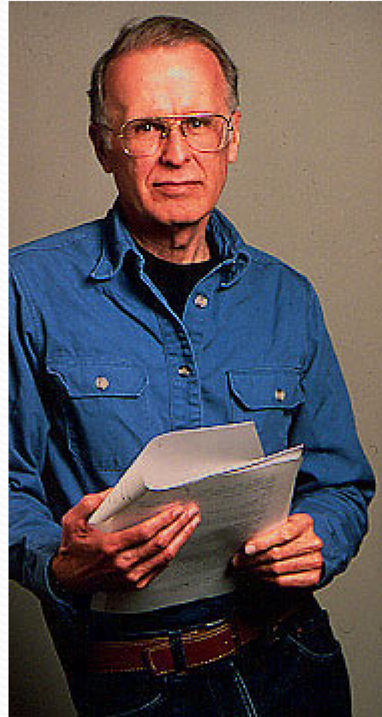
Lenguajes de Programación

Dr. Carlos A. Coello Coello
Departamento de Computación
CINVESTAV-IPN
ccoello@cs.cinvestav.mx

BNF



- John Backus sugirió en los 1950s una manera formal de describir la sintaxis de un constructor en una parte del reporte original de FORTRAN I.

BNF

- Peter Naur adaptó la notación de Backus al reporte de ALGOL-60, haciéndole una serie de mejoras. Esto condujo a la notación denominada "**Backus-Naur Form**" (BNF) que es tan común en nuestros días.

BNF

```
<postal-address> ::= <name-part> <street-address> <zip-part>

     <name-part> ::= <personal-part> <last-name> <opt-suffix-part> <EOL>
                   | <personal-part> <name-part>

 <personal-part> ::= <first-name> | <initial> "."

<street-address> ::= <house-num> <street-name> <opt-apt-num> <EOL>

      <zip-part> ::= <town-name> "," <state-code> <ZIP-code> <EOL>

<opt-suffix-part> ::= "Sr." | "Jr." | <roman-numeral> | ""
```

- La notación BNF es realmente un meta-lenguaje, porque se le usa para describir otro lenguaje (normalmente, un lenguaje de programación tal como ALGOL, Pascal, C, etc.).

BNF

BNF (Backus-Naur Form)

```
(2.0 * PI) / n

<expression>  ::=  <expression> + <term>
                |  <expression> - <term>
                |  <term>
<term>        ::=  <term> * <factor>
                |  <term> / <factor>
                |  <factor>
<factor>      ::=  number
                |   name
                |   ( <expression> )
```

CSE 341, S. Tanimoto Concepts 1- 5

- En términos llanos, la notación BNF es una nomenclatura que nos permite efectuar una descripción compacta y precisa de los constructores sintácticos usando ciertos símbolos y reglas.

BNF

- Veamos cómo funciona la notación BNF. Supongamos que queremos describir los números enteros. En notación BNF, la descripción sería la siguiente:

<integer> **::= + <unsigned integer>**
| - <unsigned integer>
| <unsigned integer>

- NOTA: Usé líneas diferentes en aras de mejorar la legibilidad. El símbolo “**::=**“ se puede leer como “**se define como**”. La barra | se interpreta como un “OR”.

BNF

- La definición anterior no deja claro cómo se pueden especificar números de varios dígitos. Esto se puede especificar si definimos “unsigned integer”:

<unsigned integer> ::= <dígito>
| <unsigned integer> <dígito>

Nótese cómo esta definición en notación BNF es recursiva.

