# Introducción a los procesadores de lenguajes

```
#gcc hello.c -o hello.exe -fsyntax-only
hello.c: In function 'main':
hello.c:3: 'i' undeclared
hello.c:8: unterminated string of char
hello.c:9: undefined reference to 'print'
```

# Procesador de lenguaje

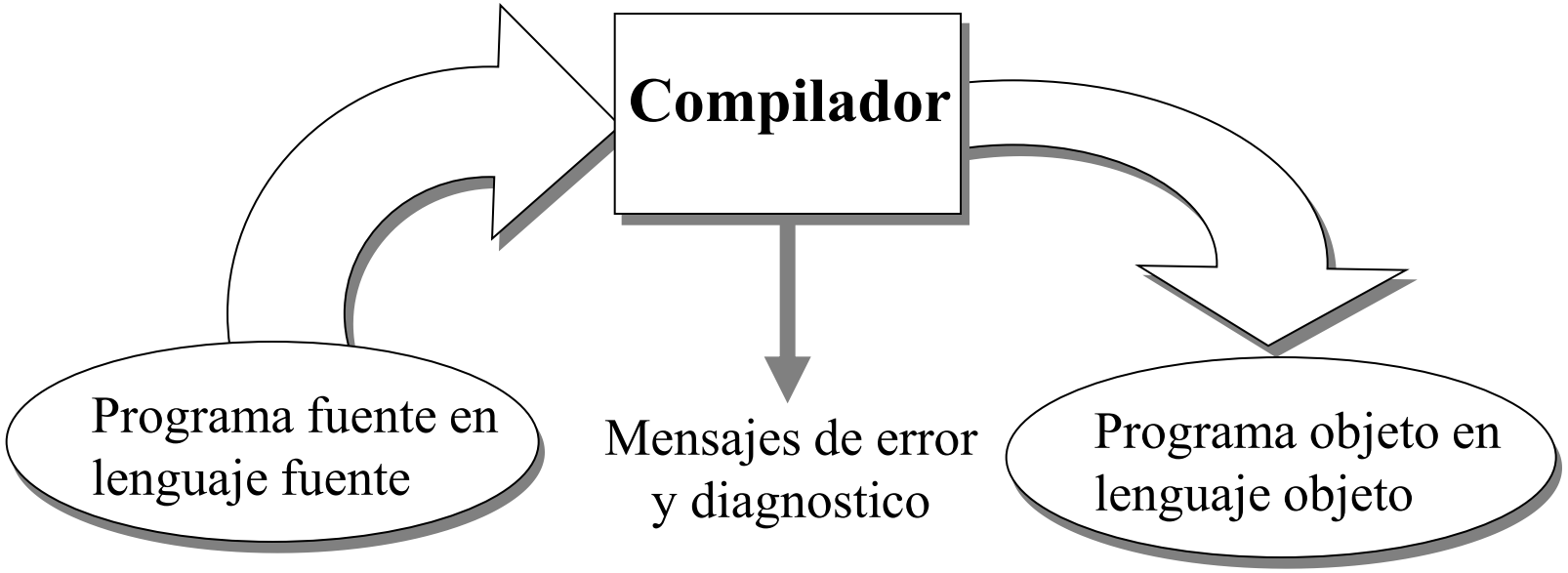
Un procesador de lenguajes es un programa informático que toma como entrada una secuencia de caracteres que forman algún tipo de texto escrito en un lenguaje, lo va a *analizar*, y *opcionalmente* va a realizar algún tipo de procesamiento con él, obteniendo por tanto un resultado.

Emilio Vivancos Rubio, Universidad Politécnica de Valencia

http://www.youtube.com/watch?v=aT5EGLgJs88

# Compilador

Un ***compilador*** es un programa que traduce un programa llamado ***programa fuente*** escrito en un cierto lenguaje denominado ***lenguaje fuente*** a un ***programa objeto*** en ***lenguaje objeto***. Además el compilador emite unos ciertos ***mensajes de error***.



