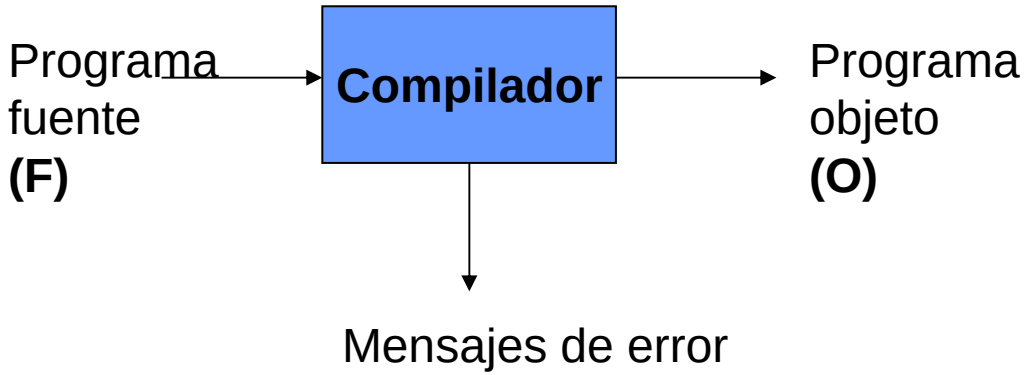
Compiladores e intérpretes
Compilación / Fases del compilador

Primer semestre 2008

Vistazo general

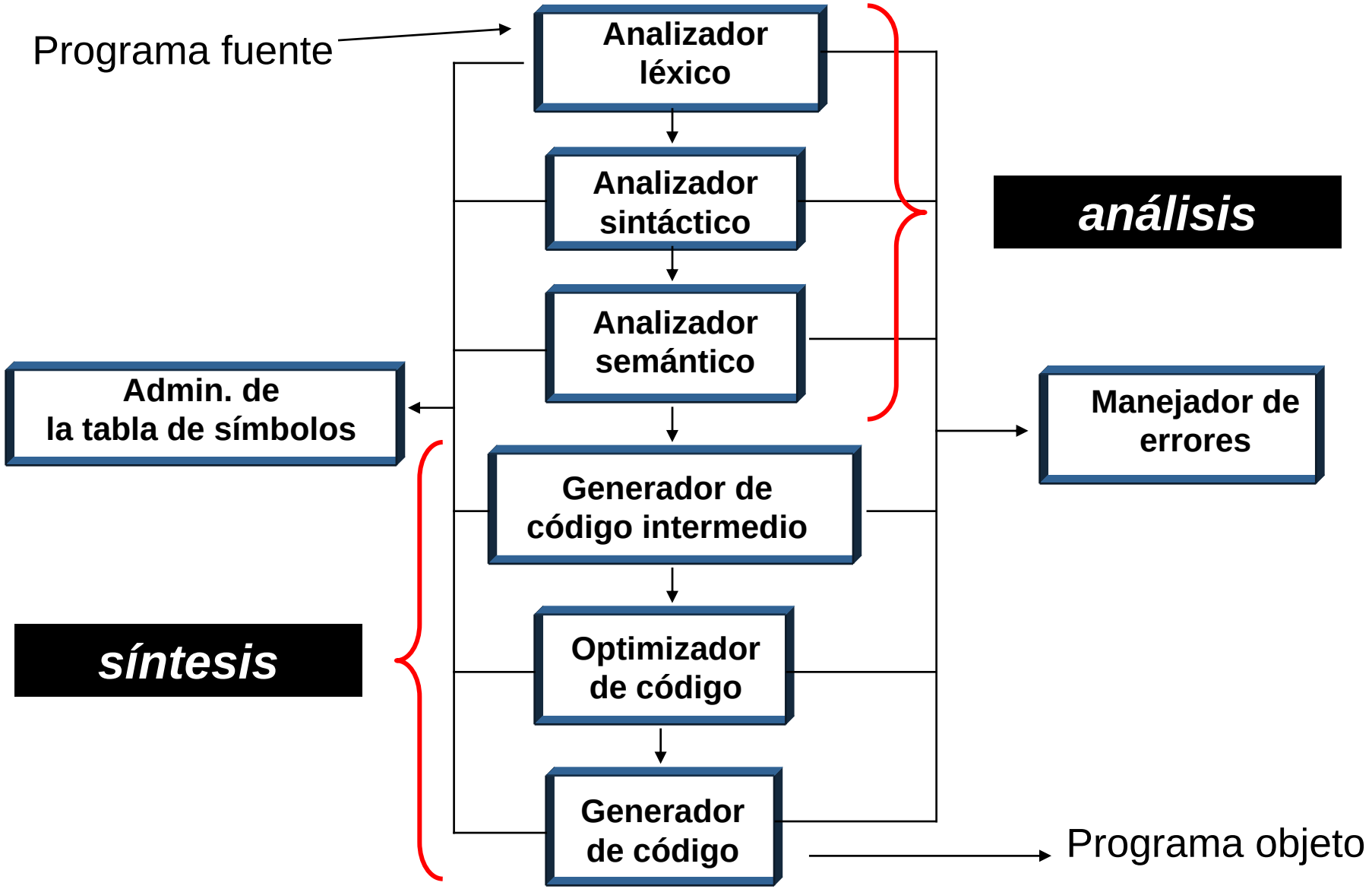
- Traducir un programa fuente (en lenguaje F) a un programa objeto equivalente (en lenguaje O)



Modelo de compilación

- Reducir complejidad
- Independencia de fuente o destino
  - Compilador ‘conectable’
- RI: con suficiente información
  - Estructura arbórea (árbol sintáctico), o
  - Formato como ensamblador (código de tres direcciones)

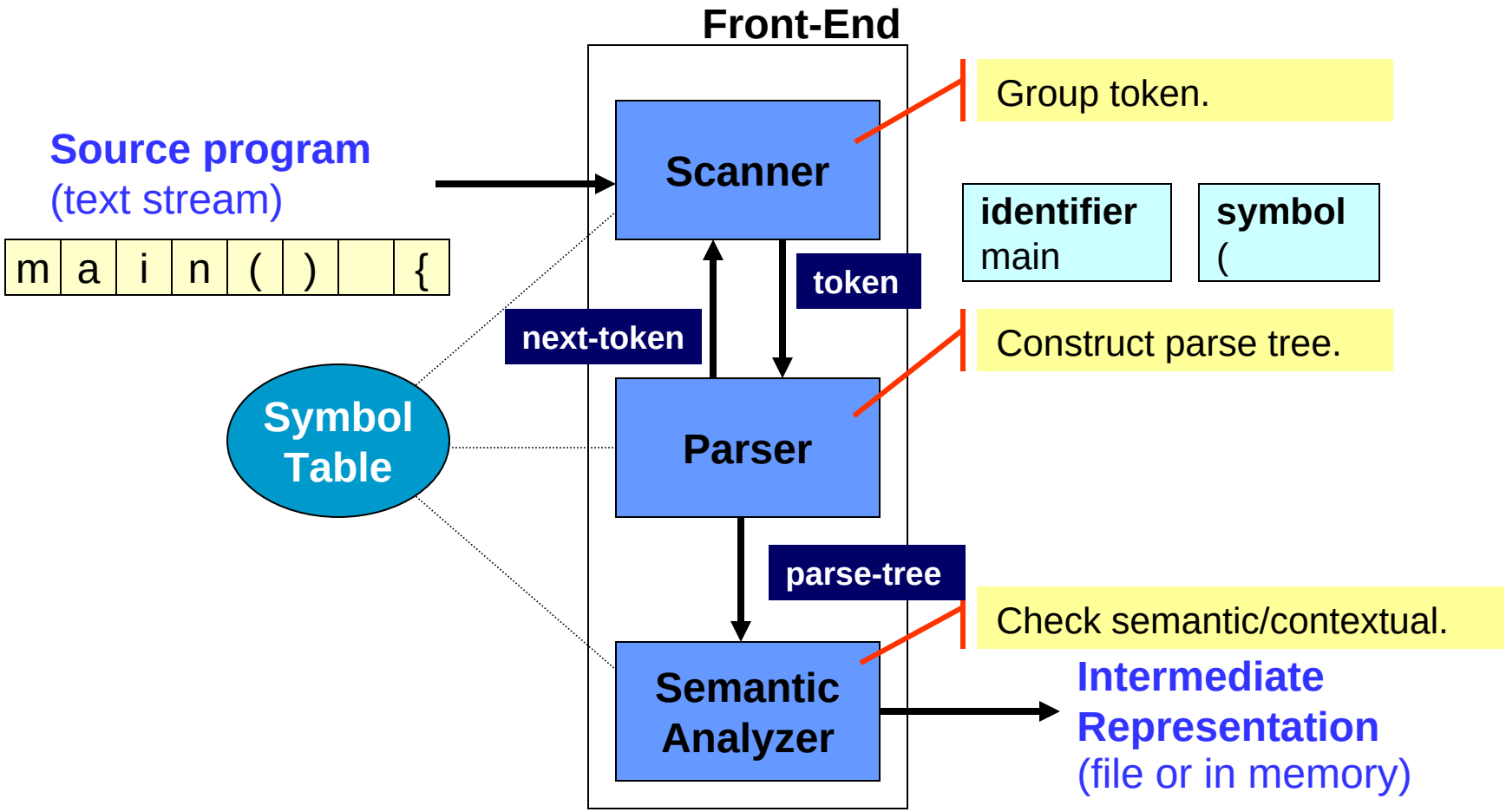
Fases en el proceso de compilación



Fases de análisis

- Análisis léxico
  - Agrupar caracteres de entrada en signos léxicos (“tokens”)
- Análisis sintáctico
  - Ver si el código fuente está bien formado
- Análisis contextual/semántico
  - Asegurar que programa tiene sentido (uso de identificadores y tipos)

Fases de análisis



Análisis léxico

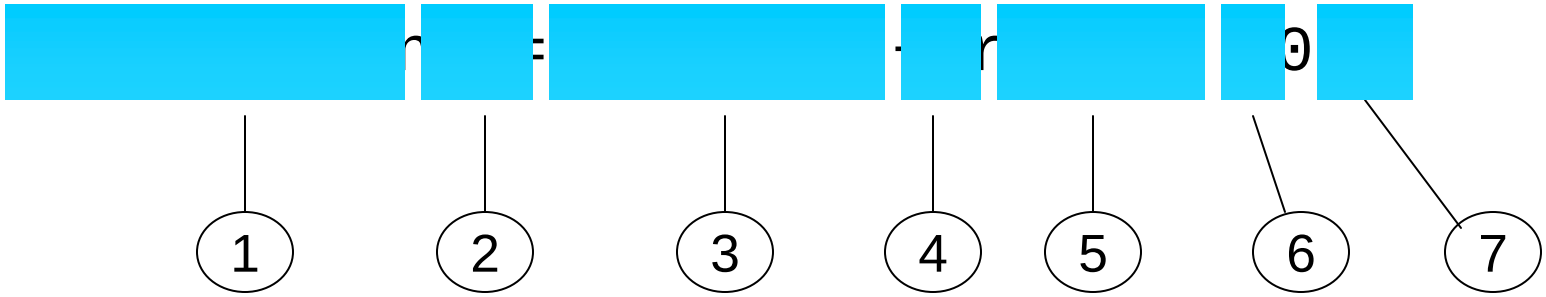
- Agrupa *componentes léxicos* de izquierda a derecha
- componentes léxicos son secuencias de caracteres con significado colectivo.
  - –un identificador
  - –una palabra clave (if, while, do, ...),
  - –un carácter de puntuación, un operador de varios caracteres como :=, <=

Unidades léxicas

- A la secuencia de caracteres que forma un componente léxico se le denomina *lexema* del componente

- A ciertos lexemas se les agrega un valor léxico y se incluyen en la tabla de símbolos (opcional)

Ejemplo de análisis léxico



1. **identifier** position
2. **assignment symbol** :=
3. **identifier** initial
4. **plus symbol** +
5. **identifier** rate
6. **muliplication symbol** *
7. **integer-literal** 60

Ejemplo con tabla de símbolos

Programa fuente:

Posición := Inicial + velocidad * 60;

Componentes léxicos

id1 := id2 + id3 * 60

Tabla de Símbolos

1	posición	. . . . .
2	inicial	. . . . .
3	velocidad	. . . . .

Otras funciones del analizador léxico

- Eliminar blancos (espacios, tabuladores, fines de línea) y comentarios
- Mantener ‘coordenadas’ del texto fuente (para listados, mensajes de error, etc.)

Analizador sintáctico

- Reconocer forma: determinar si estructura sintáctica está entre las permitidas
- Representar explícitamente la estructura (árbol de sintaxis abstracta)

Análisis sintáctico

- Agrupa los *componentes léxicos* en *frases gramaticales* que permiten luego sintetizar la salida.
  - Por lo general se representan mediante un árbol de análisis sintáctico.

