

s

Instituto Politécnico Nacional

Escuela Superior de Computo

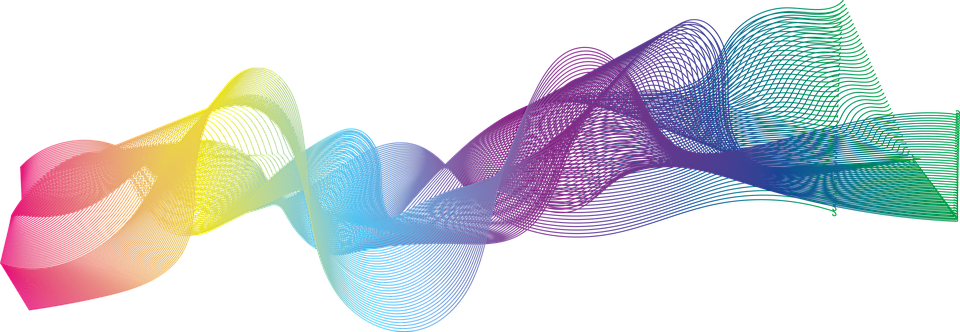
Unidad de aprendizaje: Compiladores 3CV8

Profesor: Saucedo Delgado Rafael Norman

**Proyecto: Compilador básico de lenguaje de programación C - Python**

***Alumno:*** ***García González Aarón Antonio***

Abril, 2020



Índice

[Objetivo 3](#_Toc39842583)

[Introducción 3](#_Toc39842584)

[Compilador 3](#_Toc39842585)

[Clases léxicas en lenguaje C 3](#_Toc39842586)

[Palabras clave o reservadas 3](#_Toc39842587)

[Identificadores 3](#_Toc39842588)

[Constantes 4](#_Toc39842589)

[Operadores 5](#_Toc39842590)

[Comentarios 5](#_Toc39842591)

[Desarrollo 7](#_Toc39842592)

[Gramática en BFN (Analizador sintáctico) 8](#_Toc39842593)

[Resultados 13](#_Toc39842594)

[Conclusiones 13](#_Toc39842595)

[Anexo (Código) 13](#_Toc39842596)

[Referencias 13](#_Toc39842597)

# Objetivo

El desarrollo del proyecto busca reflejar los conocimientos estudiados a lo largo del curso por medio de la implementación de un compilador básico de lenguaje de programación C a lenguaje de programación Python en las versiones 3 en adelante.

# Introducción

## Compilador

Un compilador es aquel traductor que tiene como entrada una sentencia en lenguaje formal y como salida tiene un archivo ejecutable, es decir, realiza una traducción de un código de alto nivel a código máquina (también se entiende por compilador aquel programa que proporciona un archivo objeto en lugar del ejecutable final).

## Clases léxicas en lenguaje C

Existen cinco clases de componentes sintácticos o tokens en el vocabulario del lenguaje C: palabras clave, identificadores, constantes, operadores y comentarios.

### Palabras clave o reservadas

En C existen una serie de palabras clave (keywords) que el programador no puede utilizar como identificadores (nombres de variables y/o de funciones). Estas palabras sirven para indicar al computador que realice una tarea y tienen un especial significado para el compilador. En C, existen en total 32 palabras:

auto, double, int, struct, break, else, long, switch, case, enum, register, typedef, char, extern, return, union, const, float, short, unsigned, continue, for, signed, void, default, goto, sizeof, volatile, do, if, static, while.

### Identificadores

Un identificador es un nombre con el que se hace referencia a una función o a alguna variable. En ANSI C las reglas son las siguientes:

1. Un identificador se forma con una secuencia de letras (minúsculas de la a a la z; mayúsculas de la A a la Z; y dígitos del 0 al 9).

2. El carácter subrayado o underscore (\_) se considera como una letra más.

3. Un identificador no puede contener espacios en blanco, ni otros caracteres distintos de los citados, como por ejemplo (\*,;.:-+, etc.).

4. El primer carácter de un identificador debe ser siempre una letra o un (\_). No puede ser un dígito.

5. Se hace distinción entre letras mayúsculas y minúsculas

6. ANSI C permite definir identificadores de hasta 31 caracteres de longitud.

### Constantes

Además de variables, un programa utiliza también constantes, es decir, valores que siempre son los mismos ya que una vez compilado el programa no pueden ser cambiadas.

Las hay de varios tipos.

CONSTANTES: Existen varias clases de constantes. Cada una tiene un tipo de dato. constante: constante-entera constante-de-carácter constante-flotante constante-enumeración \*Constante\_entera: Una constante entera que consiste en una secuencia de dígitos se toma como octal si inicia con 0 (cero), de otra manera es decimal. Los constantes octales no contienen los dígitos 8 6 9.

