX

2.1 1. Descripción Funcional y Conceptos Fundamentales

El análisis léxico constituye la primera línea de procesamiento en la fase de análisis de un compilador. Su tarea operativa consiste en leer secuencialmente el texto fuente carácter a carácter y agruparlos en unidades con entidad propia denominadas componentes léxicos (*tokens*).

Esta separación entre el nivel léxico y el sintáctico aporta dos ventajas de ingeniería: simplifica enormemente la estructura del analizador sintáctico (que no debe gestionar detalles triviales como espacios en blanco o retornos de carro) y confiere mayor portabilidad y eficiencia al compilador, permitiendo optimizar el escaneo continuo a nivel de búfer de entrada.

2.1.1 Diferenciación Rigurosa: Token, Lexema y Patrón

La teoría formal establece una distinción nítida entre tres conceptos complementarios:

- **Token (Componente léxico):** Categoría abstracta que engloba a todas las secuencias de caracteres que desempeñan idéntica función sintáctica dentro del lenguaje. Al analizador sintáctico solo le concierne el identificador del token (por ejemplo, NUMERO, ID, ASSIGN), no su valor textual concreto.
- **Lexema:** Secuencia exacta de caracteres leída en el texto fuente que encaja con la definición de un determinado token. Si en el código aparece `limite = 100;`, las cadenas "limite", "=", "100" y ";" son los lexemas reales.
- **Patrón:** Regla formal o descripción que delimita qué secuencias de caracteres pertenecen a un token. Los patrones se especifican matemáticamente mediante expresiones regulares y se reconocen mediante un Autómata Finito Determinista (AFD).

2.2 2. Fundamentos Formales: Álgebra de Lenguajes y Expresiones Regulares

Dado un alfabeto finito Σ de símbolos, un lenguaje formal L es cualquier subconjunto de cadenas formadas sobre Σ^* ($L \subseteq \Sigma^*$).

2.2.1 Operaciones Fundamentales sobre Lenguajes

Sobre dos lenguajes L y M se definen las siguientes operaciones cerradas:

- **Unión:** $L \cup M = \{s \mid s \in L \vee s \in M\}$.
- **Concatenación:** $LM = \{st \mid s \in L \wedge t \in M\}$.
- **Clausura de Kleene:** $L^* = \bigcup_{i=0}^{\infty} L^i$, donde $L^0 = \{\lambda\}$ (cadena vacía).
- **Clausura positiva:** $L^+ = \bigcup_{i=1}^{\infty} L^i = LL^*$.

2.2.2 Expresiones Regulares

Las expresiones regulares denotan lenguajes regulares contruidos inductivamente a partir de los casos base: 1. λ es una expresión regular cuyo lenguaje es $\{\lambda\}$. 2. Para todo símbolo $a \in \Sigma$, a es una expresión regular que denota el lenguaje $\{a\}$. 3. Si r y s son expresiones regulares que denotan $L(r)$ y $L(s)$, entonces $(r) \mid (s)$, $(r)(s)$ y $(r)^*$ son expresiones regulares que denotan respectivamente $L(r) \cup L(s)$, $L(r)L(s)$ y $(L(r))^*$.

2.3 3. Del Patrón al Autómata Finito Determinista

Para que un procesador pueda reconocer tokens en tiempo lineal sobre la entrada, las expresiones regulares deben transformarse en autómatas de estados finitos.

2.3.1 Construcción de Thompson (ER a AFN)

El teorema clásico de autómata de estados garantiza que a toda expresión regular le corresponde un Autómata Finito no Determinista con transiciones vacías (AFN- λ). El algoritmo de Thompson construye el autómata de forma compositiva: - Los casos base (λ y el símbolo individual a) se modelan con autómatas elementales de dos estados. - La unión de patrones $r \mid s$ se resuelve creando un nuevo estado inicial y otro final enlazados con transiciones λ hacia los autómatas de r y s . - La concatenación rs enlaza el estado de aceptación de r con el inicial de s . - La clausura r^* introduce bucles de retroalimentación y saltos vacíos mediante aristas λ .

2.3.2 Algoritmo de Subconjuntos: Determinización de AFN a AFD

Dado que un AFN puede residir en múltiples estados a la vez, no resulta viable ejecutarlo de forma directa sin incurrir en costes combinatorios. El método de construcción de subconjuntos construye un AFD equivalente donde cada estado del AFD representa un conjunto de estados del AFN original. Se apoya en dos operaciones:

- λ -clausura(T): Conjunto de todos los estados del AFN alcanzables desde cualquier estado $s \in T$ siguiendo únicamente transiciones λ .
- Move(T, a): Conjunto de estados del AFN hacia los cuales existe una transición con el símbolo de entrada a desde algún estado $s \in T$.

El estado inicial del AFD corresponde a λ -clausura($\{s_0\}$). A partir de ahí, para cada nuevo conjunto

de estados y cada símbolo del alfabeto se computa λ -clausura(Move(T, a)) hasta que no aparezcan nuevos conjuntos.

2.3.3 Minimización de Estados del AFD

El AFD resultante se optimiza agrupando estados indistinguibles para reducir el consumo en memoria de la tabla de transiciones. Se divide inicialmente el conjunto de estados en dos grupos disjuntos (estados de aceptación y estados de no aceptación), y se refinan sucesivamente las particiones hasta que los estados pertenecientes a un mismo bloque exhiban transiciones idénticas ante cualquier símbolo de entrada.

2.4 4. Tratamiento y Recuperación de Errores Léxicos

Un analizador léxico detecta un error cuando la secuencia de caracteres leída no encaja con ningún patrón definido en el alfabeto del lenguaje.