# Ámbitos de utilización 1

Las técnicas de construcción de compiladores se utilizan también en otros muchos ámbitos

- Los **editores** de texto con análisis sintáctico en tiempo real y tecnologías como *Language Server Protocol*.
- Los programas **formateadores de texto** (TeX, LaTeX y ConTeXt), **lenguajes de marcado ligero** (AsciiDoc, Markdown + Pandoc, ReStructuredText + Sphinx) y formateadores de código (Black, Prettier)
- **Verificadores estáticos de sintaxis** (clang-tidy, ESLint, Pylint, MyPy) para detectar errores antes de la compilación. (parecido a lo que se obtiene con la opción `-fsyntax-only` del compilador de GNU `gcc`).
- Los **interpretes** (para lenguajes como Lisp o Basic, o para programas con sus propios lenguajes interpretados como Derive o Mathematica).
- **Lenguajes de macros** de aplicaciones extensibles (*Visual Basic for Applications*, Emacs Lisp, Lua).

# Ámbitos de utilización 2

- **Lenguajes de scripts** (Bash, Zsh, PowerShell).
- El análisis de los **ficheros de configuración** requeridos en muchos modernos sistemas (ficheros `.ini` de Windows, DS_Store en Mac, ficheros de recursos de las X, YAML, TOML, JSON, XML).
- **Compiladores de circuitos de silicio y lenguajes de descripción de hardware** (lenguajes para la descripción de circuitos: VHDL, Verilog, SystemVerilog, Chisel, SpinalHDL).
- **Lenguajes de consulta a bases de datos** y API (SQL, GraphQL, Gremlin, Cypher, SPARQL, ...).
- **Análisis de especificaciones de protocolos** de comunicación en lenguajes formales de especificación (como el Abstract Syntax Notation - ASN.1).
- Traducción de hojas de estilo (LESS, SASS, Stylus a CSS).
- Empaquetadores de recursos (Webpack, Vite, Parcel, Rspack, ... ).
- Minificadores y optimizadores (UglifyJS, Terser, Google Closure Compiler, SWC, esbuild, ..., CSSO, Lightning CSS, cssnano, ...).

# Ámbitos de utilización 3

- **Preprocesadores**, procesan un texto fuente modificándolo en cierta forma previamente a la compilación. (macroinstrucciones ajenas al lenguaje para inclusión de fichero externos, definición de macros, o extensiones del lenguaje, ej, Qt). `gcc -E`
- **Transpiladores**, traducen de un lenguaje de alto nivel a otro lenguaje de alto nivel (ej. Typescript a Javascript).
- **Especificación de flujos de procesamiento**: Snakemake, Common Workflow Language, Workflow Description Language, Nextflow.
- Automatización de la infraestructura (**IaC**): Docker, Terraform, Pulumi, Open Tofu, ...
- Gestión de la configuración: Ansible, Chef, Puppet, Salt, ...
- Compiladores de grafos para redes neuronales: XLA, TVM, TensorRT, JAX, Glow,

# Un poco de historia 1

- Los conjuntos de instrucciones de las primeras máquinas eran reducidos y sencillos. Las máquinas se programaban usando directamente **secuencias de dígitos binarios**.
- En seguida se vio que era mucho mejor utilizar **códigos nemotécnicos** para las instrucciones máquina. Así nacieron los *lenguajes ensambladores*. Al principio traducidos a mano.
- A finales de los 40 se vio que la traducción de los códigos nemotécnicos a código máquina la podían hacer los propios ordenadores mediante **programas *ensambladores***.

# Un poco de historia 2

- Con el tiempo se desarrollaron lenguajes más complejos, conocidos como ***autocodes*** (*autocódigos*), permitían describir los programas de forma más concisa. Cada instrucción del *autocode* podía representar varias instrucciones en código máquina. Los programas para trasladar estos primeros *lenguajes de alto nivel* en código máquina eran más complejos que los ensambladores se llamaron ***compiladores***.

# Un poco de historia 3

- Durante los 50 los lenguajes de alto nivel evolucionaron para describir los problemas independientemente del código máquina de los ordenadores utilizados
  - Por ejemplo, en FORTRAN IV el número máximo de dimensiones de un array estaba limitado a 3, en parte porque la máquina objetivo, la IBM 709, tenía sólo 3 registros para indexar los *arrays*. Incluso en C, diseñado a mediado de los 70, el operador de incremento ++, estaba motivado por la existencia del código máquina equivalente en la PDP-11).

