Slides to Accompany *Programming Languages and Methodologies*

R. J. Schalkoff

Chapter 1, Part 1: Introduction (r2)

Definition #1

- A programming language is a notational system intended primarily to facilitate human-machine interaction.
- The notation is both human and machine-readable.

Definition #2

From a linguistic viewpoint, a (programming) language has a *syntax*, and language elements have *semantics*.

Definition #3

- A program is something that is produced by a programming language.
- A program is a structured entity with semantics.

Why?

We study programming language concepts for many reasons, including:

- To become aware of language features or capabilities which could speed the development process (increased expressive capability)
- To be able to intelligently choose an appropriate language
- To be able to learn new languages (more efficiently)
- To understand the underlying implementation issues
- To be able to modify or design new languages
- To get credit for required college courses in Computer Engineering and Science.

Mindset

Most notably, the concepts and features embraced by a particular programming language may have a significant influence on a programmer's mindset.

In other words, language design may shape a programmer's thought processes.

> *If the only tool you have is a hammer, every problem looks like a nail.*

Formal Approaches

- A formal and quantitative characterization of a programming language is based upon a formal language which itself is based upon a formal grammar.
- Denote this grammar G.
- This viewpoint allows us to alternately and quantitatively define a program as a *sentence* or *string* produced by G, and a programming language as the set of all strings (programs) producible by G.

Choices, Choices, and Choices

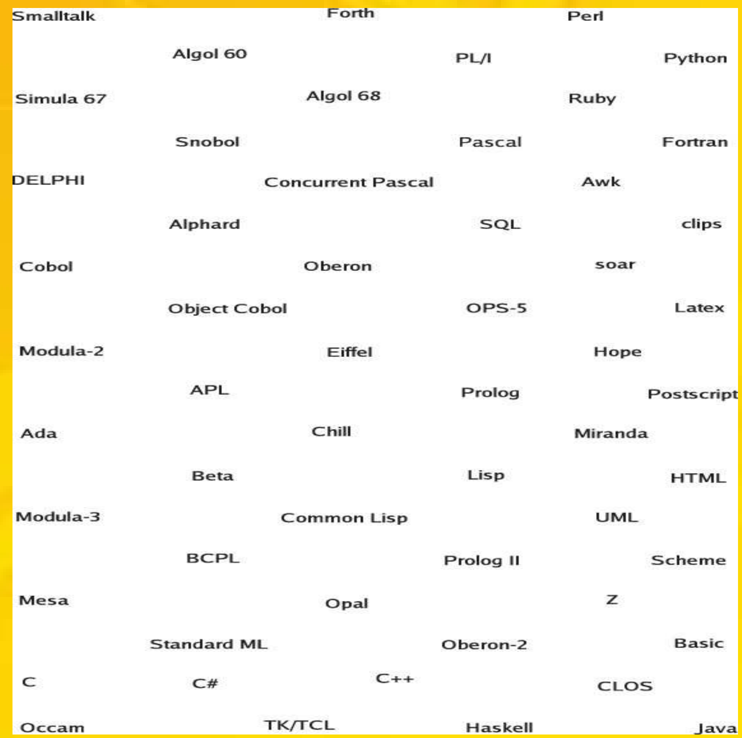

Figure 1: A Portion of the 'Sea of Languages'

Software and Moore's Law

- Moore's law, stated in 1965 by Gordon Moore, postulates a doubling in computer hardware performance (measured by component density or gates on a chip) roughly every 18 months.
- This translates into a factor of 100 every 10 years. While Moore's law is not a law of physics, it has been reasonably accurate in predicting combined computer processing, storage and communication capabilities for several decades.
- **There does not appear to be a corollary to Moore's law for software.**