2.4.1 Estrategias de Sincronización y Recuperación

- **Modo Pánico (*Panic Mode*):** Al detectar un carácter no reconocible, el analizador entra en un proceso de sincronización donde descarta caracteres sucesivos de la entrada hasta topar con un delimitador conocido (como espacios, comas o puntos y coma), reiniciando el autómata en su estado de reposo.
- **Reparación por Distancia de Edición (*Levenshtein*):** Cuando se busca mayor sofisticación, el escáner evalúa transformaciones elementales sobre el lexema con menor coste:
 - Eliminación de un carácter anómalo.
 - Inserción de un carácter omitido.
 - Sustitución de un carácter incorrecto.
 - Transposición de dos caracteres contiguos invertidos involuntariamente por el usuario (ej. `retunr` por `return`).

2.5 5. El Generador LEX: Arquitectura y Reglas Operativas

LEX es la herramienta canónica de generación de analizadores léxicos. Recibe un fichero de especificación `.l` y sintetiza una rutina en lenguaje C denominada `yylex()`.

2.5.1 Estructura de una Especificación LEX

Un archivo LEX se organiza en tres secciones separadas por delimitadores `%%`:

1. **Definiciones:** Delimitadas por `%{` y `%}` para incluir directivas de preprocesador C, cabeceras y macros con nombres de expresiones regulares (por ejemplo, `LETRA [a-zA-Z]`).
2. **Reglas:** Conjunto de pares formados por un patrón regular y su correspondiente bloque de

código C entre llaves que se ejecuta al reconocerse el patrón.

3. **Código de Usuario:** Procedimientos auxiliares en C y la función de invocación `main()`.

2.5.2 Resolución de Ambigüedades en LEX

Cuando dos o más expresiones regulares coinciden con el texto que entra por el flujo de lectura, LEX aplica dos reglas deterministas de desempate:

1. **Principio del Encaje más Largo (*Longest Match o Maximal Munch*):** LEX siempre favorece la regla que consuma la mayor cantidad de caracteres de entrada. Si la entrada es cadena, no lo segmenta como la palabra `cad` seguida de `ena`.
2. **Precedencia por Orden Textual:** A igualdad de caracteres consumidos, gana la primera regla declarada en el fichero. Por esta razón, las palabras reservadas del lenguaje (`if`, `while`) deben definirse siempre antes de la regla general de los identificadores.

2.5.3 La Acción REJECT

Cuando una regla ejecuta la macro especial REJECT, LEX ejecuta las instrucciones de esa acción pero de inmediato restituye el puntero de entrada y procede a evaluar la siguiente alternativa coincidente. Permite encadenar ejecuciones en cascada ante solapamientos de patrones.

2.5.4 Variables y Funciones Clave del Sistema

- `yytext`: Puntero al array de caracteres que contiene el lexema real recién identificado.
- `yytext`: Entero que almacena la longitud en caracteres del lexema en `yytext`.
- `yylex()`: Función principal generada por LEX que devuelve el identificador entero del token reconocido.
- `yywrap()`: Función invocada automáticamente al detectar el fin de fichero (EOF). Retorna 1 para dar por finalizado el análisis o 0 si enlaza con otro archivo.
- Acciones por defecto: Todo carácter reconocido que carezca de acción asociada no genera salida; todo carácter que no coincida con ningún patrón es enviado a la salida estándar (`stdout`, equivalente a la macro `ECHO`).

2.5.5 Condiciones de Inicio (*Start Conditions*)

Permiten conferir al analizador léxico sensibilidad al contexto izquierdo, activando y desactivando subconjuntos de reglas según el estado interno:

- **Condiciones Inclusivas (`%START o %s`):** Al activarse con `BEGIN(NOMBRE)`, entran en juego las reglas etiquetadas con `<NOMBRE>` y simultáneamente todas las reglas generales no etiquetadas.
- **Condiciones Exclusivas (`%X o %x`):** Al activarse con `BEGIN(NOMBRE)`, quedan activas de forma exclusiva las reglas marcadas con `<NOMBRE>`. Las reglas normales sin prefijo quedan suspendidas hasta que se ejecute `BEGIN(0)`, siendo el mecanismo canónico para procesar cadenas entrecomilladas o bloques de comentarios multilínea.

Fundamentos del Análisis Sintáctico

3.1 1. Misión y Descripción Funcional del Analizador Sintáctico

El analizador sintáctico opera como el núcleo articulador del compilador. Recibe como flujo de entrada la secuencia ordenada de tokens producida por el analizador léxico y debe alcanzar dos metas determinantes:

1. **Validación Sintáctica:** Verificar si la secuencia de tokens puede derivarse formalmente a partir de la gramática del lenguaje.
2. **Reconstrucción Estructural:** Identificar el orden exacto de aplicación de las producciones gramaticales para sintetizar el árbol sintáctico del programa, derivando errores con mensajes informativos si la cadena resulta inválida.

3.2 2. Gramáticas Libres del Contexto (GLC / Tipo 2)

Los lenguajes de programación no pueden describirse exclusivamente mediante expresiones regulares debido a que estas carecen de la capacidad de conteo o memoria necesaria para verificar estructuras arbitrariamente anidadas (como paréntesis balanceados, expresiones compuestas o bloques de control anidados). Se hace indispensable recurrir a las Gramáticas Libres del Contexto (tipo 2 en la jerarquía de Chomsky).