# Un poco de historia 4

- Algol 60, que en realidad apareció en el 58, fue el precursor de un nuevo enfoque en el diseño de lenguajes de alto nivel. Se diseñó teniendo en mente la resolución de problemas y se ignoraron las cuestiones relativas a como debería ser traducido para ejecutarse en una máquina real.

La mayoría de los lenguajes modernos han sido diseñados independientemente de una arquitectura de máquina concreta.

# Un poco de historia 5

- En 1972 aparece C, con gran influencia en lenguajes posteriores y base de los SOs.
- En 1985 nace C++, popularizó la **OO** a gran escala en software de sistemas y aplicaciones.
- En 1995 con Java se introdujo la **máquina virtual** como estándar industrial. El modelo «*write once, run anywhere*» sigue vigente (y la JVM hoy corre muchos lenguajes, como Scala, Kotlin).
- En 1991 nace Python que por su simplicidad se ha convertido en el lenguaje dominante para ciencia de datos, scripting e IA.

# Un poco de historia 6

- En 2011 se consolida LLVM, iniciado en el 2000, como marco modular para la infraestructura de construcción de compiladores. Hoy en día, base de Clang, Swift, Rust y otros.
- En el periodo 2015-2018, aparecen los *frameworks* y compiladores para IA (Tensorflow, Pytorch, XLA) que estandarizaron el *machine learning* a gran escala, popularizando los grafos computacionales y la optimización automática en CPU/GPU/TPU.
- WebAssembly (2017) extiende la idea de la JVM, permitiendo ejecutar aplicaciones casi nativas en navegadores.

# Uso de lenguajes de alto nivel

Los lenguajes de alto nivel compilados están en la actualidad bien establecidos.

| Ventajas | Desventajas |
| --- | --- |
| • Mejora de productividad de los programadores<br>• Reducción de errores lógicos<br>• Facilidad de depurado | • Programas más lentos que los obtenidos en ensamblador<br>• Mayor tamaño de los programas obtenidos |