Análisis sintáctico

- La estructura jerárquica se expresa mediante reglas recursivas
  - Cualquier identificador es una expresión
  - Cualquier número es una expresión
  - Si expresión1 y expresión2 son expresiones, entonces también los son:
    - expresión1 + expresión2
    - expresión1 * expresión2
    - (expresión1)

Ejemplo: sintaxis de Pascal

program ::= PROGRAM identifier (identifier ***more_identifiers***) ; ***block*** .

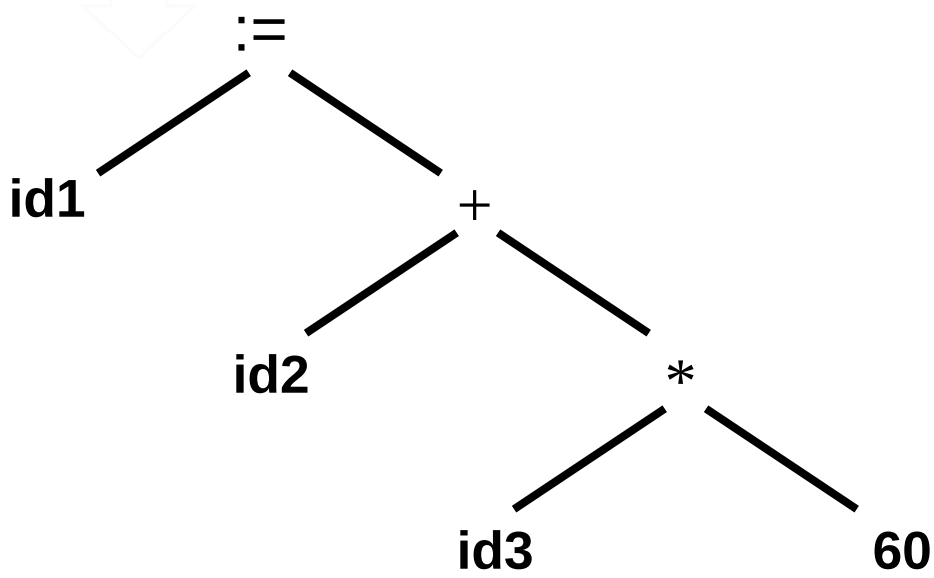
block ::= ***labels constants types variables subroutines***
BEGIN ***statement more_statements*** END

more_identifiers **::=** , identifier ***more_identifiers*** **|** ε

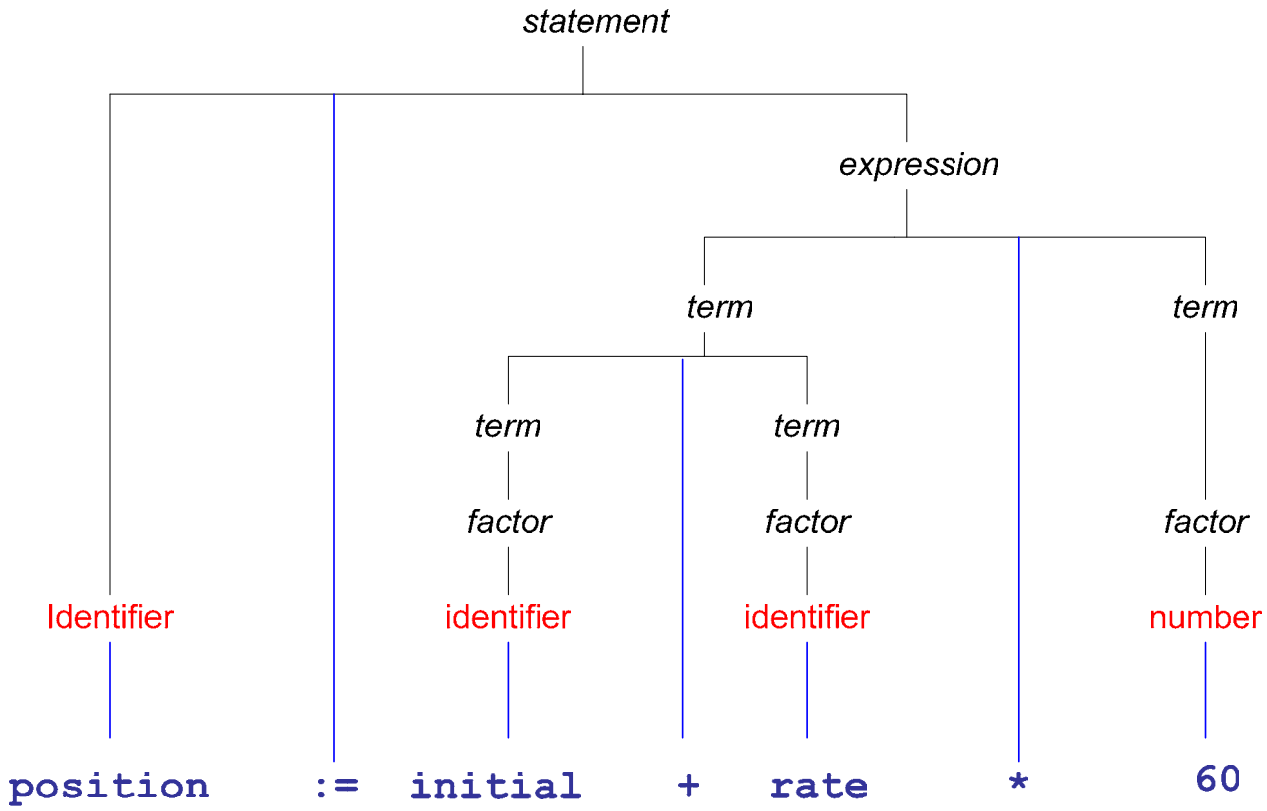
Ejemplo

Componentes léxicos id1 := id2 + id3 * 60

Árbol sintáctico



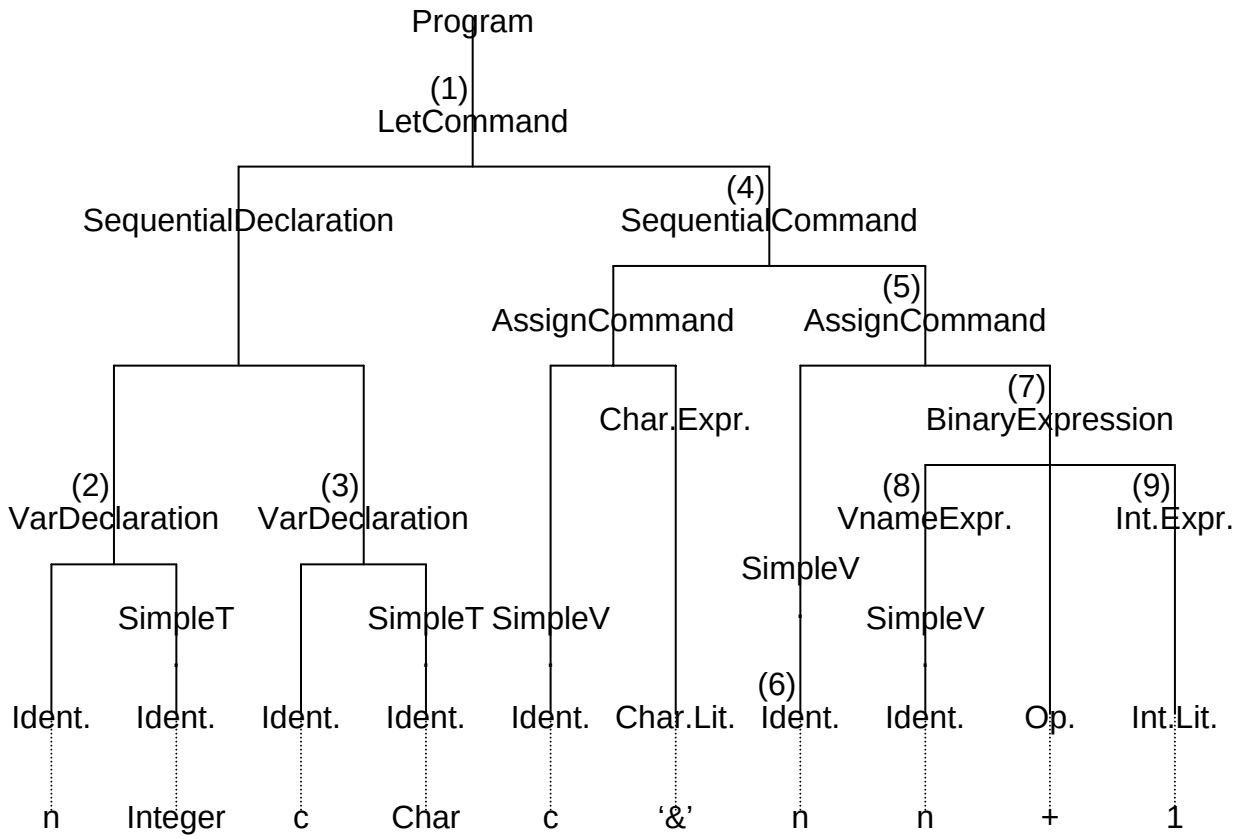
Ejemplo: árbol sintáctico

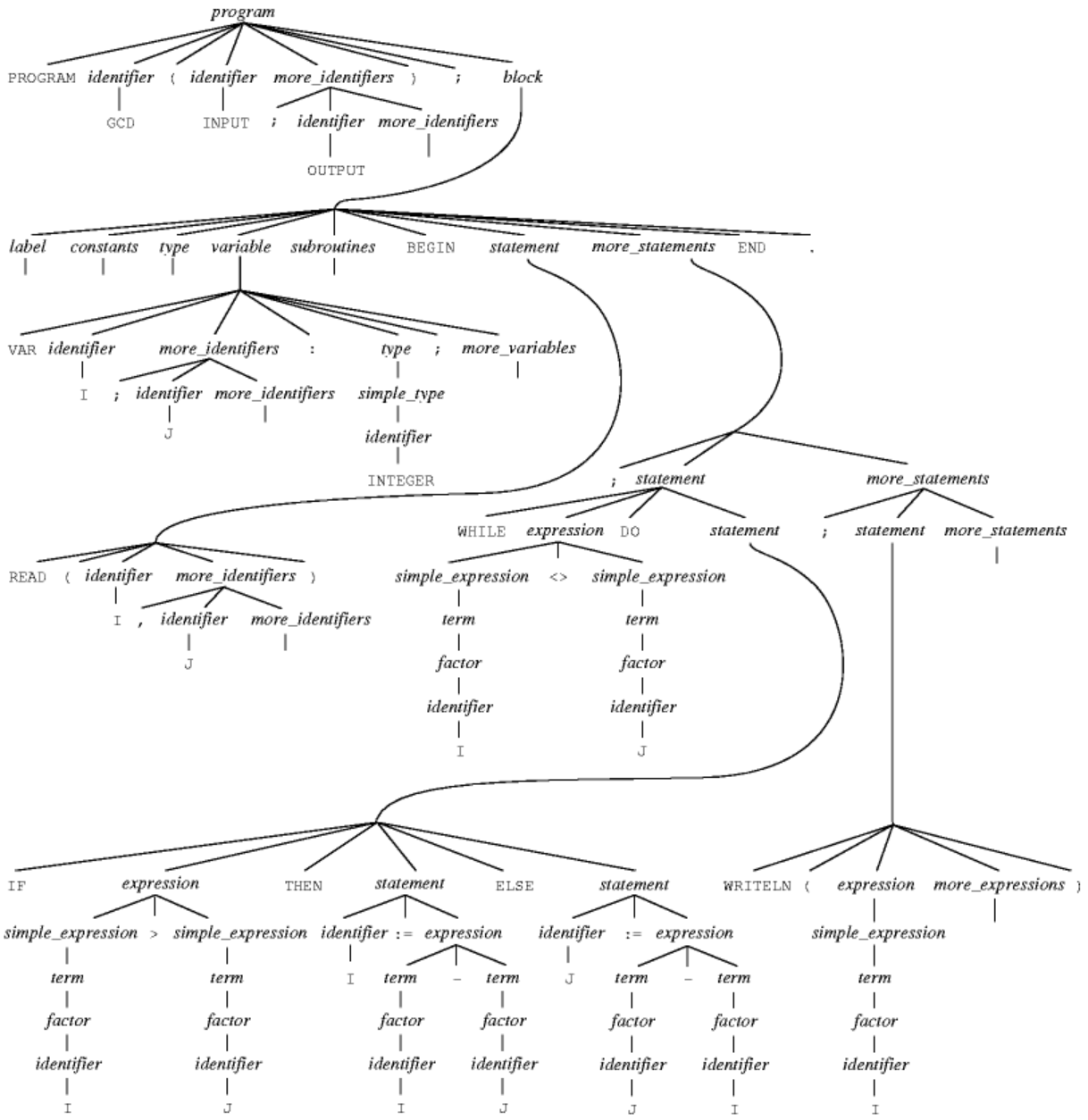


Programa en Δ

```
! Programa ilustrativo inútil
let
  var n : Integer;
  var c : Char;
in
  begin
    c  :=  ‘&’;
    n  := n+1
  end
```