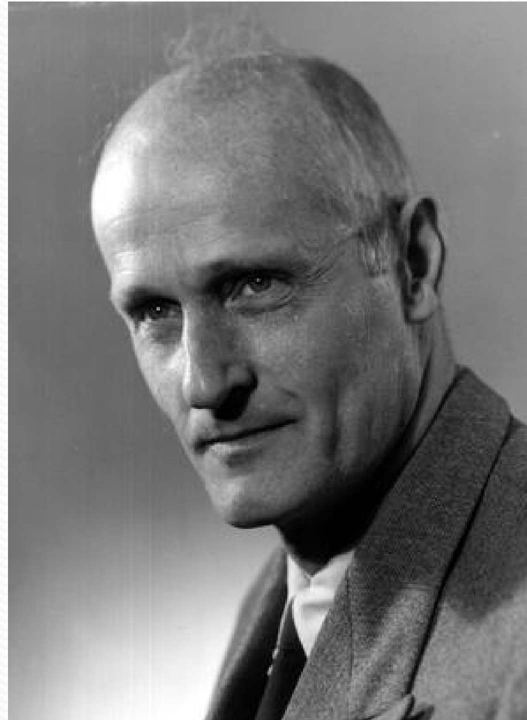
Notación BNF Extendida

- Se han propuesto varias extensiones a la notación BNF, que buscan mejorar su legibilidad. Por ejemplo, en vez de usar la definición recursiva que dimos anteriormente para "unsigned integer", podríamos usar la siguiente notación:

<unsigned integer> ::= <digit>$^{+}$

En este caso, utilizamos la **cruz de Kleene** (C^{+}), que se refiere a que la secuencia de la clase C ocurre una o más veces.

Notación BNF Extendida



- La cruz de Kleene toma su nombre del lógico Stephen Cole Kleene. También suele usarse la **estrella de Kleene** (C^*) que se refiere a la secuencia de la clase C ocurre cero o más veces.

Notación BNF Extendida

- Un ejemplo de uso de la estrella de Kleene es el siguiente:

<alfanumérico> ::= <letra> | <dígito>
<identificador> ::= <letra> <alfanumérico>*

Si <alfanumérico> se usa sólo en un lugar, entonces no tiene caso darle un nombre y sería preferible decir directamente que un identificador es una secuencia de <letra>s o <dígito>s.

Notación BNF Extendida

- Lo anterior puede expresarse de la siguiente manera en notación BNF extendida:

<identificador> ::= <letra> {<letra> | <dígito>}*

Esto puede leerse como que un "identificador" se forma con una letra, seguida por una secuencia de cero o más letras o dígitos y también puede escribirse como:

<identificador> ::= <letra> $\begin{Bmatrix} < letra > \\ < dígito > \end{Bmatrix}^{*}$

Notación BNF Extendida

- Otra posible simplificación es el uso de los corchetes rectangulares para denotar posibles alternativas que son opcionales. Por ejemplo, en vez de escribir:

<integer> ::= + <unsigned integer>
| - <unsigned integer>
| <unsigned integer>

Usaríamos:

<integer> ::= $\begin{bmatrix} + \\ - \end{bmatrix}$**<unsigned integer>**

Notación BNF Extendida

- Otro ejemplo es el de la definición de **número**:

<unsigned integer> ::= <dígito>$^{+}$

<integer> ::= $\begin{bmatrix} + \\ - \end{bmatrix}$ <unsigned integer>

<número decimal> ::= $\begin{Bmatrix} < unsigned\ integer > \\ [< unsigned\ integer >]. < unsigned\ integer > \end{Bmatrix}$

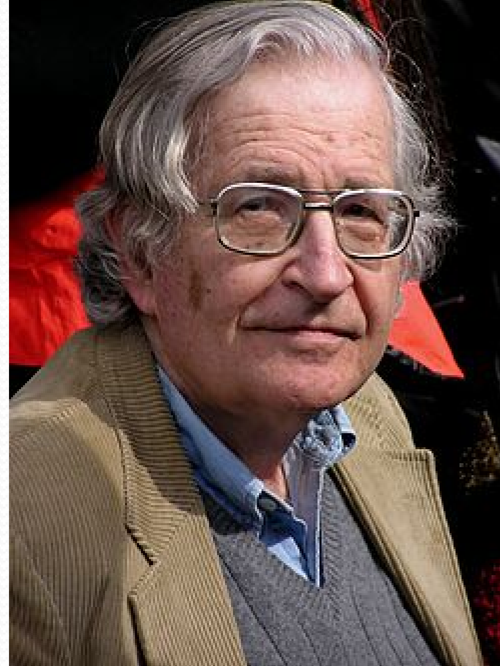
<número> ::= $\begin{bmatrix} + \\ - \end{bmatrix} \begin{Bmatrix} < número\ decimal > \\ [< número\ decimal >]_{10}\ < integer > \end{Bmatrix}$

