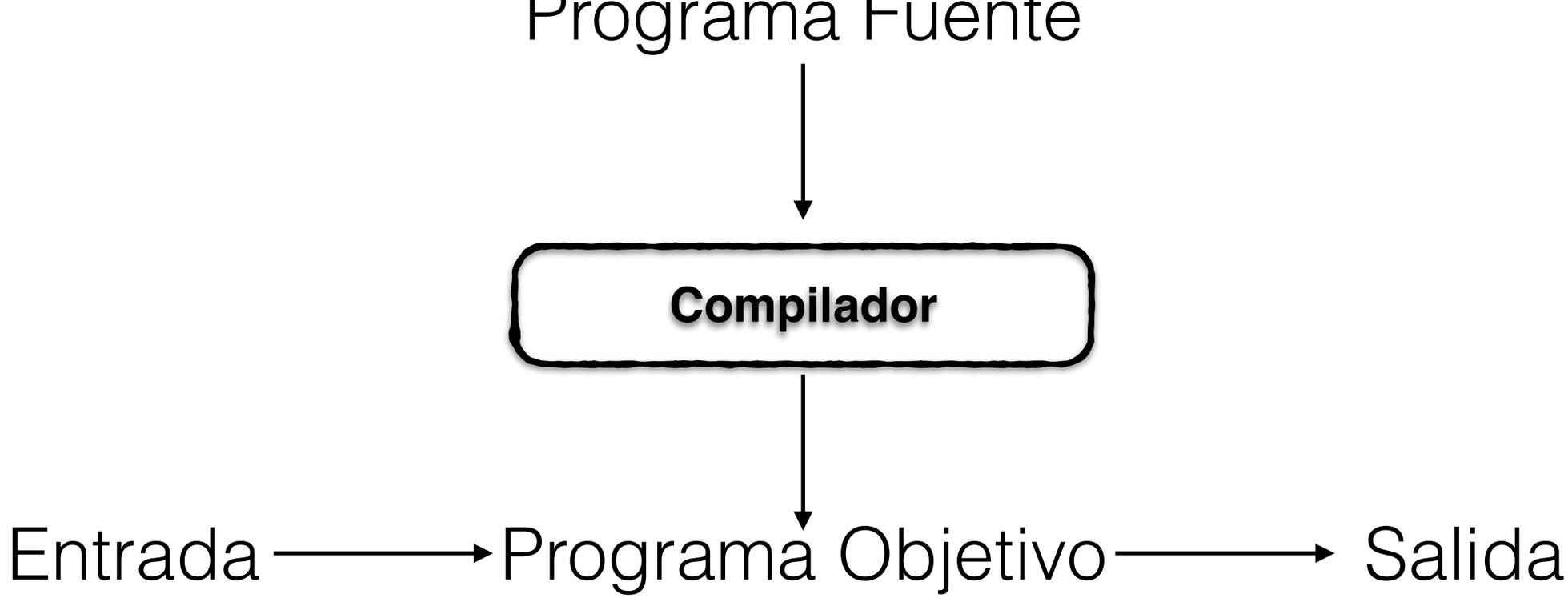
Análisis Léxico

Escuela de Ingeniería Civil Informática
Universidad de Valparaíso

Profesor: Dr. Ismael Figueroa
ifigueroap@gmail.com

Compilación



Visión de alto nivel del trabajo de un compilador

Ejemplo

Programa Fuente: gcd.c

```
int gcd(int a, int b) {
    while (a != b) {
        if (a > b) a = a - b;
        else b = b - a;
    }
}
```

archivo de texto, o sea, el compilador lo lee como un un **string**

gcc / clang

Programa Objetivo: gcd.s

```
        addiu   sp,sp,-32
        sw      ra,20(sp)
        jal     getint
        nop
        jal     getint
        sw      v0,28(sp)
        lw      a0,28(sp)
        move    v1,v0
        beq     a0,v0,D
        slt     at,v1,a0
A:      beq     at,zero,B
        nop
        b       C
        subu    a0,a0,v1
B:      subu    v1,v1,a0
C:      bne     a0,v1,A
        slt     at,v1,a0
D:      jal     putint
        nop
        lw      ra,20(sp)
        addiu   sp,sp,32
        jr      ra
        move    v0,zero
```

