LA2-Generación de código intermedio

Dr J. Guadalupe (Gualo) Ramos Díaz

Marzo 2025

Temario

- 2.1 Notaciones.
  - 2.1.1 Prefija.
  - 2.1.2 Infija.
  - 2.2.3 Postfija.
- 2.2 Representaciones de código intermedio.
  - 2.2.1 Notación Polaca.
  - 2.2.2 Código P.
  - 2.2.3 Triplos.
  - 2.2.4 Cuádruplos.
- 2.3 Esquema de generación.
  - 2.3.1 Variables y constantes.
  - 2.3.2 Expresiones.
  - 2.3.3 Instrucción de asignación.
  - 2.3.4 Instrucciones de control.
  - 2.3.5 Funciones.
  - 2.3.6 Estructuras.

Compilación vs interpretación

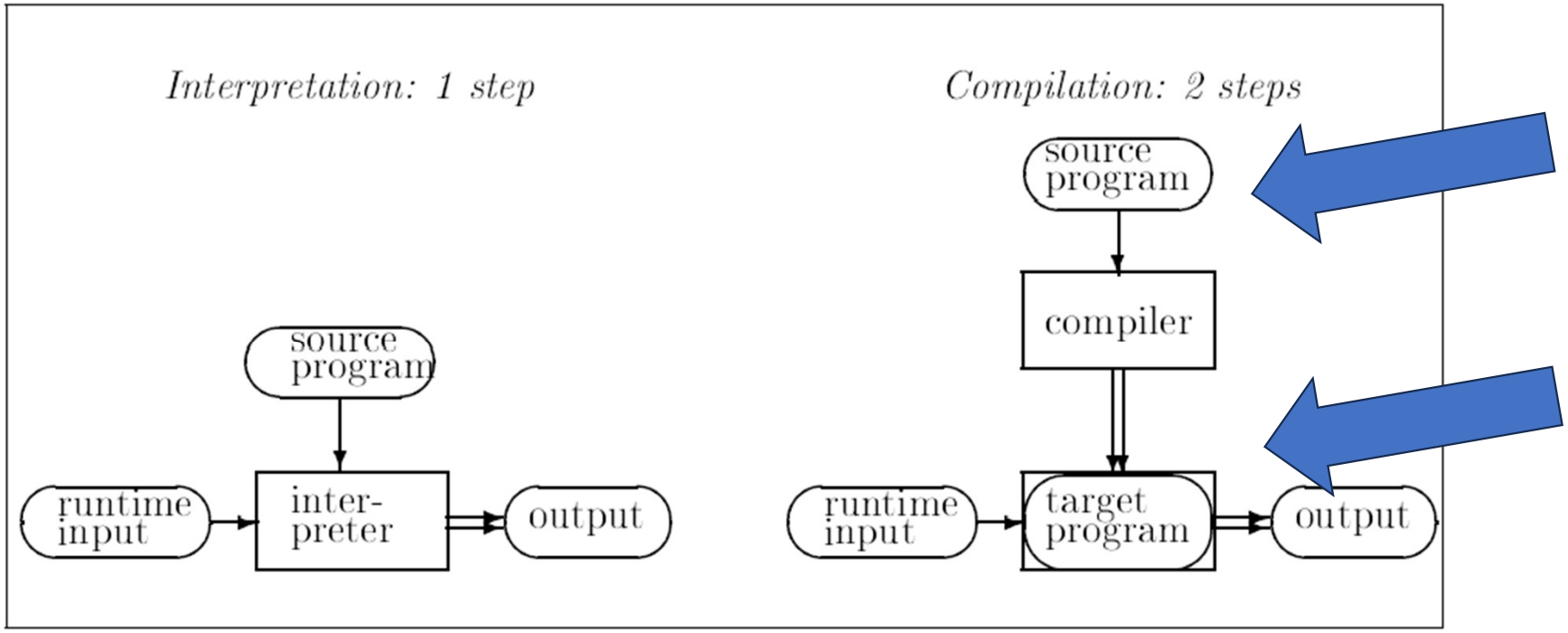
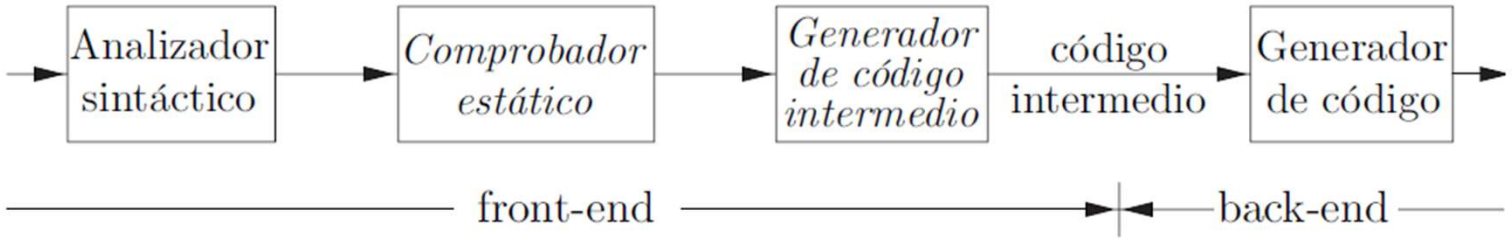


