Compiladores

Bibliografía:

- Apuntes de Compiladores, María Antonia Cárdenas Viedma, Eduardo Martínez Graciá y María Antonia Martínez Carreras (2018), Grado en Ingeniería Informática, Universidad de Murcia.

Capítulo 1

Traductores e intérpretes

Un **lenguaje de programación** es una notación formal para expresar algoritmos. Distinguimos lenguajes:

1. **De primera generación** o **lenguajes máquina**, código binario entendible directamente por un procesador y dependiente de la máquina. Los programas son difíciles de leer, escribir, modificar y corregir, por lo que el desarrollo es lento y propenso a fallos.

2. **De segunda generación** o **ensambladores**, creados al comienzo de los años 50, permiten usar abreviaturas mnemotécnicas para representar las instrucciones de la máquina y códigos octales o hexadecimales para valores. También permiten macros, secuencias de instrucciones parametrizadas para uso frecuente.

3. **De tercera generación** o **de alto nivel**, que permiten usar estructuras de control basadas en objetos lógicos y especificar los datos, funciones o procesos de forma independiente de la máquina. Fortran, Cobol, C, C++, Java.

4. **De cuarta generación**, para problemas específicos, como SQL para bases de datos o PostScript para dar formato a texto.

5. **De quinta generación** o **declarativos**, como Prolog o Haskell, con los que se especifica el cálculo que se quiere realizar más que cómo debe realizarse.

1.1. Traductores

Un **traductor** es un programa que acepta textos en un lenguaje de programación **fuente** (**programa fuente**) y genera otro semánticamente equivalente en un lenguaje de programación **destino** (**programa objeto**). Tipos:

- **Preprocesador**: traductor de una forma extendida de un lenguaje de alto nivel a su forma estándar. Reúne el programa fuente, a menudo dividido en módulos en ficheros distintos.

- **Compilador**: de un lenguaje de alto nivel a otro de bajo nivel, informando también de cualquier error detectado en el programa fuente.

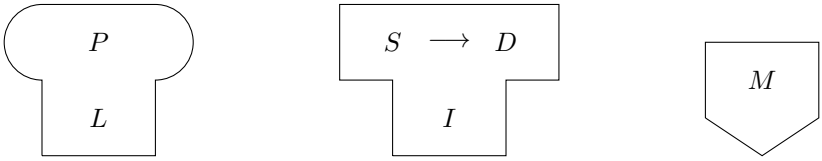

Figura 1.1: De izquierda a derecha, diagrama de un programa común, de un traductor y de una máquina.

  - **Cruzado** (*cross-compiler*): se ejecuta sobre una **máquina huésped** pero genera código para una **máquina destino** distinta.
  - ***Just-in-time*** (**JIT**): Traduce código de una máquina abstracta a código de una máquina real según se necesite en la ejecución de dicho código.
  - **Optimizador**: modifica el código intermedio o el código objeto para mejorar su eficiencia, sin alterar la funcionalidad del programa original.
- **Ensamblador**: de un lenguaje ensamblador a su correspondiente código máquina.
- **Traductor de alto nivel**: la fuente y el destino son lenguajes de alto nivel.
- **Desensamblador**: de un código máquina a su correspondiente lenguaje ensamblador.
- **Descompilador**: de un lenguaje de bajo nivel a otro de alto nivel.

Un **traductor de** n **etapas** es el traductor resultante de componer n traductores, manejando $n-1$ lenguajes intermedios. Un traductor es **de una pasada** si genera el código objeto realizando una única lectura del código fuente, o **de múltiples pasadas** si procesa varias veces el código fuente o alguna representación intermedia de este.

Representamos programas, traductores (que también son programas) y máquinas como se muestra en la figura 1.1, donde P es el nombre del programa, L es el lenguaje en que está representado, S es el lenguaje fuente del traductor, D es el lenguaje destino, I es el **lenguaje de implementación**, en el que está representado el traductor, y M es el lenguaje máquina aceptado.

Para indicar que un programa se ejecuta sobre una cierta máquina, lo situamos encima, y para indicar que un traductor traduce un programa de un lenguaje fuente a uno destino, situamos el programa fuente, cuyo lenguaje es el lenguaje fuente del traductor, a la izquierda del mismo, y el destino, de mismo nombre pero lenguaje el de destino del traductor, a la derecha. Así, un proceso traductor se representa como en la figura 1.2.

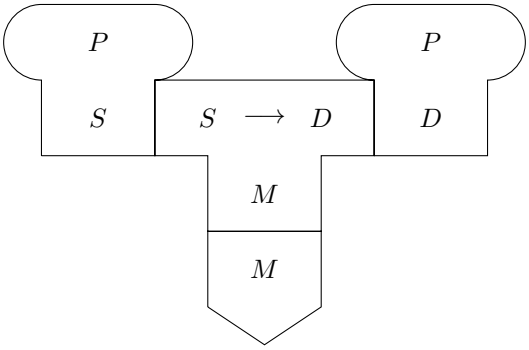

Figura 1.2: Proceso de traducción.

1.2. Compiladores

Un compilador se divide en fases:

1. Fase de **análisis** o ***front-end***: Determina la estructura y el significado de un código fuente creando una representación intermedia.

   a) **Análisis léxico**: Transforma un flujo de caracteres en un flujo de ***tokens***, identificadores de variables o funciones, palabras clave, constantes, operadores, etc.

   b) **Análisis sintáctico**: Crea un árbol sintáctico que refleja la estructura gramatical del programa. Se usan autómatas con pila para reconocer una gramática libre de contexto normalmente recursiva, si bien la mayoría de lenguajes de programación son dependientes del contexto.

   c) **Análisis semántico**: Realiza verificaciones que no se pueden incluir en gramáticas libres de contexto y calcula valores semánticos, creando un **árbol semántico**. La semántica se especifica con métodos formales.

   d) **Generación de código intermedio** en un lenguaje de bajo nivel, que debe ser fácil de producir y de traducir a código objeto.

2. **Optimización de código intermedio**, resultando en un código semánticamente equivalente pero con menor consumo de memoria o tiempo de ejecución. No se suele producir código óptimo, pues este es un problema NP-completo. Como esta fase requiere tiempo, el compilador suele ofrecer la opción de desactivarla para el desarrollo o la depuración de programas.

3. Fase de **síntesis** o ***back-end***: Convierte el código intermedio en un programa objeto semánticamente equivalente.

   a) **Generación de código**: Traducción dirigida por la sintaxis, basada en la gramática del lenguaje intermedio.

   b) **Optimización** de código dependiente de la máquina.

En la práctica, algunas de estas fases se agrupan, sin construir los datos intermedios de forma explícita.

Cuando el lenguaje de implementación y el lenguaje fuente del compilador coinciden, el compilador puede compilarse a sí mismo, y hablamos de **arranque** o *bootstrapping*. El primer compilador capaz de compilarse a sí mismo fue credo para Lisp por Hart y Levin en el MIT en 1962, y desde 1970 esto es habitual, aunque Pascal y C son alternativas muy usadas.

Cuando un programa se divide en varios ficheros fuente, en general la compilación de cada uno produce un fichero objeto con código reubicable. El **enlazador** une todos estos ficheros, así como el de las bibliotecas estáticas usadas, y el **cargador**, parte del sistema operativo, carga en memoria el ejecutable resultante con las bibliotecas dinámicas usadas, uniéndolos, y salta al punto de entrada del programa.

1.3. Jerarquía de lenguajes de Chomsky

Una **gramática** es una tupla $G \coloneqq (V_N, V_T, P, S)$ donde V_N es un alfabeto de símbolos **no terminales**, V_T es un alfabeto de **símbolos terminales** disjunto de V_N, P es un conjunto finito de **reglas de producción** de la forma $\alpha \to \beta$ con $\alpha, \beta \in (V_N \cup V_T)^*$ y $S \in V_N$ es el **símbolo inicial**.