Notación BNF Extendida

- Este mismo ejemplo sería un poco más complejo en notación **BNF pura**:

<unsigned integer> ::= <dígito> | <unsigned integer> <dígito>
<integer> ::= + <unsigned integer>
            | - <unsigned integer>
            | <unsigned integer>
<fracción decimal> ::= . <unsigned integer>
<exponente> ::= $_{10}$ <integer>
<número decimal> ::= <unsigned integer> | <fracción decimal>
                    | <unsigned integer> <fracción decimal>
<unsigned number> ::= <número decimal> | <exponente>
                    | <número decimal> <exponente>
<número> ::= + <unsigned number> | - <unsigned number>
            | <unsigned number>

Notación BNF Extendida



- A fines de los 1950s, Noam Chomsky, un lingüista del MIT, intentaba desarrollar una teoría matemática de los lenguajes naturales.

Notación BNF Extendida

- Chomsky produjo una descripción matemática de cuatro clases diferentes de lenguajes, que se conocen ahora como:

- **Chomsky tipo cero**, o *lenguajes recursivamente enumerables*.
- **Chomsky tipo uno**, o *lenguajes sensibles al contexto*.
- **Chomsky tipo dos**, o *lenguajes libres de contexto*.
- **Chomsky tipo tres**, o *lenguajes regulares*.

Cada una de estas clases, incluye a la anterior. Por ejemplo, todos los lenguajes regulares, son libres de contexto y todos los lenguajes libres de contexto son sensibles al contexto.

Notación BNF Extendida

- Chomsky también describió gramáticas, o descripciones de lenguajes, que corresponden a cada una de las clases en su jerarquía. Por ejemplo, una gramática libre de contexto, describe un lenguaje libre de contexto y viceversa.
- Se hace notar, sin embargo, que pueden existir varias formas posibles de describir el mismo lenguaje, lo que significa que pueden haber varias gramáticas correspondientes al mismo lenguaje.

Notación BNF Extendida

- Los científicos no tardaron en percatarse que la notación BNF era equivalente a las gramáticas libres de contexto (clase 2), por lo cual, describen lenguajes libres de contexto. El trabajo matemático que Chomsky había realizado previamente, permitió realizar un análisis matemático de la sintaxis y la gramática de los lenguajes de programación.

- Esto trajo muchos beneficios a la computación, puesto que permitió la generación de generadores automáticos de evaluadores de expresiones (*parsers*), que solía ser la parte más difícil del diseño de un compilador.

Gramáticas Regulares vs. Libres de Contexto

- Aunque un estudio detallado de la jerarquía de Chomsky, está más allá de los alcances de este curso, veremos rápidamente los 2 tipos de gramáticas que se usan más comúnmente en lenguajes de programación: las regulares y las libres de contexto.

- Una **gramática regular** es aquella que puede escribirse en notación BNF extendida sin usar reglas recursivas. Por ejemplo, la definición de **<número>** que vimos anteriormente en notación BNF extendida.

Gramáticas Regulares vs. Libres de Contexto

- Por lo tanto, un **lenguaje regular** es aquel que puede ser descrito por una **gramática regular**.

- Una gramática libre de contexto es aquella que, al expresarse en notación BNF extendida, requiere de reglas recursivas.

- Evidentemente, un **lenguaje libre de contexto** es aquel que puede ser descrito por una **gramática libre de contexto**.

Gramáticas Regulares vs. Libres de Contexto

- ¿Qué implicaciones prácticas tiene esto?
- El uso de reglas recursivas permite la definición de estructuras sintácticas anidadas. De esto se desprende que las gramáticas regulares no permiten especificar estructuras anidadas indefinidamente. Se hace notar, sin embargo, que si el anidamiento no es indefinido (o sea, si está limitado a ocurrir un cierto número finito de veces), no se requieren reglas libres de contexto, y una gramática regular sería suficiente.