Figure 1.3: Compilation in two steps, interpretation in one.

- Jones, Gomar, Sestoft, 1999, Partial Evaluation and Automatic Program Generation, Prentice Hall

Front-end vs Back-end Compilation (Aho, 08)

- En el modelo de análisis y síntesis de un compilador, **el front-end analiza un programa fuente y crea una representación intermedia**, a partir de la cual el back-end genera el código destino.
- Lo apropiado es que los detalles del **lenguaje fuente se confinen al front-end**, y los detalles de la **máquina de destino al back-end**.
- Con una representación intermedia definida de manera adecuada, podemos construir un compilador para el lenguaje *i* y la máquina *j* mediante la combinación del front-end para el lenguaje *i* y el back-end para la máquina *j*.
- Este método para crear una suite de compiladores puede ahorrar una considerable cantidad de esfuerzo: podemos construir ***m* × *n* compiladores** con sólo escribir *m* front-ends y *n* back-ends.
- Todos los esquemas pueden implementarse mediante la creación de un árbol sintáctico y después el recorrido de éste



Front-end vs Back-end Compilation (Aho, 08)

- Durante el proceso de traducir un programa en un lenguaje fuente dado al código para una máquina destino, un compilador puede construir una secuencia de representaciones intermedias, como en la figura 6.2.
- Las representaciones de alto nivel están cerca del lenguaje humano y las representaciones de bajo nivel están cerca de la máquina destino.
- Los **árboles sintácticos** son de alto nivel; describen la estructura jerárquica natural del programa fuente y se adaptan bien a tareas como la comprobación estática de tipos.

Figura 6.2: Un compilador podría usar una secuencia de representaciones intermedias

Front-end vs Back-end Compilation (Aho, 08)

- Una representación de bajo nivel es adecuada para las **tareas dependientes de la máquina**, como la asignación de registros y la selección de instrucciones.
- El **código de tres direcciones** puede variar de alto a bajo nivel, dependiendo de la elección de operadores.
- Para las expresiones, las diferencias entre los árboles sintácticos y el código de tres direcciones es superficial
- Por ejemplo, para las instrucciones de ciclos un árbol sintáctico representa a los componentes de una instrucción, mientras que el código de tres direcciones contiene etiquetas e instrucciones de salto para representar el flujo de control, como en el lenguaje máquina.

