Entorno de traducción

- Compilación por partes + vinculador (linker)
- Unidad de traducción (translation unit): cada fuente luego del preprocesador
- Fases de traducción (C23:5.1.1.2):
  - Conversión al juego de caracteres usado por la implementación. Nota, los trígrafos no se usan más desde C23.
  - Se unen las líneas que terminan en una barra invertida (\).
  - El fuente se divide en tokens de preprocesamiento y espacios en blanco. Los comentarios se reemplazan por un carácter de espacio.
  - Se ejecutan las directivas de preprocesamiento y se expanden las macros al archivo de código fuente.
  - Las secuencias de escape se convierten en sus equivalentes en el juego de caracteres de ejecución.
  - Se concatenan los literales adyacentes.
  - Traducción (compilación): los tokens se analizan sintáctica y semánticamente se genera el código de objeto.
  - Vinculación: Se resuelven todas las referencias externas para crear un programa ejecutable o una biblioteca de vínculos dinámicos.

Entorno de Ejecución Independiente

- Entorno Independiente (Freestanding)
  - Sin sistema operativo, típicamente un embebido.
  - El nombre de la función donde inicia el programa puede ser cualquiera (definido por la implementación)
  - Lo mismo para la firma (declaración de la misma)
  - La manera en que el programa termina está definido por la implementación.
  - Solo está obligado a dar soporte a un subconjunto de la biblioteca estándar

Entorno de Ejecución Alojado

- Entorno alojado (Hosted)
  - El programa comienza en la función main que devolverá 0 para indicar éxito y distinto de cero para indicar que terminó con error. Tendrá una de dos formas
    - `int main(void)`
    - `int main(int argc, char *argv[])`
  - argc deb ser positivo, incluye el nombre del programa en argv[0] (si está disponible), el resto son los argumentos de la función.
  - argv[argc] es un puntero nulo
  - argc y argv viven durante toda la ejecución del programa y pueden ser modificados
  - Si no hay sentencia return, devuelve cero
  - En main `return(exp)` es equivalente a llamar a `exit(exp)`

Categorías Léxicas

- Lenguaje C usa varios LR a los cuales llama categorías léxicas o tokens
- Cada palabra de alguno de estos lenguajes se conoce como lexema
- En lenguaje C se reconocen los siguientes tokens
  - ~~PalabraReservada~~ (keyword) => Palabras clave
  - Identificador
  - constante
  - LiteralCadena (string-literal)
  - CaracterPuntuación (punctuator)
    - Incluye a los operadores, que en ANSI C se los consideraba una categoría separada

Palabras Reservadas (C 17)

Entre paréntesis: alternativa en C23

Nuevo en estándar		
Ansi C	C99	C11

auto	extern	short	while
break	float	signed	_Alignas (alignas)
case	for	sizeof	_Alignof (alignof)
char	goto	static	_Atomic
const	if	struct	_Bool (bool)
continue	inline	switch	_Complex
default	int	typedef	_Generic
do	long	union	_Imaginary
double	register	unsigned	_Noreturn
else	restrict	void	_Static_assert (static_assert)
enum	return	volatile	_Thread_local (thread_local)

Cambios en C23

true	false
constexpr	nullptr
typeof	typeof_unqual
_BitInt	_Decimal128
_Decimal32	_Decimal64

Constantes Numéricas Enteras

- Decimales: dígitos del 0 al 9 pero **NO puede** comenzar con 0
- Octales: comienza con 0 y solo utilizan los dígitos del 0 al 7
- Hexadecimales: comienzan con 0x o 0X y además de los dígitos agregan las letras de la A a la F, pueden ser mayúsculas o minúsculas, incluso entremezcladas.
- Binarias (C23): comienzan con 0b o 0B seguido de uno o más dígitos binarios.
- Sin sufijo son de tipo `int` (en principio)
- Sufijos: `l L u U ll LL` pueden combinarse en cualquier orden, salvo `L` con `LL`, pueden ser mayúsculas o minúsculas

Constantes Numéricas Enteras

- Ejemplos
  - `12` //decimal, tipo int
  - `012u` //octal, tipo unsigned int
  - `0x2fLL` //hexadecimal, tipo long long int
  - `0b101UL` //binario de tipo unsigned long int

Constantes Numéricas Reales

- Decimales
  - Debo poner la parte fraccionaria o el exponente, uno de los dos es obligatorio. El exponente comienza con e o E y su signo es optativo.
- Hexadecimales
  - Comienzan con 0x y luego la parte fraccionaria en hexadecimal
  - El exponente es obligatorio y comienza con p o P y luego en decimal el exponente cuya base es 2.
- Ambos
  - El . de la parte fraccionaria puede no tener dígitos adelante o atrás
  - Sin sufijo son de tipo `double`
  - Sufijos: `l L f F` (`F` para `float` y `L` para `long double`)

Constantes Reales - Ejemplos

- `5e4` //double con valor $5x10^4$
- `.2f` //float con valor 0,2
- `3.L` //long double con valor 3,0
- `7.2E-3` //double con valor 0.0072
- `0xBp3` //double con valor $11x2^3$
- `0X12.cP-5`
  //valor: (16+2+12/16)/32 = 0,585938

Constantes de Carácter

- De carácter
  - Delimitadas entre comillas simples `''`
  - Si bien las variables las declaramos de tipo `char` las constantes son de tipo `int`.
  - El tipo `char` es tomado como `unsigned char` o signed char dependiendo de la implementación.
  - Secuencias de escape
    - `\', \", \?, \\, \dig-oct, \xdig-hex`
    - `\a , \b , \f , \n , \r , \t  \v`

Caracteres “anchos” (wide)

- `wchar_t` definido en `stddef.h` usa prefijo `L`
- `char16_t` definido en `uchar.h` usa prefijo `u`
- `char32_t` definido en `uchar.h` usa prefijo `U`
- `C23 agrega:`
  - `char8_t definido en uchar.h usa prefijo u8`
  - No es parte de los caracteres anchos.