From Gears to Software Objects

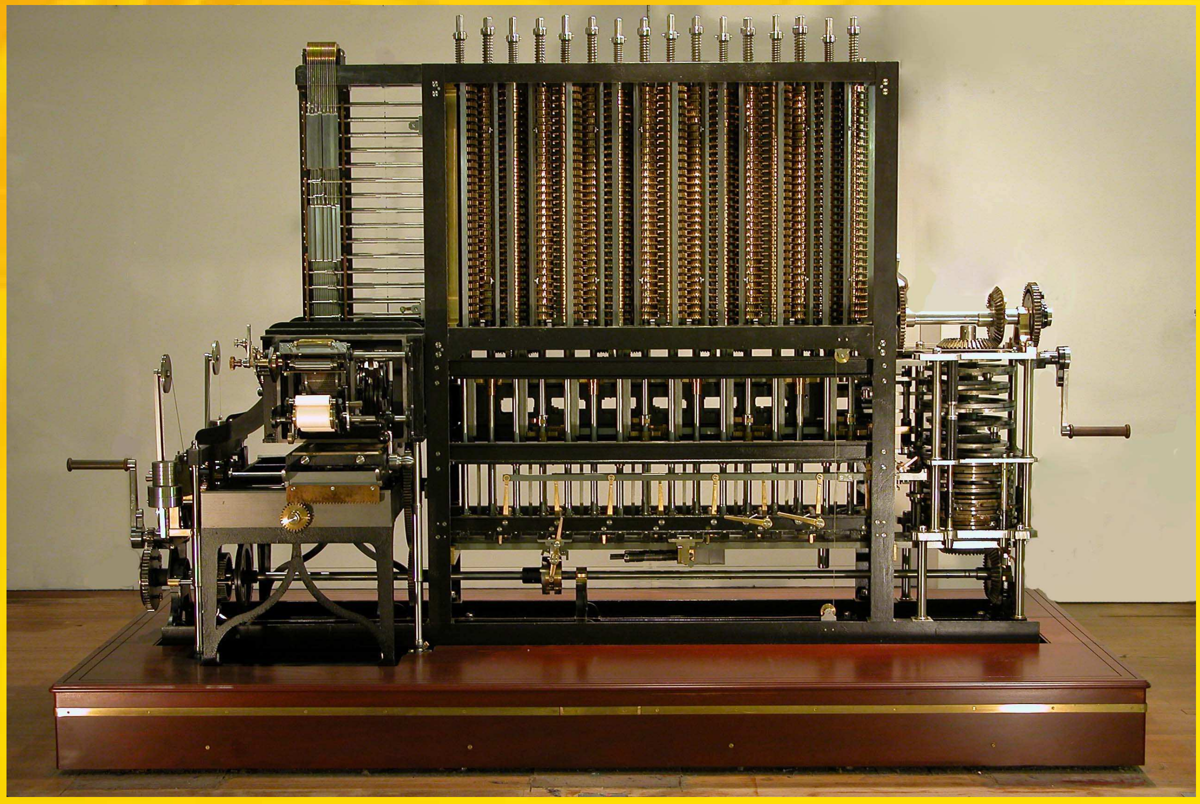

Figure 2: The Mechanical 'Difference Engine' Computer. (And you thought software development in linux was difficult?). Courtesy Doran Swade.

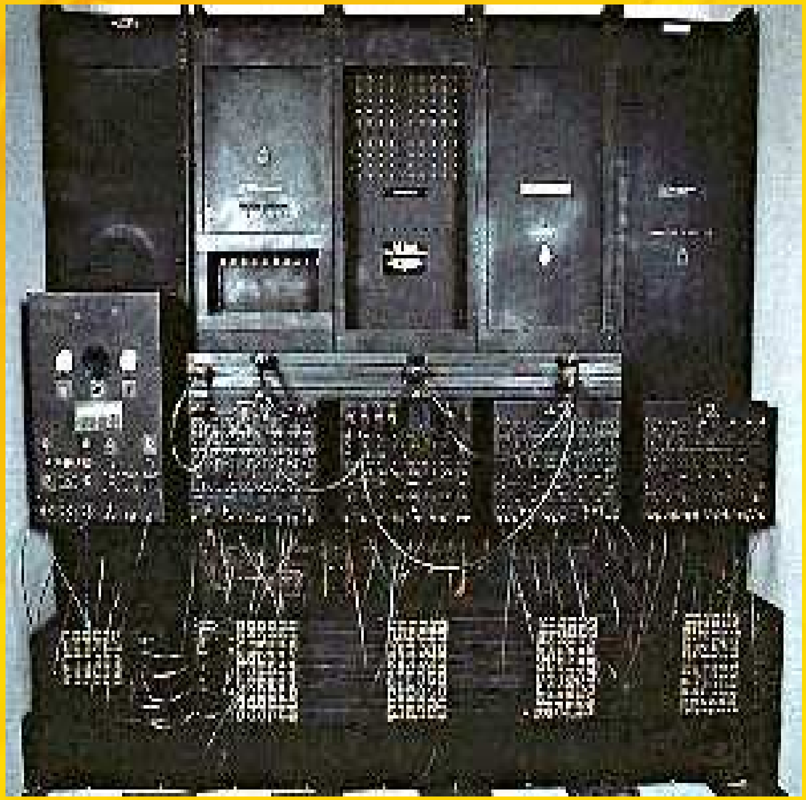

Figure 3: The ENIAC Computer, circa. 1945. ENIAC contained 20 electronic accumulators, each capable of storing a 10-digit decimal number and had a read-only memory of about 300 numbers.

hardware	software
gears	changing gears
relays/vacuum tubes	switches, cables, machine code
discrete transistors	assemblers
LSI	higher-level dev. systems
VLSI	paradigms chosen by *application*

Figure 4: Language Generations Parallel Hardware Evolution

A Point of Departure

Today, however, the situation is one of significantly greater independence. Computing hardware generally supports a number of operating systems and development tools, including language interpreters and compilers for programming languages.

