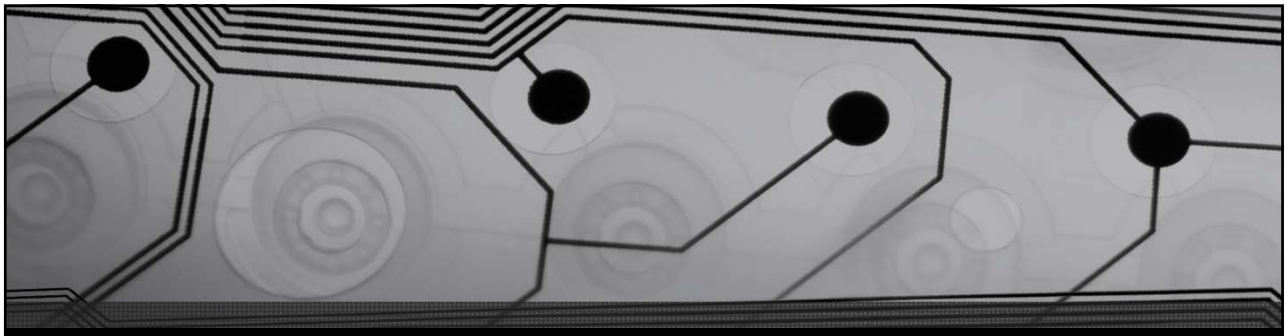
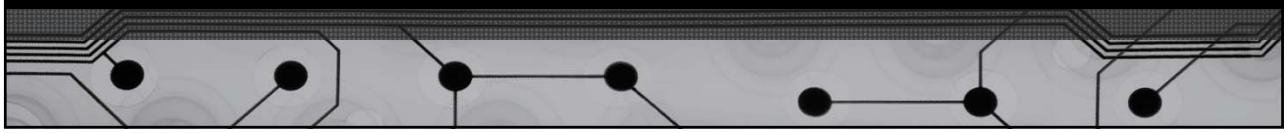


Definición de un Lenguaje de Programación

Dr. Víctor de la Cueva
vcueva@itesm.mx



Definición de un lenguaje

- La documentación para los primeros lenguajes fue escrita de una manera informal, usando el idioma inglés.
- Inmediatamente, los programadores se dieron cuenta de la necesidad de tener formas más precisas para describir un lenguaje, al punto de requerir definiciones formales como las encontradas en matemáticas.
- Sin una notación formal:
  - No se tiene una noción clara de los constructores del lenguaje y un programador no tendría idea clara de cuál es la instrucción que se está ejecutando en un momento dado.
  - No se puede razonar matemáticamente acerca de los programas, es decir, no se puede hacer verificación formal o demostraciones de su comportamiento.

Estandarización

- Uno de los objetivos de la creación de lenguajes fue lograr la independencia de la implementación sobre la máquina.
- La mejor forma de lograr esto es mediante la estandarización, la cual requiere una definición del lenguaje precisa e independiente que sea universalmente aceptada.
- Las organizaciones de estándares como la ANSI (*American National Standards Institute*) o la ISO (*International Organization for Standardization*) han publicado definiciones para muchos lenguajes incluyendo: C, C++, Ada, Common LISP y Prolog.

Preguntas difíciles

- Una razón más para la definición formal es que, en el proceso de programación, inevitablemente surgen cuestiones difíciles de contestar acerca del comportamiento y la interacción del programa.
- Los programadores necesitan formas adecuadas para contestar estas preguntas más que hacerlo a prueba y error (comúnmente usado).
- Es común que estas preguntas surjan en la etapa de diseño de un programa y que sus respuestas ocasionen cambios mayores en el diseño.