Árbol de sintaxis abstracta




program
PROGRAM identifier (identifier more_identifiers) ; block
GCD
INPUT ; identifier more_identifiers
OUTPUT
label constants type variable subroutines BEGIN statement more_statements END .
VAR identifier more_identifiers : type ; more_variables
I ; identifier more_identifiers
J
simple_type
identifier
INTEGER
; statement
more_statements
READ (identifier more_identifiers)
I , identifier more_identifiers
J
WHILE expression DO statement
; statement more_statements
simple_expression <> simple_expression
term
factor
identifier
I
term
factor
identifier
J
IF expression THEN statement ELSE statement
WRITELN (expression more_expressions)
simple_expression > simple_expression
identifier := expression
identifier := expression
simple_expression
term
factor
identifier
I
term
factor
identifier
J
I term - term
J term - term
term
factor
identifier
I
J
I
J
I

Análisis contextual (semántico)

- Análisis estático
- Busca en el fuente errores de contexto:
  - Variables declaradas antes de usarlas
  - Uso de identificadores: reglas de alcance
  - Reglas de tipos
  - Llamadas a procedimientos/funciones con número correcto de argumentos
  - …

Análisis contextual (semántico)

- Reúne la información sobre los tipos para la fase posterior de generación de código
  - permite verificar si cada operador tiene los operandos permitidos por la especificación del lenguaje.
  - Algunos lenguajes permiten coerciones sobre tipos

Ejemplo

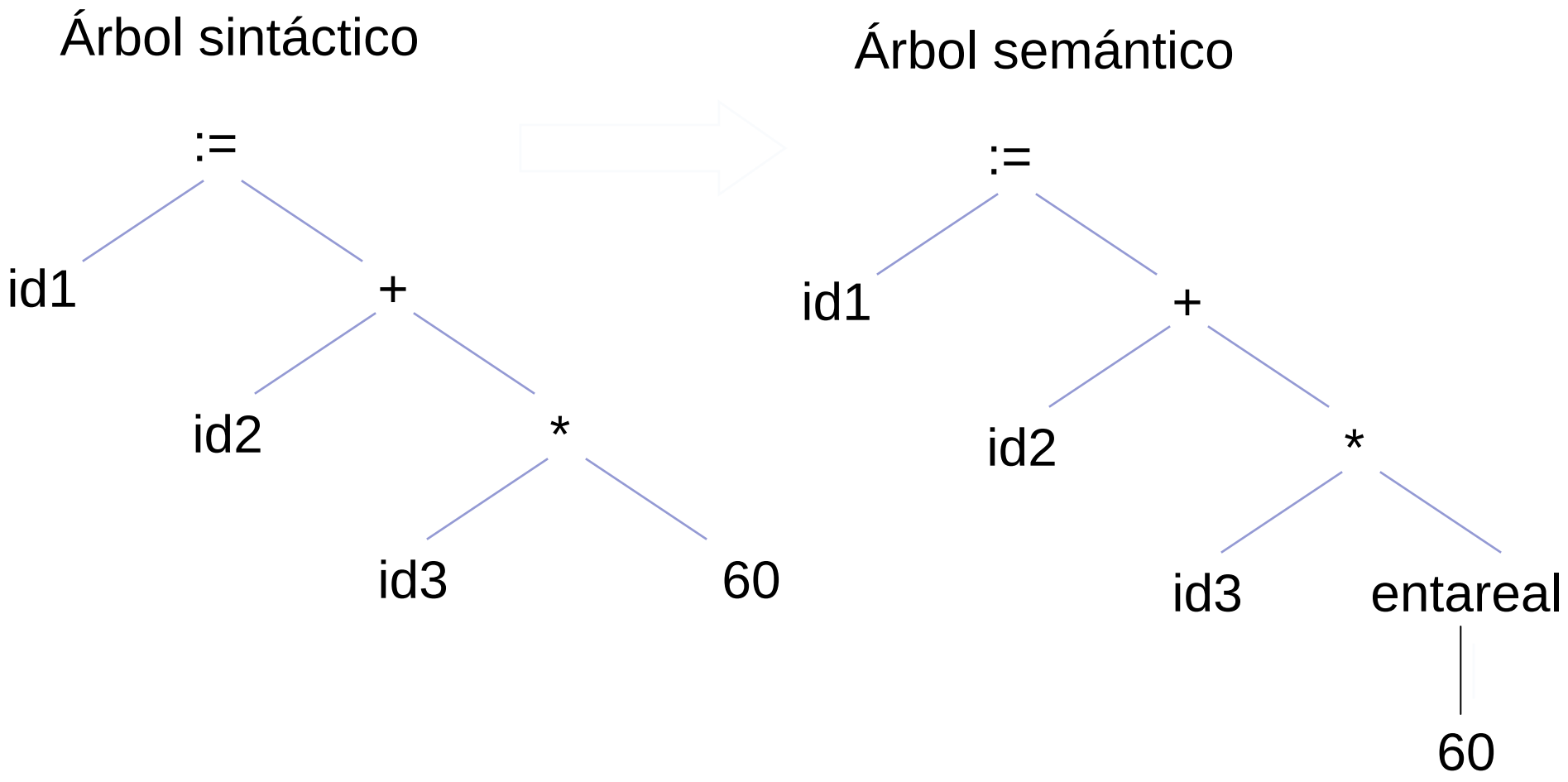

Árbol sintáctico
:=
id1
+
id2
*
id3
60
Árbol semántico
:=
id1
+
id2
*
id3
entareal
60

Tabla de símbolos (diccionario, tabla de identificación)

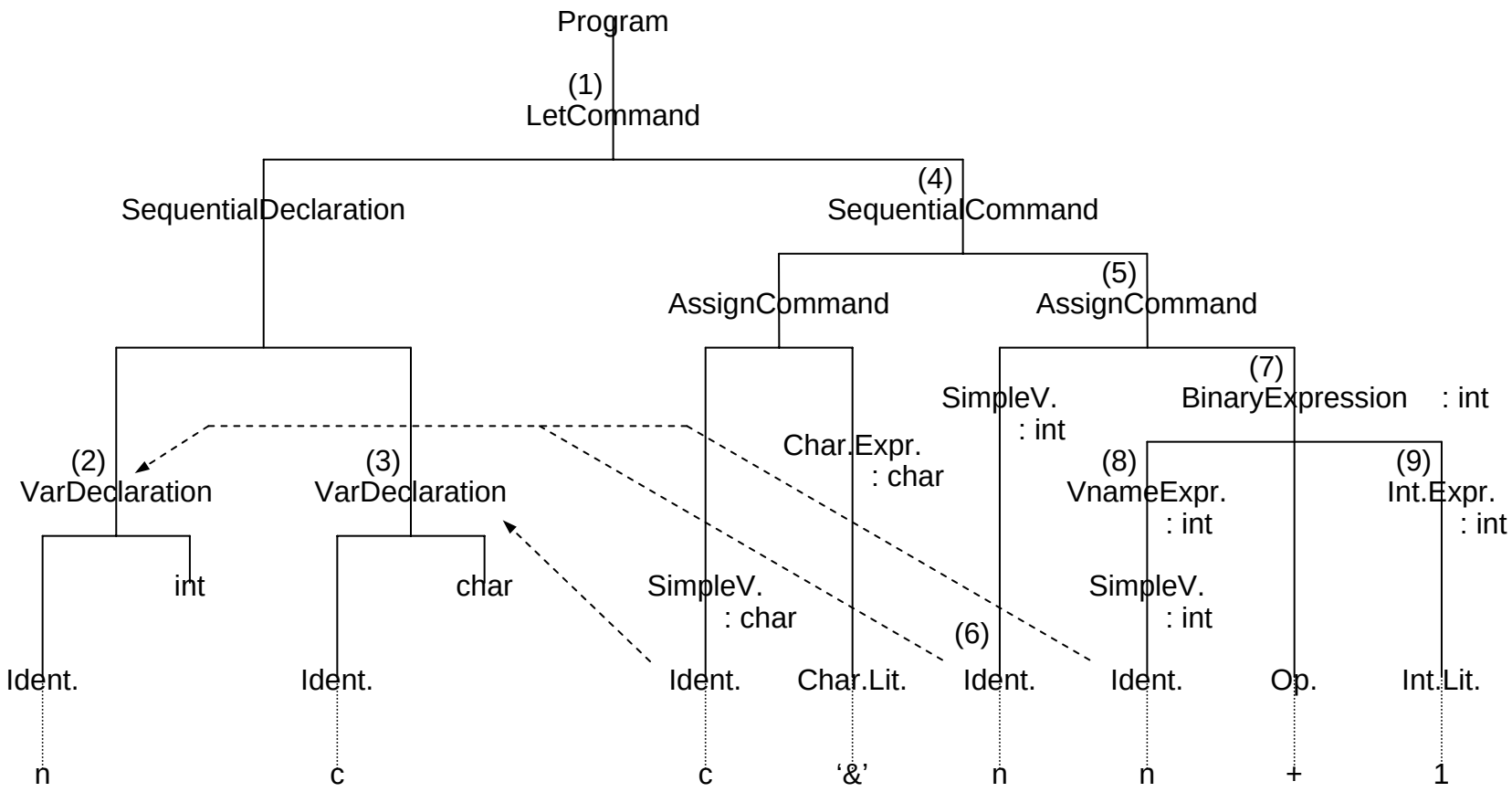
- El análisis contextual construye una *tabla de símbolos* que registra la definición de cada identificador
  - Las variables no declaradas y los errores de tipo pueden ser detectados en esta etapa

Nombre	Tipo	Dirección
x	Integer	200
y	Integer	204

Árbol sintáctico decorado

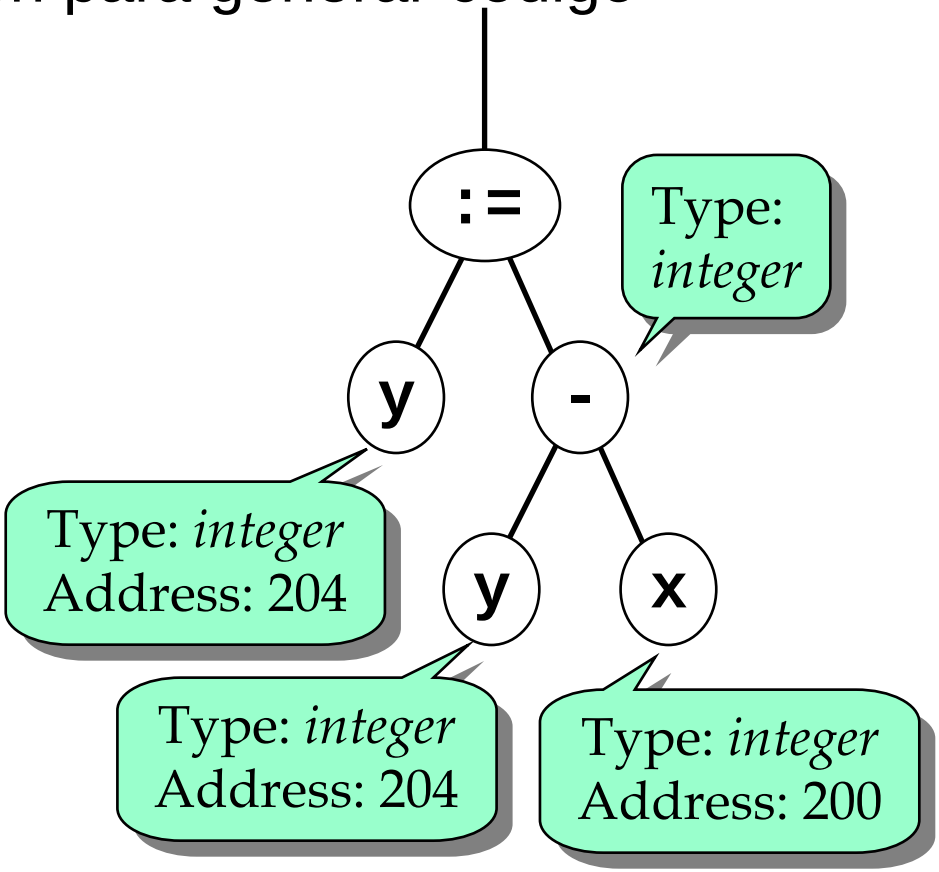
- Enlazar cada identificador a su declaración
- Decorar cada expresión con su tipo