Para $\alpha, \beta \in (V_N \cup V_T)^*$, α **deriva directamente** en β, $\alpha \Rightarrow \beta$, si existen $\delta, \gamma, \mu, \sigma \in (V_N \cup V_T)^*$ tales que $\alpha = \delta\gamma\mu$, $\beta = \delta\sigma\mu$ y $\gamma \to \sigma \in P$. Si $\alpha =: \gamma_0 \Rightarrow \cdots \Rightarrow \gamma_n \coloneqq \beta$, $(\gamma_0, \ldots, \gamma_n)$ es una **derivación** de **longitud** n de α a β, y en particular (α) es una derivación de longitud 0 de α a α. Decimos que α deriva en β, $\alpha \Rightarrow^* \beta$, si existe una derivación de α a β, y que $\alpha \Rightarrow^+ \beta$ si dicha derivación se puede tomar de longitud positiva.

Una **forma sentencial** es un elemento de $D(G) \coloneqq \{\alpha \in (V_N \cup V_T)^* \mid S \Rightarrow^* \alpha\}$, y una **sentencia** es un elemento de $\mathcal{L}(G) \coloneqq D(G) \cap V_T^*$, el **lenguaje definido** por la gramática.

Tipos de lenguajes:

- **Tipo 0**, definidos por gramáticas **sin restricciones** o **con estructura de frase**, con reglas $\alpha X \beta \to \gamma$, siendo X cualquier símbolo, terminal o no. Reconocidos por máquinas de Turing.
- **Tipo 1**, definidos por gramáticas **dependientes del contexto**, con reglas $\alpha A \beta \to \alpha\gamma\beta$, siendo A no terminal. Reconocidos por autómatas linealmente acotados.
- **Tipo 2**, definidos por gramáticas **libres de contexto**, con reglas $A \to \alpha$ siendo A no terminal. Reconocidos por autómatas con pila.
- **Tipo 3**, definidos por gramáticas **regulares**, con reglas $A \to a$, $A \to aB$ o $A \to \lambda$, siendo A y B no terminales y a terminal. Reconocidos por autómatas finitos.

Los lenguajes de un tipo también son de todos los tipos anteriores, aunque muchos lenguajes no son de tipo 0. La mayoría de lenguajes de programación son de tipo 1, aunque muchas de sus reglas gramaticales pueden reducirse al tipo 2 y, para los símbolos básicos, al tipo 3.

1.4. Intérpretes

Un **intérprete** es un programa que acepta un **programa fuente** en un **lenguaje fuente** y lo ejecuta inmediatamente, sin traducirlo a un código objeto. Es un buen método cuando el

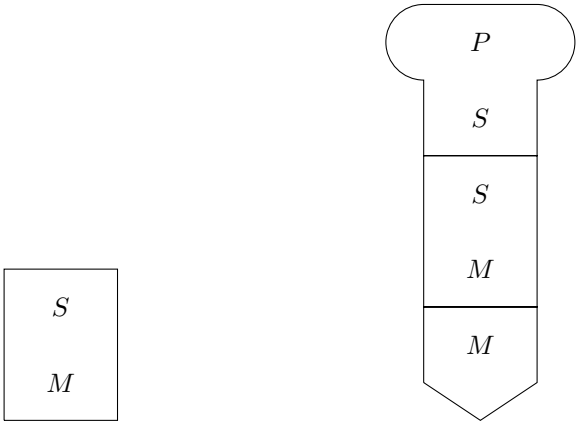

Figura 1.3: Intérprete (izquierda) e interpretación de un programa (derecha).

programador está trabajando de forma interactiva; el programa se va a utilizar pocas veces, con lo que el rendimiento no es importante; se espera que cada instrucción se ejecute una sola vez, y las instrucciones tienen un formato simple.

Un intérprete muy usado es el intérprete de comandos de Unix, la *shell*. Los intérpretes los representamos como en la figura 1.3, donde S es el lenguaje fuente y M es la máquina sobre la que se ejecuta el intérprete.

La interpretación de un programa en un lenguaje de alto nivel es unas 100 veces más lenta que la ejecución de un programa equivalente en código máquina, por lo que esto no interesa cuando el programa se va a ejecutar en producción ni cuando las instrucciones se van a ejecutar frecuentemente o tienen formatos complicados.

Un **emulador** es un intérprete cuyo lenguaje fuente es de bajo nivel, útil para verificar un diseño hardware. Así, podemos ver una máquina como un intérprete implementado en hardware, un intérprete como una máquina implementada en software y un código máquina como un lenguaje para el que existe un intérprete hardware. A veces llamamos **máquinas abstractas** a los intérpretes para diferenciarlos de las máquinas reales.

Un **compilador interpretado** es un traductor de un lenguaje fuente a un lenguaje intermedio diseñado para que la traducción sea rápida, el nivel de abstracción sea intermedio entre el lenguaje fuente y el código maquina y las instrucciones tengan un formato simple, facilitando su interpretación.

El código de la máquina virtual de Java (**JVM**) es un lenguaje de este tipo, pues proporciona instrucciones que corresponden directamente a operaciones como creación de objetos, llamadas a métodos e indexado de matrices, facilitando la traducción de Java a código intermedio, pero estas tienen un formato sencillo como las instrucciones máquina, con campos de operación y operandos, facilitando la interpretación. El kit de desarrollo de Java (**JDK**) contiene un traductor de Java a código JVM, un intérprete de código JVM y un compilador JIT que complementa al intérprete.

1.5. Portabilidad

Un programa es **portable** si puede ser compilado y ejecutado en cualquier máquina. La **portabilidad** de un programa es la porción de código que no haría falta cambiar cuando el programa se mueve entre máquinas diferentes.

Para el código ensamblador, la portabilidad es, en general, del 0 %, mientras que para un lenguaje de alto nivel, es del 95–100 %. Los procesadores de lenguajes se escriben en lenguajes de alto nivel, aunque los compiladores no pueden ser totalmente portables si queremos que compilen al lenguaje de la implementación.

Capítulo 2

Análisis léxico

Un ***token*** es un par formado por un **nombre de *token***, que representa un tipo de unidad léxica, y un atributo opcional, cuyo tipo depende del nombre de *token*. Todo nombre de *token* lleva asociado un **patrón**, un lenguaje regular que describe las secuencias de caracteres que deben ser convertidas a un *token* con dicho nombre, llamadas **lexemas**.

Un **analizador léxico** es un programa que separa una secuencia de caracteres de entrada en lexemas y genera una lista de *tokens*, informando de errores y pudiendo hacer otras tareas como ignorar comentarios y caracteres de espaciado.

El analizador sintáctico usa los nombres de *token* como símbolos terminales de una gramática libre de contexto, pues aunque esta gramática podría incluir la definición de los *tokens*, dividir el análisis en dos fases simplifica el diseño, y mejora la portabilidad al no tener que preocuparse el analizador sintáctico de la codificación de caracteres. En general el analizador léxico está subordinado al sintáctico, ofreciéndole una interfaz de llamadas.

2.1. Fundamentos teóricos

Dado un alfabeto V, los símbolos λ y $\emptyset$ y los elementos de V son **expresiones regulares**; también lo son $(\alpha|\beta)$, $(\alpha \circ \beta)$ y (α^*) si α y β son expresiones regulares, y solo son expresiones regulares las expresiones que se obtienen aplicando estas reglas. Se pueden omitir los paréntesis si no causa ambigüedad, entendiendo que $*$ tiene más precedencia que $\circ$ y $\circ$ más que $|$ y que los tres operadores son asociativos por la izquierda, y se puede escribir $\alpha\beta \coloneqq \alpha \circ \beta$.

Toda expresión regular α lleva asociada un lenguaje $L(\alpha)$, dado por $L(\emptyset) \coloneqq \emptyset$; $L(\lambda) \coloneqq \{\lambda\}$; si $a \in V$, $L(a) \coloneqq \{a\}$, y si α y β son expresiones regulares, $L(\alpha|\beta) \coloneqq L(\alpha) \cup L(\beta)$, $L(\alpha\beta) \coloneqq L(\alpha)L(\beta)$ y $L(\alpha^*) \coloneqq L(\alpha)^*$. Un lenguaje es regular si es el asociado a alguna expresión regular. Dos expresiones regulares son **equivalentes**, $\alpha = \beta$, si $L(\alpha) = L(\beta)$.