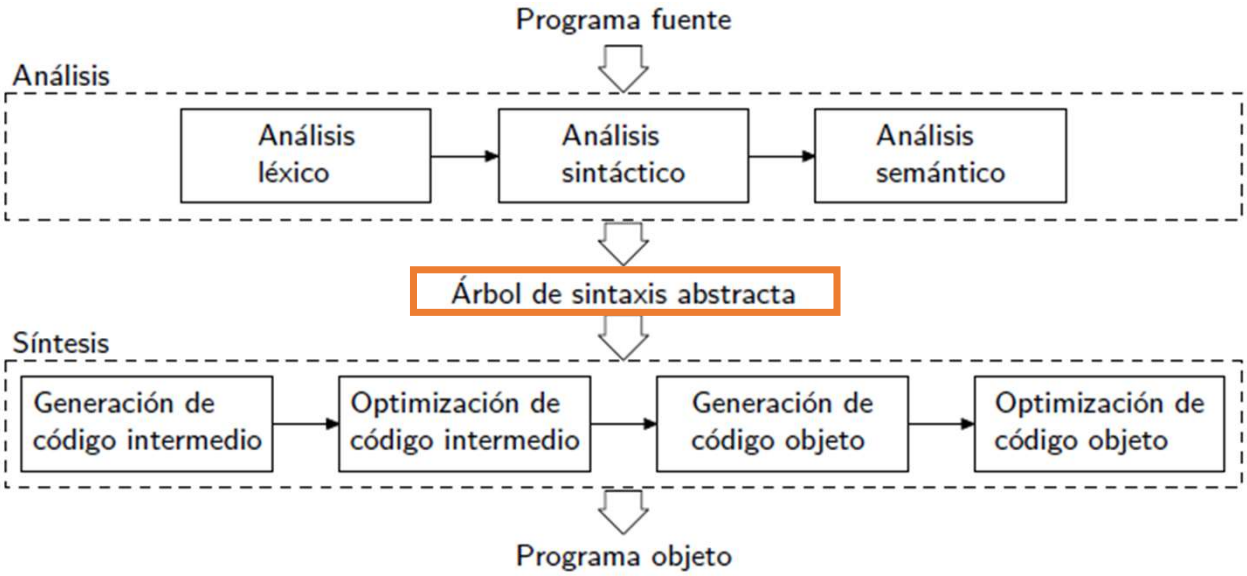
Front-end vs Back-end Compilation (Aho, 08)

- La elección o diseño de una representación intermedia varía de un compilador a otro.
- **Una representación intermedia puede ser un verdadero lenguaje, o puede consistir en estructuras de datos internas que se comparten mediante las fases del compilador.**
- C es un lenguaje de programación, y aún así se utiliza con frecuencia como forma intermedia, ya que es flexible, se compila en código máquina eficiente y existe una gran variedad de compiladores.
- El compilador de C++ original consistía en una front-end que generaba C, y trataba a un compilador de C como back-end.

Definición, ventajas e inconvenientes (Ribadas, 2006)

- **Representación Intermedia (RI)**
  - Definición (página 2)
  - **Ventajas** (página 2)
  - **Inconvenientes** (página 3)
  - **Características** deseables (página 4)
  - Tipos (página 4)
    - Lineales y Arbóreas
  - Ejemplos de representaciones Intermedias industriales(página 4)
    - **Práctica de código de 3 direcciones con GCC**
  - Introducción a las representaciones arbóreas (página 5)

RI Arbóreas

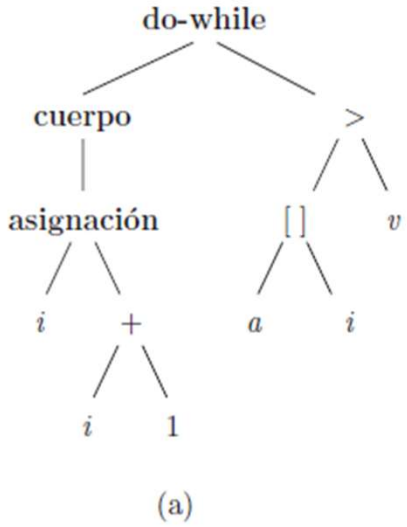

Programa fuente
Análisis
Análisis léxico
Análisis sintáctico
Análisis semántico
Árbol de sintaxis abstracta
Síntesis
Generación de código intermedio
Optimización de código intermedio
Generación de código objeto
Optimización de código objeto
Programa objeto