Ejemplo


Program
(1)
LetCommand
SequentialDeclaration
(4)
SequentialCommand
AssignCommand
(5)
AssignCommand
(7)
SimpleV.
: int
BinaryExpression
: int
(2)
VarDeclaration
(3)
VarDeclaration
Char.Expr.
: char
(8)
VnameExpr.
: int
(9)
Int.Expr.
: int
int
char
SimpleV.
: char
SimpleV.
: int
(6)
Ident.
Ident.
Ident.
Char.Lit.
Ident.
Ident.
Op.
Int.Lit.
n
c
c
‘&’
n
n
+
1

Anotación del árbol

- El analizador contextual pasa información semántica mediante anotaciones en el árbol de sintaxis abstracta
- Se continúan añadiendo anotaciones, hasta tener suficiente información para generar código

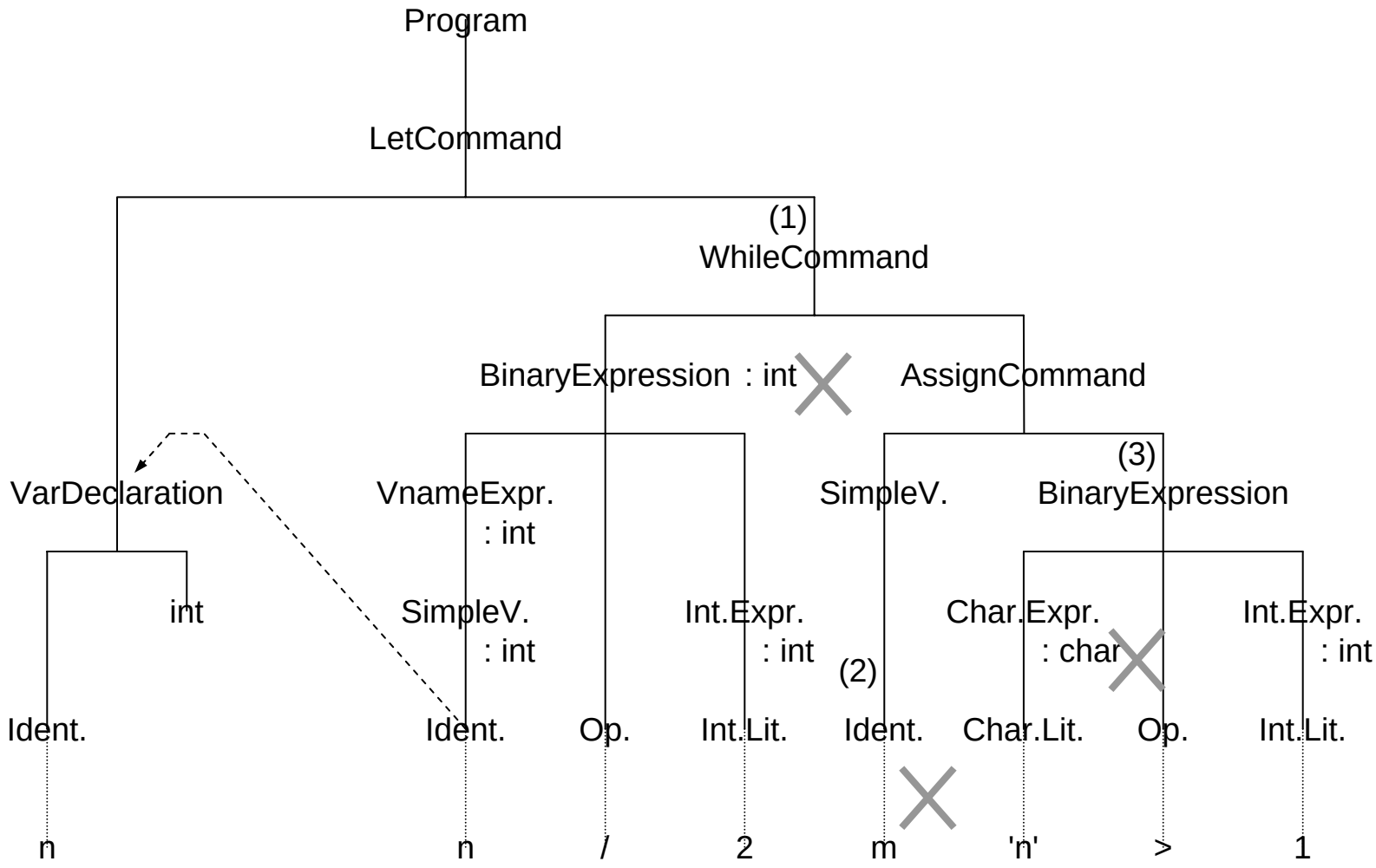


Programa con errores de contexto

```
let
  var n : Integer
in
  while n/2 do
    m := ‘n’ > 1
```

- *Encuentre los errores*

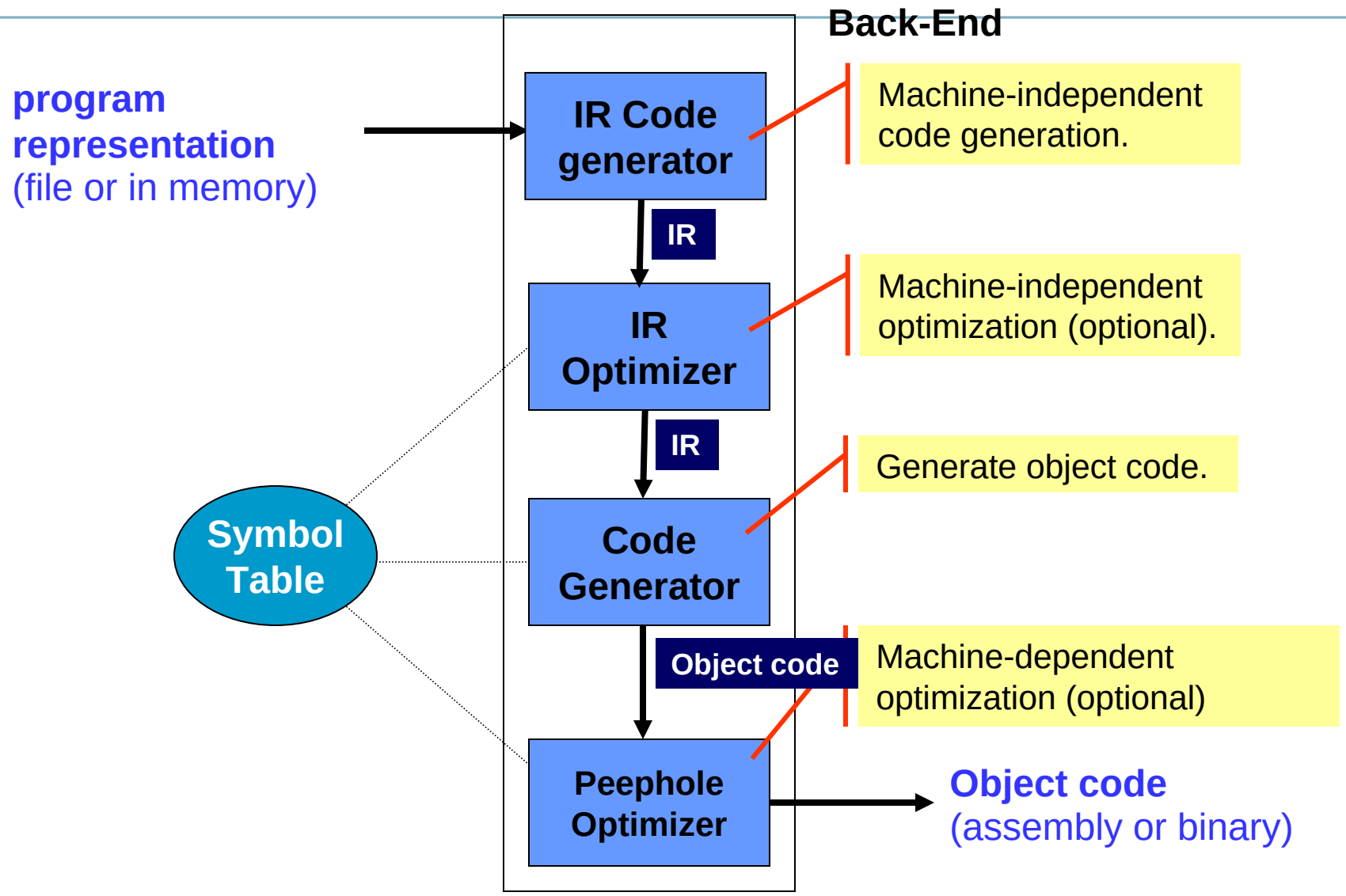
Errores reflejados en el árbol



Fases de síntesis

- Generación de código intermedio
  - Cambio a representación de bajo nivel
  - Preservar significados
- Optimización de código
  - Mejora de código intermedio, independiente de la máquina
  - Mejora de código objeto: dependiente de la máquina final
- Generación de código
  - Generar código objeto
  - Asignar ubicaciones de memoria o registros
  - Seleccionar instrucciones

Fases de síntesis



Generación de código

- Después de los análisis sintáctico y contextual sabemos que el programa está bien formado
- Ahora debemos traducir al código objeto

Generación de código intermedio

- Representación intermedia del fuente
- Debe ser fácil de producir y entender
- Existen diversos métodos
  - Tres direcciones: A lo sumo un operador además de la asignación, empleando elementos temporales para almacenar cálculos
  - Máquina de pila: facilita traducción de lenguajes con estructura de bloques (TAM para Δ, por ejemplo)

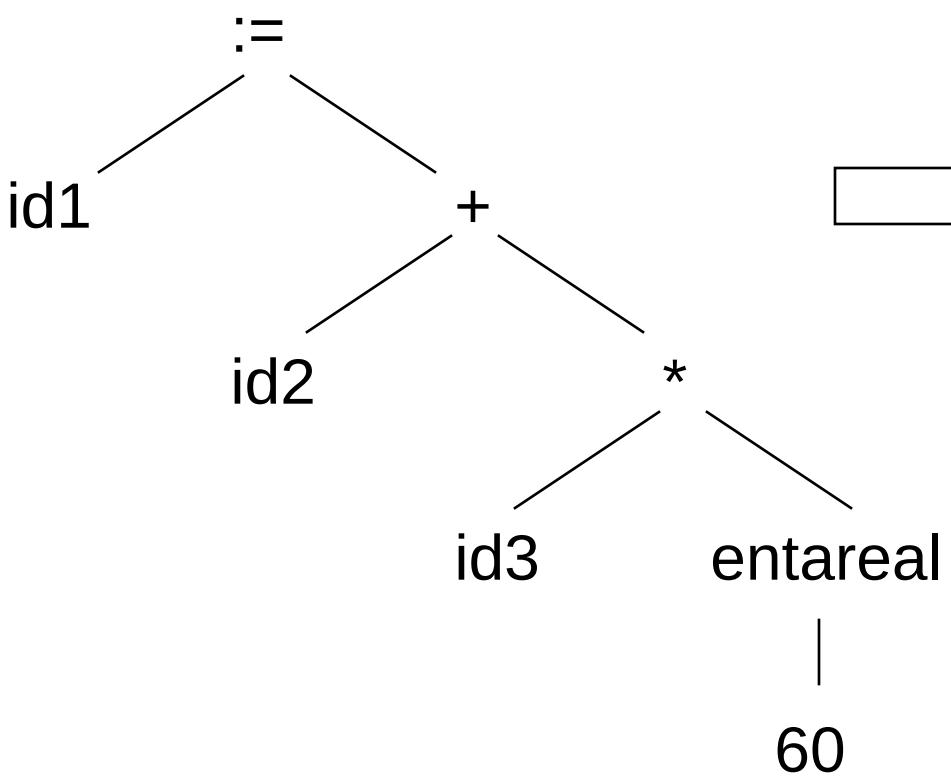
Ejemplo de representación intermedia
Código de tres direcciones

```
position := initial + rate * 60
```

```
tmp := rate * 60
tmp := initial + tmp
position := tmp
```

Ejemplo

Árbol semántico




```
temp1 :=  entareal(60)
temp2 :=  id3  *  temp1
temp3 :=  id2 + temp2
id1     :=  temp3
```

Ejemplo: de Δ a TAM

```
let
  var n : Integer;
  var c : Char;
in
  begin
    c  :=  ‘&’;
    n  := n+1
  end
```

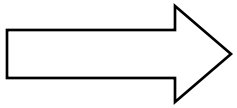
```
PUSH 2
LOADL 38
STORE 1 [SB]
LOAD 0 [SB]
LOADL 1
CALL add
STORE 0 [SB]
POP 2
HALT
```