Disciplina

- Los requerimientos de un diseño formal aseguran disciplina cuando un lenguaje está siendo diseñado.
- Es muy común que los diseñadores de lenguajes no tengan en cuenta las consecuencias de ciertas decisiones de diseño hasta que se requiere producir una definición clara.
- La definición de un lenguaje puede ser dividida en dos partes:
  - Sintaxis o estructura
  - Semántica o significado

Sintaxis

- La sintaxis de un lenguaje de programación es como la gramática de un lenguaje natural.
- Es la descripción de la forma en la que diferentes partes del lenguaje pueden ser combinadas para formar frases y, finalmente, instrucciones.
- Ej, la sintaxis del estatuto `if` en C puede ser definida como:
  - Un estatuto `if` consiste de la palabra "if", seguida de una expresión entre paréntesis, seguida por un estatuto (instrucción), seguida por una parte `else` opcional, que consiste de la palabra "else" y otra instrucción.
- La descripción de la sintaxis de un lenguaje es una de las área donde las definiciones formales han ganado mucha aceptación y la sintaxis de todos los lenguajes son dadas usando una gramática.

Léxico

- La estructura léxica de un lenguaje de programación es la estructura de las palabras del lenguaje, las cuales son generalmente llamadas tokens.
- La estructura léxica es similar al deletreo en lenguaje natural.
- En el ejemplo de la instrucción `if` en C, las palabras `if` y `else` son ejemplos de tokens.
- Otros tokens en los lenguajes de programación son los identificadores (o nombres), símbolos de operaciones (operadores), y símbolos de puntuación especial (e.g. punto y coma, dos puntos, etc.)

Semántica

- La sintaxis representa sólo la estructura superficial del lenguaje y esto es sólo un pequeña parte de su definición.
- La semántica o significado del lenguaje es mucho más complicada de describir en forma precisa.
- La primera dificultad es que "significado" puede ser definido en muchas formas.
- Típicamente, la descripción del significado de una pieza de código involucra describir los efectos de la ejecución de dicho código, pero no es una forma estándar de hacerlo.
- Además, el significado de un mecanismo particular puede involucrar interacciones con otros mecanismos de tal forma que la comprensión del significado en todos los contextos se vuelve muy compleja.

El estatuto if en C

- Su semántica puede ser descrita como sigue (adaptado de Kernighan and Richie [1988]):
  - Un estatuto `if` es ejecutado primero evaluando su expresión, la cual debe tener un tipo aritmético o de apuntador, incluyendo todos los efectos colaterales, y si no es igual a 0, se ejecuta el estatuto que sigue a la expresión. Si hay una parte `else` y la expresión es 0, se ejecuta el estatuto que sigue al "else".
- Esta definición simple muestra algunas de las complicaciones de la semántica, por ejemplo, no dice qué pasa si la expresión es 0 y no hay parte `else` (presumiblemente, no pasa nada y el programa sigue su ejecución en el punto siguiente al estatuto `if`).
- Otro punto a verificar es si el estatuto `if` es "seguro".

Descripción formal para la semántica

- No existe un método formal aceptado en general (como lo son las gramáticas libres de contexto para la sintaxis) para describir la semántica.
- De hecho, todavía no es habitual una definición formal de la semántica de un lenguaje de programación.
- Sin embargo, se han desarrollado muchos sistemas de notación para definiciones formales de la semántica de un lenguaje de programación y se ha incrementado su uso:
  - Semántica operacional
  - Semántica denotacional
  - Semántica axiomática
- En muchas partes de este curso la semántica del lenguaje se dará como implícita.

Herramientas Formales

Antecedentes Formales

- Todos los lenguajes de programación actuales se desarrollan basados en la teoría de lenguajes formales (desarrollada por los científicos de la computación y lingüistas, en particular, Noam Chomsky).
- Un lenguaje formal sigue reglas matemáticas preestablecidas muy precisas y se ajustan con todo rigor a ellas:
  - Lenguajes matemáticos (álgebra)
  - Lenguajes lógicos (lógica de predicados)
  - Lenguajes de computadoras
- Gracias a esta adhesión a las reglas es posible construir traductores automáticos de los lenguajes de programación (y por eso no se cuenta con un algoritmo eficiente para el lenguaje natural).

Lenguaje formal

- Es un lenguaje (un sistema de comunicación entre dos entidades) cuyos símbolos primitivos y reglas para unirlos están formalmente especificadas.
- Al conjunto de símbolos primitivos se les llama alfabeto (o vocabulario) del lenguaje y al conjunto de reglas se le llama gramática (o sintaxis).
- A una secuencia de símbolos formada de acuerdo a la gramática se le llama fórmula bien formada (FBF o palabra) del lenguaje.
- Un lenguaje formal es idéntico al conjunto de todas sus FBF.
- El alfabeto debe ser finito y cada FBF debe tener una longitud finita, pero el lenguaje puede estar compuesto por un número infinito de FBF.

