LITTLE LANGUAGES

BY Jon Bentley

PRESENTED BY: Alejandra Yepez
It is a particular kind of system for encoding and decoding information.

Combination of symbols in order to produce any meaning that can be understood by a group domain.

Examples?
## LANGUAGES

### Examples:
- Human Languages

<table>
<thead>
<tr>
<th>Arabic (عربية)</th>
<th>Armenian (งզունթք)</th>
<th>Chinese (中文)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Français (Français)</td>
<td>ΕΛΛΗΝΙΚΑ (Ελληνικά)</td>
<td>Kreyòl Ayisyen (Haitian-Creole)</td>
</tr>
<tr>
<td>한국어 (한국어)</td>
<td>Polski (Polski)</td>
<td>Português (Português)</td>
</tr>
<tr>
<td>Español (Español)</td>
<td>Tiếng Việt (Vietnamese)</td>
<td>فارسی (Farsi)</td>
</tr>
</tbody>
</table>
Examples:

Artificial Languages

![Hobbit Elvish Script](image-url)
LANGUAGES

Examples:
- Artificial Languages
Examples:
- Artificial Languages: Adunaic
Examples:

Artificial Languages
Programming languages can be used to create programs that control the behavior of a machine, to express algorithms precisely, or as a mode of human communication.
Root of some Human Languages

Natural, pre-existing languages may also be used in this way; their developers merely catalogued and standardized their vocabulary and identified their grammatical rules.

One such language, **Latino Sine Flexione**, was the based for the development of new human languages like Spanish, Portuguese, French, Italian, and others.
“Any Mechanism to express intent, and the input to many programs can be viewed as statement in a language”.

Little languages = computer languages (Jon Bentley, 1986).

Little languages commonly known as domain-specific languages (DSLs), or micro-languages.

Domain-Specific Language (DSL): “A small, usually declarative, language expressive over the distinguishing characteristics of a set of programs in a particular problem domain” (Walton, 1996).

In contrast to a general purpose language (GPL), a domain-specific language (DSL) is a language that is expressive uniquely over the specific features of programs in a given problem domain.
LITTLE LANGUAGES

Little languages commonly known as domain-specific languages (DSLs), or micro-languages.

In contrast to a general purpose language (GPL), a domain-specific language (DSL) is a language that is expressive uniquely over the specific features of programs in a given problem domain.

Someone asked: "What do little programming languages have to do with domain analysis/reuse?"

Someone asked: "Currently, there’re many systems that we can learn to use and master it within a short period of time but I’m sure that they’re not small languages so is there any other outstanding characteristic of little language?"
LITTLE LANGUAGES

Some widely used DSLs with their application domains:

<table>
<thead>
<tr>
<th>DSL</th>
<th>Application Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>BNF</td>
<td>Syntax specification</td>
</tr>
<tr>
<td>Excel</td>
<td>Spreadsheets</td>
</tr>
<tr>
<td>HTML</td>
<td>Hypertext web pages</td>
</tr>
<tr>
<td>LaTeX</td>
<td>Typesetting</td>
</tr>
<tr>
<td>Make</td>
<td>Software building</td>
</tr>
<tr>
<td>MATLAB</td>
<td>Technical computing</td>
</tr>
<tr>
<td>SQL</td>
<td>Database queries</td>
</tr>
<tr>
<td>VHDL</td>
<td>Hardware design</td>
</tr>
<tr>
<td>Java</td>
<td>General-purpose</td>
</tr>
</tbody>
</table>

Someone asked: "Is the Microsoft Visual Studio a little language, since it has ways for users to point and click to put together a Windows application?"

Someone asked: "Are little languages common in today's market?"

Someone asked: "Are the principles introduced in this paper still valid these days?"

Someone asked: "This is a 1986 paper when computer technology was used by pretty advanced users and scientists. However, in current globalized market, how feasible it is to develop little languages from business point of view?"

Someone asked: "Although the advantages of the little languages, I really doubt about their abilities to handling complex problems nowadays, considering such idea was proposed nearly twenty years ago and problems become much complex."
Created by Brian Kernighan.

He is the creator of the expression WYSIAYG, and WYSIWWYG.
WYSIAYG: "What You See Is All You Get"

WYSIWYG: "What You See Is What You Get"
**PIC – How do we use it?**

- Pic is a domain-specific language for drawing pictures.
- Pic provides boxes, lines, arrows, circles, ellipses, arcs, and splines, plus facilities for positioning and labeling them.
PIC – How do we use it?

- Each picture begins with `.PS` and ends with `.PE`; between them are commands to describe the picture.

- Each command is typed on a line by itself.

- Pic is a troff preprocessor; it passes most of its input through untouched, but translates commands between `.PS` and `.PE` into troff commands that draw the pictures.
PIC – Basic!!

For example:

```plaintext
.PS
box "this is" "a box"
.PE
```

Result

```
this is a box
```
PIC – Basic!!