Formalmente, una gramática libre del contexto se define mediante la cuádrupla:

$$G = (V_N, V_T, P, S)$$

- V_N : Conjunto finito de símbolos no terminales (variables sintácticas intermedias como *Expresión*, *Sentencia*).
- V_T : Conjunto finito de símbolos terminales, equivalentes a los tokens generados por el analizador léxico.
- P : Conjunto finito de reglas de producción. Su rasgo definitorio es que la parte izquierda contiene un único símbolo no terminal:

$$A \rightarrow \alpha \quad \text{donde } A \in V_N \text{ y } \alpha \in (V_N \cup V_T)^*$$

La denominación “libre del contexto” radica en que la sustitución de A por la forma sentencial α puede aplicarse con total independencia de los símbolos adyacentes a A en la cadena. - S : Símbolo inicial o axioma de la gramática ($S \in V_N$).

3.3 3. Árboles Sintácticos y Ambigüedad Gramatical

El árbol sintáctico es la representación gráfica de una derivación: - La raíz del árbol es siempre el axioma inicial S . - Los nodos interiores son no terminales (V_N), reflejando las producciones aplicadas. - Las hojas son terminales (V_T) y leídas de izquierda a derecha componen la secuencia de entrada analizada.

3.3.1 El Problema de la Ambigüedad

Una gramática se dice que es ambigua cuando admite **más de un árbol sintáctico distinto** para una misma secuencia de entrada (lo que equivale a tener más de una derivación más a la izquierda o más a la derecha).

En la construcción de procesadores de lenguajes, la ambigüedad constituye un problema crítico. Si consideramos la clásica gramática de expresiones aritméticas:

$$E \rightarrow E + E \mid E * E \mid id$$

La secuencia de tokens $id + id * id$ admite dos árboles de derivación diferentes: uno agrupa la suma en primer lugar y otro prioriza la multiplicación. Dado que el árbol sintáctico gobierna la generación del código y el orden de evaluación matemática, tolerar ambigüedad llevaría a cálculos indeterministas. La solución estándar exige reescribir la gramática introduciendo niveles no terminales intermedios que capturen la precedencia y la asociatividad de las operaciones.

3.4 4. Formas Normales de Gramáticas Libres del Contexto

La teoría de lenguajes estandariza el formato de las producciones para habilitar algoritmos de análisis sintáctico formal:

3.4.1 Forma Normal de Chomsky (FNC)

Toda gramática libre de contexto cuyas producciones responden exclusivamente a los formatos:

$$A \rightarrow BC \quad \text{o bien} \quad A \rightarrow a$$

donde $A, B, C \in V_N$ y $a \in V_T$. Es la base sobre la que operan algoritmos generales de reconocimiento

como CYK (*Cocke-Younger-Kasami*), con coste de análisis $O(n^3)$.

3.4.2 Forma Normal de Greibach (FNG)

Toda gramática libre de contexto cuyas reglas de derivación presentan la estructura:

$$A \rightarrow a\alpha$$

donde $A \in V_N$, $a \in V_T$ y $\alpha \in V_N^*$. La ventaja de Greibach es que cada paso de derivación consume exactamente un símbolo terminal de entrada, facilitando el diseño de reconocedores descendentes deterministas.

3.5 5. Autómatas de Pila (AP)

Así como los autómatas finitos modelan los lenguajes regulares, el autómata de pila es el modelo computacional canónico que reconoce los lenguajes libres de contexto. Su capacidad para apilar y desapilar símbolos en una memoria LIFO le permite verificar construcciones anidadas y correspondencias emparejadas.

Formalmente se define como una séptupla:

$$AP = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

- Q : Conjunto finito de estados del autómata.
- Σ : Alfabeto de entrada (finito).
- Γ : Alfabeto de la pila (permite almacenar terminales y no terminales).
- δ : Función de transición de tipo:

$$\delta : Q \times (\Sigma \cup \{\lambda\}) \times \Gamma \rightarrow \mathcal{P}_{\text{fin}}(Q \times \Gamma^*)$$

- $q_0 \in Q$: Estado inicial.
- $Z_0 \in \Gamma$: Símbolo inicial que reside en el fondo de la pila antes de arrancar.
- $F \subseteq Q$: Conjunto de estados finales de aceptación.

Una **Descripción Instantánea (DI)** representa la fotografía del sistema en un instante determinado mediante la terna (q, ω, γ) , con $q \in Q$ como estado actual, $\omega \in \Sigma^*$ como la cadena de entrada restante y $\gamma \in \Gamma^*$ como la secuencia de elementos contenidos en la pila.

Los teoremas de equivalencia demuestran que un lenguaje L es independiente del contexto si y solo si es aceptado por algún autómata de pila (bien por alcanzar un estado final de F , bien por vaciado total de la pila).

3.6 6. Estrategias de Análisis Sintáctico

La verificación estructural de una cadena de entrada ω puede abordarse desde dos ópticas metodológicas opuestas:

- **Análisis Descendente (*Top-Down*):**

- Parte del símbolo inicial S (la raíz del árbol) y aplica reglas de expansión hacia abajo hasta coincidir en las hojas con la secuencia de entrada ω ($S \Rightarrow^* \omega$).
- Construye derivaciones más a la izquierda (*leftmost derivations*).
- Es la base de los analizadores predictivos recursivos y de las gramáticas $LL(1)$.
- **Análisis Ascendente (*Bottom-Up*):**
 - Parte de los tokens leídos en las hojas ω y localiza subcadenas que encajen con la parte derecha de una regla para reemplazarlas por el no terminal correspondiente (*reducción*), ascendiendo hasta converger en el símbolo inicial S ($\omega \Leftarrow^* S$).
 - Construye en sentido inverso la derivación canónica más a la derecha (*rightmost derivation in reverse*).
 - Es la base de los métodos de precedencia y de los analizadores LR (SLR, LALR y LR canónico).