Ejemplo simple

- Alfabeto = {a, b}
- Gramática: una FBF es aquella que tenga el mismo número de a's que de b's.
- Ejemplos de FBF: ab, ba, abaabb, …
- ¿Para un lenguaje de programación?

Notas

- Para algunos lenguajes formales (como los lenguajes de programación) existe una semántica que puede interpretar y dar significado a las FBF, pero no es una condición necesaria.
- Algunos lenguajes formales permiten la palabra vacía, normalmente representada por $\in$ o por λ.
- Ejemplos de lenguajes formales:
  - Gramáticas formales
  - Expresiones regulares
  - Autómatas (e.g. Máquinas de Turing)

Herramientas

- En el caso de los lenguajes de programación se requieren herramientas formales para definir los lenguajes que ayuden a verificar un programa en:
  - Léxico
  - Sintaxis
  - Semántica
- Cada una de estas secciones utilizan ciertas herramientas formales que se adaptan mejor a sus necesidades, por ejemplo:
  - Léxico
    - Definición: Expresiones Regulares
    - Implementación: Autómatas Finitos
  - Sintaxis
    - Definición: Gramáticas Libres de Contexto

Expresiones regulares

- Definición: Sea Σ un alfabeto. Las expresiones regulares sobre Σ son definidas recursivamente como sigue:
  i. Base: λ y a, para cada $a \in \Sigma$, son expresiones regulares sobre Σ.
  ii. Paso recursivo: Sean u y v expresiones regulares sobre Σ. Las expresiones
     $(u \cup v)$ Unión
     (uv) Concatenación
     (u^*) Iteración
     Son expresiones regulares sobre Σ.
  iii. Cerradura: u es una expresión regular sobre Σ si puede ser obtenida de los elementos base por medio de un número finito de aplicaciones del paso recursivo.

Notación extra ER

- Las ER tienen tantas aplicaciones que su notación se ha ido adaptando para facilitar su uso, sobre todo con los editores de texto:
  - Unión: |
  - Una o más repeticiones: +
  - Series de caracteres: [a-z]
  - Excepto un carácter: [^a]
  - Opcional: ?

Definiciones básicas

- Un autómata finito (FA) consiste de un conjunto finito de estados y un conjunto de transiciones de un estado a otro que ocurren de acuerdo a símbolos de entrada seleccionados de un alfabeto Σ.
- Para cada símbolo de entrada hay exactamente una transición hacia afuera del estado (posiblemente de regreso al mismo estado).
- Un estado, usualmente denotado por q_o, es el estado inicial, en el cual el autómata inicia.
- Algunos estados son designados como estados finales o de aceptación.

Diagrama de Transición

- Un grafo dirigido, llamado un diagrama de transición, es asociado con un autómata finito como sigue:
  - Los vértices del grafo corresponden a los estados de un FA.
  - Si existe una transición de un estado q a un estado p en una entrada a, entonces, hay un arco etiquetado con una a, del estado q al estado p en el diagrama de transición.
  - El FA acepta un string x si la secuencia de transiciones correspondiente a los símbolos de x lleva al FA de un estado de inicio a un estado de aceptación.

Notación de una diagrama de transición

- Los estados son representados por círculos, con un nombre único.
- Hay arcos dirigidos entre los estados (flechas) que representan transiciones entre los estados.
  - Los arcos son etiquetados por "entradas", las cuales representan influencias externas al sistema.
- Uno de los estados es designado como el estado de inicio (*start state*) y representa el estado en el cual el sistema es colocado inicialmente.
  - Por convención, el estado inicial se indica con una flecha que apunta hacia dicho estado y que se etiqueta con la palabra "Start".
- Normalmente, es necesario indicar uno o más estados "finales" o de "aceptación", los cuales se representan con un doble círculo.
  - El entrar en uno de los estados finales, después de una secuencia de entradas, indica que la secuencia de entrada es "buena", en algún sentido.