Constantes de enumeración

- De enumeración
  - Las que se definen con enum
    - `enum colores {ROJO, AMARILLO = 3, VERDE};`
    - `enum colores col = VERDE; //asigna 4`
  - Son de tipo `int`
  - En C son simples constantes, no definen un nuevo tipo de datos como si puede hacer C++
    - `col = 300;`
      - válido en C
      - error: invalid conversion from ‘int’ en C++

Constantes de enumeración

- C23 agrega con tipo de dato explícito
  - `Genéricamente enum nombre : tipo { … };`
  - `enum colores : short int {ROJO, AMARILLO = 3, VERDE};`

Literales de cadena

- Se encierran entre comillas dobles
- El final está delimitado por `'\0'`
- Se puede aplicar \ para poder incluir una comillas doble y cualquier secuencia de escape válida para constantes de carácter.
- El tipo de dato es `char[]` (generalmente usado como `char*` que es a lo que degrada)
- Cadenas adyacentes se concatenan
- Se pueden agregar prefijos: u8 para utf8, L, u y U con el mismo significado que para caracteres

Puntuación (punctuator - C23)

Del estándar C23 6.4.6: Un signo de puntuación es un símbolo que tiene un significado sintáctico y semántico independiente. Dependiendo del contexto, puede especificar una operación a realizar (que a su vez puede generar un valor o el designante de una función, producir un efecto lateral o alguna combinación de los mismos) en cuyo caso se conoce como operador (otras formas de operador también existir en algunos contextos).

```
punctuator: one of
[ ] ( ) { } . ->
++ -- & * + - ~ !
/ % << >> < > <= >= == != ^ | && ||
? : :: ; ...
= *= /= %= += -= <<= >>= &= ^= |=
, # ##
```

Operadores

Fuente: https://es.cppreference.com/w/c/language/operator_precedence

Precedence	Operator	Description	Associativity
1	`++ --`	Suffix/postfix increment and decrement	Left-to-right
	`()`	Function call	
	`[]`	Array subscripting	
	`.`	Structure and union member access	
	`->`	Structure and union member access through pointer	
	`(`*`type`*`){`*`list`*`}`	Compound literal(C99)	
2	`++ --`	Prefix increment and decrement	Right-to-left
	`+ -`	Unary plus and minus	
	`! ~`	Logical NOT and bitwise NOT	
	`(`*`type`*`)`	Type cast	
	`*`	Indirection (dereference)	
	`&`	Address-of	
	`sizeof`	Size-of	
	`_Alignof`	Alignment requirement(C11)	
3	`* / %`	Multiplication, division, and remainder	Left-to-right

Operadores

Fuente: https://es.cppreference.com/w/c/language/operator_precedence

Precedence	Operator	Description	Associativity
4	+ -	Addition and subtraction	Left-to-right
5	<< >>	Bitwise left shift and right shift	
6	< <=	For relational operators < and ≤ respectively	
	> >=	For relational operators > and ≥ respectively	
7	== !=	For relational = and ≠ respectively	
8	&	Bitwise AND	
9	^	Bitwise XOR (exclusive or)	
10	|	Bitwise OR (inclusive or)	
11	&&	Logical AND	
12	||	Logical OR	
13	?:	Ternary conditional	Right-to-Left
14	=	Simple assignment	
	+= -=	Assignment by sum and difference	
	*= /= %=	Assignment by product, quotient, and remainder	
	<<= >>=	Assignment by bitwise left shift and right shift	
	&= ^= |=	Assignment by bitwise AND, XOR, and OR	
15	,	Comma	Left-to-right

Puntuación – Otros casos

- { } delimitan bloques
- ; transforma una expresión en sentencia o separan expresiones dentro de for
- , se utiliza para separar variables en su declaración o definición y para separar argumentos de una función.
- : se usa en etiquetas, ya sea para goto o para los distintos casos dentro de un switch
- ... se utilizan para declarar o definir funciones con cantidad variable de argumentos
- # directiva del preprocesador u operador para convertir a cadena
- ## operador para “pegar” tokens

Detección de lexemas

```
/* fragmento de ejemplo*/
int mifunc(int a)
{
        int r,aux=1;
        r = aux+++a;  /*solo para mostrar +++ */
        return r;
}
```

int	palabraReservada	**,**	caracterPuntuación	**++**	operador
mifunc	identificador	**aux**	identificador	**+**	operador
(	caracterPuntuación	**=**	caracterPuntuación	**a**	identificador
int	palabraReservada	**1**	constante	**;**	caracterPuntuación
a	identificador	**;**	caracterPuntuación	**return**	palabraReservada
)	caracterPuntuación	**r**	identificador	**r**	identificador
{	caracterPuntuación	**=**	operador	**;**	caracterPuntuación
int	palabraReservada	**aux**	identificador	**}**	caracterPuntuación
r	identificador				

Licencia