Leer un archivo en C

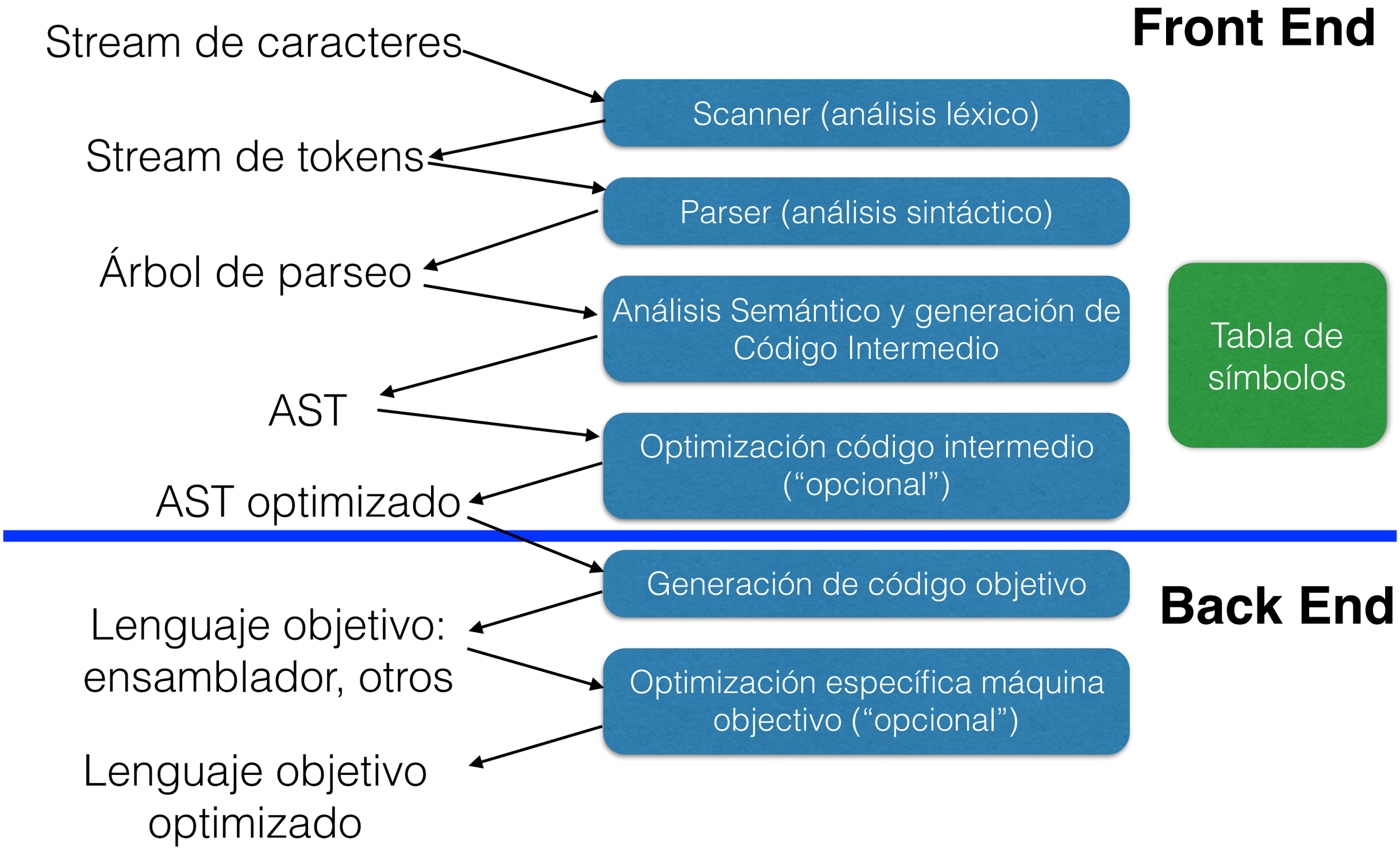
```c
int main(int argc, char** argv[])
{
   char ch, file_name[25];
   FILE *fp;

   // Archivo es el primer parámetro de la línea de comandos
   // e.g.: "gcc gcd.c"
   file_name = argv[1];

   fp = fopen(file_name,"r"); // read mode

   if( fp == NULL )
   {
      perror("Error while opening the file.\n");
      exit(EXIT_FAILURE);
   }

   // Lectura caracter por caracter ...
   while( ( ch = fgetc(fp) ) != EOF ) {
     // COMPILACION AQUI??
   }

   fclose(fp);
   return 0;
}
```

no se ve muy factible ...