Gramáticas formales

- Las gramáticas formales normalmente se representan por medio de producciones. Ej:

  $A \rightarrow bA$

  $A \rightarrow c$

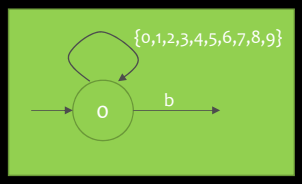
- En el caso de los lenguajes de programación hay un lenguaje formal para definir los tokens (léxico) y otro para definir las FBF (sintaxis).
- Además, hay una semántica que define el significado de las FBF para poder generar código ejecutable.

Gramáticas Libres de Contexto (CFG)

- Una CFG o simplemente gramática se denota como G=(V,T,P,S), donde V y T son conjuntos finitos de variables y terminales, respectivamente. Se asume que V y T son disjuntos. P es un conjunto finito de producciones; cada producción es de la forma $A \rightarrow \alpha$, donde A es una variables y α es un string de símbolos de $(V \cup T)^*$. Finalmente, S es una variable especial llamada símbolo de inicio.
  - Ej. La gramática de las operaciones aritméticas es ({E},{+,*,(,),**id**},P,E)
  - El que el LI tenga una sola variable le da el nombre de "Libre de Contexto".

Reconociendo tokens

- En los compiladores, los tokens son reconocidos normalmente usando un autómata (para describir expresiones regulares).
- Ej. Reconocimiento de números enteros:
  - Autómata para números enteros:



  - Expresión regular $\{0,1,2,3,4,5,6,7,8,9\}^+$, que en formato general se pone $[0\text{-}9]^+$.
- Cualquier de estos dos se implementa computacionalmente usando una tabla que se forma de la función de transición del autómata.

Ejemplo

- Hacer un autómata que detecte sumas y restas de números enteros y reales sin signo
- Hacer la tabla de transición que representa dicho autómata
- Algunas expresiones que se aceptarían son:
  - 43 + 22
  - 29.47 – 89.524
  - 67 + 74.213

pizarrón

Especificación de la sintaxis

- La sintaxis de un lenguaje especifica cómo están construidos los programas de dichos lenguajes.
- Casi siempre se especifican usando alguna variante de las gramáticas libres de contexto (GLC), por lo que se definen mediante producciones:
  - Forma Backus-Naur (BNF)
  - Forma Backus-Naur Extendida (BNFE)
  - Esquemas de Sintaxis (formato gráfico de la BNF)

Gramáticas Libres de Contexto

- Def. Una GLC o simplemente gramática, consta de 4 partes:
  1. Un conjunto de símbolos terminales
  2. Un conjunto de símbolos no terminales
  3. Un conjunto de producciones, donde cada producción tiene solamente un símbolo no terminal del lado izquierdo (libre de contexto), un símbolo de producción (normalmente $\rightarrow$ o ::=), y del lado derecho, una cadena construida con los conjuntos de símbolos terminales o no terminales.
  4. Un símbolo no terminal especial designado como símbolo no terminal inicial (generalmente se el primero que aparece en la primera producción o el símbolo S).

  A menos que se diga otra cosa, las producciones del símbolo no terminal inicial son las primeras que aparecen.

Otros conceptos

- Dado un conjunto de símbolos:
  - Cadena: es una secuencia finita de 0 o más símbolos.
  - Longitud de la cadena: el número de símbolos en la secuencia.
  - Los símbolos atómicos de un lenguaje se conocen como componentes léxicos (tokens) o símbolos terminales (*terminals*).
  - Los constructores de un lenguaje están representados por símbolos no terminales.
  - El símbolo no terminal que representa al constructor principal de un lenguaje se llama símbolo no terminal inicial.
  - Los componentes de un constructor se identifican con reglas llamadas producciones.

Forma de Backus-Naur (BNF)


- El concepto de Gramática Libre de Contexto es independiente de la notación utilizada para escribir gramáticas.
- En BNF [John Backus, 1960]:
  - Los símbolos no terminales se encuentran entre braquets '<' y '>'
  - La cadena vacía se representa como <vacía>
  - Los símbolos terminales van sueltos
  - ::= representa la flecha de producción de las GLC
  - | indica opciones, es decir, una u otra
- Peter Naur la complementó con varias mejoras en 1963.