We can use circle or ellipse in place of a box:

- Text is centered on lines and arrows; if there is more than one line of text, the lines are centered above and below:

  line "this is" "a line"

  arrow "this is" "an arrow"

- Boxes and lines may be dashed or dotted; just add the word dashed or dotted after box or line:

  line dashed "dashed" "line"
PIC – Basic!!

Arcs by default turn 90 degrees counterclockwise from the current direction; you can make them turn clock-wise by saying arc cw:

line; arc; arc cw; arrow

What is the shape of arc; arc; arc; arc?
PIC – Basic!!

What is the shape of arc; arc; arc; arc???
PIC – Basic!!

Objects are normally drawn one after another, left to right, and connected at the obvious places:

arrow;
box "input";
arrow;
box "process";
arrow;
box "output";
Arrow;
Each object that PIC knows about (boxes, circles, etc.) has associated dimensions, like height (ht), width (wid), radius (rad), and so on.

- **Box**: 3/4 " wide × 1/2 " high
- **Circle**: 1/2 " diameter
- **Ellipse**: 3/4 " wide × 1/2 " high
- **Arc**: 1/2 " radius
- **Line or arrow**: 1/2 " long
- **Move**: 1/2 " in the current direction

**For example:**

- box width 3 height 0.1;
- circle radius 0.1;
PIC – Controlling Positions!!

You can place things anywhere you want.

Pic uses a standard Cartesian coordinate system with \( x \) increasing rightwards and \( y \) increasing upwards, so any point or object has an \( x \) and \( y \) position, measured in inches.

```
box ht 0.2 wid 0.2 at 0,0 "1"
move right 0.5              # or "move to 0.5,0"
box ht 0.2 wid 0.2 "2"
move right 0.5              # or "move 0.5" or "move same"
box ht 0.2 wid 0.2 "3"
```
Each Lines and arrows are most easily drawn by specifying the amount of motion from where you are right now, in terms of directions.

Accordingly the words up, down, left and right and an optional distance can be attached to line, arrow, and move.

For example:

```plaintext
line up 1 right 2
arrow left 2
move left 0.1
line <-> down 1 "height" rjust
```
PIC – Labels!

Objects can be labelled or named so that you can talk about them later.

For example,

Box1: box
Let’s do an exercise!

The GOAL is create a picture using a given PIC’s input.

Rules!

You have some symbols in this moment!

Let’s create a picture on the wall with the symbols you have. ONE AT A TIME!

You have around 30 seconds to put a symbol in its respective position on the figure. After 30 seconds if you have the following symbol and you did not put it, you’re out!

If you make a mistake you’re out!

The person who wins will receive a gift! : )
Let's do an exercise!

Let's create a picture using the following input:

```
.PS
  arrow "source" "code"
LA:
  box "lexical" "analyzer"
  arrow "tokens" above
P:
  box "parser"
  arrow "intermediate" "code"
Sem:
  box "semantic" "checker"
  arrow
  arrow <-> up from top of LA
LC: 
  box "lexical" "corrector"
  arrow <-> up from top of P
Syn: 
  box "syntactic" "corrector"
  arrow up
DMP: 
  box "diagnostic" "message" "printer"
  arrow <-> right from right of DMP
ST:
  box "symbol" "table"
  arrow from LC.ne to DMP.sw
  arrow from Sem.nw to DMP.se
  arrow <-> from Sem.top to ST.bot
.PE
```
Let’s do an exercise!
PIC’s Compiler Vs Compiler

FIGURE 2. A Detailed View of a Compiler

FIGURE 6. A Detailed View of PIC
PIC’s Compiler Vs Compiler

**FIGURE 2. A Detailed View of a Compiler**

**FIGURE 6. A Detailed View of PIC**
PIC's Compiler Vs Compiler

FIGURE 2. A Detailed View of a Compiler

FIGURE 6. A Detailed View of PIC
PIC’s Compiler Vs Compiler

Lexical Analysis

The lexical analyzer breaks the input text into units called “tokens”.

When a pattern (on the left-hand side) is recognized, the action on the right is performed.

For example, “>” → return(GT);  

For example, the PIC input line: line down from B1.S  

Will be return by the lexer using the following sequence:

<table>
<thead>
<tr>
<th>LINE</th>
<th>DOWN</th>
<th>FROM</th>
<th>SYMBOL: B1</th>
<th>SOUTH</th>
</tr>
</thead>
</table>

PIC input line
Someone asked: "LEX uses a little language for regular expressions to specify lexical analyzers. What other programs on your system employ regular expressions? How do they differ and why?"
**PIC’s Compiler Vs Compiler**

**Lexical Analysis**

A C++ sample:  
\[ \text{Sum} = 4; \]

Will be return by the lexer using the following sequence:

<table>
<thead>
<tr>
<th>LEXEME</th>
<th>TOKEN</th>
<th>TOKEN #</th>
<th>VALUE/NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sum</td>
<td>IDENT</td>
<td>30</td>
<td>Sum</td>
</tr>
<tr>
<td>=</td>
<td>ASSIGN</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>INTEGER</td>
<td>58</td>
<td>4</td>
</tr>
<tr>
<td>;</td>
<td>ENDSTATM</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>
PIC’s Compiler Vs Compiler

FIGURE 2. A Detailed View of a Compiler

FIGURE 6. A Detailed View of PIC

YACC
YACC = “Yet Another Compiler- Compiler.”