Métodos de Análisis Sintáctico Descendentes

4.1 1. El Método Intuitivo y el Problema de la Recursividad Izquierda

El análisis sintáctico descendente busca reconstruir el árbol sintáctico desde la raíz hacia las hojas, derivando el axioma S hasta alcanzar la secuencia de tokens leída en la entrada.

En su formulación más simple (método intuitivo o por fuerza bruta), el analizador prueba sucesivamente las alternativas de cada producción mediante vuelta atrás (*backtracking*): - Expande la primera producción del símbolo inicial. - Toma el no terminal más a la izquierda y aplica la primera producción disponible. - Si en algún punto los terminales generados no coinciden con la entrada, retrocede la lectura y prueba con la siguiente alternativa gramatical.

4.1.1 El Problema Crítico de la Recursividad a la Izquierda

Una gramática es recursiva a la izquierda si existe algún símbolo no terminal X tal que $X \Rightarrow^* X\alpha$. Si se intenta aplicar un método descendente sobre una regla con recursividad inmediata:

$$A \rightarrow A\alpha \mid \beta$$

El algoritmo intentará expandir A sustituyéndolo nuevamente por $A\alpha$, y este a su vez por $A\alpha\alpha$, entrando en un bucle infinito en tiempo de ejecución sin llegar a consumir jamás un solo token de la entrada.

4.2 2. Eliminación Sistemática de la Recursividad a la Izquierda

Para hacer viable el análisis descendente, es obligatorio transformar la gramática eliminando cualquier ciclo recursivo por la izquierda.

4.2.1 Transformación para Recursividad Inmediata

Dada la regla $A \rightarrow A\alpha \mid \beta$ (donde β no comienza por A), la recursividad se traslada a la derecha mediante la introducción de un nuevo no terminal A' :

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \epsilon$$

Justificación matemática: Ambas estructuras generan idéntico lenguaje regular $\beta\alpha^*$. No obstante, en la versión transformada el analizador descendente expande primero β , lo que permite consumir tokens de la entrada antes de evaluar las posibles repeticiones de α .

4.2.2 Generalización a Múltiples Alternativas

Si un no terminal presenta varias ramas recursivas y varias ramas base:

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

Se transforma en:

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A'$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \epsilon$$

4.3 3. Factorización a la Izquierda para Analizadores Predictivos

Un analizador predictivo persigue seleccionar la producción correcta en cada paso inspeccionando únicamente el siguiente símbolo de entrada (un único token de anticipación, *lookahead*), erradicando por completo el coste del *backtracking*.

Sin embargo, si dos o más alternativas de un no terminal arrancan con el mismo prefijo:

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$$

El analizador no puede determinar qué producción escoger al leer el inicio de α . La **factorización a la izquierda** resuelve este problema aplazando la decisión hasta haber consumido el prefijo común:

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta_1 \mid \beta_2$$

4.4 4. Analizador Predictivo no Recursivo con Autómata de Pila

Para evitar la sobrecarga de implementar funciones recursivas en el código del compilador, se utiliza una máquina de pila gobernada por una tabla de análisis sintáctico bidimensional $M[A, a]$.

4.4.1 Componentes y Dinámica de Ejecución

- **Cinta de entrada:** Cadena de tokens finalizada por el delimitador de cierre $\$$ ($a_1 a_2 \dots a_n \$$).
- **Pila del autómata:** Almacena terminales y no terminales, con el símbolo inicial S en la cima sobre el marcador de fondo $\$$: $\text{cima} \rightarrow [S, \$]$.
- **Tabla de análisis** $M[A, a]$: Matriz donde las filas corresponden a los no terminales V_N y las columnas a los terminales $V_T \cup \{\$\}$.

La rutina cíclica evalúa en cada paso el elemento en la cima de la pila X y el token actual de entrada a :

1. Si $X = a = \$$, el análisis concluye con éxito (cadena aceptada).
2. Si $X = a \neq \$$, se desapila el terminal X y se avanza la lectura al siguiente token (*match*).
3. Si X es un no terminal, se consulta la celda $M[X, a]$:
 - Si contiene la regla $X \rightarrow Y_1 Y_2 \dots Y_k$, se extrae X de la pila y se apilan los símbolos en orden inverso ($Y_k \dots Y_2 Y_1$), de modo que Y_1 quede en la cima.
 - Si la producción es $X \rightarrow \epsilon$, simplemente se extrae X de la pila sin apilar nada nuevo.
 - Si la celda está vacía, se ha detectado un error sintáctico.

4.5 5. Cálculo Formal de los Conjuntos INICIALES y SEGUIDORES

Para construir la tabla predictiva M sin ambigüedades, se calculan dos conjuntos matemáticos para cada símbolo de la gramática.

4.5.1 Conjunto INICIALES (*First*)

Representa el conjunto de símbolos terminales que pueden aparecer al principio de cualquier cadena derivada a partir de una forma sentencial α :

$$\text{INICIALES}(\alpha) = \{a \in V_T \mid \alpha \Rightarrow^* a\beta\}$$

Reglas formales de cálculo:

1. Si $X \in V_T$, entonces $\text{INICIALES}(X) = \{X\}$.
2. Si existe la producción $X \rightarrow \epsilon$, entonces $\epsilon \in \text{INICIALES}(X)$.
3. Para producciones $X \rightarrow Y_1 Y_2 \dots Y_k$:
 - Se añade $(\text{INICIALES}(Y_1) \setminus \{\epsilon\})$ a $\text{INICIALES}(X)$.
 - Si $Y_1 \Rightarrow^* \epsilon$, se añade también $(\text{INICIALES}(Y_2) \setminus \{\epsilon\})$, y así sucesivamente mientras los Y_i anteriores deriven a ϵ .
 - Si todos los símbolos $Y_1, \dots, Y_k$ derivan a ϵ , entonces $\epsilon \in \text{INICIALES}(X)$.