Fuente de fotos:
https://en.wikipedia.org/wiki/John_Backus
https://en.wikipedia.org/wiki/Peter_Naur

Ejemplo: BNF para números reales

NOTA: sin notación exponencial

<número-real> ::= <secuencia-dígitos> . <secuencia-dígitos>

<secuencia-dígitos> ::= <dígito> | <dígito> <secuencia-dígitos>

<dígito> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

::= significa "es" o "produce"

| significa "o"

Ejemplo

- Hacer una GLC en BNF que defina expresiones aritméticas:
  - Los operadores son +, -, / y *
  - Se pueden usar paréntesis
  - Los operandos son Números o Variables (por ahora las manejamos sin definición)

Pizarrón

Una posible solución

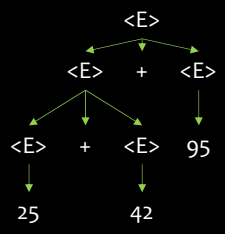
<expresión> ::= <expresión> + <expresión> |
<expresión> - <expresión> |
<expresión> * <expresión> |
<expresión> / <expresión> |
(<expresión>) | <elemento>

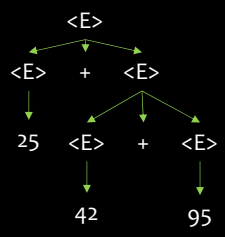
<elemento> ::= <variable> | <constante>

El problema con esta solución es que es una gramática ambigua, es decir, produce más de un árbol de parseo para la misma entrada

Árbol de parseo

- Es el árbol que resulta de aplicar la sintaxis defina en BNF para derivar una entrada dada.
- Ejemplo: Genere el árbol para la expresión 25 + 42 + 95





Cuando la gramática produce Más de un árbol de parseo se Dice que es ambigua

Ambigüedad

- Para efectos de la implementación de un analizador sintáctico basado en la representación BNF de un lenguaje de programación, se requiere que la representación no sea ambigua
- Existen varias formas de quitar la ambigüedad, que fueron tratadas en Teoría de la Computación

NOTA: Otra característica importante para varias de las implementaciones de un analizador sintáctico es que la BNF no tenga recursión por la izquierda sino por la derecha (para lo cual también hay varias técnicas que verán en Compiladores)

Ejemplo: BNF para expresiones aritméticas

<expresión> ::=	<expresión> + <término> |
	<expresión> - <término> |
	<término>
<término> ::=	<término> * <factor> |
	<término> / <factor> |
	<factor>
<factor> ::=	(<expresión>) |
	<variable> |
	<contantes>

Faltarían las definiciones de:

<variable>

<constante>

las cuales se pueden tomar como resueltas por la parte léxica.

BNF Extendida (BNFE)

- El uso de braquets complica el entendimiento de una gramática escrita en BNF.
- Por otro lado, se sabe que las Expresiones Regulares logran representar muchas veces lo mismo con una notación más compacta.
- BNFE es BNF con expresiones regulares.
- Sus reglas son:
  - Los Símbolos No Terminales inician con MAYÚSCULA.
  - Los terminales de un solo caracter se colocan entre comillas simples (e.g. '+', '-')
  - Los terminales en negritas aparecen tal cual.
  - |, representa una opción.
  - ( ), se usan para agrupar.
  - { }, representan cero o más repeticiones.
  - [ ], representan una construcción opcional.

Ejemplo

- Hacer la BNFE para la gramática de las expresiones vista anteriormente, tanto la ambigua como la no ambigua

pizarrón

Ejemplo: BNFE para expresiones aritméticas

Expresión ::= Término { ('+' | '-') Término }

Término::= Factor { ('*' | '/') Factor }

Factor ::= '(' Expresión ')' | Variable | Constante

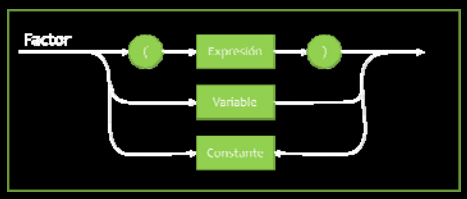
NOTA: Las palabras reservadas del lenguaje aparecerían en negritas.