Parser generator.

The input has the same form as LEX.

For example,

- **Primitive:** 
  
  ```
  BOX attriblist \[
  \rightarrow \{ \text{boxgen($1$);} \}
  ```

- **Attrlist:** 
  
  ```
  attriblistattr;
  ```

- **Attr:** 
  
  ```
  DIR expr \{ \text{storefattr($1, !DEF, $2);} \}
  ```
PIC’s Compiler Vs Compiler
Symbol Table

FIGURE 2. A Detailed View of a Compiler

FIGURE 6. A Detailed View of PIC
PIC’s Compiler Vs Compiler

Code Generation

**FIGURE 2. A Detailed View of a Compiler**
PIC’s Compiler Vs Compiler

MAKE: keeping up-to-date versions of the files containing header code, source code, object code, documentation, and test cases.

```
OFILES = picy.o picl.o main.o print.o \
     misc.o symtab.o blockgen.o \
     ...
CFILES = main.c print.c misc.c symtab.c \
     blockgen.c boxgen.c circgen.c \
     ...
SRCFILES = picy.y picl.l pic.h $(CFILES)
pic:  $(OFILES)
     cc $(OFILES) -lm
$(OFILES):  pic.h y.tab.h
memo:
     pic memo | eqn | troff -ms >memo.out
backup:  $(SRCFILES) makefile pictest.a
     push safemachine $? /usr/bwk/pic
touch backup
bundle:  
     bundle $(SRCFILES) makefile README
```

PROGRAM 3. PIC’s MAKE file
PIC’s Advantages

FIGURE 6. A Detailed View of PIC
Someone asked: “How can processors of little languages respond to linguistic errors?”

Someone asked: “How can a better understanding of linguistic Insight can give you a better understanding of the tools you now use and how can it help you in programming better?”
One of the greatest advantages of little languages is that one processor’s input is another processor’s output.

This structure is easy to implement as a UNIX pipeline of processes:

```
scatter infile | pic | troff >outfile
```
Principles and Advantage of Little Languages

Little languages are important part of the fourth- and Fifth Generation Languages and Application Generators.

Provide an elegant interface for humans to control complex programs.

It is simpler to have a large system composed of modules.

It is better before designing a language, carefully study the problem and dividing in parts.

A smaller language is easier to design, build, document, and maintain.

Little languages are creative.
Principles and Advantage of Little Languages

It is simpler to have a large system composed of modules.

Someone asked: "Are the reusable components considered the Little Languages we build as programmers?"
## Principles and Advantage of Little Languages

<table>
<thead>
<tr>
<th>#</th>
<th>Maintainability Factor</th>
<th>DSL Influence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Ease of expressing anticipated modifications</td>
<td>++</td>
</tr>
<tr>
<td>2.</td>
<td>Small development costs per application</td>
<td>++</td>
</tr>
<tr>
<td>3.</td>
<td>Small code size (low LOC)</td>
<td>++</td>
</tr>
<tr>
<td>4.</td>
<td>Low annual change traffic (ACT)</td>
<td>0</td>
</tr>
<tr>
<td>5.</td>
<td>Code readability</td>
<td>++</td>
</tr>
<tr>
<td>6.</td>
<td>System modularity</td>
<td>++</td>
</tr>
<tr>
<td>7.</td>
<td>Locality of changes</td>
<td>++</td>
</tr>
<tr>
<td>8.</td>
<td>Testability</td>
<td>+</td>
</tr>
<tr>
<td>9.</td>
<td>Code portability</td>
<td>+</td>
</tr>
<tr>
<td>10.</td>
<td>Maintenance process followed</td>
<td>+</td>
</tr>
<tr>
<td>11.</td>
<td>Maintainability as an objective</td>
<td>+</td>
</tr>
<tr>
<td>12.</td>
<td>Quality of configuration management</td>
<td>0</td>
</tr>
<tr>
<td>13.</td>
<td>Repository for modification requests</td>
<td>0</td>
</tr>
<tr>
<td>14.</td>
<td>Small number of languages used</td>
<td>--</td>
</tr>
</tbody>
</table>

Figure 3: Maintainability factors, and the best possible effect (ranging from negative -- via neutral 0 to positive ++) the use of a DSL has on each of these factors.
Why is LT or DSL so important within the Software Reuse area?

- They offer appropriate domain-specific notations from the start.
- Their primary contribution is to enable reuse of software artifacts.
- Software architectures are commonly reused when DSLs are employed; for example, GAL (Graphics Adaptor language).

Some people asked: How does "Little Languages" apply to software reuse?

Someone asked: Using/creating little languages is one way to reuse software assets (e.g. tools) to build the program but using big language also reuse some assets too so which one is better?
Bibliography

Slides from Dr. Boggess class in Programming Languages.


THANK YOU