Una constante entera puede tener la letra u o U como sufijo, lo cual especifica que es unsigned. También puede tener como sufijo la letra l o L para estipular que es long.

\*Constante de carácter: Una constante dc carácter es una secuencia de uno o ms caracteres encerrados entre apóstrofos, como 'x'. El valor de una constante de carácter con un solo carácter es de valor numérico del carácter en el conjunto de caracteres de la maquina al tiempo de ejecución. Ei valor de una constante multicarácter esta definido por la implantación. Las constantes de carácter no contienen el carácter ' o nueva línea; para representarlos, así como a algunos otros caracteres, se pueden utilizar las siguientes secuencias dc escape.

A screenshot of a cell phone

Description automatically generated

\*Constantes flotantes: Una constante flotante consta de una parte entera, un punto decimal, una parte fraccionaria, una e o E, un exponente entero signado optativo y un tipo sufijo optativo entre f 0 F, 1 o L. Las partes entera y fraccionaria constan de una secuencia de dígitos. Cualquiera de las partes entera o fraccionaria (no ambas) puede omitirse; cualquiera dc las partes del punto decimal o la e y ei exponente (no ambas) pueden omitirse. Ei tipo esta determinado por el sufijo; F o f la hacen float, L o l la hacen long double; de otra manera double.

### Operadores

Los operadores en C son cadenas (de uno o dos caracteres) que indican al programa lo que debe hacer, el lenguaje es especialmente rico en estos signos:

Aritméticos

+, -, \*, / y %

Asignación

=, +=, -=, \*= y /=

Incrementales

++ y –

Relacionales

<, >, >=, <=, == y ¡=

Lógicos

&& y ||

### Comentarios

El lenguaje C permite que el programador introduzca comentarios en los ficheros fuente que contienen el código de su programa. El compilador ignora por completo los comentarios.

Los caracteres (/) se emplean para iniciar un comentario introducido entre el código del programa; el comentario termina con los caracteres (/). No se puede introducir un comentario dentro de otro. Todo texto introducido entre los símbolos de comienzo (/) y final (/) de comentario son siempre ignorados por el compilador.

El lenguaje ANSI C permite también otro tipo de comentarios, tomado del C++. Todo lo que va en cualquier línea del código detrás de la doble barra (//) y hasta el final de la línea, se considera como un comentario y es ignorado por el compilador.

# Desarrollo

## Analizador léxico

## Gramática en BFN (Analizador sintáctico)

<translation-unit> ::= {<external-declaration>}\*

<external-declaration> ::= <function-definition>

| <declaration>

<function-definition> ::= {<declaration-specifier>}\* <declarator> {<declaration>}\* <compound-statement>

<declaration-specifier> ::= <storage-class-specifier>

| <type-specifier>

| <type-qualifier>

<storage-class-specifier> ::= auto

| register

| static

| extern

| typedef

<type-specifier> ::= void

| char

| short

| int

| long

| float

| double

| signed

| unsigned

| <struct-or-union-specifier>

| <enum-specifier>

| <typedef-name>

<struct-or-union-specifier> ::= <struct-or-union> <identifier> { {<struct-declaration>}+ }

| <struct-or-union> { {<struct-declaration>}+ }

| <struct-or-union> <identifier>

<struct-or-union> ::= struct

| union

<struct-declaration> ::= {<specifier-qualifier>}\* <struct-declarator-list>

<specifier-qualifier> ::= <type-specifier>

| <type-qualifier>

<struct-declarator-list> ::= <struct-declarator>

| <struct-declarator-list> , <struct-declarator>

<struct-declarator> ::= <declarator>

| <declarator> : <constant-expression>

| : <constant-expression>

<declarator> ::= {<pointer>}? <direct-declarator>

<pointer> ::= \* {<type-qualifier>}\* {<pointer>}?

<type-qualifier> ::= const

| volatile

<direct-declarator> ::= <identifier>

| ( <declarator> )

| <direct-declarator> [ {<constant-expression>}? ]

| <direct-declarator> ( <parameter-type-list> )

| <direct-declarator> ( {<identifier>}\* )