Esquemas o Diagramas de Sintaxis

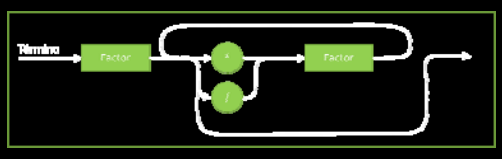
- Es el equivalente gráfico a la BNFE, por lo que generalmente se escribe a partir de ella.
- Constituye otra forma para escribir una gramática y se construye de la siguiente manera:
  - Existe un diagrama para cada símbolo no terminal.
  - Cada producción para un símbolo no terminal se transforma en una trayectoria a través del diagrama.
  - Los símbolos terminales se encuentran en rectángulos redondeados y los no terminales en rectángulos.
  - Las llaves, que indican cero o más repeticiones, sugieren diagramas con ciclos.
  - Una forma sintetizada de construir diagramas con ciclos es dibujando un ciclo alrededor del diagrama que representa esa cadena.

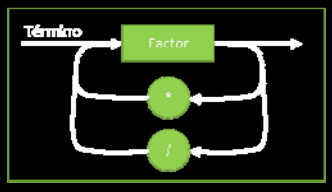
Ejemplo: Diagrama de sintaxis

- Para el No Terminal Factor:



- { (* | /) Factor } es un ciclo en Término::=Factor { (‘*’ | ‘/’) Factor }





Ejercicios

- Genere la sintaxis en BNF, BNFE y DS para los siguientes problemas:
  - Una lenguaje especial [Sebasta, 2012]:
    - Un lenguaje que tiene solo una instrucción: la asignación.
    - Un programa consiste de la palabra reservada **begin**, seguida de una lista de instrucciones separadas por punto-y-coma, seguida de la palabra reservada **end**.
    - Una expresión es una sola variable o dos variables separadas por el operador + o -.
    - Los únicos nombres de variables permitidos son A, B y C.
  - El estatuto **for** de Java
  - El estatuto **switch** de C++

Parseadores

- Actualmente hay unos programas que realizan el parseo de sintaxis dada una gramática, o de léxico dadas las expresiones regulares
- Uno de los más famosos es YACC (*Yet Another Compiler-Compiler*) o su versión gratuita llamada Bison
- También hay paseadores de sólo léxico (tokenizadores) como:
  - LEX (*A Lexical Analyzer Generator*): genera el parseador en C
  - JAX (): general el parseador en Java
  - Flex: es una alternativa (gratuita) a Lex y también genera el autómata en C

Semántica

- Para los lenguajes de programación la semántica define el significado de las FBF, es decir, el significado del programa (para poder generar código ejecutable).
- Ejemplo: Fechas

Sintaxis <fecha> ::= DD/DD/DDDD

donde / es un separador (no un 'o') y D es la abreviación de <dígito>

Si esto está definido entonces una entrada como 01/02/2016 se interpreta como una fecha, sin embargo, no se sabe si el 01 es el día o el mes.

- Esta sintaxis tiene semánticas diferentes en diferentes partes del mundo.

NOTA

- Es indispensable contar con descripciones completas y claras de la sintaxis y de la semántica de un lenguaje, para asegurar que todas las implementaciones de dicho lenguaje acepten los mismos programas.
- De lo contrario, la portabilidad seria imposible.

Tarea 1

Criterios de diseño de un lenguaje

Criterios de diseño de un lenguaje

- ¿Qué hace a un lenguaje de programación exitoso?
- ¿Con qué criterios lo podríamos juzgar?
- Anteriormente comentamos que los criterios claves son:
  - Legibilidad (que se lea fácilmente)
  - Mecanismos para lograr abstracción
  - Control de la complejidad
- Pero es sumamente complicado analizar éxito o fracaso por estos puntos.
- Por ahora, podemos decir que un lenguaje es exitoso si satisface alguno o todos de los siguientes criterios:
  - Alcanza las metas de sus diseñadores.
  - Logra ser muy usado en alguna área de aplicación.
  - Sirve como modelo para otros lenguajes que también son exitosos.