Un **AFD** es una tupla (Q, V, δ, q_0, F) formada por un conjunto finito de **estados** Q, un alfabeto V, un **estado inicial** $q_0 \in Q$, un conjunto de **estados finales** $F \subseteq Q$ y una **función de transición** $\delta : D \subseteq (Q \times V) \to Q$.

Un **AFND** es una tupla (Q, V, δ, q_0, F), definida como un AFD salvo que $\delta : Q \times (V \dot{\cup} \{\lambda\}) \to \mathcal{P}(Q)$.

2.2. Diseño

Para diseñar un analizador léxico usamos *Flex*, una herramienta multiplataforma que genera código en C basándose en un fichero de especificación de expresiones regulares, sucesora de *Lex* para Unix. Los ficheros de especificación suelen tener extensión `.l` y constan de 3 partes separadas por una línea `%%`:

1. **Definiciones**: Macros para expresiones regulares auxiliares con formato *`nombre expresión`* en una línea. Código C, entre una línea `%{` y una `}%`. Opciones con `%option` *`opción`*. Condiciones de contexto, tablas, etc.

2. **Reglas**: Con forma *`expresión código`*, donde *`expresión`* es una expresión regular que puede incluir macros con formato `{`*`nombre`*`}` y *`código`* es una línea de C o un bloque código entre llaves que se ejecuta cuando se reconoce un lexema asociado a la expresión regular. Si en el código hay una sentencia `return`, esta sale del analizador léxico devolviendo el entero indicado, normalmente el código de *token*. De lo contrario sigue leyendo, por lo que es habitual usar reglas para espacios y comentarios con la sentencia vacía (`;`).

3. **Subrutinas de usuario**: Código C auxiliar que se añade al final del fichero generado.

La orden `flex [-o` *`SALIDA`*`] `*`FICHERO`* genera un analizador léxico *`SALIDA`* (por defecto, `lex.yy.c`) desde la especificación *`FICHERO`*. Este debe enlazarse a la biblioteca `fl`, lo que en `gcc` se hace con la opción `-lfl`.

2.2.1. Identificar los *tokens* y definir los patrones

Se suelen considerar *tokens*:

- Los identificadores.
- Las **palabras clave**. Están **reservadas** si usuario no puede modificar su significado, en cuyo caso el analizador léxico puede reconocerlas directamente. Si no lo están, el analizador léxico debe reconocerlas como identificadores para que una fase posterior las distinga.
- Los operadores y signos de puntuación.
- Las constantes simples, como reales o enteros sin el signo, caracteres o cadenas de caracteres.
- Los espacios en blanco y comentarios, aunque en general el analizador léxico los ignora, no los devuelve.

Las expresiones regulares en *Flex* se forman combinando símbolos y operadores. Los operadores son, en orden de prioridad:

1. `?`, `*` y `+`: respectivamente, de 0–1, 0 o más y 1 o más repeticiones de lo anterior.
2. Concatenación de símbolos.
3. `{...}`, de repetición.

4. `^`, de comienzo de línea, y `$`, de final.

5. `|` de disyunción entre los elementos a izquierda y derecha.

6. `/`, para reconocer la expresión regular de la izquierda solo si va seguida de la de la derecha (*look-ahead*).

También se puede agrupar con paréntesis `(...)`, preceder un símbolo de `\` para evitar que se trate como un operador o indicar una cadena de caracteres literal con `"..."`. Otros operadores:

1. `[...]` indica disyunción entre los caracteres del conjunto indicado: *`a`*`-`*`b`* indica todos los caracteres entre *`a`* y *`b`*, inclusive; `^` al principio complementa el conjunto; `\` permite indicar caracteres de escape; `[:...:]` indica una clase de caracteres como `[:digit:]` o `[:alnum:]`, y cualquier otro caracter dentro de los corchetes se indica a sí mismo.

2. `.` indica cualquier caracter salvo salto de línea.

3. `<<EOF>>` indica final de fichero.

Llamamos **caracteres de anticipación** a los que se leen al hacer *look-ahead* para determinar el final de un *token*, lo que hace falta cuando hay reglas que acaban en `+`, `*`, `?`, `$` y `/...`, y cuando un lexema de una expresión regular puede ser prefijo de uno de otra. *Flex* maneja esto de forma automática, devolviendo a la entrada los caracteres no usados (*push back*).

En caso de ambigüedad en las reglas, *Flex* reconoce el lexema más largo posible que pueda estar asociado a una expresión regular, y si este puede corresponder a varias, se asocia a la que aparezca antes. Así, las palabras reservadas deben estar antes que la expresión regular que reconoce los identificadores.

2.2.2. Construcción e implementación del autómata

Se podría obtener un AFD correspondiente a las expresiones regulares directamente, pero esto es difícil, por lo que se obtiene un AFND y, como su simulación es costosa, se construye entonces un AFD equivalente y se minimiza. Podemos simular directamente la tabla de transiciones con los estados o simular el diagrama de transiciones mediante código específico. De esto se encarga *Flex*.

2.2.3. Diseño de la interfaz de entrada y salida

Para la entrada, es muy ineficiente leer el código de caracter a caracter, por lo que se lee en bloques que se guardan en *buffers*, que deben diseñarse permitiendo secuencias largas de caracteres de anticipación. La salida suele ser una o varias funciones que el analizador sintáctico llama para obtener el siguiente *token*. *Flex* proporciona la siguiente interfaz:

`yytext` Último lexema leído.

`yyleng` Longitud de `yytext`.

`yylineno` Activando la opción `yylineno`, número de línea actual.

`yymore()` Indica a *Flex* que, la próxima vez que se lea un lexema, este se debería concatenar al actual (`yytext`) en vez de reemplazarlo.

`yyless(int)` Retrasa el puntero de lectura para que apunte al caracter de `yytext` con el índice dado.

`yywrap()` Se ejecuta al alcanzarse el final del fichero.

`yyin` Descriptor de fichero de donde se lee.

`yyout` Descriptor de fichero al que se escribe al usar la opción `echo`.

`input()` Lee el siguiente caracter del flujo de entrada.

`output(char)` Escribe el caracter dado en la salida.

`unput(char)` Coloca el caracter dado en el flujo de entrada, para que sea el primero leído en la próxima ocasión.

2.2.4. Manejo de errores

Una forma sencilla es la **recuperación en modo pánico**: ignorar caracteres hasta encontrar un caracter válido para un nuevo *token*. Para implementarlo es necesaria una expresión regular capaz de reconocer un conjunto de caracteres erróneos antes del comienzo de un nuevo lexema. Otras técnicas permiten borrar un caracter extraño, insertar uno que falta, reemplazar un caracter incorrecto por otro correcto o intercambiar caracteres adyacentes.

Capítulo 3

Análisis sintáctico

Un **analizador sintáctico** toma una cadena de componentes léxicos y, si esta verifica la gramática libre de contexto del lenguaje fuente, genera un árbol sintáctico, y de lo contrario genera un informe con la lista de errores detectados. También puede hacer tareas de análisis semántico como completar la tabla de símbolos, verificar los tipos y generar código intermedio.

3.1. Recuperación de errores

El informe de errores debe ser claro y preciso, la recuperación debe ser rápida para poder continuar con el procesamiento y el mecanismo de detección no debe ralentizar programas correctos.

- **Recuperación en modo pánico**: Cuando se recibe un *token* que no concuerda con la especificación sintáctica del lenguaje, se empiezan a desechar *tokens* hasta que se encuentra uno de un **conjunto de sincronización**, normalmente con terminadores de estructuras sintácticas. Esto desecha muchos *tokens*.
- **Recuperación a nivel de frase**: Cuando se descubre un error, se inserta una cadena que permita continuar con el análisis. Da problemas si el error se produjo antes del punto de detección.
- **Producciones de error**: Si se sabe en qué puntos están los errores más frecuentes, ampliar la gramática con reglas de producción que simulen la producción de errores.
- **Corrección global**: Se calcula la distancia mínima de la cadena incorrecta a una correcta y se sustituye. Es muy costoso, y se usa para la evaluación de otras técnicas o de forma local para encontrar cadenas de sustitución óptimas en una recuperación a nivel de frase.