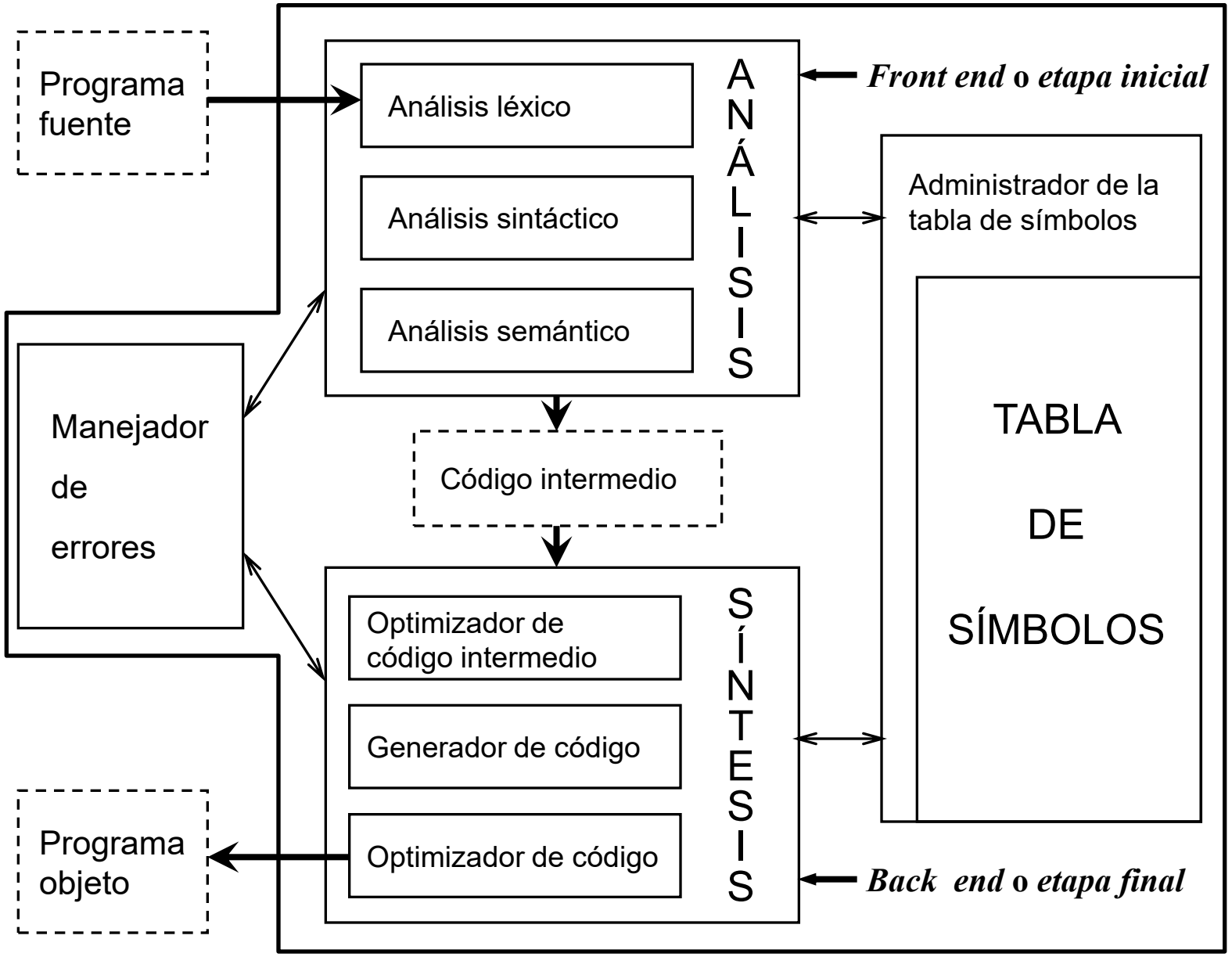
# Interpretar versus compilar

- Hay dos maneras de ejecutar un programa escrito en un lenguaje de alto nivel.
  - **Compilación**: traducirlo todo el programa a otro programa equivalente en código máquina del ordenador. Entonces se ejecuta el programa obtenido.
  - **Interpretación**: interpretar las instrucciones del programa escrito en lenguaje de alto nivel y ejecutarlas una por una.

| Interprete | Compilador |
|---|---|
| • Fácil localización de errores.<br>• Cada vez que se ejecuta el programa es necesaria su interpretación.<br>• Adecuado en la etapa de desarrollo y depuración. | • Difícil localización de errores.<br>• Sólo es necesaria una compilación. Y una vez realizada, la velocidad de ejecución es alta.<br>• Adecuado cuando no hay más errores (etapa de explotación) |

https://www.youtube.com/watch?v=_C5AHaS1mOA

# Estructura de un compilador 1



# Estructura de un compilador 2

- El proceso de compilación se puede dividir en una serie de **fases**, que pueden llevarse a cabo simultáneamente o consecutivamente y cada una de las cuales transforma el programa fuente de una representación en otra.

> En la práctica se pueden agrupar algunas fases, y las representaciones intermedias entre las fases agrupadas no necesitan ser construidas explícitamente.

***front end*** = *a. léxico + a. sintáctico + a. semántico*

(***middle end*** = *generación de código intermedio*)

***back end*** = *opt. de cód. intermedio + gen. de cód. + opt. de cód*

# Estructura de un compilador 3

- Si el *front end* (etapa de análisis o etapa inicial), que comprende aquellas fases que dependen principalmente del lenguaje fuente), se diseña adecuadamente, es posible usar distintos *back end* (etapa de síntesis o etapa final), que incluye las fases que dependen de la máquina objeto, de modo que se puede obtener código para distintas máquinas.