4.5.2 Conjunto SEGUIDORES (*Follow*)

Representa el conjunto de terminales que pueden aparecer inmediatamente a la derecha de un no terminal A en alguna derivación sentencial:

$$\text{SEGUIDORES}(A) = \{a \in V_T \mid S \Rightarrow^* \alpha A a \beta\}$$

Reglas formales de cálculo:

1. El marcador de fin de entrada pertenece al axioma: $\$ \in \text{SEGUIDORES}(S)$.
2. Si existe una producción de la forma $A \rightarrow \alpha B \beta$, todo símbolo perteneciente a $\text{INICIALES}(\beta)$ (excepto ϵ) se añade a $\text{SEGUIDORES}(B)$.
3. Si existe una producción $A \rightarrow \alpha B$, o bien $A \rightarrow \alpha B \beta$ donde $\beta \Rightarrow^* \epsilon$, entonces todo elemento de $\text{SEGUIDORES}(A)$ pasa a formar parte de $\text{SEGUIDORES}(B)$.

4.6 6. Algoritmo de Construcción de la Tabla Predictiva M

Con los conjuntos de INICIALES y SEGUIDORES computados, la tabla $M[A, a]$ se rellena aplicando el siguiente procedimiento:

Para cada producción $A \rightarrow \alpha$ en la gramática G :

Para cada terminal a perteneciente a $\text{INICIALES}(\alpha)$:

Añadir " $A \rightarrow \alpha$ " en $M[A, a]$

Si ϵ pertenece a $\text{INICIALES}(\alpha)$:

Para cada terminal b perteneciente a $\text{SEGUIDORES}(A)$ (incluyendo $\$$):

Añadir " $A \rightarrow \alpha$ " en $M[A, b]$

Todas las celdas indefinidas en M se marcan como **ERROR**.

4.7 7. Gramáticas $LL(1)$ y Condiciones de no Ambigüedad

Una gramática se define como $LL(1)$ (lectura de izquierda a derecha, derivación más a la izquierda, 1 token de anticipación) si su tabla de análisis predictivo carece de entradas múltiples. Una celda con más de una regla implica un conflicto de predicción.

Formalmente, una gramática es $LL(1)$ si y solo si para cada par de alternativas $A \rightarrow \alpha \mid \beta$:

1. $\text{INICIALES}(\alpha) \cap \text{INICIALES}(\beta) = \emptyset$.
2. Como máximo una de las dos formas sentenciales (α o β) puede derivar en ϵ .
3. Si $\beta \Rightarrow^* \epsilon$, entonces $\text{INICIALES}(\alpha) \cap \text{SEGUIDORES}(A) = \emptyset$.

Una **S-Gramática** (o $LL(1)$ simple) es una versión restringida donde todas las alternativas de un no terminal comienzan por terminales distintos y no existen producciones a la cadena vacía.

4.8 8. Tratamiento y Recuperación de Errores Sintácticos

Cuando el analizador accede a una celda vacía de la tabla M , se activa el protocolo de recuperación para seguir procesando el fichero:

- **Recuperación a Nivel de Frase:** Rellena los huecos vacíos con punteros a rutinas específicas que corrigen errores previsibles (por ejemplo, insertar un paréntesis de cierre omitido o suprimir un token repetido).
- **Recuperación en Modo Pánico:** El autómata descarta tokens de la entrada hasta topar con un símbolo de sincronización predeterminado:
 - La elección estándar consiste en tomar los símbolos de $\text{SEGUIDORES}(A)$ como conjunto de sincronización. Al alcanzar uno de ellos, el analizador extrae A de la pila y reanuda el análisis regular.
 - Para límites de sentencias o bloques de código, se suelen usar los propios terminales de inicio de sentencia (INICIALES) para evitar saltarse bloques completos de instrucciones.

Métodos de Análisis Sintáctico Ascendentes y Precedencias

5.1 1. El Motor de Desplazamiento y Reducción (*Shift-Reduce*)

El análisis sintáctico ascendente (*Bottom-Up*) aborda la construcción del árbol en orden inverso: arranca en las hojas con la secuencia de tokens leída de la entrada y busca reducir subcadenas sucesivas hacia los no terminales de la gramática hasta alcanzar el axioma inicial S .

Esta familia de analizadores se apoya en una máquina de pila que ejecuta cuatro operaciones atómicas:

- **Desplazar (*Shift*):** Lee el siguiente token de la cinta de entrada y lo apila en la cima de la pila.
- **Reducir (*Reduce*):** Localiza en la cima de la pila una subcadena τ que coincide exactamente con el lado derecho de una producción $A \rightarrow \tau$. Extrae los símbolos de τ y apila en su lugar el no terminal A .
- **Aceptar (*Accept*):** Si la cinta de entrada se ha consumido por completo y en la pila solo resta el axioma inicial S sobre el centinela de fondo $\$,$ el análisis concluye satisfactoriamente.
- **Error:** El autómata no puede ni desplazar el token actual ni reducir válidamente la cima de la pila.

5.2 2. Los Conflictos Ascendentes Clásicos

En gramáticas no restringidas, el analizador puede encontrarse en situaciones de indecisión determinista:

- **Conflicto Desplaza/Reduce (*Shift/Reduce*):** El autómata no puede decidir si debe desplazar el siguiente token de entrada a la pila o si debe reducir los elementos que ya residen en la cima. Es el conflicto típico en construcciones como sentencias *if-then-else*.
- **Conflicto Reduce/Reduce:** La subcadena alojada en la cima de la pila coincide con el lado derecho de dos o más producciones distintas ($A \rightarrow \tau$ y $B \rightarrow \tau$), desconociendo a qué no terminal debe reducir.