- To a great extent, language choice is independent of hardware.
- Hardware is (relatively) inexpensive (e.g., Raspberry Pi).
- Software development is (relatively) expensive and time-consuming.
- A possible point of view: **software, not hardware, is holding back advances in computing.**

Language Chronology

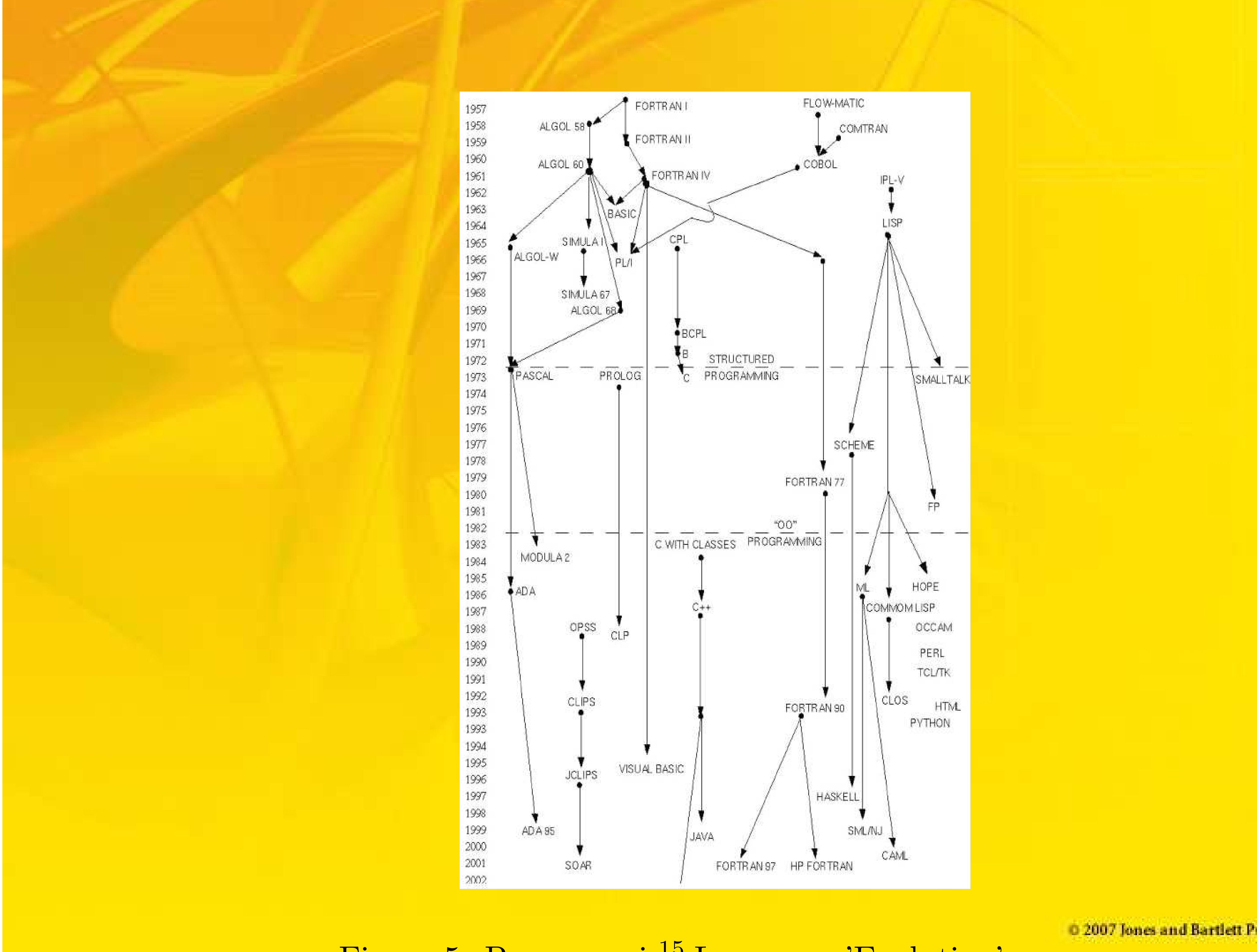

Figure 5: Programming Language 'Evolution'