# Etapa de análisis

- La **etapa de análisis** (***front end*** o **etapa inicial**) agrupa aquellas fases que dependen principalmente del lenguaje fuente, y comprende:
  - El **analizador léxico** (también llamada ***scanner***): agrupar los caracteres individuales en entidades lógicas.
  - El **analizador sintáctico** (también llamado ***parser***) analiza la estructura general de todo el programa, agrupando las entidades simples identificadas por el scanner en construcciones mayores, como sentencias, bucles, rutinas, que componen el programa completo. Normalmente se utiliza la representación de **árboles sintácticos** para reflejar dicha estructura.
  - Una vez determinada la estructura del programa se puede analizar su significado (*semántica*) mediante el **analizador semántico**. Se determina que variables almacenaran enteros, cuales número en coma flotante, si el acceso a los *arrays* cae dentro del rango fijado en su definición, ... , etc.

# Etapa de síntesis

- La **etapa de síntesis** (***back end*** o **etapa final**) agrupa aquellas fases que dependen principalmente de la máquina objetivo, y comprende:
  - El **optimizador de código intermedio** que transforma la representación intermedia en otra equivalente pero más eficiente.
  - El **generador de código** genera un programa equivalente para su ejecución en la máquina objetivo, añadiéndole posiblemente rutinas de biblioteca y código de inicialización.
  - Finalmente puede haber un **optimizador de código** para mejorar aún más el código generado.