3.2. Fundamentos teóricos

Dada una gramática libre de contexto (GLC) $G \coloneqq (V_N, V_T, P, S)$, una derivación directa $\alpha A \mu \Rightarrow \alpha\gamma\mu$ con $A \to \gamma \in P$ es **más a la izquierda**, $\alpha \Rightarrow_{\text{mi}} \beta$, si $\alpha \in V_T^*$, y es **más a la derecha**, $\alpha \Rightarrow_{\text{md}} \beta$, si $\mu \in V_T^*$.

Una **derivación más a la izquierda** de α es una de la forma $S \Rightarrow_{\text{mi}}^* \alpha$, y una **derivación más a la derecha** es una de la forma $S \Rightarrow_{\text{md}}^* \alpha$. A una derivación más a la derecha

$$S \Rightarrow_{\text{md}} \gamma_1 \Rightarrow_{\text{md}} \cdots \Rightarrow_{\text{md}} \gamma_n$$

con $n \geq 0$ le corresponde una **reducción por la izquierda** $\gamma_n \Rightarrow_{\text{mi}}^R \cdots \Rightarrow_{\text{mi}}^R \gamma_1 \Rightarrow_{\text{mi}}^R S$, donde escribimos $\alpha \Rightarrow^R \beta$ si $\alpha \Rightarrow \beta$ en la gramática con reglas de producción $\{\gamma \to \delta\}_{\delta \to \gamma \in P}$.

Si $\gamma A \mu \Rightarrow \alpha \coloneqq \gamma\beta\mu$, β es una **frase simple** de α respecto a A. Llamamos **forma sentencial derecha** o **más a la derecha** al resultado de una derivación más a la derecha, y llamamos **pivote** de una forma sentencial derecha α a la frase simple asociada a la última derivación directa en una derivación más a la derecha de α.

Un **árbol de derivación** para la GLC G es un árbol ordenado y etiquetado con raíz S tal que, para todo nodo con etiqueta X, bien $X \in V_T$ y el nodo es hoja, bien $X \in V_N$ y los hijos del nodo tienen etiquetas $X_1, \ldots, X_k$ con $X \to X_1 \cdots X_k$.

Una sentencia de G es **ambigua** si existen dos árboles de derivación en G distintos tales que la sentencia es la concatenación ordenada de las etiquetas de todos los nodos hoja, si y sólo si esta admite dos derivaciones más a la izquierda distintas, si y sólo si admite dos derivaciones más a la derecha distintas. Una gramática es ambigua si tiene alguna sentencia ambigua.

La ambigüedad de una gramática es indecidible. Un lenguaje de tipo 2 es **inherentemente ambiguo** si toda gramática que lo genere es ambigua.

Algunos casos de ambigüedad:

- La debida a la precedencia y asociatividad de los operadores. Una gramática ambigua como $E \to E + E \mid E * E \mid id$ es equivalente a una no ambigua $E \to E + T \mid T; T \to T * F \mid F; F \to id$.
- La debida a las sentencias `if-then-else`. Una gramática como $S \to iEtS \mid iEtSeS \mid s$, donde i, t y e son los *tokens* respectivos `if`, `then` y `else`, es ambigua porque $iEtiEtSeS$ tiene dos árboles de derivación. Esto se puede solucionar haciendo que las sentencias `if-then` e `if-then-else` terminen con un *token* `endif` o usando la gramática equivalente

  $$S \to S_e \mid S_{ne}; S_e \to iEtS_eeS_e \mid s; S_{ne} \to iEtS \mid iEtS_eeS_{ne},$$

  aunque esto no se suele hacer porque es menos eficiente, y en su lugar se incluye en el analizador un caso especial.

Un **autómata de pila** es una tupla $M \coloneqq (Q, V, \Sigma, \delta, q_0, z_0, F)$ donde Q es un conjunto finito de **estados**, V es el **alfabeto de entrada**, Σ es el **alfabeto de la pila**, $q_0 \in Q$ es el **estado inicial**, $z_0 \in \Sigma$ es el **símbolo inicial** de la pila, $F \subseteq Q$ es el conjunto de **estados finales** y $\delta : Q \times (V \cup \{\lambda\}) \times \Sigma \to \mathcal{P}(Q \times \Sigma^*)$ es la **función de transición**.

Una **configuración** de un autómata con pila es una tupla $(q, w, \alpha) \in Q \times V^* \times \Sigma^*$. Un **movimiento** es una transición de una configuración a otra de la forma $(q, a\omega, z\alpha) \Rightarrow (q', \omega, \beta\alpha)$ con $(q', \beta) \in \delta(q, a, z)$ o de la forma $(q, \omega, z\alpha) \Rightarrow (q', \omega, \beta\alpha)$ con $(q', \beta) \in \delta(q, a, z)$. En general los autómatas de pila son **no deterministas**, es decir, existen varios movimientos con la misma configuración inicial (a la izquierda).

Una cadena $\omega \in V^*$ es **reconocida** por un autómata **por estado final** si $(q_0, \omega, z_0) \Rightarrow^* (q_f, \lambda, \gamma)$ para ciertos $q_f \in F$ y $\gamma \in \Sigma^*$ o **por vaciado de pila** si $F = \emptyset$ y $(q_0, w, z_0) \Rightarrow^* (q, \lambda, \lambda)$ para cierto $q \in Q$. El **lenguaje reconocido** por un autómata de pila es el conjunto de cadenas

en Σ^* que reconoce. Un lenguaje es reconocido por algún autómata de pila si y sólo si es de tipo 2.

Algunas características de los lenguajes de programación no son libres de contexto:

- Declaración de identificadores. $L_1 \coloneqq \{wcw \mid w \in \{a,b\}^*\}$, donde en wcw, la primera w representa una declaración de identificador, la c un fragmento de programa cualquiera y la segunda w un uso del identificador, no es de tipo 2.
- Número de parámetros de las funciones. Si $L_2 \coloneqq \{a^n b^m c^n d^m \mid n, m \geq 1\}$, donde una a es un parámetro de una primera función, una b es un parámetro de una segunda función, una c es el paso de un argumento a la primera y una d es el paso de un argumento a la segunda, entonces L_2 no es de tipo 2.

Para estas, se asigna el mismo *token* a todos los identificadores y se permite en la gramática enviar un número arbitrario de argumentos a las funciones, y en el análisis semántico se comprueba que los identificadores han sido declarados y el número de argumentos es el correcto.

3.3. Métodos de análisis semántico

Suelen ser de estos tipos:

1. **Descendentes**: Se construye el árbol de derivación de la raíz a las hojas usando derivaciones izquierdas.
2. **Ascendentes**: Se construye el árbol de las hojas a la raíz usando reducciones izquierdas.

Los métodos **generales** pueden analizar cualquier GLC, pero son bastante ineficientes, por lo que vemos métodos eficientes que admiten subconjuntos de las gramáticas libres de contexto suficientemente expresivos.

Dada una GLC (V_N, V_T, P, S), definimos $\mathsf{PRIMERO} : (V_N \cup V_T)^* \to \mathcal{P}(V_T \sqcup \{\lambda\})$ como

$$\mathsf{PRIMERO}(\alpha) \coloneqq \{a \in V_T \mid \exists \beta : \alpha \Rightarrow^* a\beta\} \cup \{\lambda \mid \alpha \Rightarrow^* \lambda\}.$$

Para calcularlo, usamos el algoritmo 1 para obtener $\sigma : V_N \cup V_T \to V_T^*$ y calculamos

$$\mathsf{PRIMERO}(X_1 \cdots X_n) = \\ = \bigcup_{i=1}^{\min(\{i \mid X_1 \cdots X_i \not\Rightarrow^* \lambda\} \cup \{n\})} (\sigma(X_i) \setminus \{\lambda\}) \cup \{\lambda \mid X_1 \cdots X_n \Rightarrow^* \lambda\}.$$

Definimos $\mathsf{SIGUIENTE} : V_N \to \mathcal{P}(V_T \sqcup \{\$\})$ como

$$\mathsf{SIGUIENTE}(A) \coloneqq \{a \in V_T \mid \exists \alpha, \beta : S \Rightarrow^+ \alpha A a \beta\} \cup \{\$ \mid \exists \alpha \mid S \Rightarrow^* \alpha A\},$$

lo que podemos calcular con el algoritmo 2.