5.3 3. Análisis de Precedencia de Operador

Propuesto originalmente por Robert Floyd en 1963, este método predictivo ascendente fue diseñado para compilar expresiones algebraicas complejas prescindiendo de la construcción de gramáticas completas.

5.3.1 Restricciones de una Gramática de Operador

Para poder aplicar precedencia de operador, la gramática debe cumplir dos restricciones fundamentales:

1. No existen producciones que deriven a la cadena vacía ($A \rightarrow \epsilon$).
2. No aparecen dos símbolos no terminales contiguos en la parte derecha de ninguna producción (no se permiten formas $A \rightarrow \alpha BC\beta$).

5.3.2 Relaciones de Precedencia entre Operadores

Las relaciones se evalúan estrictamente entre terminales (operadores e identificadores), sin intervenir los no terminales:

- $a < b$: El terminal a tiene menor precedencia que b . Ordena la acción de **desplazar** b a la pila.
- $a \doteq b$: Ambos terminales tienen igual precedencia (emparejamientos de ámbito, como paréntesis $(\doteq)$).
- $a > b$: El terminal a tiene mayor precedencia que b . Indica que se ha alcanzado el límite derecho de un subcálculo y ordena la acción de **reducir**.

Propiedad matemática formal: Estas relaciones no constituyen relaciones de orden convencionales, ya que carecen de transitividad y no son simétricas (que $a < b$ se cumpla no excluye que en otro contexto se dé $b < a$).

5.3.3 Cálculo Formal mediante PRIMERO y ULTIMO (*FirstOp* y *LastOp*)

Para cada símbolo no terminal A , se definen los conjuntos de terminales extremos:

$$\text{PRIMERO}(A) = \{a \in V_T \mid A \Rightarrow^* a \cdots \vee A \Rightarrow^* Xa \cdots \quad (X \in V_N)\}$$

$$\text{ULTIMO}(A) = \{a \in V_T \mid A \Rightarrow^* \cdots a \vee A \Rightarrow^* \cdots aX \quad (X \in V_N)\}$$

Con estos conjuntos, la tabla de relaciones de precedencia se construye inspeccionando las subcadenas del lado derecho de las producciones: - Si aparece aB con $a \in V_T$ y $B \in V_N$, entonces $a < b$ para todo $b \in \text{PRIMERO}(B)$. - Si aparece Ba con $B \in V_N$ y $a \in V_T$, entonces $b > a$ para todo $b \in \text{ULTIMO}(B)$. - Para los delimitadores externos: $\$ < a$ para todo $a \in \text{PRIMERO}(S)$ y $b > \$$ para todo $b \in \text{ULTIMO}(S)$.

5.3.4 Limitaciones de la Precedencia de Operador

- **Ceguera de no terminales:** Durante la reducción solo se verifica el operador terminal, ignorando los no terminales intermediarios. En consecuencia, el analizador acepta como sintácticamente válidas secuencias erróneas como $\text{id} + + \text{id}$ o $\text{id} * \text{id}$.
- **Incapacidad ante operadores polimórficos:** Presenta serias deficiencias al manejar símbolos con distinta precedencia según el contexto sintáctico, como el operador menos ($-$), que tiene precedencia unaria alta y precedencia binaria media.

5.4 4. Análisis de Precedencia Simple (Wirth y Weber, 1966)

Para subsanar las limitaciones de la precedencia de operador, Niklaus Wirth y Helmut Weber extendieron las relaciones de precedencia a **todos los símbolos** del vocabulario gramatical, comparando indistintamente terminales y no terminales ($V_N \cup V_T$).

5.4.1 El Concepto de Pivote (*Handle* o Mango)

El pivote es la subcadena alojada en la cima de la pila que coincide con la parte derecha de una regla gramatical, cuya sustitución por la parte izquierda representa un paso formal en la inversión de una derivación canónica por la derecha.

En precedencia simple, el pivote queda delimitado inequívocamente por las relaciones de precedencia calculadas sobre los símbolos adyacentes:

$$\langle X_1 \doteq X_2 \doteq \dots \doteq X_k \rangle$$

5.4.2 Reglas para la Tabla de Precedencia Simple

Sobre cualquier par de símbolos $X, Y \in (V_N \cup V_T)$:

1. $X \doteq Y$: Si existe una producción $A \rightarrow \alpha XY\beta$ (ambos símbolos cuelgan de un mismo nodo en el árbol sintáctico).
2. $X < Y$: Si existe una producción $A \rightarrow \alpha XB\beta$ tal que $B \Rightarrow^+ Y\gamma$ (X cuelga de un nodo a la izquierda e inmediatamente superior al nodo del que desciende Y).
3. $X > Y$: Si existe una producción $A \rightarrow \alpha B\beta$ tal que $B \Rightarrow^+ \gamma X$ y además $\beta \Rightarrow^* Y\delta$ o bien $Y \in \text{SEGUIDORES}(A)$.

5.4.3 Restricciones de una Gramática de Precedencia Simple

1. **Unívocamente Invertible:** No existen dos producciones con idéntica parte derecha ($A \rightarrow \alpha$ y $B \rightarrow \alpha \implies A = B$).
2. **Ausencia de ϵ -producciones:** No existen reglas que deriven en la cadena vacía.
3. **Relaciones Disjuntas:** Entre cualquier par de símbolos X, Y existe como máximo una única relación de precedencia. Si para un par aparecen simultáneamente $<$ y $>$, la gramática no es de precedencia simple.