Optimización de código

- Trata de mejorar el código intermedio producido en la fase anterior
  - Código de máquina más rápido de ejecutar, o
  - Código más corto (compacto), o
  - Variables más accesibles (en registros), o
  - Reordenamiento de instrucciones (en “pipeline”)

Ejemplo

```
temp1 :=  entareal(60)
temp2 :=  id3  *  temp1
temp3 :=  id2 + temp2
id1     :=  temp3
```



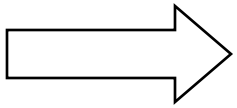
```
temp1 :=  id3 *  60.0
id1     :=  id2 +  temp1
```

Generación de código

- Fase final del compilador
- Es la generación de código objeto
- Código de máquina relocalizable o ensamblador
- Traducción a una secuencia de instrucciones de máquina que ejecuta la misma tarea
- Variables se traducen a registros o a celdas de memoria

Ejemplo de generación de código

```
temp1 :=  id3 *  60.0
id1   :=  id2 +  temp1
```



```
MOVF   id3, R1
MULF  #60.0, R1
MOVF   id2, R2
ADDF   R1, R2
MOVF   R2,  id1
```

```
    addiu   sp,sp,-32       # reserve room for local variables
    sw      ra,20(sp)       # save return address
    jal     getint          # read
    nop
    sw      v0,28(sp)       # store i
    jal     getint          # read
    nop
    sw      v0,24(sp)       # store j
    lw      t6,28(sp)       # load i
    lw      t7,24(sp)       # load j
    nop
    beq     t6,t7,D         # branch if i = j
    nop
A:  lw      t8,28(sp)       # load i
    lw      t9,24(sp)       # load j
    nop
    slt     at,t9,t8        # determine whether j < i
    beq     at,zero,B       # branch if not
    nop
    lw      t0,28(sp)       # load i
    lw      t1,24(sp)       # load j
    nop
    subu    t2,t0,t1        # t2 := i - j
    sw      t2,28(sp)       # store i
    b       C
    nop
B:  lw      t3,24(sp)       # load j
    lw      t4,28(sp)       # load i
    nop
    subu    t5,t3,t4        # t5 := j - i
    sw      t5,24(sp)       # store j
C:  lw      t6,28(sp)       # load i
    lw      t7,24(sp)       # load j
    nop
    bne     t6,t7,A         # branch if i <> j
    nop
D:  lw      a0,28(sp)       # load i
    jal     putint          # writeln
    nop
    move    v0,zero         # exit status for program
    b       E               # branch to E
    nop
    b       E               # branch to E
    nop
E:  lw      ra,20(sp)       # retrieve return address
    addiu   sp,sp,32        # deallocate space for local variables
    jr      ra              # return to operating system
    nop
```

Tabla de símbolos

- Estructura de datos
- Mantiene un registro por cada identificador, con sus atributos (nombre, tipo, alcance)
- Permite encontrar rápidamente el registro
- Identificador es frecuentemente introducido en el análisis léxico y sus atributos descriptivos en las otras fases

Detección e informe de errores

- Cada fase detecta y maneja errores
- Qué hacer cuando se encuentran errores en el fuente

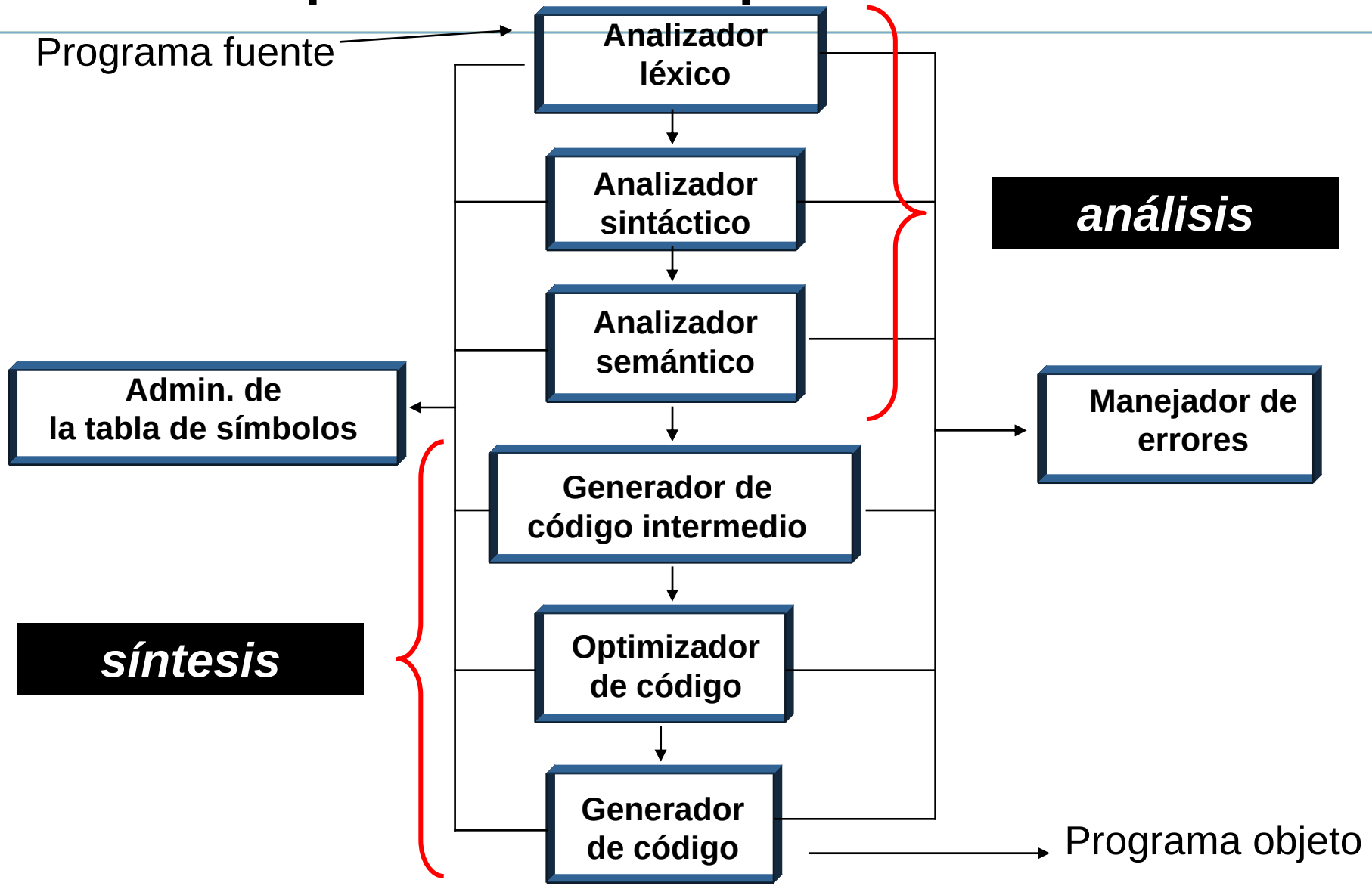
Agrupamiento de fases

- Con frecuencia se emplean dos o más fases
- Las iniciales tratan tareas dependientes del código (lenguaje) fuente e independientes del código (lenguaje) objeto
- Incluye las etapas de análisis, detección de errores, tabla de símbolos, código intermedio, cierto grado de optimización (del código intermedio)

Agrupamiento de fases

- Las fases finales están relacionadas con el código de máquina
- Su inicio depende del lenguaje intermedio
- Optimización y generación de código, manejo de errores y operaciones de tabla de símbolos