Entrada: Gramática (V_N, V_T, P, S)
Salida: Mapeo $\sigma := \mathsf{PRIMERO}|_{V_N \cup V_T}$.
para $X \in V_T$ **hacer** $\sigma(X) \leftarrow \{X\}$;
para $X \in V_N$ **hacer**
  si $X \Rightarrow^* \lambda$ **entonces** $\sigma(X) \leftarrow \{\lambda\}$;
  sinó $\sigma(X) \leftarrow \emptyset$;
fin
repetir
  para $X \to Y_1 \cdots Y_k \in P$ **hacer**
    para $i \in \{1, \ldots, k\} : Y_1 \cdots Y_{i-1} \Rightarrow^* \lambda$ **hacer**
      Añadir todo $\sigma(Y_i) \setminus \{\lambda\}$ a $\sigma(X)$
    fin
    si $Y_1 \cdots Y_k \Rightarrow^* \lambda$ **entonces** Añadir λ a $\sigma(X)$;
  fin
hasta que *no se hayan añadido más elementos a los conjuntos*;

Algoritmo 1: Cálculo de PRIMERO para símbolos.

Entrada: Gramática (V_N, V_T, P, S).
Salida: Mapeo $\sigma :=$ SIGUIENTE
SIGUIENTE$(S) \leftarrow \{\$\}$;
para $A \in V_N \setminus \{S\}$ **hacer** SIGUIENTE$(A) \leftarrow \emptyset$;
para $A \to X_1 \cdots X_k \in P$ **hacer**
  para $i \leftarrow 1$ **a** $k - 1$ **hacer**
    si $X_i \in V_n$ **entonces**
      Añadir todo PRIMERO$(X_{i+1} \cdots X_k) \setminus \{\lambda\}$ a SIGUIENTE(X_i);
    fin
  fin
fin
repetir
  para $A \to X_1 \cdots X_k : k \geq 1$ **hacer**
    para $i \leftarrow 0$ **a** k **hacer**
      si $X_i \in V_N \wedge \lambda \in$ PRIMERO$(X_{i+1} \cdots X_k)$ **entonces**
        Añadir todo SIGUIENTE(A) a SIGUIENTE(X_i);
      fin
    fin
  fin
hasta que *no se hayan añadido más elementos a los conjuntos*;

Algoritmo 2: Cálculo de SIGUIENTE.

3.4. Análisis ascendente

Una GLC es $LR(k)$ para un $k \geq 0$ si existe un algoritmo que, leyendo *tokens* de izquierda a derecha (*Left to right*), construya una derivación más a la derecha (*Rightmost derivation*) de la entrada, en el caso de que esta esté en la gramática, sin necesidad de examinar más de k *tokens* de anticipación.

Un **analizador LR** es una estructura formada por:

1. Un alfabeto de **símbolos terminales** V_T.
2. Un alfabeto de **símbolos no terminales** V_N disjunto de V_T.
3. Un conjunto finito de **estados** S.
4. Un **estado inicial** $s_0 \in S$.
5. Una tabla $\mathsf{Acción} : S \times (V_T \sqcup \{\$\}) \to X$, donde X puede ser *Desplazar* s con $s \in S$, *Reducir* $A \to \beta$ con $A \in V_N$ y $\beta \in (V_N \cup V_T)^*$, *Aceptar* o *Error*. A veces se abrevia *Desplazar* s como ds, *Reducir* $A \to \beta$, siendo $A \to \beta$ la regla i-ésima, como ri, y *Aceptar* como *Acc*, y se omite *Error* al representar la tabla.
6. Una tabla $\mathsf{IrA} : S \times V_N \to S \sqcup \{\emptyset\}$.

Una **configuración** del analizador es un par $(s_0\, X_1\, s_1\, \ldots\, X_m\, s_m, a_i a_{i+1} \cdots a_n\$)$, donde $s_0, \ldots, s_m \in S$ y $X_1, \ldots, X_m \in V_N \cup V_T$, $a_i, \ldots, a_n \in V_T$. El primer elemento del par es el **contenido de la pila** y el segundo elemento es la cadena de símbolos de entrada que queda por reconocer. Ejecutar el analizador sobre una entrada es aplicar el algoritmo 3, que recibe una entrada y devuelve un árbol de derivación de esta si cumple una cierta gramática $LR(1)$, o bien encuentra un error. Un analizador LR se puede convertir en un autómata de pila.

3.4.1. Método SLR

Dadas una gramática $G \coloneqq (V_N, V_T, P, S)$ y $\gamma \in (V_N \cup V_T)^*$, γ es un **prefijo viable** de G si existen $\alpha, \beta, \omega \in (V_N \cup V_T)^*$ y $A \in V_N$ tales que $S \Rightarrow^*_{\text{md}} \alpha A \omega \Rightarrow_{\text{md}} \alpha\beta\omega$ y γ es prefijo de $\alpha\beta$.

Llamamos **ítem** ($LR(0)$) de una gramática a una regla de producción de esta a la que se añade un punto en alguna posición de la cadena a la derecha, que indica la porción de la regla que ha sido reconocida en la entrada, de modo que cuando el punto está a la derecha podemos aplicar una reducción.

Llamamos **gramática aumentada** o **extendida** de G a $(V_N \cup \{S'\}, V_T, P \cup \{S' \to S\}, S')$, donde $S' \notin V_N \cup V_T$. Definimos la **clausura** de un conjunto I de ítems, $\mathsf{Clausura}(I)$, según el algoritmo 4, y dados un conjunto de ítems I y $X \in V_N \cup V_T$, definimos

$$\mathsf{Goto}(I, X) := \mathsf{Clausura}(\{A \to \alpha X \cdot \beta\}_{A \to \alpha \cdot X \beta \in I}).$$

Con esto definimos la **colección** $LR(0)$ según el algoritmo 5.

Un ítem $LR(0)$ $A \to \beta \cdot \gamma$ es **válido** para un prefijo viable $\alpha\beta$ si existe una derivación $S' \Rightarrow^*_{\text{md}} \alpha A \omega \Rightarrow_{\text{md}} \alpha\beta\gamma\omega$ con $\omega \in V_T^*$.

El método SLR para generar un analizador ascendente es el algoritmo 6. Una gramática es $SLR(1)$ si el método no genera conflictos. Toda gramática ambigua genera conflictos, pero no toda gramática que genera conflictos es ambigua.

Entrada: Un analizador LR y una entrada de *tokens* que termina con el símbolo \$.
Salida: Un árbol sintáctico.
Establecer la pila a (s_0). En esta versión del algoritmo, la pila no contiene símbolos de V_T o V_N sino árboles completos;
mientras *haya entrada* **hacer**
  Llamar s al estado en la cima de la pila;
  Leer un *token* a;
  seleccionar $\mathsf{Acción}(s, a)$ **hacer**
    caso Desplazar s' **hacer**
      Apilar a y s';
    caso Reducir $A \to \beta$ **hacer**
      $k := |\beta|$;
      Desapilar $2k$ entradas, guardando los árboles como $T_1, \ldots, T_k$;
      Llamar s' al estado en la cima de la pila;
      Apilar el árbol $A(T_1, \ldots, T_k)$ y el estado $\mathsf{IrA}(s', A)$;
      Devolver a a la entrada;
    caso Aceptar **hacer**
      devolver *el último árbol apilado*;
    caso Error **hacer**
      Informar del error e intentar continuar;
  fin
fin

Algoritmo 3: Análisis sintáctico LR.

Entrada: GLC (V_N, V_T, P, S) y conjunto de ítems I de esta.
Salida: Clausura J de I
$J \leftarrow I$;
repetir
  para $A \to \alpha \cdot B\beta \in J$ **hacer**
    para $B \to \gamma \in P$ **hacer** Añadir $B \to \cdot\gamma$ a J;
  fin
hasta que *no se añadan más elementos*;

Algoritmo 4: Clausura de un conjunto de ítems $LR(0)$.