Gramáticas Regulares vs. Libres de Contexto

- Un ejemplo es el uso de expresiones basadas en el uso de paréntesis balanceados:

(((()))()((()())))

Para este tipo de expresiones, se requiere una **gramática libre de contexto**:

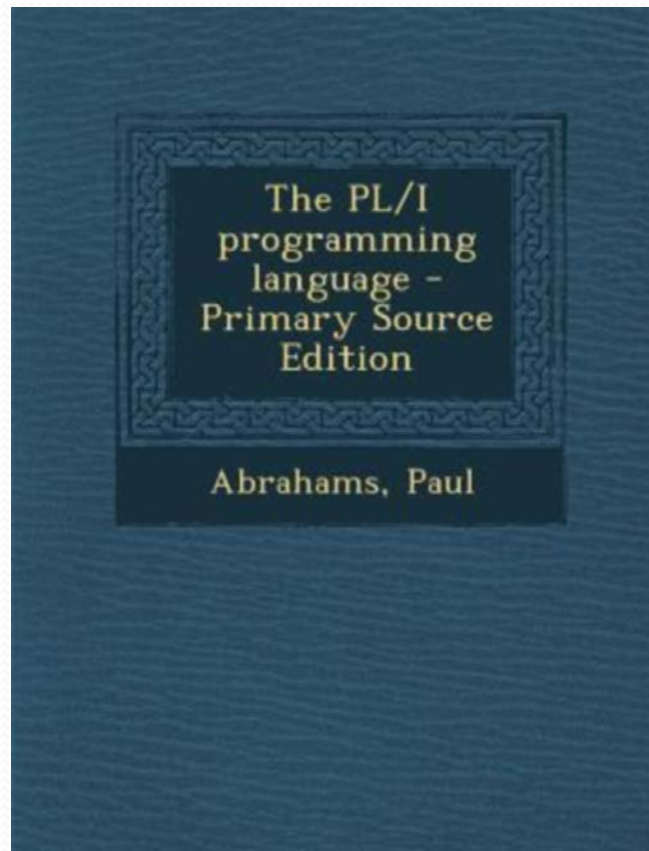
<expresión> ::= <balanceado>*

<balanceado> ::= (<expresión>)

Pascal

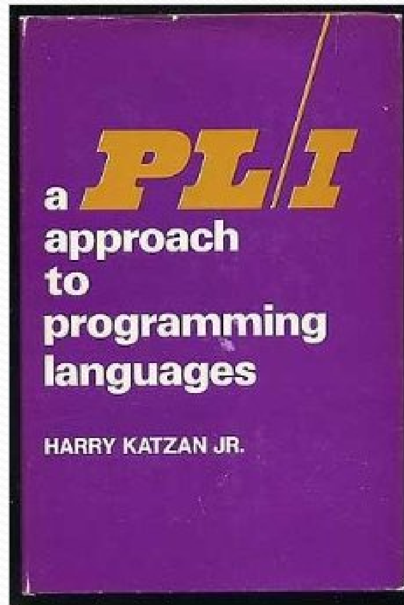
- Aunque ALGOL nunca se volvió un lenguaje muy popular, su influencia fue enorme sobre los lenguajes imperativos (o sea, estructurados) que le sucedieron.
- El PL/I fue un intento fallido de IBM por combinar ALGOL, FORTRAN y COBOL en un solo lenguaje de programación.

Pascal



- El resultado fue un auténtico desastre, porque este lenguaje tenía demasiadas interacciones de funciones, además de ser enorme y difícil de controlar.

Pascal



- Consecuentemente, el PL/I nunca se volvió muy popular y ha sido criticado constantemente por la mayor parte de los diseñadores de lenguajes debido a su horrible estructura y su tremenda redundancia.

Pascal

- Otro enfoque distinto de diseño fue el denominado “lenguajes expandibles”, el cual consistía en desarrollar un “kernel” o lenguaje base y un mecanismo que permitiera extenderlo hacia el tipo de aplicación que se deseara.

- Habían varios tipos posibles de extensión, tales como la “extensión de operadores” y las “macros sintácticas”.