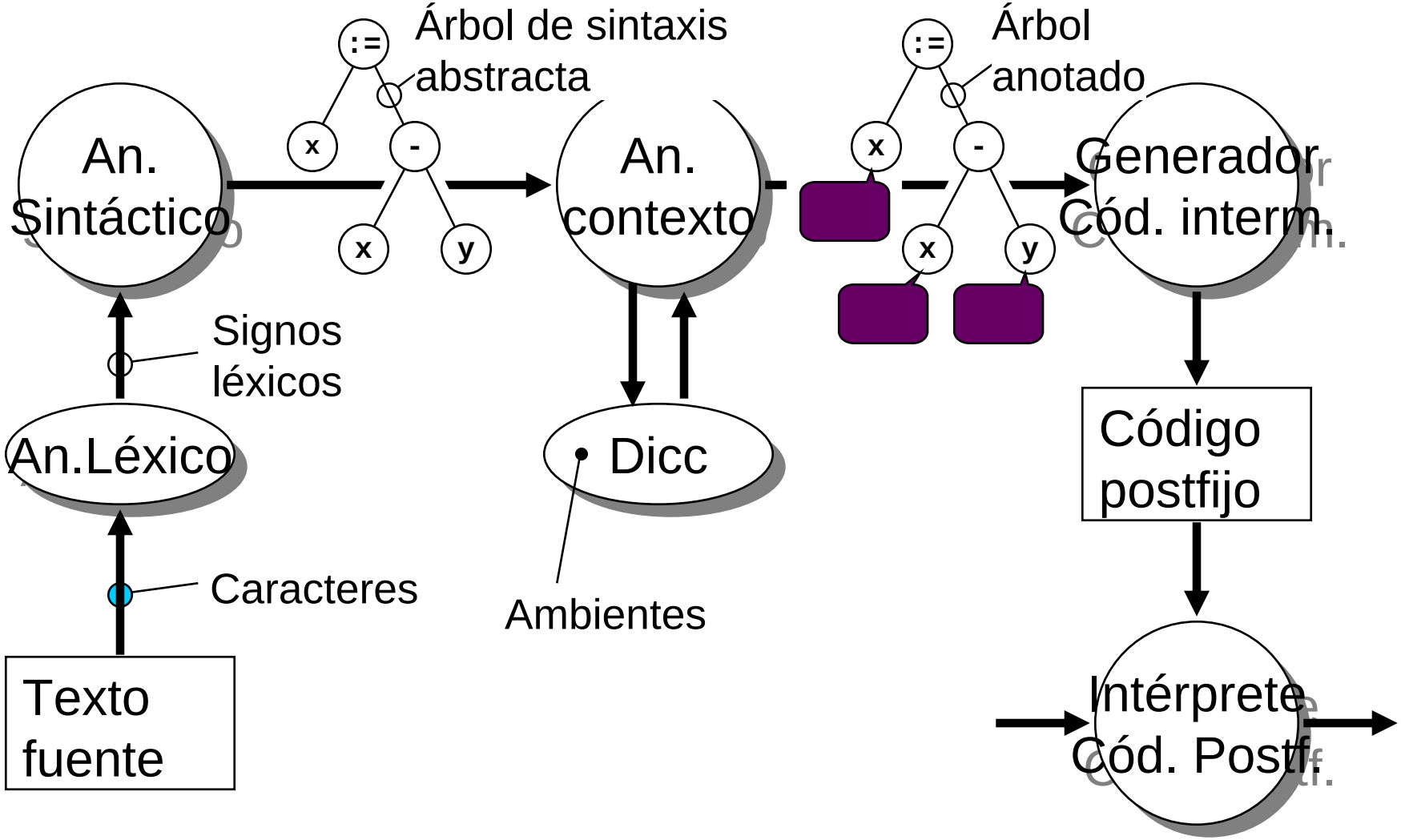
Fases en el proceso de compilación



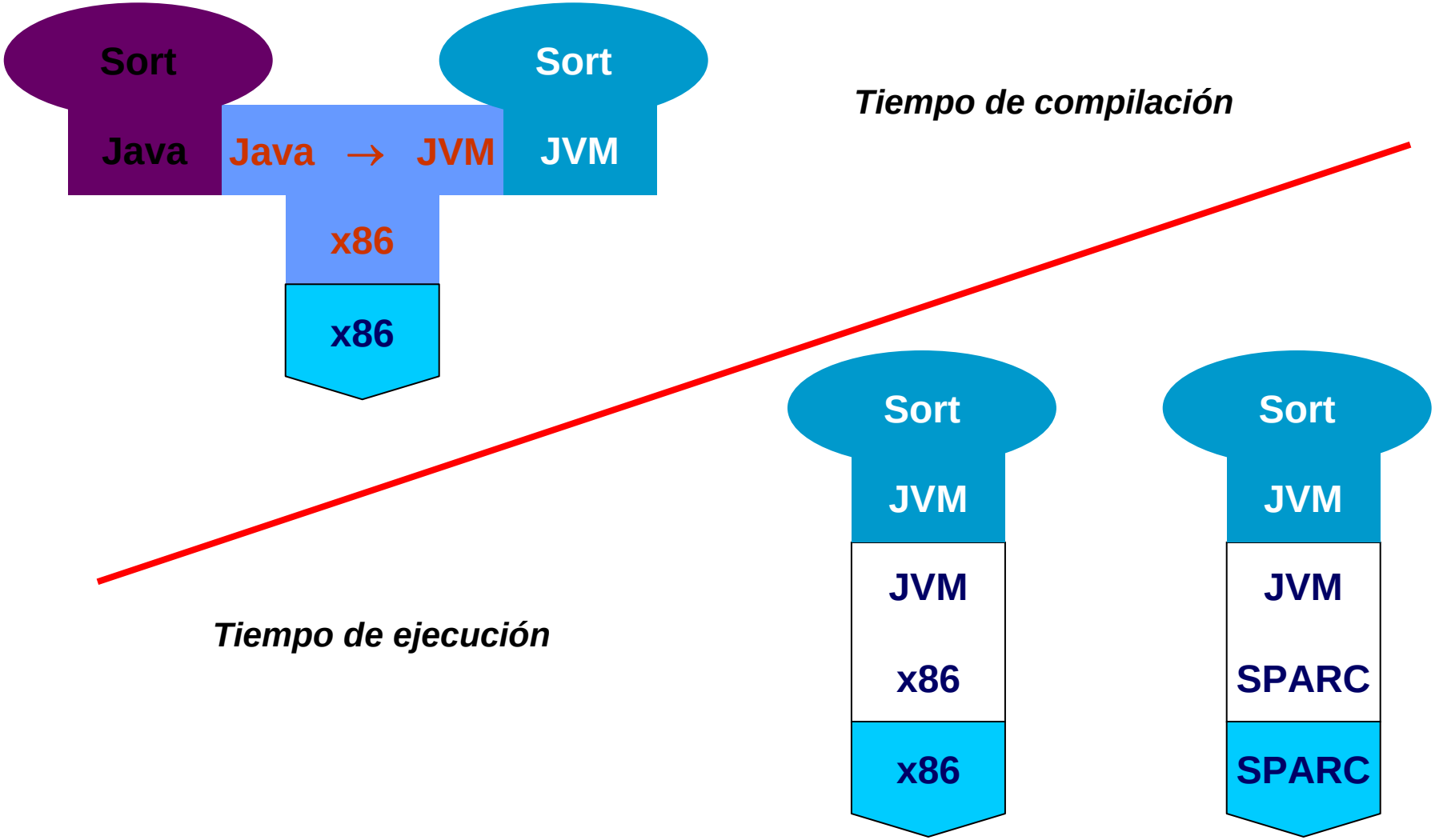
Algunas opciones

- Veamos algunas opciones de compilación o interpretación

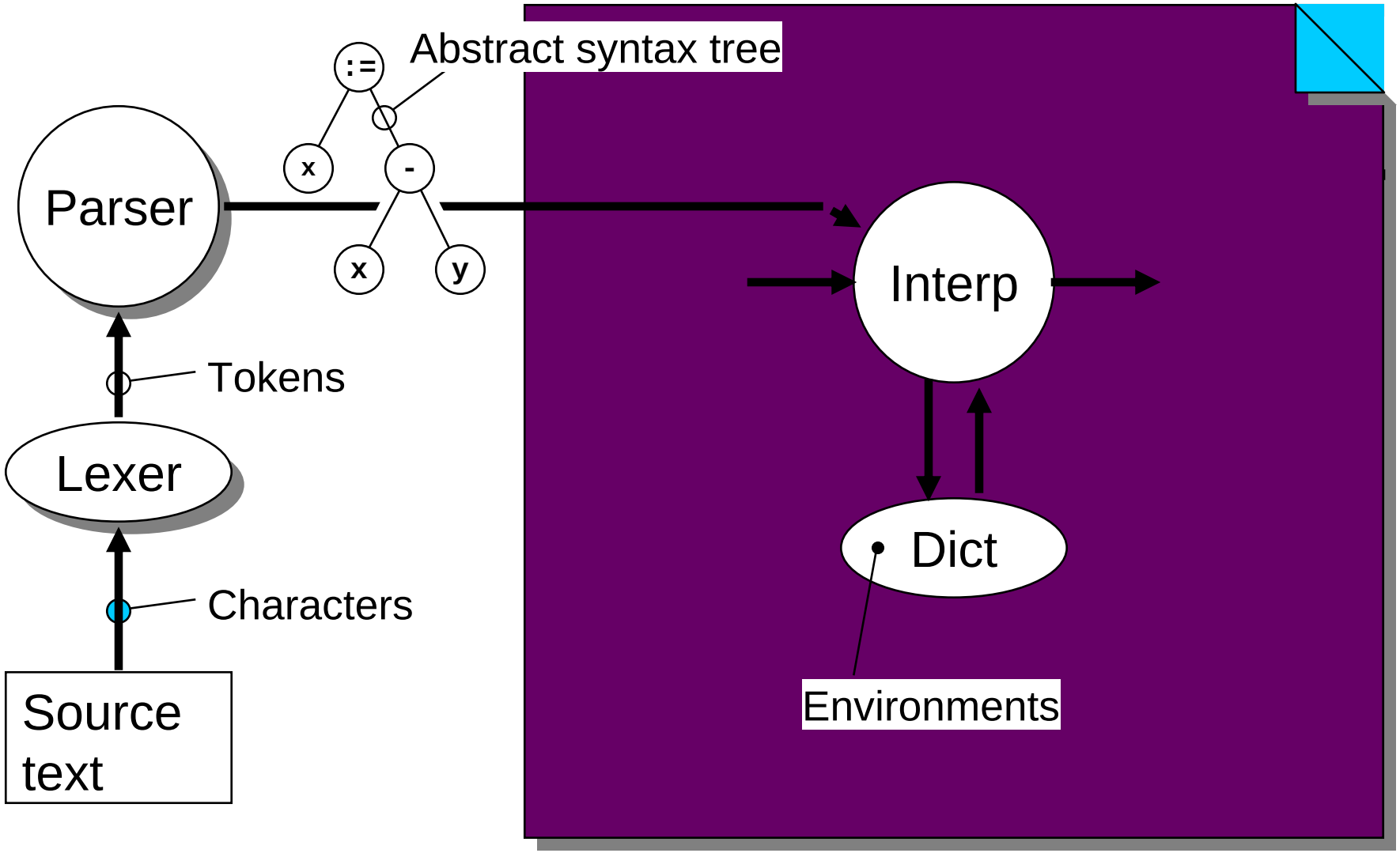
Vistazo general
Generar código intermedio



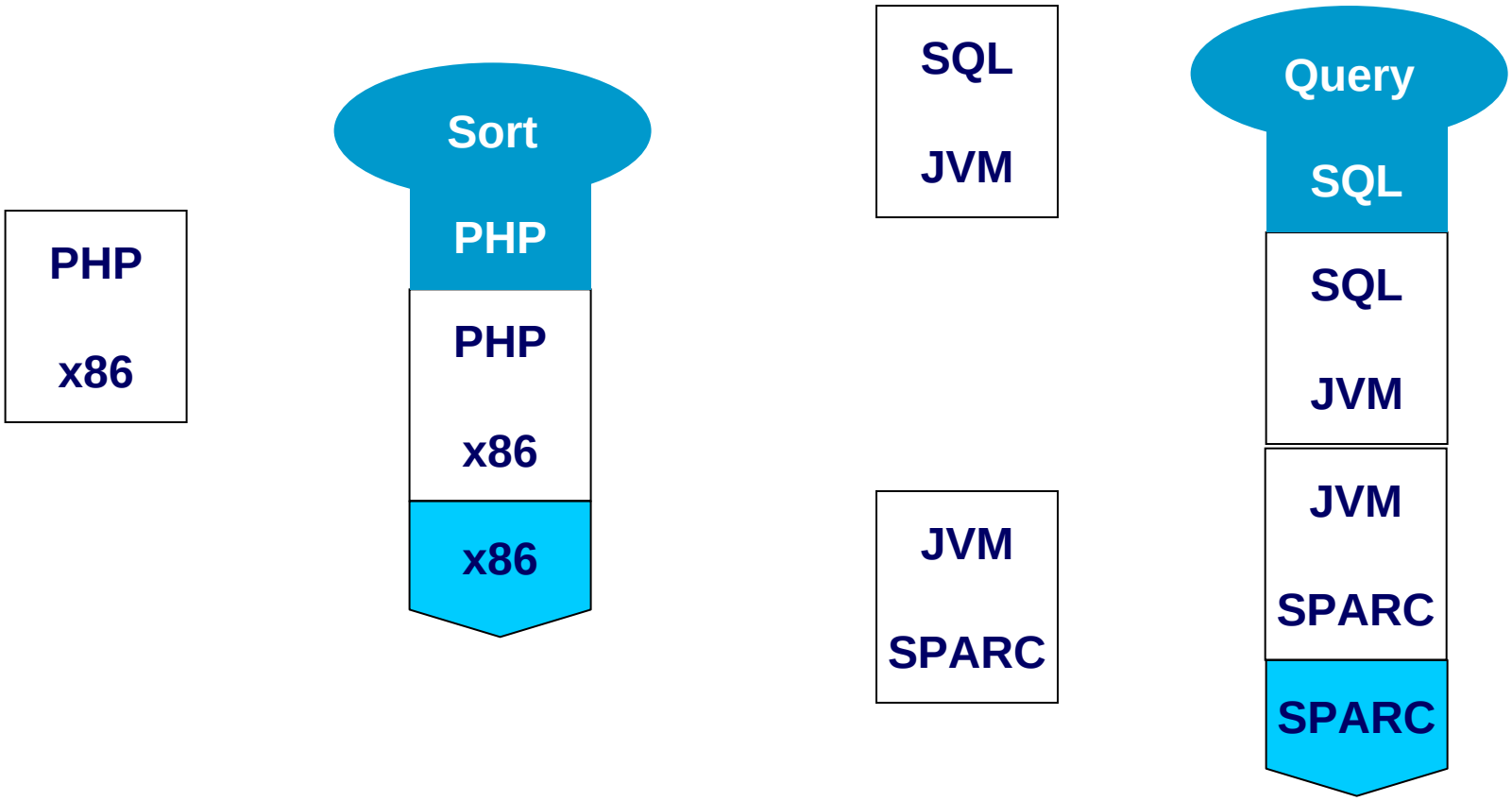
Compilar a código intermedio / Ejecutar código intermedio