Entrada: GLC (V_N, V_T, P, S) extendida con un cierto S'
Salida: Colección $LR(0)$ C.
$C \leftarrow \{\mathsf{Clausura}(\{S' \to \cdot S\})\}$;
repetir
  para $I \in C$ **hacer**
    para $X \in V_N \cup V_T \cup \{S'\}$ **hacer**
      si $\mathsf{Goto}(I, X) \neq \emptyset$ **entonces** Añadir $\mathsf{Goto}(I, X)$ a C;
    fin
  fin
hasta que *no se añadan más elementos*;

Algoritmo 5: Colección $LR(0)$.

Entrada: GLC aumentada con S'.
Salida: Tablas Acción e IrA del analizador, conjunto de estados C y estado inicial s, o conflicto.

Construir la colección $LR(0)$ C;
para $I \in C$ **hacer**
  para $a \in V_T$ **hacer** $\mathsf{Acción}(I, a) \leftarrow \mathit{Error}$;
  para $A \to \alpha \cdot \beta \in I$ **hacer**
    si $A \to \alpha \cdot \beta = S' \to \cdot S$ **entonces** $s \leftarrow I$;
    si $|\beta| \geq 1$ **entonces**
      $a \leftarrow \beta_0$;
      $N \leftarrow \mathit{Desplazar}\ \mathsf{Goto}(I, a)$;
      si $\mathsf{Acción}(I, a) \neq \text{Error}, N$ **entonces** Conflicto;
      $\mathsf{Acción}(I, a) \leftarrow N$;
    sinó, si $A \neq S'$ **entonces**
      $N \leftarrow \mathit{Reducir}\ A \to \alpha$;
      para $a \in \mathsf{SIGUIENTE}(A)$ **hacer**
        si $\mathsf{Acción}(I, a) \neq \text{Error}, N$ **entonces** Conflicto;
        $\mathsf{Acción}(I, a) \leftarrow N$;
      fin
    sinó
      $\mathsf{Acción}(I, \$) \leftarrow \mathit{Aceptar}$; // Caso $S' \to S$
    fin
  fin
fin
para $I \in C, A \in V_N$ **hacer** $\mathsf{IrA}(I, A) \leftarrow \mathsf{Goto}(I, A)$;

Algoritmo 6: Método SLR.

Los conflictos pueden ser:

- Desplaza-reduce (*shift-reduce*): Se podría tanto desplazar el siguiente símbolo a la pila como reducir.
- Reduce-reduce (*reduce-reduce*): Hay al menos dos reglas de producción aplicables.

En estos casos se puede especificar manualmente la acción a tomar o usar un método más potente.

3.4.2. Método LR-canónico

Un **ítem** $LR(1)$ de una gramática aumentada es un par $[A \to \beta \cdot \gamma, a]$ formado por un ítem $LR(0)$ y un símbolo $a \in V_T \sqcup \{\$\}$. Es **válido** para el prefijo viable $\alpha\beta$ si existe $\omega \in V_T^*$ tal que $S' \Rightarrow_{\text{md}}^* \alpha A\omega \Rightarrow_{\text{md}} \alpha\beta\gamma\omega$ y, bien $\omega \neq \lambda$ y $a = \omega_0$, bien $\omega = \lambda$ y $a = \$$.

Definimos la clausura de un conjunto de ítems $LR(1)$ según el algoritmo 7, y para un conjunto de ítems I y un $X \in V_N \cup V_T$, llamamos $\mathsf{Goto}(I, X) \coloneqq \mathsf{Clausura}(\{[A \to \alpha X \cdot \beta, a]\}_{[A \to \alpha \cdot X\beta, a] \in I})$.

Entrada: GLC (V_N, V_T, P, S) y conjunto I de ítems $LR(1)$ de esta.
Salida: Clausura J de I.
$J \leftarrow I$;
repetir
  para $[A \to \alpha \cdot B\beta, a] \in J$ **hacer**
    para $b \in \mathsf{PRIMERO}(\beta a)$ **hacer**
      para $B \to \gamma \in P$ **hacer**
        Añadir $[B \to \cdot\gamma, b]$ a J;
      fin
    fin
  fin
hasta que *no se hayan añadido elementos*;

Algoritmo 7: Clausura de un conjunto de ítems $LR(1)$.

El algoritmo para obtener la colección $LR(1)$ de una GLC es el mismo que para obtener la colección $LR(0)$, pero inicializando el conjunto a $\{\mathsf{Clausura}(\{[S' \to \cdot S, \$]\})\}$.

El método LR-canónico es el algoritmo 8 y funciona para toda gramática $LR(1)$, pero genera muchos estados.

3.4.3. Método LALR

Tiene una potencia intermedia entre el método LR-canónico y el SLR, y consigue una tabla de análisis de igual tamaño que con SLR. Pasos:

1. Construir la colección $LR(1)$ C.
2. Si, para $I \in C$, $\rho(I) \coloneqq \{R \mid \exists a \in V_T \mid [R, a] \in I\}$, para $I, J \in C$ tales que $\rho(I) = \rho(J)$, sustituir I y J en C por su unión.
3. Aplicar el método LR-canónico a la colección resultante.

Entrada: GLC (V_T, V_N, P, S) aumentada con S'.
Salida: Conjunto de estados C, estado inicial s y tablas Acción e IrA, o conflicto.
Obtener la colección $LR(1)$ C;
para $I \in C$ **hacer**
  para $a \in V_T$ **hacer** Acción$(I, a) \leftarrow$ *Error*;
  para $[A \to \alpha \cdot \beta, b] \in I$ **hacer**
    si $A \to \alpha \cdot \beta = S' \to \cdot S$ **entonces** $s \leftarrow I$;
    si $|\beta| \geq 1$ **entonces**
      $a \leftarrow \beta_0$;
      si $a \in V_T$ **entonces**
        $N \leftarrow$ *Desplazar* Goto(I, a);
        si Acción$(I, a) \neq$ Error$, N$ **entonces** Conflicto;
        Acción$(I, a) \leftarrow N$;
      fin
    sinó, si $A \neq S'$ **entonces**
      $N \leftarrow$ *Reducir* $A \to \alpha$;
      si Acción$(I, b) \neq$ Error$, N$ **entonces** Conflicto;
      Acción$(I, b) \leftarrow N$;
    sinó
      Acción$(I, \$) \leftarrow$ *Aceptar* ; // Caso $[S' \to S\cdot, \$]$
    fin
  fin
fin
para $I \in C; A \in V_N$ **hacer** IrA$(I, A) \leftarrow$ Goto(I, A);

Algoritmo 8: Método LR canónico.

Una gramática es $LALR(1)$ si en este último paso no se producen conflictos. Si se aplica el método LALR a una gramática $LR(1)$:

- Pueden surgir conflictos *reduce-reduce*.

  La gramática $S \to XY; X \to Aa \mid Bb; Y \to Ac \mid Ba; A \to a; B \to a$ es $LR(1)$, pero al fusionar los conjuntos $\{[A \to a\cdot, a], [B \to a\cdot, b]\}$ y $\{[A \to a\cdot, c], [B \to a\cdot, a]\}$ de su colección $LR(1)$ en un conjunto K, obtenemos que $\mathsf{Acción}[I, a]$ puede ser *Reduce* $A \to a$ o *Reduce* $B \to a$.

- No pueden surgir conflictos *shift-reduce*.

  Si hubiera uno, habría $\alpha, \beta, \gamma \in (V_N \cup V_T)^*$, $a, x \in V_T$ y $A, B \in V_N$ tales que existe un J en la colección de conjuntos LALR de la gramática con $[A \to \alpha \cdot a\gamma, x], [B \to \beta\cdot, a] \in J$. Si $J = I_1 \cup \cdots \cup I_n$ con $I_1, \ldots, I_n$ conjuntos de la colección $LR(1)$, existe k con $[B \to \beta\cdot, a] \in I_k$ y existe $y \in V_T$ con $[A \to \alpha \cdot a\gamma, y] \in I_k$, luego el conflicto ya existiría en el método LR-canónico.