Algunos factores prácticos

- Que no están directamente conectados a la definición del lenguaje también tienen un gran efecto en el éxito o fracaso:
  - Disponibilidad
  - Precio
  - Calidad de los traductores
  - Política
  - Geografía
  - Sincronización
  - Mercados

Ejemplos

- C, por el éxito de UNIX
- COBOL, (ignorado por los científicos de la computación) por la permanencia de sus aplicaciones.
- Ada, porque su uso era requerido en proyectos del Departamento de Defensa de USA.
- Java y Python, por el crecimiento de Internet y la distribución gratuita de sus ambientes de programación.
- Smalltalk, nunca tuvo un gran uso pero todos los OOL tienen un gran número de características de él.

Grupos

- Algunos diseñadores de lenguajes argumentan que un individuo o un grupo pequeño tienen mayor oportunidad de crear un lenguaje exitoso porque imponen un concepto de diseño uniforme.
  - Pascal, C, C++, APL, SNOBOL y LISP, fueron diseñados así y fueron exitosos
  - COBOL, Algol y Ada, fueron diseñados por grandes comités y también fueron exitosos

Objetivo en mente

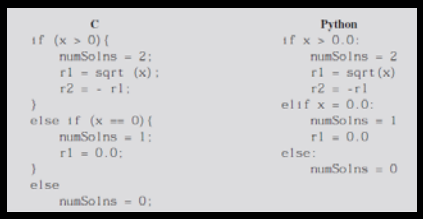
- Cuando se diseña un nuevo lenguaje es esencial decidir una meta general para el lenguaje y mantenerla en mente durante todo el proceso de diseño.
- Esto es particularmente importante para lenguajes de propósito especial (e.g. Lenguajes de bases de datos, Lenguajes gráficos, Lenguajes de tiempo real, etc.), porque las abstracciones particulares del área de aplicación objetivo deben ser construidas en el diseño del lenguaje.
- También esto es cierto para los lenguajes de propósito general:
  - FORTRAN, se centró en una ejecución eficiente.
  - COBOL, en una legibilidad no técnica tipo inglés.
  - Algol60, para proporcionar un lenguaje estructurado en bloques para describir algoritmos.
  - Pascal, para proporcionar un lenguaje instruccional simple para promover el diseño top-down.
  - C++, en las necesidades del usuario de grandes abstracciones, conservando la eficiencia y compatibilidad con C.

Muy complicado

- La realidad es que es sumamente complicado establecer algunos criterios generales.
- Incluso los grandes diseñadores de lenguajes y científicos de la computación no han logrado ponerse de acuerdo.
- Desde luego que se pueden especificar algunos criterios generales, siempre y cuando se tenga en mente que no son una regla que se debe seguir "como manual de operación":
  - Eficiencia
  - Regularidad
  - Seguridad
  - Extensibilidad

Eficiencia

- La eficiencia se puede ver desde diferentes puntos de vista:
  - Eficiencia en el código para correr más rápido
    - traductores eficientes
    - Tipado estático
    - Asignación de memoria estática (arreglos fijos y no recursión)
  - Eficiencia en la programación
    - Qué tan rápido puede una persona leer y escribir un programa
    - Está afectada por la expresividad del lenguaje (e.g. mecanismos de abstracción, mecanismos de control estructurado)
    - Sintaxis simple
  - Confiabilidad. Si no lo es se incurre en costos adicionales para garantizar buen comportamiento.



C

```
if (x > 0){
    numSolns = 2;
    r1 = sqrt (x);
    r2 = - r1;
}
else if (x == 0){
    numSolns = 1;
    r1 = 0.0;
}
else
    numSolns = 0;
```

Python

```
if x > 0.0:
    numSolns = 2
    r1 = sqrt(x)
    r2 = -r1
elif x = 0.0:
    numSolns = 1
    r1 = 0.0
else:
    numSolns = 0
```

Regularidad

- Es el concepto más importante para lograr que los programas sean legibles y mantenibles (entendibles).
- Se refiere a qué tan bien son integradas las características de un lenguaje.
- A mayor regularidad menos:
  - Restricciones inusuales en los constructores
  - Interacciones extrañas entre constructores
  - Menos sorpresas en el comportamiento de las características del lenguaje
- Se divide en 3 componentes:
  - Generalidad
  - Diseño ortogonal
  - Uniformidad