Vistazo general
Intérprete

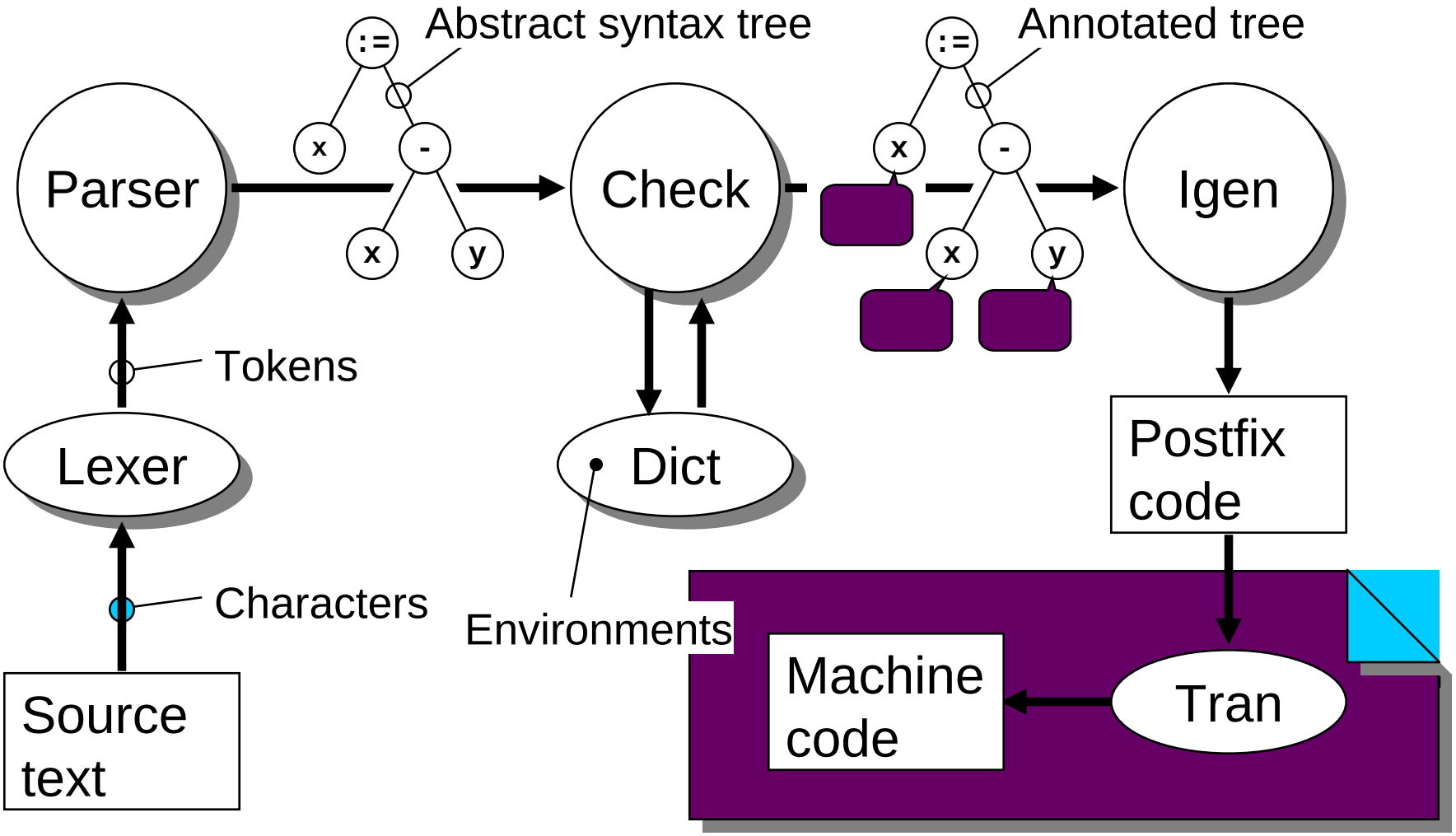


Interpretación

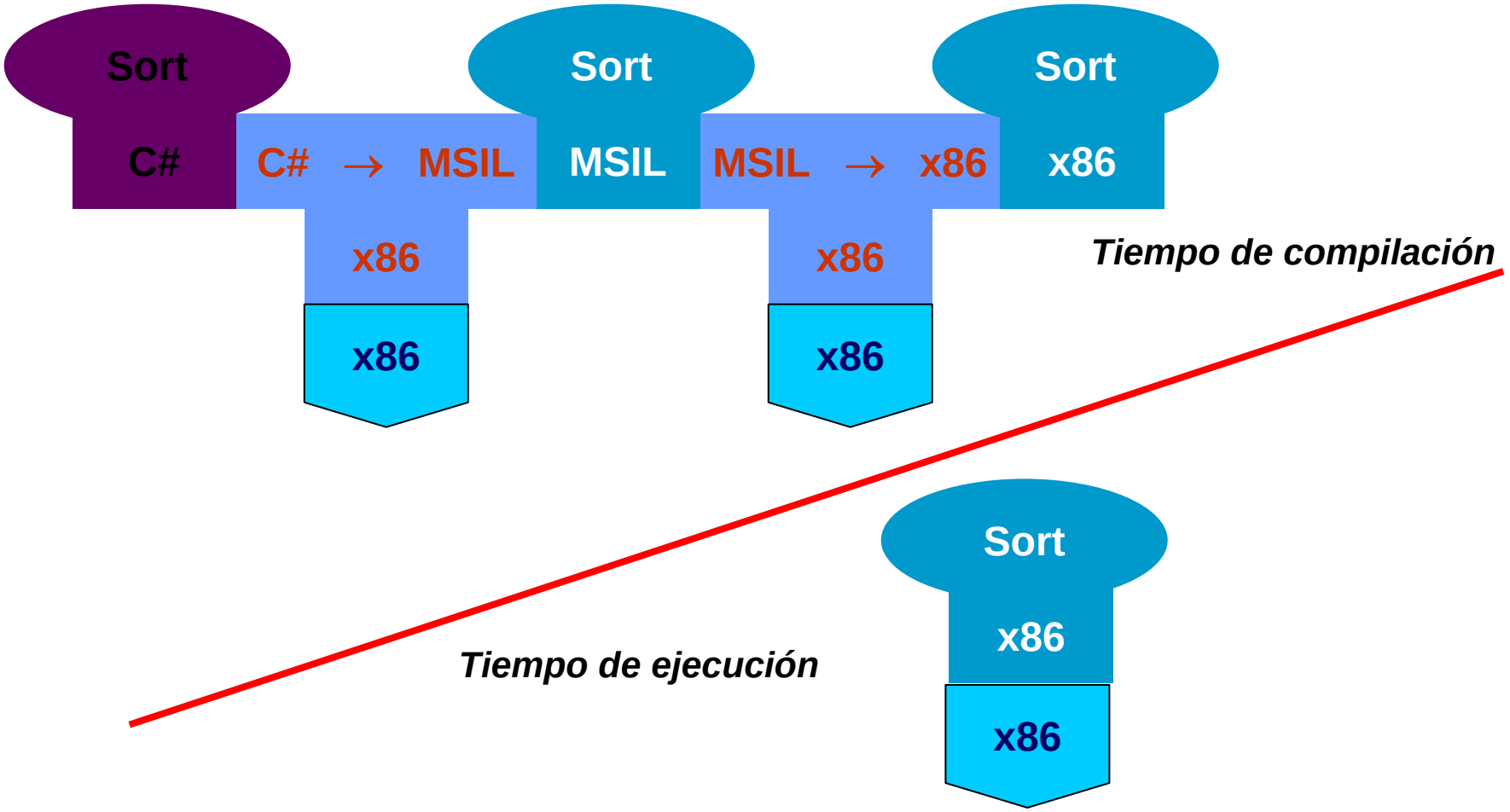


Tiempo de interpretación = tiempo de ejecución

Vistazo general
Generar código de máquina



Compilación a código de máquina



Pasadas

- Una “pasada” es un recorrido completo del programa fuente o de alguna representación interna del programa fuente
  - Texto fuente original
  - Secuencia de *tokens*
  - Árbol de sintaxis abstracta
  - Representación intermedia
  - Etc.

Compiladores de una pasada

- Generalmente gobernados por el analizador sintáctico
- No construyen explícitamente los árboles sintácticos
- El analizador léxico es invocado como una sub-rutina del analizador sintáctico
- Para cada estructura de frase significativa se invocan, en secuencia:
  - Análisis contextual (identificadores y tipos)
  - Generación de código
- Limitados en cuanto a optimización

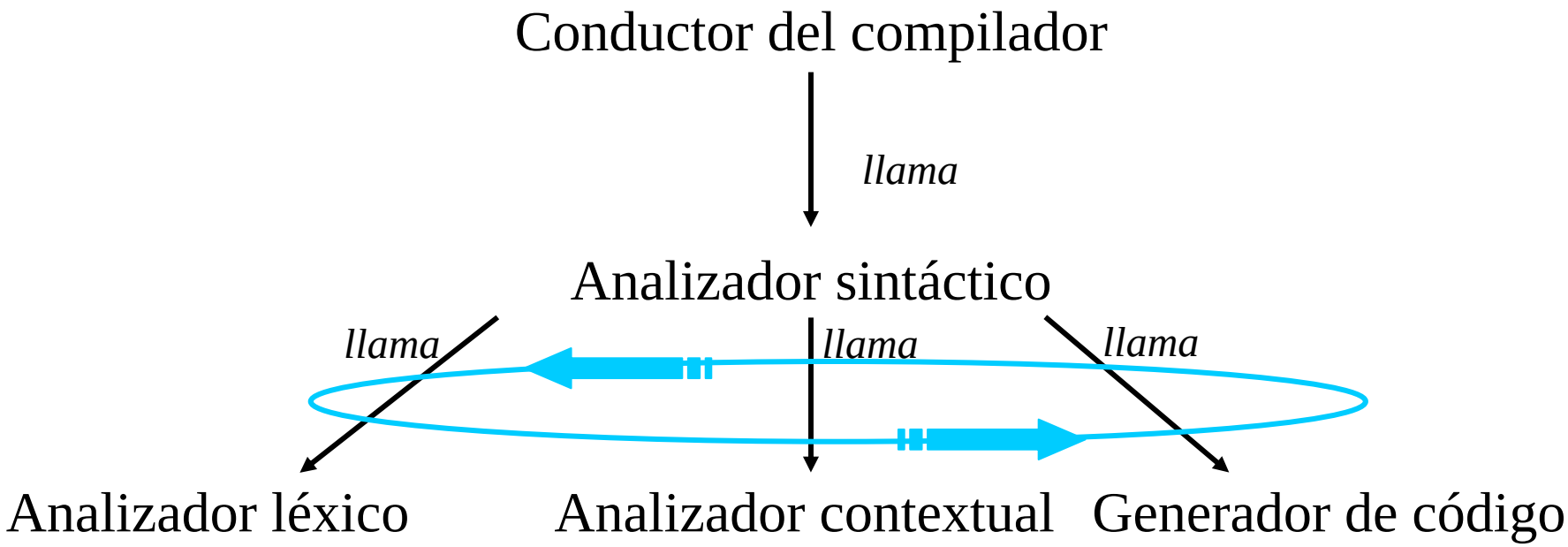
Una pasada

- Repetidamente:
  - Consumir poco a poco el texto fuente
  - Generar árbol sintáctico para una frase
  - ‘Decorar’ el árbol con información de tipos y alcance
  - Generar código para la frase
  - Seguir con siguiente frase
- Eficiente, pero difícil de mantener: mezcla muchas responsabilidades en un mismo lugar

Una pasada

- Generación de código
  - En lugar de construir árbol, se toma información de las partes de las frases
  - Generar código ‘al vuelo’
  - Requiere tabla de símbolos para las variables

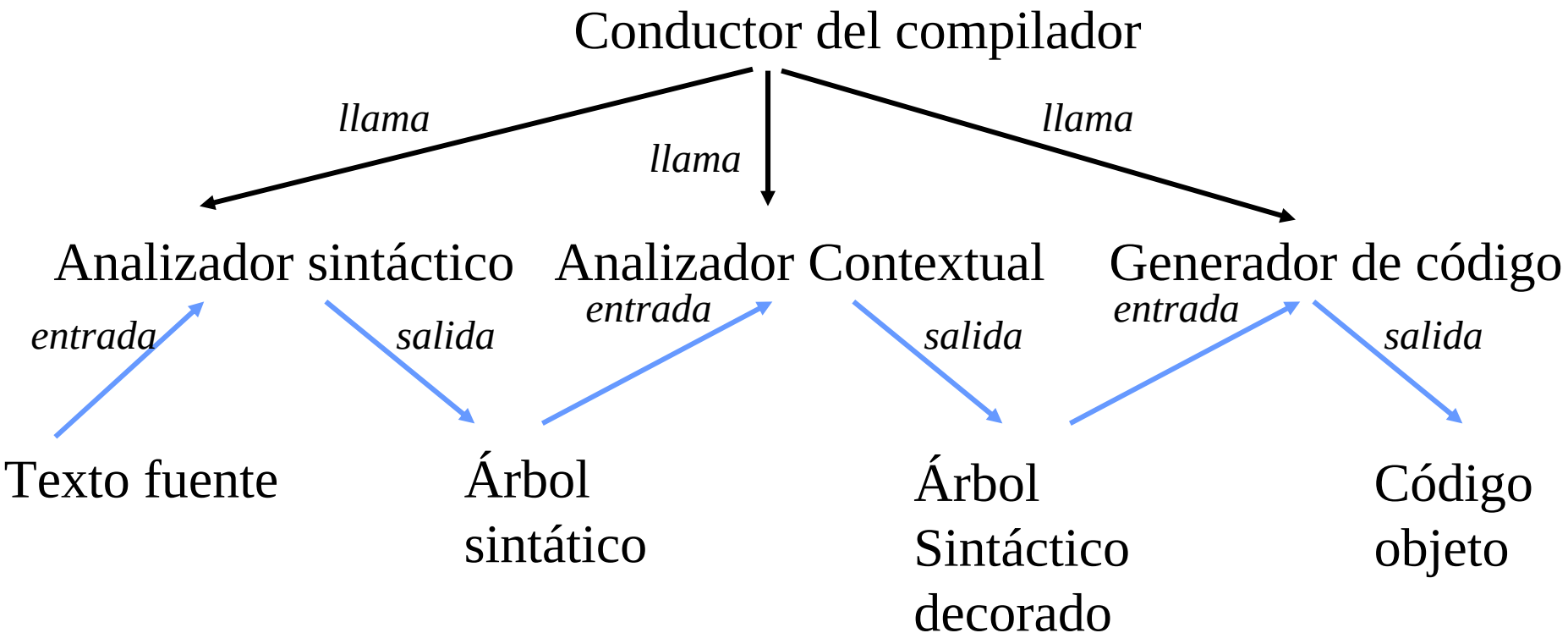
Compilador de un pasada: organización típica



Múltiples pasadas

- Generar árbol sintáctico para todo el programa
- Analizar alcance y tipos, decorando el árbol completo
- Generar código a partir del árbol sintáctico