<constant-expression> ::= <conditional-expression>

<conditional-expression> ::= <logical-or-expression>

| <logical-or-expression> ? <expression> : <conditional-expression>

<logical-or-expression> ::= <logical-and-expression>

| <logical-or-expression> || <logical-and-expression>

<logical-and-expression> ::= <inclusive-or-expression>

| <logical-and-expression> && <inclusive-or-expression>

<inclusive-or-expression> ::= <exclusive-or-expression>

| <inclusive-or-expression> | <exclusive-or-expression>

<exclusive-or-expression> ::= <and-expression>

| <exclusive-or-expression> ^ <and-expression>

<and-expression> ::= <equality-expression>

| <and-expression> & <equality-expression>

<equality-expression> ::= <relational-expression>

| <equality-expression> == <relational-expression>

| <equality-expression> != <relational-expression>

<relational-expression> ::= <shift-expression>

| <relational-expression> < <shift-expression>

| <relational-expression> > <shift-expression>

| <relational-expression> <= <shift-expression>

| <relational-expression> >= <shift-expression>

<shift-expression> ::= <additive-expression>

| <shift-expression> << <additive-expression>

| <shift-expression> >> <additive-expression>

<additive-expression> ::= <multiplicative-expression>

| <additive-expression> + <multiplicative-expression>

| <additive-expression> - <multiplicative-expression>

<multiplicative-expression> ::= <cast-expression>

| <multiplicative-expression> \* <cast-expression>

| <multiplicative-expression> / <cast-expression>

| <multiplicative-expression> % <cast-expression>

<cast-expression> ::= <unary-expression>

| ( <type-name> ) <cast-expression>

<unary-expression> ::= <postfix-expression>

| ++ <unary-expression>

| -- <unary-expression>

| <unary-operator> <cast-expression>

| sizeof <unary-expression>

| sizeof <type-name>

<postfix-expression> ::= <primary-expression>

| <postfix-expression> [ <expression> ]

| <postfix-expression> ( {<assignment-expression>}\* )

| <postfix-expression> . <identifier>

| <postfix-expression> -> <identifier>

| <postfix-expression> ++

| <postfix-expression> --

<primary-expression> ::= <identifier>

| <constant>

| <string>

| ( <expression> )

<constant> ::= <integer-constant>

| <character-constant>

| <floating-constant>

| <enumeration-constant>

<expression> ::= <assignment-expression>

| <expression> , <assignment-expression>

<assignment-expression> ::= <conditional-expression>

| <unary-expression> <assignment-operator> <assignment-expression>

<assignment-operator> ::= =

| \*=

| /=

| %=

| +=

| -=

| <<=

| >>=

| &=

| ^=

| |=

<unary-operator> ::= &

| \*

| +

| -

| ~

| !

<type-name> ::= {<specifier-qualifier>}+ {<abstract-declarator>}?

<parameter-type-list> ::= <parameter-list>

| <parameter-list> , ...

<parameter-list> ::= <parameter-declaration>

| <parameter-list> , <parameter-declaration>

<parameter-declaration> ::= {<declaration-specifier>}+ <declarator>

| {<declaration-specifier>}+ <abstract-declarator>

| {<declaration-specifier>}+

<abstract-declarator> ::= <pointer>

| <pointer> <direct-abstract-declarator>

| <direct-abstract-declarator>

<direct-abstract-declarator> ::= ( <abstract-declarator> )

| {<direct-abstract-declarator>}? [ {<constant-expression>}? ]

| {<direct-abstract-declarator>}? ( {<parameter-type-list>}? )

<enum-specifier> ::= enum <identifier> { <enumerator-list> }

| enum { <enumerator-list> }

| enum <identifier>

<enumerator-list> ::= <enumerator>

| <enumerator-list> , <enumerator>

<enumerator> ::= <identifier>

| <identifier> = <constant-expression>

<typedef-name> ::= <identifier>

<declaration> ::= {<declaration-specifier>}+ {<init-declarator>}\* ;

<init-declarator> ::= <declarator>

| <declarator> = <initializer>

<initializer> ::= <assignment-expression>

| { <initializer-list> }

| { <initializer-list> , }

<initializer-list> ::= <initializer>

| <initializer-list> , <initializer>

<compound-statement> ::= { {<declaration>}\* {<statement>}\* }

<statement> ::= <labeled-statement>

| <expression-statement>

| <compound-statement>

| <selection-statement>

| <iteration-statement>

| <jump-statement>

<labeled-statement> ::= <identifier> : <statement>

| case <constant-expression> : <statement>

| default : <statement>

<expression-statement> ::= {<expression>}? ;

<selection-statement> ::= if ( <expression> ) <statement>

| if ( <expression> ) <statement> else <statement>

| switch ( <expression> ) <statement>

<iteration-statement> ::= while ( <expression> ) <statement>

| do <statement> while ( <expression> ) ;

| for ( {<expression>}? ; {<expression>}? ; {<expression>}? ) <statement>

<jump-statement> ::= goto <identifier> ;

| continue ;

| break ;

| return {<expression>}? ;

# Resultados

# Conclusiones

# Anexo (Código)

# Referencias