Programming Languages by *Paradigm*

A wide variety of programming paradigms exist. Examples are:

1. Procedural or imperative (the best-known, e.g., c, Java)
2. Functional (or applicative)
3. Declarative
4. Object-oriented
5. Rule-based
6. Event-driven
7. Parallel or concurrent
8. Agent-oriented

Example: Visual Programming (khoros)

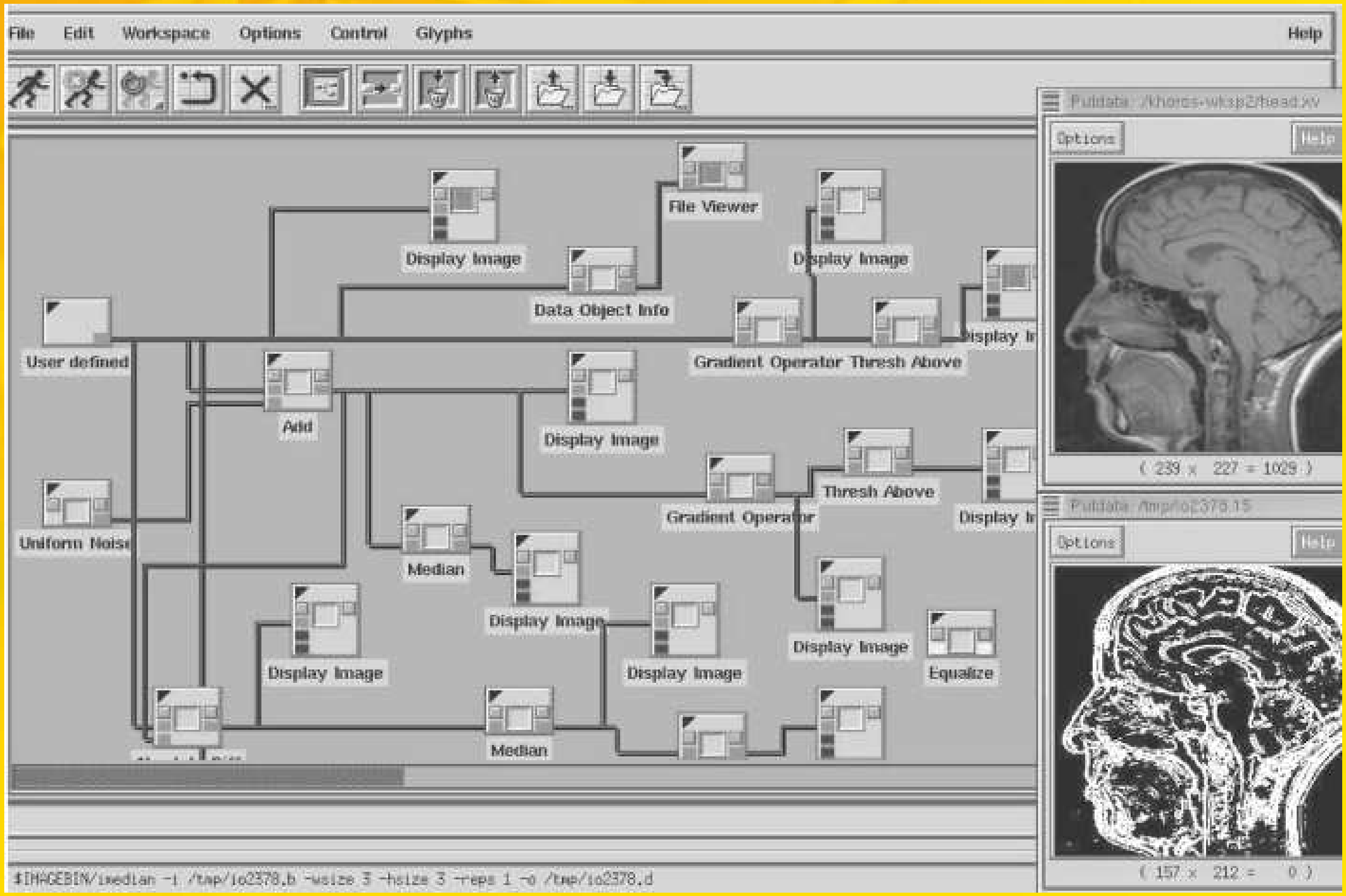

Figure 6: Visual Programming a.k.a. 'Connect the Gliphs'

Syntax

Programming language syntax defines the allowable arrangement of symbols in programs, especially program fragments. For example, the syntax may constrain fragments to have matching parentheses, e.g.,

```
<main-decl> ::= <type> main <arg>
<type> ::= int | void
<arg> ::= lparens <type> rparens
```

Syntax also catalogs the basic elements of the language, most importantly the **structure** of the language.