Compilador de múltiples pasadas: organización típica



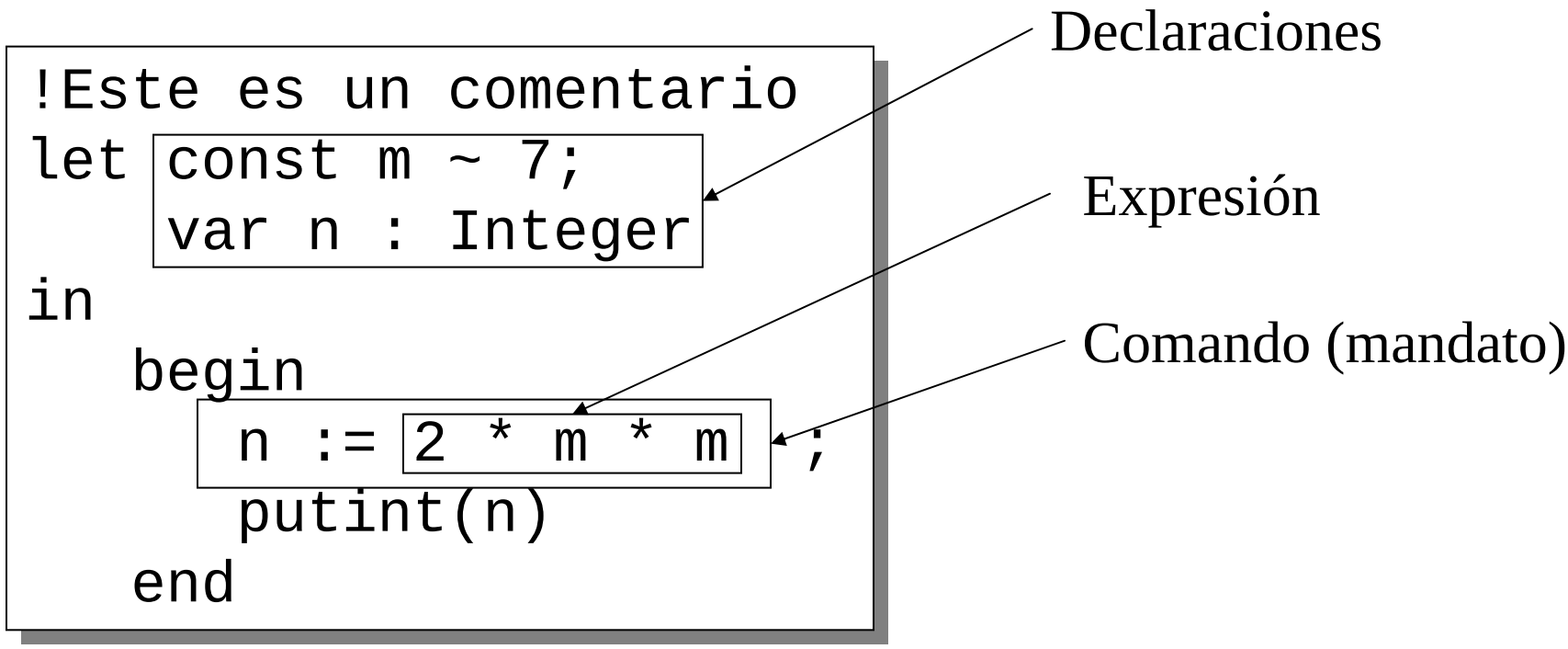
Pasadas múltiples versus única

- Una pasada
  - Rápido
  - Usa menos memoria
  - Difícil de construir y modificar
  - Más rígido
  - Imposible optimizar globalmente
- Múltiples pasadas
  - Más lento
  - Más fácil de construir o modificar
  - Diseño más modular
  - Más flexible (p.ej. para portar)
  - Posible realizar optimizaciones
- Algunos lenguajes requieren múltiples pasadas (al menos parcialmente)
  - Funciones mutuamente recursivas en ML
  - Métodos ‘adelante’ en Java

El compilador de Δ

- Compilador de tres pasadas
  - Analizador sintáctico transforma fuente en árbol sintáctico
  - Analizador contextual revisa árbol sintáctico y lo decora con información de identificación y de tipos
  - Generador de código parte del árbol sintáctico decorado
- Δ podría ser compilado en una pasada
  - No lo hacemos, por razones didácticas
  - El diseño en múltiples pasadas es más modular

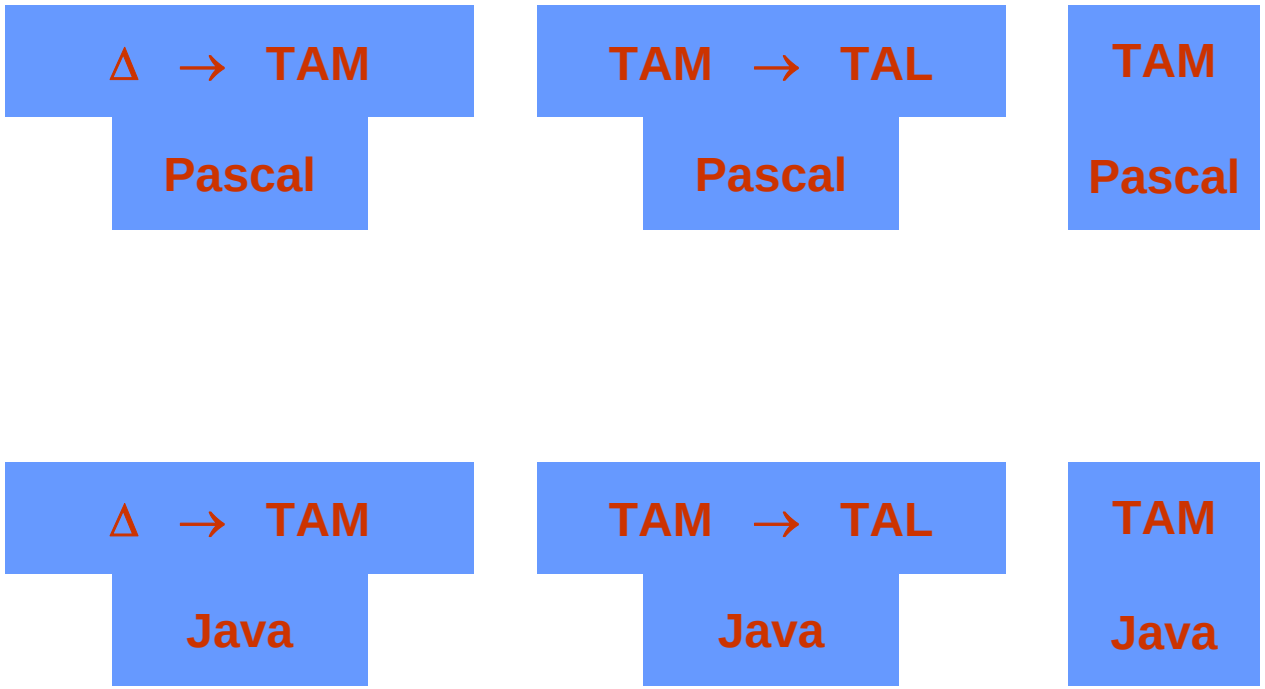
Ejemplito en Δ



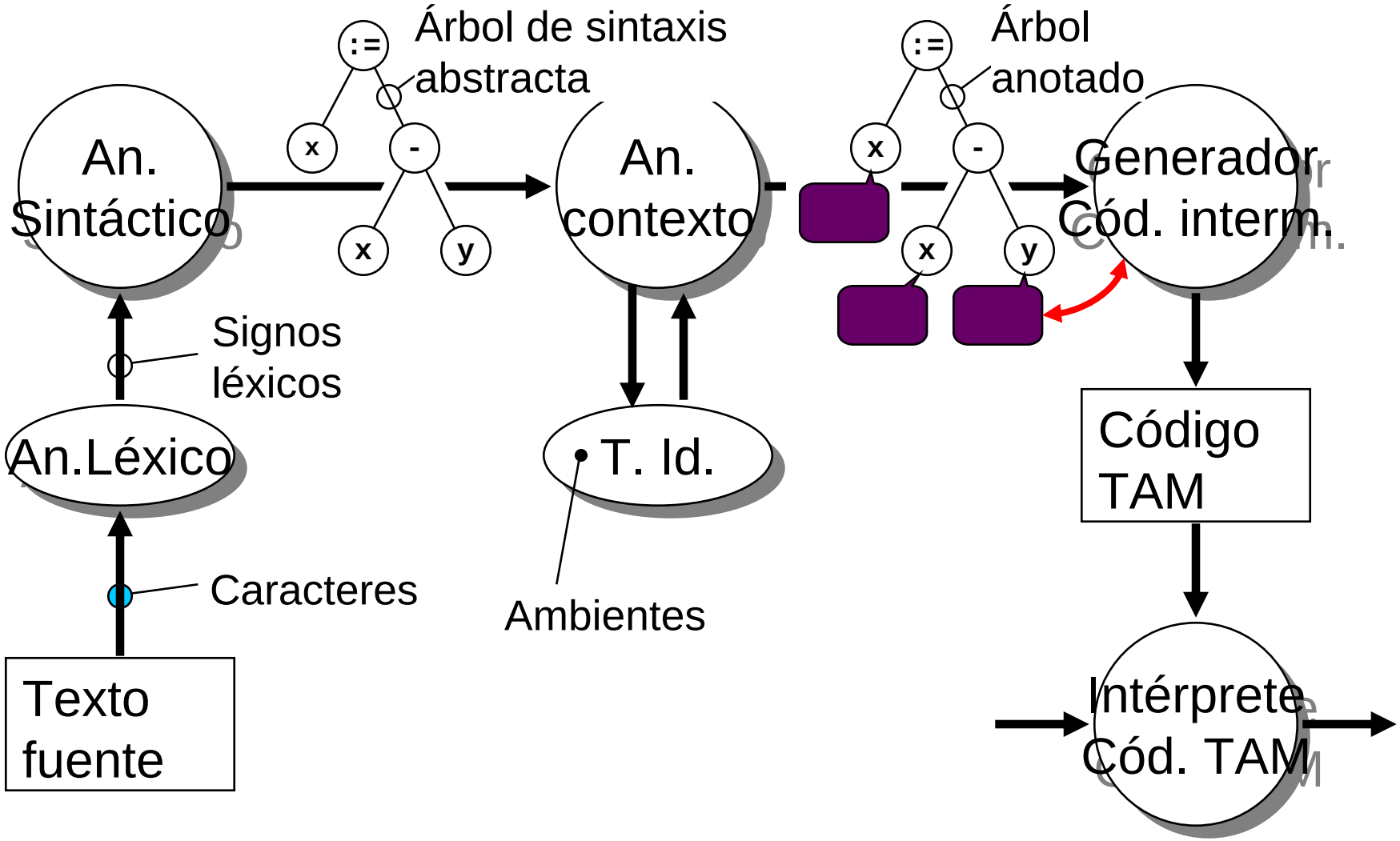
El procesador de Δ

- Tres lenguajes
  - Δ (Triángulo): el lenguaje fuente, estilo Pascal
  - TAM (Triangle Abstract Machine): código de máquina abstracta
  - TAL (Triangle Assembly Language): TAM con mnemónicos
- Tres programas
  - Compilador de Δ a TAM compiler, en Java o Pascal
  - Intérprete de TAM, en Java o Pascal
  - Desensamblador de TAM a TAL, en Java o Pascal

Los procesadores de Δ



Procesador de Δ



Compilación de Δ
Versión Pascal (editada)

```
procedure compileProgram (showingAST, showingTable: Boolean; var successful: Boolean);
                var theAST: AST;
begin
    initCompilation;
    startErrorReporting;
...
    parseProgram(theAST);
    if noErrors then
      begin
        checkProgram(theAST);
        if showingAST then
          drawAST(theAST);
        if noErrors then
          begin
            encodeRun(theAST, showingTable)
          end
      end;
    disposeAST(theAST);
    if noErrors then begin
      disassembleProgram;
    end
    else
      Console.Screen.Lines.Add('Compilation was NOT successful.')
     end; {compileProgram}
```

Conductor del compilador de Δ
Versión Java

```
public class Compiler {
      public static void compileProgram(...) {
       Parser parser = new Parser(...);
       Checker checker = new Checker(...);
       Encoder generator = new Encoder(...);

       Program theAST = parser.parse();
       checker.check(theAST);
       generator.encode(theAST);
      }
      public void main(String[] args) {
       ... compileProgram(...) ...
      }
  }
```

Herramientas para construir compiladores

- Generadores de analizadores sintácticos (“parser generators”)
  - Generar un analizador sintáctico a partir de una gramática independiente del contexto
  - Yacc, Bison, MLLama, JavaCC
- Generadores de analizadores léxicos
  - Generar un analizador léxico a partir de expresiones regulares
  - Lex, Flex