RI Arbóreas: Árbol de sintaxis abstracta (Aho, 2008)

- En la figura se ilustran dos formas de código intermedio. Una forma, conocida como **árboles sintácticos abstractos o simplemente árboles sintácticos**, representa la estructura sintáctica **jerárquica** del programa fuente.
- En el modelo de la figura el analizador sintáctico **produce un árbol sintáctico, que se traduce posteriormente en código de tres direcciones**.

RI Arbóreas: Árbol de sintaxis abstracta (Aho, 2008)

- Algunos compiladores combinan el análisis sintáctico y la generación de código intermedio en un solo componente



(a)

```
1: i = i + 1
2: t1 = a [ i ]
3: if t1 < v goto 1
```

(b)

Figura 2.4: Código intermedio para "`do i=i+1; while (a[i] <v);`"

La raíz del árbol sintáctico abstracto en la figura 2.4(a) representa un ciclo do-while completo. El hijo izquierdo de la raíz representa el cuerpo del ciclo, que consiste sólo de la asignación `i=i+1;`. El hijo derecho de la raíz representa la condición `a[i]<v`. En la sección 2.8(a) aparece una implementación de los árboles sintácticos.

```
fun longitud(x) =
        if null(x) then 0 else longitud(tl(x)) + 1;
```

Ejemplo 6.15: El árbol sintáctico abstracto de la figura 6.29 representa la definición de *longitud* en la figura 6.28. La raíz del árbol, etiquetada como **fun**, representa a la definición de la función. El resto de los nodos que no son hojas pueden verse como aplicaciones de funciones. El nodo etiquetado como + representa la aplicación del operador + a un par de hijos. De manera similar, el nodo etiquetado como **if** representa la aplicación de un operador **if** a una tripleta formado por sus hijos (para la comprobación de tipos, no importa si se va a evaluar la parte **then** o **else**, pero no ambas).

RI Arbóreas: Árbol de sintaxis abstracta (Aho, 2008)

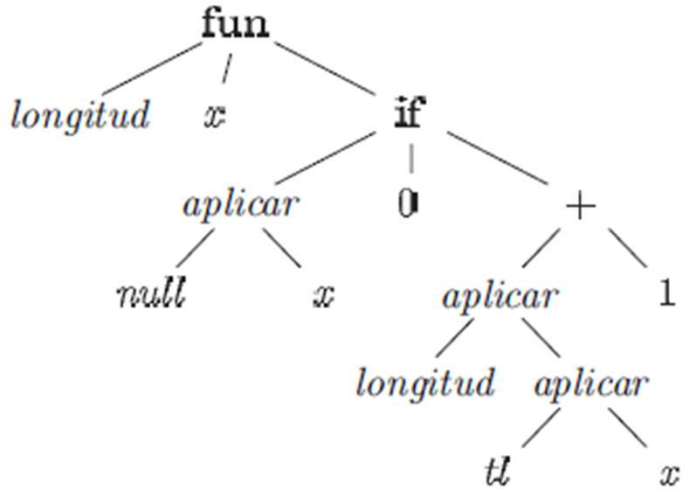


Figura 6.29: Árbol de sintaxis abstracto para la definición de la función de la figura 6.28

RI Arbóreas: Árbol de sintaxis abstracta (Aho, 2008)

- Los nodos en un árbol sintáctico representan construcciones en el programa fuente; los hijos de un nodo representan los componentes significativos de una construcción. Un grafo acíclico dirigido (de aquí en adelante lo llamaremos *GDA*) para una expresión identifica a las *subexpresiones comunes* (subexpresiones que ocurren más de una vez) de la expresión