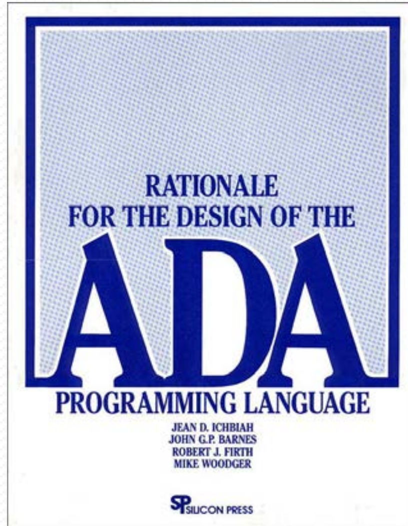
Pascal

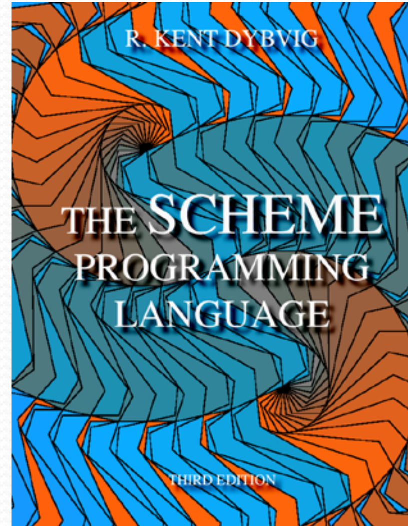
- En el primer caso, podían definirse nuevos operadores de acuerdo a las necesidades del usuario.
- En el segundo casi, podía introducirse una nueva sintaxis en el lenguaje.
- El problema principal con los lenguajes expandibles era su ineficiencia, debida al hecho de que las extensiones no se implementaban directamente en el compilador y por lo tanto había un costo de indirección asociado a ellas.

Pascal

- Otro problema era el diagnóstico pobre de los lenguajes expandibles.
- Esto se debía a que el chequeo de errores lo hacía el kernel del lenguaje.
- Debido a esto, la mayor parte de los diagnósticos se realizaban en términos de constructores del lenguaje usado por el kernel, el cual resultaba sumamente confuso para el usuario.

Pascal





- Aunque los lenguajes expandibles no lograron sobrevivir hasta nuestros días, algunos lenguajes de programación tales como Ada y Scheme todavía incorporan extensiones a los operadores y macros sintácticas como mecanismos alternativos para extender el lenguaje.

Pascal

- Niklaus Wirth empezó a trabajar en su propia implementación de ALGOL (llamada Algol-W) en 1968 y para 1970 ya tenía un compilador completamente funcional.

Pascal

- Las ideas originales de Wirth en torno a posibles extensiones al ALGOL fueron rechazadas por el comité que desarrolló este lenguaje.
- En vez de hacerle caso a Wirth, el comité optó por desarrollar un lenguaje más grande, más sutil y excesivamente complejo, al que denominaron ALGOL-68.

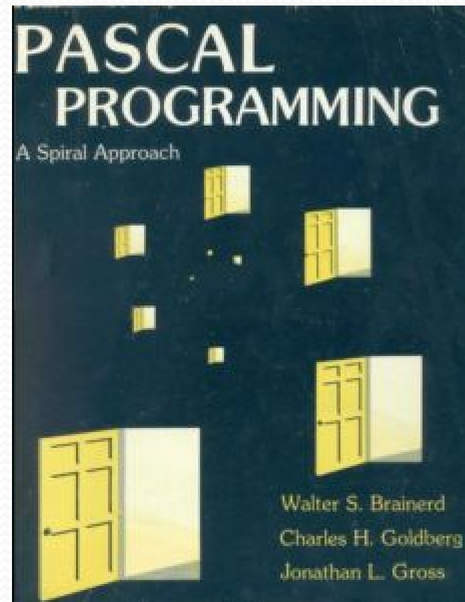
Pascal

- Sin embargo, Algol-W fue implementado en la Universidad de Stanford y usado durante varios años para la enseñanza.

Pascal

- El lenguaje fue renombrado **Pascal** y fue revisado ligeramente en 1972, volviéndose un estándar internacional en 1982.

Pascal



- Aunque eventualmente fue desplazado por C/C++, Pascal fue un lenguaje muy importante en la enseñanza de programación y diseño de compiladores, usándosele por muchos años en todo el mundo.