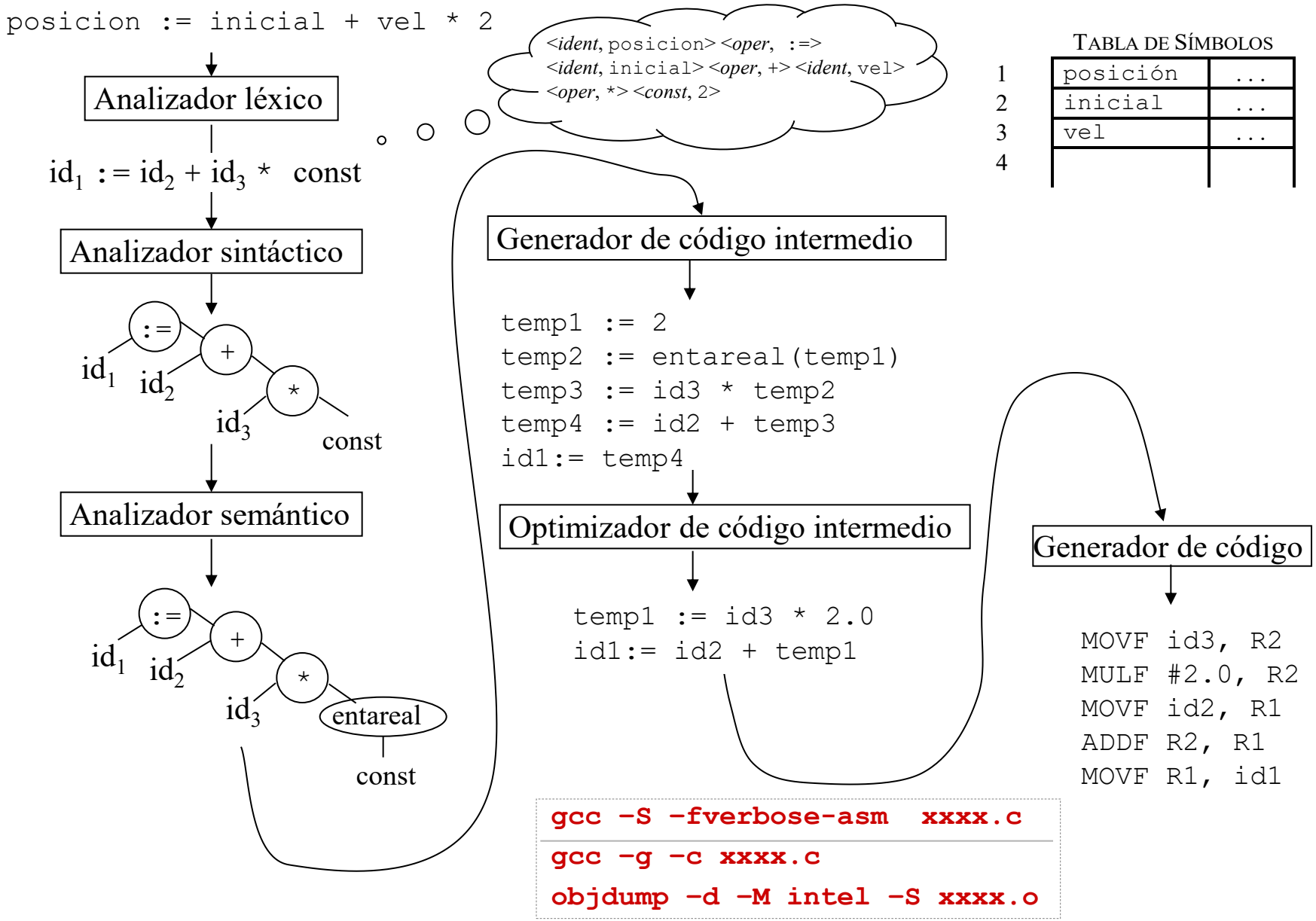
# El manejador de errores

- Un compilador tiene que tener un determinado comportamiento ante programas erróneos. Este proceso se agrupa en una fase llamada **manejador de errores**. Cada una de las fases anteriores interacciona con el manejador de errores.

# El administrador de la tabla de símbolos

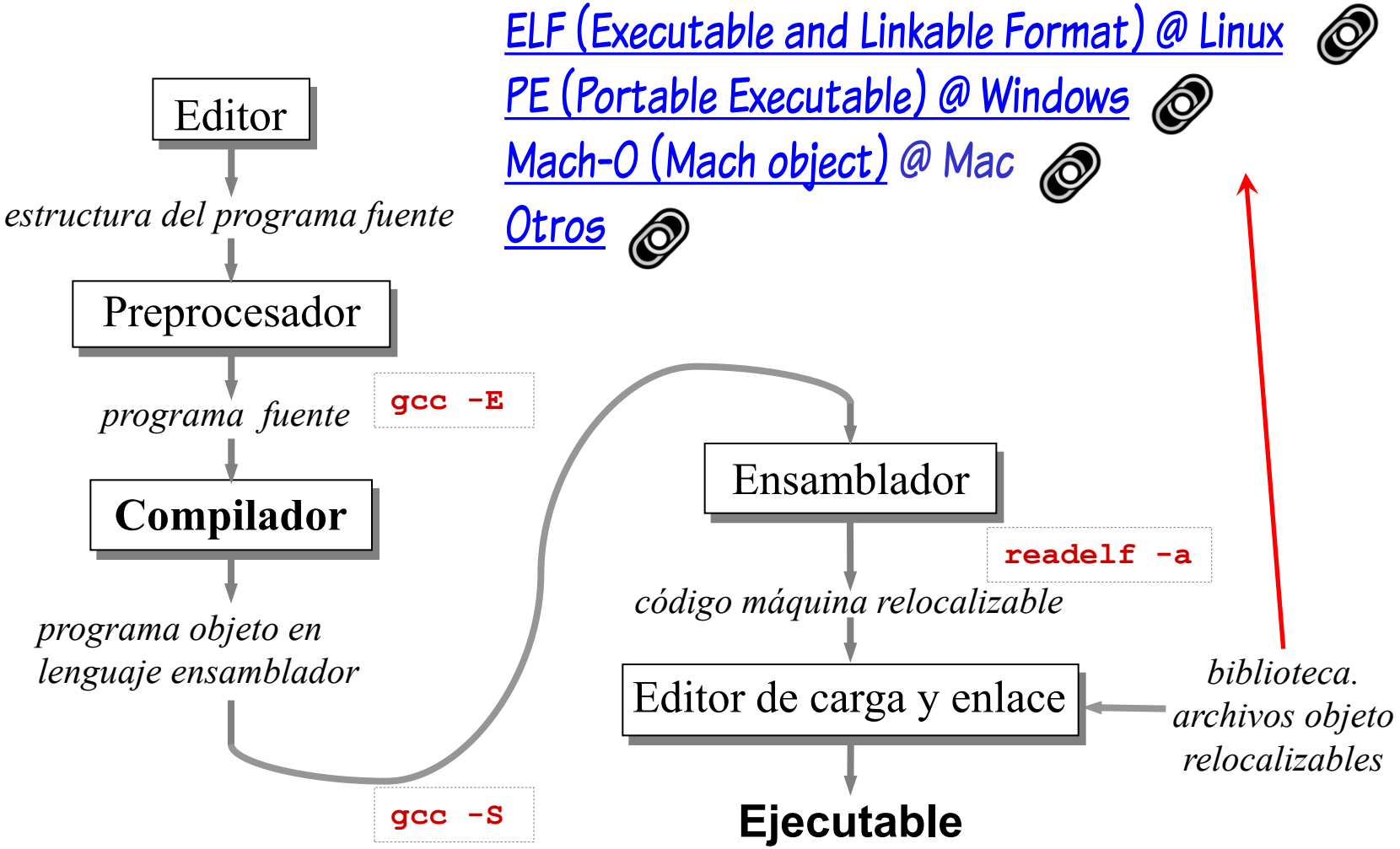
- Otro elemento en la compilación es la tabla de símbolos a la que acceden cada una de las fases a través del **administrador de la tabla de símbolos**. En la tabla de símbolos se almacena información sobre los distintos identificadores.
  - De variables, constantes, funciones y procedimientos, como: tipo, memoria asignada, ámbito, alcance, ..., etc. Las fases anteriores introducen estas información sobre los identificadores en la tabla de símbolos y después la utilizan de varias formas).