En la práctica…

- Un compilador procesa un archivo a través de una serie de **fases** (*compiler phases*), bien definidas.
- Cada fase **descubre información** que es útil a las fases posteriores, o bien **transforma** el programa en una forma que facilita el trabajo de la fase siguiente

Fases de Compilación



Fases de Compilación

- El Front End es independiente del lenguaje objetivo, y tiene por objetivo *capturar el significado de un programa en el lenguaje fuente*

- El Back End es específico para cada lenguaje o máquina objetivo. Su objetivo es *generar un programa objetivo equivalente al programa fuente.*

Compiladores en el Mundo Real

- GCC:
  - Front ends: C/C++, Objective-C, Ada, Fortran, Java, Go, y otros…
    - Back ends: https://gcc.gnu.org/backends.html
- LLVM: http://llvm.org/Features.html
- Java JDK: el bytecode es el producto final del front-end. El JRE es la puerta de entrada del back-end.

Sintáxis

- El problema de definir la sintáxis de un lenguaje de programación es que ésta debe ser ***precisa***.

- No debe existir ambigüedad, de manera que tanto los desarrolladores y los computadores puedan entender los programas.

- Para obtener la precisión necesaria, se utiliza ***un método formal de notación.***

Ejemplo: números naturales

- Un número es una concatenación de *dígitos*, donde el primer *dígito es distinto a cero.*

- Por ejemplo: “7645” = “7” + “6” + “4” + “5”

- Pero también hay números que tienen ceros:

  - “76405” = “7” + “6” + ”4” + “0” + ”5”

- ***¿Cómo definir un patrón general para calzar contra strings arbitrarios y determinar si son números?***

Expresiones Regulares (RE)

- Recordatorio “Lenguajes y Autómatas”
- Un *lenguaje* es un conjunto (posiblemente infinito) de *strings*. Y un *string* es una secuencia finita de *símbolos*. Los símbolos pertenecen a un *alfabeto* finito.
- Una *expresión regular* es una notación formal para denotar un lenguaje.

Expresiones Regulares (RE)

- **Símbolos**: si el símbolo **a** está en el alfabeto, la expresión regular **a** denota el lenguaje que contiene sólo el string *a*.

- **Disyunción**: dadas dos expresiones regulares M y N, la disyunción denotada M | N es también una expresión regular. Un string pertenece al lenguaje M | N si está contenido el lenguaje de M o bien en el lenguaje de N.

- **Concatenación**: dadas dos expresiones regulares M y N, la concatenación M · N es una nueva expresión regular. Un string pertence a M · N si es la concatenación de dos strings *alpha* y *beta*, tales que *alpha* está en M y *beta*

Expresiones Regulares (RE)

- **Epsilon**: la expresión regular **ε** representa un lenguaje que sólo contiene el string vacío.

- **Repetición:** dada una expresión regular M, su “clausura de Kleene” es M*, y también es una expresión regular. Un string pertence a M* si es la concatenación de cero o más strings en M.

Expresiones Regulares (RE)

- Ejemplos.
  - a | b = { “a”, “b” }
  - (a | b) · a = { “aa”, “ba” }
  - (a · b) | ε = { “ab”, “ “ }
  - ((a | b) · a)* = { “ “, “aa”, “aaaa”, “ba”, “baba”, … }

RE: Notación

- Omitir el operador de concatenación y epsilon. El operador * tiene más precedencia que la concatenación, y la concatenación tiene más precedencia que la disyunción.

- Los corchetes cuadrados [ ] se usarán como abreviación para la disyunción. Por ejemplo [**abcd**] = **a** | **b** | **c** | **d**

- Dada una expresión regular M, M+ = (M · M*). Además, M? = (M | **ε**)

- Rangos de caracteres, por ejemplo [**b**-**g**] = [**bcdefg**]

Tokens y RE

- DIGITO = [0-9]
- DIGITO_NO_CERO ::= [1-9]
- NATURAL ::= DIGITO_NO_CERO DIGITO*
- BIN_OP ::= + | -
- Los digitos reconocidos no tienen significado, **sólo son símbolos**. El análisis léxico de un programa consiste en construir un ***flujo de tokens*** que será consumido por el analizador sintáctico.

Tokens

- Un *token* es una secuencia de caracteres que puede ser tratada como una unidad en la gramática de un lenguaje de programación

- Un lenguaje clasifica los tokens léxicos en un conjunto finito de *tipos de token*. Por ejemplo:

Ejemplos Tipos de Token

Tipo	Ejemplo
ID	foo, n14, last
NAT	73, 0, 515, 082
REAL	66.1, .5, 10., 1e67, 5.5e-10
IF	if
COMMA	,
NOTEQ	!=, <>, =/=
LPAREN	(, {, [
RPAREN	), },]
BIN_OP	+, .

Analizadores Léxicos en Racket

- https://docs.racket-lang.org/parser-tools/Lexers.html?q=define-tokens#%28part._.Lexer_.S.R.E_.Operators%29