Pascal

- Las metas principales del diseño de Pascal fueron:
  - El lenguaje debía ser adecuado para enseñar programación en una forma sistemática.
  - La implementación del lenguaje debía ser confiable y eficiente, tanto en tiempo de compilación como en tiempo de ejecución.
  - Dicha eficiencia estaba especificada de acuerdo al equipo disponible en la época (fines de los 1960s).

Pascal

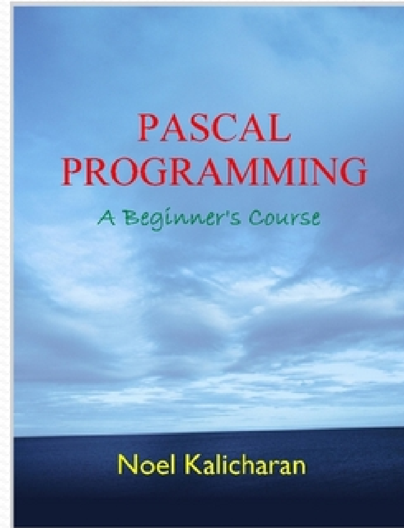
- Pascal es un lenguaje muy simple y general. Es algo así como un ALGOL sin ningún exceso, ya que todas las cosas barrocas de este lenguaje se removieron de Pascal.

- Asimismo, cuenta con algunas estructuras de datos adicionales, pero éstas tienen como fin facilitar y hacer más eficiente el lenguaje.

Pascal

- El punto más fuerte de Pascal es su simplicidad y a ella se atribuye su éxito prolongado a lo largo de tantos años en todo el mundo.
- Pascal tiene una sintaxis muy similar a la de ALGOL, pero realiza importantes adiciones a los mecanismos de identificación, datos y estructuras de control.

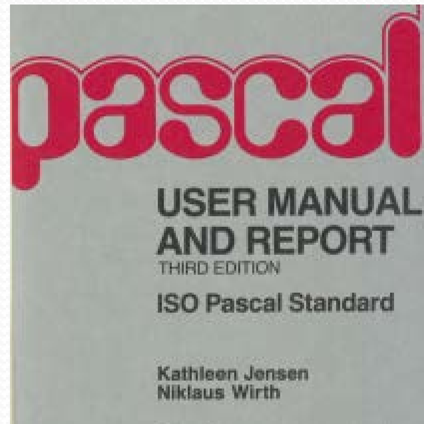
Pascal



- Además de permitir declarar variables y procedimientos, Pascal introduce declaraciones de constantes y de nuevos tipos de datos.

- También cuenta con estructuras **if-then-else** y con ciclos “**for**” (en una forma muy simplificada).

Pascal



- Asimismo, hay ciclos con la decisión al inicio y al final y una sentencia "**case**" para manejar casos.

- A continuación analizaremos algunas de las características más importantes de Pascal.

Enumeraciones

- Pascal cuenta con un tipo de datos para las enumeraciones.
- Este mecanismo es de muy alto nivel y muy orientado a las aplicaciones.
- Las enumeraciones permiten a los programadores expresarse directamente, sin tener que usar variables enteras.

Enumeraciones

- Ejemplo:

```
type
  DiaSemana = (Dom, Lun, Mar, Mie, Jue, Vie, Sab);
```

Enumeraciones

- Las enumeraciones son eficientes porque permiten que el compilador optimice espacio de almacenamiento y las operaciones en que se les involucra pueden realizarse rápidamente.

Enumeraciones

- Puesto que el compilador sabe con anticipación cuál es el número total de valores que contendrá la enumeración, puede asignar exactamente la cantidad de bits que se requiera para representarla, en vez de tener que usar el número total de bits que requeriría normalmente un entero.

Enumeraciones

- Por ejemplo, si nuestra enumeración tendrá 7 valores posibles, el compilador puede usar 3 bits para representar cada elemento.

- Esto contrasta con los 16 ó 32 bits que normalmente requeriría el tipo "integer" para cada elemento de la enumeración.

Enumeraciones

- Ejemplo:

Dom	Lun	Mar	Mie	Jue	Vie	Sáb
000	001	010	011	100	101	110