Generalidad y diseño ortogonal

- Se logra la generalidad evitando casos especiales en la disponibilidad y uso de los constructores y combinando constructores muy relacionados en uno solo.
  - En C, el operador == no es general ya que no se pueden comparar arreglos.
- Un diseño ortogonal permite a los constructores del lenguaje ser combinados en cualquier forma válida, sin comportamientos o restricciones inesperadas que surjan de la interacción de constructores o del contexto de uso.
  - Tipos que puede regresar una función: En Pascal las funciones sólo pueden regresar escalares o apuntadores. En C y C++, se pueden regresar valores de cualquier tipo excepto arreglos. En Ada, Python y la mayoría de los lenguajes funcionales son ortogonales.
  - Lugares de declaración de variables. En C sólo al inicio.
  - Tipos de referencia y primitivos. En Java, los tipos primitivos (char, int, float, etc.) no son objetos.

Uniformidad

- Se refiere a que cosas similares, se ven similares y se comportan en forma similar. De la misma forma, las cosas distintas se ven distintas.
- Es decir, se refiere a la consistencia en apariencia y comportamiento para constructores similares.
  - En C++, se requiere un ; al final de la declaración de una clase pero se prohíbe al final de la declaración de una función.
  - En Pascal, el valor de regreso de una función se da por medio de una asignación := (cosas diferentes y se ven iguales). La mayoría de los lenguajes usan una instrucción *return*.

Seguridad

- La confiabilidad de un lenguaje de programación se pude ver seriamente afectada si no se ponen ciertas restricciones a ciertas características:
  - En C, los apuntadores están muy poco restringidos y esto los hace más propensos al mal uso y al error.
  - En Java se eliminaron los apuntadores (quedan implícitos en la creación de objetos).
- Un lenguaje diseñado con seguridad:
  - Evita que los programadores cometan errores
  - Permite la detección y reporte de errores
- Si la seguridad es la meta normalmente se fuerza al programador a tener mucha especificidad:
  - LISP y Python decidieron que la declaración de variables y el tipado estático hacían difícil la codificación pero, por otro lado, las aplicaciones industriales, comerciales o de defensa requieren un lenguaje que tenga características adicionales para la seguridad.
  - El punto fundamental es cómo crear lenguajes que sean seguros y maximicen la expresividad y la generalidad. Ejemplos de avances en esta dirección son ML y Haskell (funcionales, sin declaración y con tipado estático).

Semánticamente seguro

- El tipado estático es sólo un componente de la seguridad.
- Muchos lenguajes modernos como Python, LISP y Java, van más allá de esto y son considerados semánticamente seguros.
- Significa que previenen al programador de compilar y ejecutar estatutos o expresión que violen la definición del lenguaje.
- C y C++ no son semánticamente seguros.
  - El error de "índice fuera de límite", causa un error de compilación o de ejecución en Python, LISP y Java, pero puede no detectarse en C y C++.
  - *Memory leaks* en C y C++ porque el programador falla al reciclar el almacenamiento dinámico. En Java, Python y LISP, el recolector de basura evita que ocurra.

Extensibilidad

- Un lenguaje extensible permite el uso de características adicionales a las de él.
- La mayoría de los lenguajes son extensibles hasta cierto punto:
  - Permiten que el programador defina nuevos tipos de datos y nuevas operaciones.
  - Algunos lenguajes permiten al programador incluir estos recursos en unidades tales como paquetes o módulos.
- Los diseñadores de lenguajes modernos permite extensiones al lenguaje mediante nuevas versiones, las cuales son, típicamente compatibles hacia atrás (no afectan a código legalmente escrito anteriormente).
- Algunos pocos lenguajes permiten al programador agregar nueva sintaxis y semántica (e.g. LISP).

Referencias

- R. Sethi. Programming Languages: concepts and constructs. Addison-Wesley, 2nd edition (1996).
- K.C. Louden and K. A. Lambert. Programming Languages: Principles and Practice. Cengage 3rd edition (2011).
- R.W. Sebasta. Programming Languages. 3rd ed. Pearson (2012).