Syntax is often indicated by a metalanguage, i.e., a language used to quantify another language. Examples are the language of regular expressions and BNF.

The Concept and Use of an API

In many cases, 'real' programming means writing code/developing software which interfaces to a substantial amount of (and investment in) existing code.

The existing code was (we hope) designed for this type of interface through an API or **A**pplication **P**rogramming **I**nterface.

API: Application Programming Interface- a set of routines (usually functions) and accompanying protocols for using these functions which provide building blocks for software development.

Examples of programming to an API include:

1. The X-windows system (common on Unix/linux platforms);
2. Microsoft Windows (probably one of the more difficult and simultaneously popular API families); and
3. The Palm OS.
4. Android development.

Other noteworthy examples are:

- `wine`, an open-source implementation of the Windows API built upon X and linux; and
- `cygwin`, an environment for Windows providing a linux-like API.

Reading Code Is Good for You

- Professional software developers spend a significant fraction of their time not in actually writing code, but instead in reading, understanding, and modifying existing code.
- An emerging notion is that learning how to produce good code is based upon reading good code.
- Lots of Open Source code corresponding to popular programs such as web browsers, graphics file manipulation programs, language interpreters/compilers, and even entire operating systems is available for reading and review.
- A key element for the success of this approach is the ability to distinguish good programming practices from 'not-so-good'.

Some Tools

- editors:
  - syntax highlighting
  - support language features (matching parens)
- diff
- indent

Other Tools

`gprof`: A code profiler.

`cdecl`: For deciphering complex type declarations in `c` source.

`RCS` and `CVS`: These are tools for the management of multi-programmer projects and keeping track of revisions of source code.

`strace (linux)` and `API Spy (Windows)`: Facilitate checking on operating system calls and may be used to check or debug an executable.

`gctags` and `ctags`: Help to find declarations and definitions; produce output compatible with the regular expression search facility in `vi`.

`hexedit`: Useful for examining object code, exploring binary data structures, and deciphering non-ASCII image files.

IDEs and Editors

The notion of an integrated development editor (IDE) is appealing to programmers and involves combining an editor, a source compiler (or interpreter), debugging, revision control, project management, etc. all in one application.

diff

See the text.

indent: visual appearance

```
/* game-no-format.c  */
/* for ece352 sp2007 */

int boardprint(void){int i,j;
printf("\nThe current board state is:\n\n");for(i=0;i<BOARDDIM;i++)
{printf("\n");for(j=0;j<BOARDDIM;j++) printf("  (%d,%d):%s  ",i,j,board[i][j
printf("\n\n");}return 1;
}

int gen_and_est_motion_stats(void)
{float iavg=0.0;float javg=0.0;int themove,k,stepi,stepj;printf("\nGeneratin
for(k=0;k<100;k++){themove = random(DIRECTIONS);stepi=moves[themove].im;iavg
}
iavg=iavg/50.0;javg=javg/50.0;printf("Estimated expected motion vector value
iavg,javg); return (1);
} /* gen_and_est_motion_stats */
```

```
/* game-format.c  */
/* for ece352 sp2007 */

int
boardprint (void)
{
  int i, j;
  printf ("\nThe current board state is:\n\n");
  for (i = 0; i < BOARDDIM; i++)
    {
      printf ("\n");
      for (j = 0; j < BOARDDIM; j++)
printf ("  (%d,%d):%s  ", i, j, board[i][j]);
      printf ("\n\n");
    }
  return 1;
}

int
gen_and_est_motion_stats (void)
{
```

```
  float iavg = 0.0;
  float javg = 0.0;
  int themove, k, stepi, stepj;
  printf ("\nGenerating 100 random motion vectors\n");
  for (k = 0; k < 100; k++)
    {
      themove = random (DIRECTIONS);
      stepi = moves[themove].im;
      iavg = stepi + iavg;
      stepj = moves[themove].jm;
      javg = stepj + javg;
    }
  iavg = iavg / 50.0;
  javg = javg / 50.0;
  printf ("Estimated expected motion vector value is: %E %E\n\n", iavg, javg
  return (1);
} /* gen_and_est_motion_stats */
```

Tools We'll Use

- Prolog: tracing (CLI and GUI)
- ocaml: tracing, profiling, documentation, compilation