3.4.4. Conflictos `if-then-else`

La gramática no ambigua que vimos para `if-then-else` es $LR(1)$, pero genera muchos estados. En la gramática

$$S \to iSeS \mid iS \mid s,$$

una versión simplificada de la gramática ambigua, ocurre un conflicto *shift-reduce*, pero podemos hacer que el `else` colgante se asocie al `if` más cercano en SLR priorizando el desplazamiento en el conflicto. En efecto, la colección $LR(0)$ es

$$\begin{aligned}\{&\{S' \to \cdot S, S \to \cdot iSeS, S \to \cdot iS, S \to \cdot s\}, \{S' \to S\cdot\},\\ &\{S \to i \cdot SeS, S \to i \cdot S, S \to \cdot iSeS, S \to \cdot iS, S \to \cdot s\},\\ &\{S \to iS \cdot eS, S \to iS\cdot\},\\ &\{S \to iSe \cdot S, S \to \cdot iSeS, S \to \cdot iS, S \to \cdot s\}, \{S \to s\cdot\}\},\end{aligned}$$

y el único conflicto sucede en $\{S \to iS \cdot eS, S \to iS\cdot\}$, luego si queremos que el `else` que viene a continuación se asocie al `if` más cercano debemos hacer un desplazamiento.

3.4.5. Recuperación de errores

Un acceso a IrA no produce errores, pues cuando se aplica una producción, sabemos que el nuevo conjunto de símbolos de la pila forma un prefijo viable. El método LR-canónico detecta los errores de la entrada inmediatamente, mientras que los métodos LALR y SLR pueden hacer varias reducciones antes de detectar el error, pero ninguno desplaza un símbolo de entrada erróneo en la pila.

Recuperación:

- Tratamiento a nivel de frase: Para cada entrada en blanco de Acción, se introduce una rutina de manejo que inserte o elimine símbolos de la pila o de la entrada, o que altere o transponga símbolos de la entrada. Debe evitarse extraer de la pila un símbolo no terminal, pues indica que la estructura se había reconocido con éxito.

- Modo pánico: Se extraen pares de elementos de la pila hasta encontrar un estado s para el que exista A con $s' \coloneqq \mathsf{IrA}(s, A)$ definido, que no se llega a extraer; se introducen A y s' en la pila para simular una reducción, y se descartan elementos de la entrada hasta encontrar un símbolo de $\mathsf{Siguiente}(A)$, que tampoco se descarta.

3.5. Transformación de gramáticas

Entrada: GLC (V_N, V_T, P, S).
Salida: GLC λ-libre equivalente (V'_N, V_T, P', S')
$N \leftarrow \{A\}_{A \to \lambda \in P}$;
$(V'_N, P', S') \leftarrow (V_N, P, S)$;
repetir
  para $A \to X_1 \cdots X_k \in P : X_1, \ldots, X_k \in N$ **hacer** Añadir A a N;
hasta que *no se hayan añadido elementos*;
para $A \to X_1 \cdots X_k$ **hacer**
  $t \leftarrow |\{i : X_i \in N\}|$;
  Definir $\sigma : \{1, \ldots, t\} \to \{1, \ldots, k\}$ inyectiva con $X_{\sigma(i)} \in N$ para todo i;
  para $j_1, \ldots, j_t \leftarrow 0$ **a** *1* **hacer**
    $\alpha \leftarrow \lambda$;
    para $i = 1$ **a** k **hacer**
      si $X_i \notin N \vee j_{\sigma^{-1}(i)} = 1$ **entonces** $\alpha \leftarrow \alpha X_i$;
    fin
    Añadir $A \to \alpha$ a P';
  fin
fin
$P' \leftarrow \{A \to \alpha \in P' : \alpha \neq \lambda\}$;
si $S \in N$ **entonces**
  si *S no aparece a la derecha de una regla* **entonces**
    Añadir $S \to \lambda$ a P'
  sinó
    Añadir un símbolo S' a V'_N que no esté en $V_N \cup V_T$;
    Añadir $S' \to S$ y $S' \to \lambda$ a P';
  fin
fin

Algoritmo 9: Eliminación de λ-reglas.

Una GLC (V_N, V_T, P, S) es **λ-libre** si no contiene producciones de la forma $A \to \lambda$, salvo quizá $S \to \lambda$, en cuyo caso S no aparece a la derecha de ninguna regla de producción. Toda GLC es equivalente a una λ-libre por el algoritmo 9.

Una **producción unitaria** es una regla de la forma $A \to B$ con $A, B \in V_N$. Toda GLC es equivalente a una que no contiene producciones unitarias por el algoritmo 10.

$X \in V_N \cup V_T$ es una **variable improductiva** si no existe $\omega \in V_T^*$ con $X \Rightarrow^* \omega$, es un **símbolo inaccesible** si no aparece en ninguna forma sentencial, es **inútil** si es inaccesible o una variable improductiva y es **útil** en caso contrario.

Entrada: GLC (V_N, V_T, P, S).
Salida: GLC (V_N, V_T, P', S) equivalente sin reglas unitarias.
para $A \in V_N$ **hacer** $U(A) \coloneqq \{B \in V_N \setminus \{A\} : A \Rightarrow^* B\}$;
$P' \leftarrow \{A \to \alpha \in P : \nexists B \in V_N : \alpha = B\}$;
para $A \in V_N$ **hacer**
  para $B \in U(A)$ **hacer**
    para $B \to \beta \in P'$ **hacer** Añadir $A \to \beta$ a P';
  fin
fin

Algoritmo 10: Eliminación de reglas unitarias.

Entrada: GLC (V_N, V_T, P, S).
Salida: GLC (V'_N, V_T, P', S) equivalente sin variables improductivas.
$V'_N \leftarrow \{A \in V_N : \exists \omega \in V_T^* : A \to \omega \in P\}$;
repetir
  para $A \to X_1 \cdots X_k \in P$ **hacer**
    si $X_1, \ldots, X_k \in M$ **entonces** Añadir A a V'_N;
  fin
hasta que *no se hayan añadido más elementos*;
$P' \leftarrow \{A \to X_1 \cdots X_k \in P : X_1, \ldots, X_k \in V'_N\}$;

Algoritmo 11: Eliminación de variables improductivas.

Entrada: GLC (V_N, V_T, P, S).
Salida: GLC (V'_N, V'_T, P', S) equivalente sin símbolos inaccesibles.
$R \leftarrow \{S\}$;
repetir
  para $A \to X_1, \ldots, X_k \in P : A \in R$ **hacer** Añadir $X_1, \ldots, X_k$ a R;
hasta que *no se hayan añadido más elementos*;
$(V'_N, V'_T) \leftarrow (V_N \cap R, V_T \cap R)$;
$P' \leftarrow \{A \to X_1, \ldots, X_k \in P : A \in R\}$;

Algoritmo 12: Eliminación de símbolos inaccesibles.

Toda GLC admite una GLC equivalente sin símbolos inútiles, aplicando sucesivamente los algoritmos 11 y 12. En efecto, el algoritmo 12 no resulta en variables improductivas si no las había de antes, pero el algoritmo 11 sí que puede hacer a algunos símbolos inaccesibles.

Una GLC es **propia** si es λ-libre y no tiene símbolos inútiles ni ciclos. Toda GLC admite una GLC equivalente propia, que se puede obtener aplicando sucesivamente y en orden los algoritmos 9–12.

$A \in V_N$ es **recursiva por la izquierda** si $\exists \alpha : A \Rightarrow^+ A\alpha$, **recursiva por la derecha** si $\exists \alpha : A \Rightarrow^+ \alpha A$ y **recursiva** si $\exists \alpha, \beta : A \Rightarrow^+ \alpha A \beta$. Una gramática es recursiva por la izquierda, por la derecha o recursiva (a secas) si tiene alguna variable que lo sea. Una regla de producción es recursiva por la izquierda si es de la forma $A \to A\alpha$, por la derecha si es de la forma $A \to \alpha A$ y recursiva si es de la forma $A \to \alpha A \beta$.