5.5 5. Detección y Tratamiento de Errores

En el análisis sintáctico por precedencia, los fallos emergen en dos escenarios concretos:

- **Error de relación (hueco en la tabla):** Al consultar la relación entre el símbolo en la cima de la pila y el símbolo actual de la entrada, la celda correspondiente carece de relación definida.
- **Error de reducción:** El analizador identifica los límites del pivote ($\langle \dots \rangle$), pero la subcadena aislada no se corresponde con el lado derecho de ninguna producción registrada en la gramática. La recuperación pasa por vaciar estados de la pila o insertar operandos asumidos según las heurísticas de Aho y Ullman.

Análisis Sintáctico Ascendente LR y Generación con YACC

6.1 1. La Familia de Analizadores LR(k): Fundamentos

Los analizadores LR(k) representan la técnica determinista más potente para el reconocimiento de lenguajes libres de contexto. Su nomenclatura técnica sintetiza su comportamiento operativo:

- **L** (*Left-to-right*): El texto fuente se lee y procesa de izquierda a derecha.
- **R** (*Rightmost derivation in reverse*): Reconstruye la derivación canónica por la derecha en sentido inverso (ascendente).
- **k**: Número de símbolos terminales de anticipación (*lookahead*) que inspecciona para tomar decisiones operativas (en la práctica estándar, $k = 1$).

Frente a los métodos descendentes $LL(1)$, los analizadores LR admiten gramáticas con recursividad por la izquierda sin requerir modificaciones estructurales y soportan un repertorio de construcciones de lenguaje mucho más amplio. Frente a los métodos de precedencia, un analizador LR no solo analiza el tope de la pila, sino que codifica todo el prefijo procesado mediante estados de un autómata finito.

6.2 2. Arquitectura y Dinámica de un Analizador LR

Un analizador LR se compone de una cinta de entrada, una pila de estados y una tabla de control estructurada en dos matrices complementarias:

6.2.1 La Pila de Estados

La pila almacena una secuencia alternada de estados y símbolos gramaticales:

$$s_0 X_1 s_1 X_2 s_2 \dots X_m s_m$$

Dado que cada estado resume toda la información contenida en la pila que le precede, en implementaciones reales basta con apilar únicamente los identificadores de estado $(s_0 s_1 \dots s_m)$.

6.2.2 Las Tablas ACCIÓN e IR_A

- **Tabla ACCIÓN (*Action*):** Matriz indexada por $[estado, terminal]$. Determina la acción del analizador ante el estado en la cima s_m y el token de entrada a :
 - **Desplazar (d_j):** Apila el terminal a y el nuevo estado j , avanzando el puntero de lectura de la entrada.
 - **Reducir (r_k):** Dada la producción $k : A \rightarrow \beta$, extrae de la pila $2 \times |\beta|$ elementos (los $|\beta|$ símbolos y sus estados asociados). Al quedar al descubierto el estado anterior $s_{m-|\beta|}$, consulta la tabla IR_A.
 - **Aceptar (acc):** Da por concluido con éxito el análisis sintáctico.
 - **Error (celda vacía):** Dispara la rutina de recuperación de errores sintácticos.
- **Tabla IR_A (*Goto*):** Matriz indexada por $[estado, no_terminal]$. Indica el nuevo estado que debe apilarse inmediatamente después de efectuar una reducción a un no terminal A .

6.3 3. Construcción del Analizador SLR(1) (Simple LR)

El método SLR(1) es la aproximación más directa y pedagógica para generar analizadores LR.

6.3.1 Gramática Aumentada e Ítems LR(0)

- **Gramática Aumentada:** Se añade a la gramática G la regla inicial ficticia $S' \rightarrow S$. Su propósito formal es fijar una única condición de parada en el autómata $(S' \rightarrow S \cdot)$, evitando confundir el inicio con una reducción interna a S .
- **Ítem LR(0):** Es una regla de producción con un punto $(\cdot)$ situado en alguna posición de su parte derecha. Indica cuánto de la regla ha sido reconocido en la pila y qué símbolos esperamos leer a continuación:
 - $A \rightarrow \cdot \alpha$: En espera de reconocer la cadena α .
 - $A \rightarrow \alpha \cdot \beta$: Habiendo reconocido ya el prefijo α , esperando leer β .
 - $A \rightarrow \alpha \cdot$: Ítem completo, candidato a ejecutar una reducción.

6.3.2 Operaciones Clausura e Ir_A

- **Clausura(I):** Si un ítem contiene el punto inmediatamente antes de un no terminal ($A \rightarrow \alpha \cdot B\beta$), se incorporan recursivamente al conjunto todos los ítems iniciales $B \rightarrow \cdot \gamma$ para cada producción de B en la gramática.
- **Ir_a(I, X):** Toma todos los ítems de I donde el punto precede al símbolo X ($A \rightarrow \alpha \cdot X\beta$), traslada el punto una posición a la derecha ($A \rightarrow \alpha X \cdot \beta$) y computa la clausura del conjunto resultante.

6.3.3 Colección Canónica LR(0) y Prefijos Viables

La aplicación repetida de las operaciones Clausura e Ir_A desde el ítem inicial $\{S' \rightarrow \cdot S\}$ genera la **Colección Canónica LR(0)**. Este conjunto define formalmente los estados del Autómata Finito Determinista (AFD) que reconoce los **prefijos viables** de la gramática (prefijos de formas sentenciales que pueden figurar en la pila sin rebasar el extremo derecho del pivote).