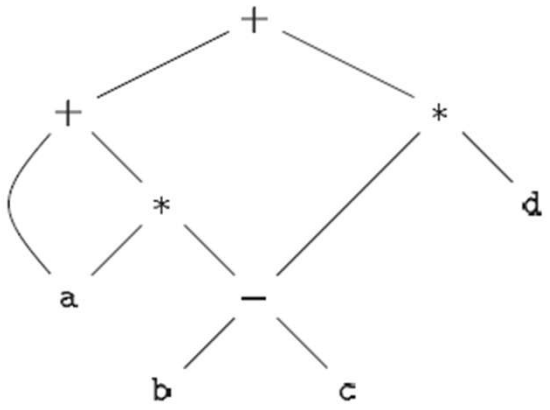


Figura 6.3: GDA para la expresión a + a * (b - c) + (b - c) * d

Práctica de AST en ANTLR (tomada del libro de ANTLR de Terence Parr)

```
grammar Expr;
options {      output=AST;
               ASTLabelType=CommonTree; // type of $stat.tree ref etc...
}

// START:stat
/** Match a series of stat rules and, for each one, print out the   tree stat returns, $stat.tree.  toStringTree() prints the tree   out in form: (root child1
... childN).  ANTLR's default tree    construction mechanism will build a list (flat tree) of the stat  result trees.  This tree will be the input to the tree
parser. */

prog:   ( stat {System.out.println(
          $stat.tree==null?"null":$stat.tree.toStringTree());} )+ ;

stat:   expr NEWLINE       -> expr
    |   ID '=' expr NEWLINE -> ^('=' ID expr)
    |   NEWLINE            ->
    ;

expr     :   multExpr (('+'^|'-'^) multExpr)*      ;
multExpr :   atom ('*'^ atom)*                     ;
atom     :   INT  |   ID     |   '('! expr ')'!    ;

// START:tokens
ID  :   ('a'..'z'|'A'..'Z')+ ;
INT :   '0'..'9'+ ;
NEWLINE:'\r'? '\n' ;
WS  :   (' '|'\t'|'\n'|'\r')+ {skip();} ;
// END:tokens
```

6.1 Notaciones (Olivares)

- Las **notaciones** sirven de base para expresar sentencias bien definidas.
- El uso más extendido de las notaciones sirve para expresar operaciones aritméticas.
- Las expresiones aritméticas se pueden expresar de tres formas distintas: **infija, prefija y postfija**.
- La diversidad de notaciones corresponde en que para algunos casos es más sencillo un tipo de notación.
- Las notaciones también dependen de cómo se **recorrerá el árbol** sintáctico, el cual puede ser en inorden, preorden o postorden; teniendo una relación de uno a uno con la notación de los operadores

6.1 Notaciones (Olivares)

- La notación infija es la más utilizada por los humanos porque es la más comprensible ya que ponen el operador entre los dos operandos.
- Por ejemplo **a+b-5**.
- No existe una estructura simple para representar este tipo de notación en la computadora por esta razón se utilizan otras notaciones.