# Proceso de compilación 1




```
gcc -S -fverbose-asm  xxxx.c
gcc -g -c xxxx.c
objdump -d -M intel -S xxxx.o
```

# Entorno de trabajo de un compilador



# Construcción de un compilador

- En la construcción de un compilador se puede proceder siguiendo varios enfoques, pudiéndose escribir:
  - en lenguaje **ensamblador**, con lo que se consiguen compiladores muy eficientes pero difíciles de mantener.
  - en un lenguaje de **alto nivel**, con lo que se hace más fácil el mantenimiento, pero aún requiere mucho tiempo de desarrollo.
  - utilizando **herramientas** de construcción de compiladores como flex, bison, JavaCC, JJTree, JTB.

# Herramientas

- Generadores de analizadores léxicos
  - Generan automáticamente analizadores léxicos partiendo de una especificación basada en expresiones regulares (ej. Lex/Flex/JFlex, JavaCC, ANTLR).
- Generadores de analizadores sintácticos
  - Producen analizadores sintácticos partiendo de una gramática independiente del contexto (ej. Yacc/Bison,
  - JavaCC, ANTLR, Lemon, PLY, Lark, SableCC, CUP, ...)
- Generadores de árboles sintácticos
  - Proporcionan medios para facilitar la creación de árboles sintácticos (ej. JJTree, JTB)

# Herramientas

- Dispositivos de traducción dirigida por la sintaxis
  - Producen grupos de rutinas que recorren el árbol de análisis sintáctico generando código intermedio
- Generadores automáticos de código
  - Toman un conjunto de reglas que definen la traducción de cada operación del lenguaje intermedio al lenguaje máquina objeto. La técnica fundamental es la concordancia de plantillas.
- Dispositivos para análisis de flujo de datos
  - Permiten la optimización de código en función de una recolección de información sobre la forma en que se transmiten los valores de una parte de un programa a cada una de las otras partes.

# Herramientas

- LLVM (*Low Level Virtual Machine*), *framework* de desarrollo de compiladores.
  - Proporciona una infraestructura modular con elementos reutilizables.
  - Soporta la generación de código para múltiples *backends*, para CPU, GPU y aceleradores,
  - Ha transformado el diseño de compiladores: en vez de crear un compilador monolítico, se construyen sobre LLVM componentes especializados.
  - Usado en lenguajes modernos (Rust, Swift, Julia, Kotlin/Native), navegadores (WebAssembly), e incluso *frameworks* de IA (XLA, MLIR)

# Herramientas

- Herramientas formales:
  - Gramáticas.
  - Máquinas de Turing.
  - Autómatas finitos deterministas y no deterministas.
  - Autómatas de pila.
  - Análisis LL, LR, SLR y LALR.
- Herramientas de programación:
  - Yacc/Bison.
  - Lex/Flex/JFlex.
  - JavaCC, ANTLR, Lemon, PLY, Lark, SableCC, CUP, ...