6.3.4 Reglas de Relleno de la Tabla SLR

Dado el conjunto de estados de la colección canónica $\{I_0, I_1, \dots, I_n\}$:

1. Si $[A \rightarrow \alpha \cdot a\beta] \in I_i$ y $\text{Ir_a}(I_i, a) = I_j$, se asigna d_j en $\text{ACCIÓN}[i, a]$ (para $a \in V_T$).
2. Si $[A \rightarrow \alpha \cdot] \in I_i$, se asigna $r_{A \rightarrow \alpha}$ en $\text{ACCIÓN}[i, a]$ **únicamente para los terminales** $a \in \text{SEGUIDORES}(A)$.
3. Si $[S' \rightarrow S \cdot] \in I_i$, se asigna acc en $\text{ACCIÓN}[i, \$]$.
4. Para la tabla de transiciones: Si $\text{Ir_a}(I_i, A) = I_j$, se asigna j en $\text{IR_A}[i, A]$ (para $A \in V_N$).

Si ninguna celda de la tabla presenta solapamientos de acciones, la gramática es **SLR(1)**. Si una celda contiene a la vez un desplazamiento y una reducción se produce un conflicto *desplaza/reduce*; si contiene dos reducciones distintas, un conflicto *reduce/reduce*.

6.4 4. Comparativa Teórica: SLR(1), LR(1) Canónico y LALR(1)

6.4.1 La Limitación de SLR y la Solución LR(1)

El método SLR(1) utiliza $\text{SEGUIDORES}(A)$ para autorizar una reducción. Esto representa una aproximación excesivamente laxa: un terminal a puede pertenecer al conjunto global de seguidores de A en algún punto del lenguaje, pero ser completamente ilícito en el contexto del estado específico en el que se encuentra el autómata. Esta pérdida de contexto induce conflictos ficticios.

Los analizadores **LR(1) Canónicos** solventan este déficit integrando en cada ítem un símbolo explícito de anticipación contextual:

$$[A \rightarrow \alpha \cdot \beta, a]$$

donde $a \in V_T \cup \{\$\}$ indica que la reducción $A \rightarrow \alpha$ solo es válida si el siguiente token de entrada es exactamente a . - *Ventaja*: Es el modelo determinista más potente que existe para gramáticas libres de contexto. - *Inconveniente*: Produce una explosión exponencial en el número de estados (pasa de decenas de estados a miles en gramáticas reales).

6.4.2 El Compromiso Óptimo: LALR(1) (*Look-Ahead LR*)

Desarrollado para retener la potencia contextual de LR(1) sin asumir el coste de su tabla de estados.

- **Principio de Fusión de Núcleos**: El algoritmo construye el autómata LR(1) y **fusiona aquellos estados que poseen el mismo núcleo** (*core*, es decir, que contienen exactamente los mismos ítems LR(0) prescindiendo del símbolo de anticipación), uniando sus conjuntos de *lookahead*.
- **Dimensiones**: Un autómata LALR(1) tiene exactamente el mismo número de estados que el autómata SLR(1)/LR(0) equivalente.
- **Teorema de Preservación de Conflictos (pregunta canónica de examen)**:
 - La fusión de estados con idéntico núcleo **nunca genera conflictos desplaza/reduce** nuevos.
 - La unión de los símbolos de anticipación **sí puede generar nuevos conflictos**

reduce/reduce, aunque en gramáticas reales de lenguajes de programación esta anomalía prácticamente nunca se manifiesta.

6.5 5. El Generador YACC (*Yet Another Compiler-Compiler*)

YACC es la herramienta estándar de generación de compiladores que sintetiza un analizador sintáctico LALR(1) en C a partir de un fichero de especificación `.y`.

6.5.1 Estructura de un Fichero YACC

1. **Declaraciones:** Código C delimitado por `%{ y %}`, declaración de tokens mediante `%token`, definición del axioma mediante `%start`, y fijación de precedencias con `%left`, `%right` y `%nonassoc`.
2. **Reglas Gramaticales:** Reglas de producción asociadas a acciones semánticas ejecutadas durante la reducción. Permite referenciar los valores calculados por los símbolos del lado derecho mediante `$1`, `$2`, `\dots`, `$n`, y escribir el valor resultante en la variable `$$` asociada al no terminal de la parte izquierda.
3. **Código de Usuario en C:** Funciones auxiliares, la rutina `main()` que invoca al analizador mediante `yyparse()` y el procedimiento de diagnóstico `yyerror(char *s)`.

6.5.2 Resolución de Ambigüedades por Defecto en YACC

Cuando una gramática presenta conflictos y no se establecen directivas explícitas de precedencia, YACC aplica reglas deterministas por defecto:

- En conflictos **desplaza/reduce**: YACC opta siempre por **desplazar**. Esta regla resuelve automáticamente el dilema del *dangling-else*, asociando el `else` al `if` anidado más cercano.
- En conflictos **reduce/reduce**: YACC resuelve en favor de la producción que aparezca declarada **en primer lugar** en el fichero `.y`.
- Precedencia explícita: Declaraciones como `%left '+' '-'` y `%left '*' '/'` resuelven los conflictos de expresiones aritméticas evaluando el orden de definición (las líneas más bajas tienen mayor precedencia).

6.5.3 Recuperación de Errores con el Token Reservado `error`

YACC proporciona una estrategia robusta basada en el pseudotoken `error`:

- Al producirse una incoherencia sintáctica, el analizador empieza a vaciar estados de la pila hasta alcanzar un estado que contenga una transición activa sobre el símbolo `error`.
- El autómata desplaza el token `error` a la pila y entra en modo de sincronización, descartando tokens de la cinta de entrada hasta encontrar un símbolo delimitador especificado en la regla (típicamente el punto y coma `;` en una sentencia).
- Al encontrar el delimitador, reduce por la regla de error y reanuda el análisis normal del resto del programa, evitando abortar la compilación.