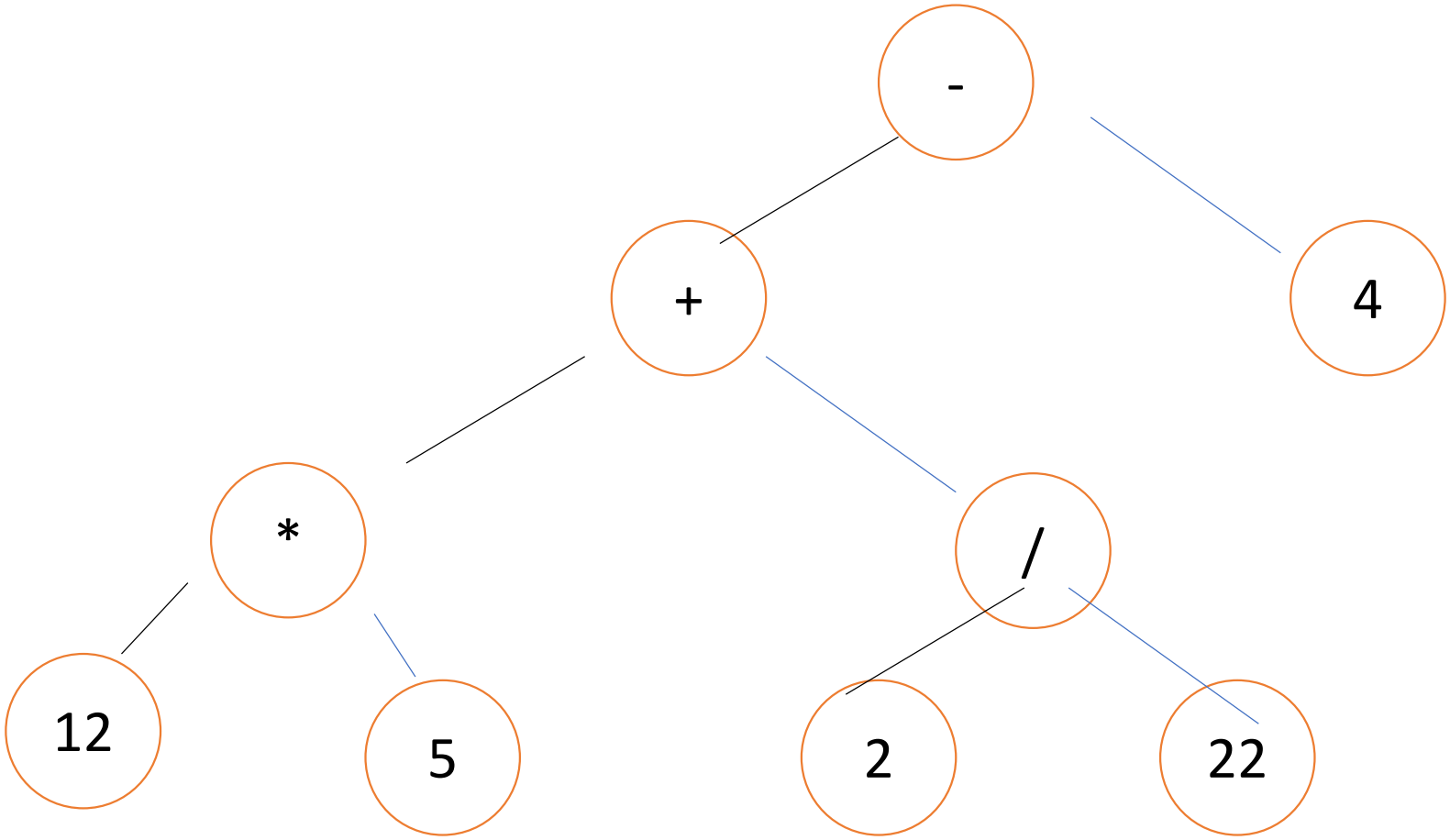
6.1 Notaciones (Olivares)

- La notación **prefija** pone el operador primero que los dos operandos, por lo que la expresión anterior queda:
- **- + a b 5 .**
- Esto se representa con una estructura del tipo FIFO (First In First Out) o cola.
- Las estructuras FIFO son ampliamente utilizadas pero tienen problemas con el anidamiento aritmético.

6.1 Notaciones (Olivares)

- La notación postfija pone el operador al final de los dos operandos, por lo que la expresión queda:
- a b + 5 -
- La notación posftfija utiliza una estructura del tipo LIFO (Last In First Out) pila, la cual es la más utilizada para la implementación.
- También llamada Notación Polaca Inversa (RPN), véase Ribadas página 6, algoritmo de interpretación

6.1 Pasar una expresión de un árbol a las notaciones



Representaciones intermedias lineales

- Revisar material de Ribadas
  - Convertir expresiones aritméticas infijas a postfijas (página 6)
  - Ejecutar algoritmo de pila para computar expresiones aritméticas (página 6)
- Código de 3 direcciones (página 7)
- Tercetos y cuartetos (página 8 y 9)