Entrada: GLC (V_N, V_T, P, S) propia.
Salida: GLC equivalente (V'_N, V_T, P', S) sin recursividad por la izquierda.
Llamar $\{A_1, \ldots, A_n\} \coloneqq V_N$ con $A_1 = S$;
$(V'_N, P') \leftarrow (V_N, P)$;
repetir
  para $i \leftarrow 1$ **a** n **hacer**
    si A_i *tiene producciones recursivas por la izquierda* **entonces**
      Llamar a las producciones de A_i como $A_i \to A_i\alpha_1 \mid \cdots \mid A_i\alpha_m \mid \beta_1 \mid \cdots \mid \beta_n$, donde $\beta_{j0} \neq A_i$ para ningún i;
      Dado un $A' \notin V'_N \cup V_T$, $V'_N \leftarrow V_N \cup \{A'\}$;
      Sustituir las producciones de A_i en P' por
      $\{A_i \to \beta_1 \mid \cdots \mid \beta_n \mid \beta_1 A' \mid \cdots \mid \beta_n A',$
       $A' \to \alpha_1 \mid \cdots \mid \alpha_m \mid \alpha_1 A' \mid \cdots \mid \alpha_m A'\}$;
    fin
    para $j \leftarrow 1$ **a** $i - 1$ **hacer**
      para $R := (A_i \to A_j\alpha) \in P'$ **hacer**
        Si las producciones de A_j son $A_j \to \gamma_1 \mid \cdots \mid \gamma_p$, sustituir R por $A_i \to \gamma_1\alpha \mid \cdots \mid \gamma_p\alpha$;
      fin
    fin
  fin
hasta que *no se cambie* P;

Algoritmo 13: Eliminación de la recursividad por la izquierda.

Toda GLC recursiva por la izquierda admite otra equivalente que no es recursiva por la izquierda, por el algoritmo 13.

Si existen reglas $A \to \alpha\beta, A \to \alpha\gamma \in P$ con $|\alpha| > 0$, α es un **factor común por la izquierda**. Toda GLC admite una equivalente sin factores comunes.

3.6. Análisis descendente

Una GLC es $LL(k)$ si, leyendo una entrada de izquierda a derecha (*Left-to-right*), se puede construir la derivación más a la izquierda correspondiente (*Leftmost derivation*) sin más de k

Entrada: GLC (V_N, V_T, P, S).
Salida: GLC equivalente (V'_N, V_T, P', S) sin factores comunes por la izquierda.
$(V'_N, P') \leftarrow (V_N, P)$;
para $A \in V_N$ **hacer**
  repetir
    Encontrar el prefijo α más largo común a dos producciones de A;
    si $\alpha \neq \lambda$ **entonces**
      Llamar a estas producciones $A \to \alpha\beta_1 \mid \cdots \mid \alpha\beta_n \mid \gamma_1 \mid \cdots \mid \gamma_k$, donde ningún γ_i tiene prefijo α;
      Dado $A' \notin V'_N \cup V_T$, añadir A' a V'_N;
      Sustituir las producciones de A por $A \to \alpha A' \mid \gamma_1 \mid \cdots \mid \gamma_k$ y $A' \to \beta_1 \mid \cdots \mid \beta_n$;
    fin
  hasta que $\alpha = \lambda$;
fin

Algoritmo 14: Factorización por la izquierda.

tokens de anticipación. Toda gramática $LL(k)$ es $LR(k)$.

Saber si un lenguaje es o no $LL(1)$ es indecidible, pero a veces podemos transformar una gramática para obtener otra equivalente $LL(1)$, por ejemplo, eliminando ambigüedad y recursividad por la izquierda o factorizando por la izquierda.

Dada una GLC (V_N, V_T, P, S), definimos $\mathsf{Predict} : P \to \mathcal{P}(V_T \sqcup \{\$\})$ como

$$\mathsf{Predict}(A \to \alpha) := \begin{cases} (\mathsf{PRIMERO}(\alpha) \setminus \{\lambda\}) \cup \mathsf{SIGUIENTE}(A), & \lambda \in \mathsf{PRIMERO}(\alpha); \\ \mathsf{PRIMERO}(\alpha), & \text{en otro caso.} \end{cases}$$

Una **tabla de análisis** de un analizador descendente predictivo no recursivo es una función $M : V_N \times (V_T \cup \{\$\}) \to \mathcal{P}(P)$ dada por $M(A, a) \coloneqq \{A \to \alpha \in P \mid a \in \mathsf{Predict}(A \to \alpha)\}$, que a cada no terminal a derivar y terminal siguiente en la entrada le asocia un conjunto de reglas que pueden aplicarse para realizar la siguiente derivación.

Con esto, una GLC es $LL(1)$ si ningún $M(A, a)$ tiene más de un elemento, si y sólo si para cualquier par de producciones de un mismo no terminal $A \to \alpha$ y $A \to \beta$, $\mathsf{Predict}(A \to \alpha)$ y $\mathsf{Predict}(A \to \beta)$ son disjuntos.

La ejecución de un analizador descendente predictivo no recursivo se hace con el algoritmo 15, en el que, si ω es la porción de entrada reconocida hasta ahora, la pila contiene una $\alpha \in (V_N \cup V_T)^*$ seguida de \$ (al fondo) de forma que $S \Rightarrow^*_{\text{mi}} \omega\alpha$.

3.6.1. Conflictos `if-then-else`

La gramática no ambigua que vimos para `if-then-else` no es $LL(1)$, pues $\mathsf{Predict}(S \to S_e) = \{i, s\}$ y $\mathsf{Predict}(S \to S_{ne}) = \{i\}$, pero una solución común es partir de la gramática inicial ambigua, factorizarla si se puede y construir la tabla de análisis seleccionando la regla que resuelve cada conflicto.

En este caso, factorizando la gramática $S \to iEtS \mid iEtSeS \mid s; E \to x$ obtenemos

$$S \to iEtSS' \mid s; S' \to eS \mid \lambda; E \to x,$$

Entrada: La tabla de análisis M para una gramática (V_N, V_T, P, S) $LL(1)$ y un flujo de *tokens* acabado en la marca de fin \$.
Salida: Derivación más a la izquierda de la entrada, o error.
Inicializar la pila P con \$ y S;
repetir
  Sacar un símbolo X de la pila;
  Leer un *token* a;
  si $X \neq a$ **entonces**
    si $X \in V_T \vee M[X, a] = \emptyset$ **entonces**
      Error.
    sinó, si $M[X, a] = \{X \to Y_1 \cdots Y_k\}$ **entonces**
      Introducir $Y_k, \ldots, Y_1$ en orden en P;
      Devolver a a la entrada;
      Emitir la derivación por la izquierda dada por $X \to Y_1 \cdots Y_k$;
    fin
  fin
hasta que $X =$ \$;

Algoritmo 15: Ejecución de un analizador descendente no recursivo.

que también es ambigua, y tenemos

$$\mathsf{Predict}(S \to iEtSS') = \{i\}, \qquad \mathsf{Predict}(S \to s) = \{s\},$$
$$\mathsf{Predict}(S' \to eS) = \{e\}, \qquad \mathsf{Predict}(S' \to \lambda) = \{e, \$\}, \qquad \mathsf{Predict}(E \to x) = \{x\}.$$

El único conflicto se da entre $S' \to eS$ y $S' \to \lambda$, que tienen en común el símbolo e, y para asociar el `else` con el `if` más próximo, se debe elegir $S' \to eS$ en la entrada (S', e) de la tabla de análisis.

3.6.2. Recuperación de errores

Los errores que pueden surgir son que el terminal en la cima de la pila no coincida con el de la entrada y que no haya una producción en la tabla para la variable y el *token* de entrada.
Recuperación:

- A nivel de frase: En cada casilla en blanco de la tabla, que se amplía al dominio $(V_N \cup V_T) \times (V_T \cup \{\$\})$, se añade una rutina de manejo del error, que hace operaciones como cambiar, eliminar o añadir caracteres y emite un mensaje de error. Esto es complicado, pues requiere considerar todos los casos de error posibles.

- En modo pánico: Si el terminal en la cima de la pila no coincide con el de la entrada, se extrae el de la cima de la pila pero no el de la entrada, lo que simula la inserción de este en la entrada. Si la tabla no tiene una producción para la variable A de la pila y el *token* de entrada, se descartan *tokens* de la entrada hasta encontrar uno en $\mathsf{PRIMERO}(A)$, en cuyo caso se continúa con el análisis, o en $\mathsf{SIGUIENTE}(A)$, en cuyo caso se saca A de la pila simulando así que se ha reconocido una cadena derivable de A. No se descarta el *token* de entrada que cumple la condición.