Language definition

- **Syntax**: the structure of a program. Usually given a formal (i.e., mathematical) definition using a context-free language. (Lexical structure - the structure of the words or tokens - uses regular expressions.)

- **Semantics**: the actual result of execution. Usually described in English, but can be specified mathematically.

Language translation

**Compiler**: two-step process that

1. translates source code into executable code (typically in machine code) and then goes away.;
2. the system executes the executable code.
Language translation

Interpreter: one-step process in which the source code is executed directly.

Hybrids are also possible (Java).

Bytecode is composed of instructions that have been brought to the lowest level possible without making them machine dependent.

Analogy

- Compiling - Cooking with a precise recipe.
  You know all ingredients, utensils, amounts, oven temperatures, etc. Adapted for a specific kitchen and set of ingredients.

- Interpretation – Cooking without knowing ahead of time exactly what ingredients you have, what pans you will need, what cooking utensils/temperatures will be appropriate.

- Hybrid – The basic instructions and kitchen specifications are known, but there is room for last minute changes.
In compiling/interpreting, part of the work comes from parsing.

**Parsing**

- NP: noun phrase
- VP: verb phrase
- N: noun
- A: article
- V: verb
- PP: Prepositional Phrase

**What is Parsing in English?**

```
S → NP VP
NP → A N
NP → NP PP
VP → V NP
VP → VP PP
PP → P NP
```

```
S
  /\  
Papa
 /   \  
VP   PP
|   / \|
V   A   N
|   |  /
ate the caviar
|   |  /
A   N
|  /
with a spoon
```

```
NP → Papa
N → caviar
N → spoon
V → spoon
V → ate
P → with
A → the
A → a
```
Programming languages

```c
printf ("/charset [%s", (re_opcode_t) *(p - 1) == charset_not ? "^" : "");
assert (p + *p < pend);
for (c = 0; c < 256; c++)
  if (c / 8 < *p && (p[1 + (c/8)] & (1 << (c % 8)))) {
    /* Are we starting a range? */
    if (last + 1 == c && ! inrange) {
      putchar ('-');
      inrange = 1;
    }
    /* Have we broken a range? */
    else if (last + 1 != c && inrange) {
      putchar (last);
      inrange = 0;
    }
    if (! inrange)
      putchar (c);
  last = c;
  }
```

- Easy to parse.
- Designed that way!

Natural languages

```c
printf="/charset %s", (re_opcode_t) *p - 1 == charset_not ? "^": "");
assert *p < pend; for c = 0; c < 256; c++ if c / 8 < *p && p[1 + (c/8)] & (1 << (c % 8))
  if (last + 1 == c && inrange) {
    putchar ('-');
    inrange = 1;
  }
  else if (last + 1 != c && inrange) {
    putchar (last);
    inrange = 0;
  }
  if (! inrange)
    putchar (c);
  last = c;
```

- No {}, () [], to indicate scope & precedence
- Lots of overloading (arity varies)
- Grammar isn’t known in advance!
- Context-free grammar not best formalism
Ambiguity

Ambiguity -eating two things?
• Compiling: we would create a file of machine instructions that would only work for one architecture.

• Interpreting: our interpreter would have to be able to understand the high level code AND would repeatedly parse it (if the code was in a loop or called multiple times).

Parsing is the process of analyzing a text, made of a sequence of tokens (for example, words), to determine its grammatical structure with respect to a given grammar.

• When we use the hybrid approach of Java, we produce the file Test.class, which is a binary file that’s not readable by most humans.

Language Translation Steps

Compilation:

• lexical analysis: characters grouped into logical chunks (keywords, constants, etc)

• syntax analysis: figure out what it means - usually use parse trees (grammars to define). Like diagraming sentences.

  I have a picture of John at my home – what is meant? What does “at my home” modify?

• code generation

• optimization - to make smaller or faster

• linking: supplying missing addresses to system code

• load module: user code augmented with system code
Language Implementation Steps

**Pure Interpretation:**
- no translation phase - fetch, decode, and execute the source code (not machine code)

**Advantages/Disadvantages**
1. easy to write debugger - as source lines are unchanged (no cryptic messages from debugger)
2. execution is 10-100 times slower; statement decoding is bottleneck
3. better for language with simple structure - as not so slow to decode
4. natural for some kinds of features - like dynamic binding of type. **Ex:** \( c = c + b \) If \( c \) may be integer, string, or a set, how can we know what code to generate?
5. Nice for executing user produced code at runtime

**Ex.** boolean expression is easy to evaluate if known at compile time, but NOT if produced at run time.

---

What is meant by dynamic binding?

**Girls choice dance:**
- Will you go with Sofie? (early binding)
- Will you go with Sofie/Ann/Betty (whoever shows up at your door)? (delayed binding)
- No specific partner assigned, but will change throughout the night. (changing binding)

**Lots of the interesting issues involve binding times.**
Passing in a polygon, actual type can be any descendant.

```cpp
class Polygon {
    int numVertices;
    float *xCoord, float *yCoord;
    public:
        void set(float *x, float *y, int nV);
        String display(String s) {
            return "POLYGON" + s;
        }
        float area();
};

class Rectangle: public Polygon {
    public:
        float area();
        String display(String s) {
            return "RECTANGLE" + s;
        }
};

class Triangle: public Polygon {
    public:
        float area();
        String display(String s) {
            return "TRIANGLE" + s;
        }
};
```

Class binding (C++)

Suppose I have

```cpp
Polygon * p = new Rectangle()
p->display(); Which display is called?
```

virtual – function call is resolved at run time
Dictionary Moment

- **Lexical**: of or relating to words or the vocabulary of a language as distinguished from its grammar and construction
- **Lexical Analysis**: finding the individual words (tokens) of the code.

Error classification

- **Lexical**: character-level error, such as illegal character (hard to distinguish from syntax error).
- **Syntax**: error in structure (e.g., missing semicolon or keyword).
- **Static semantic**: non-syntactic error prior to execution (e.g., undefined vars, type errors).
- **Dynamic semantic**: non-syntactic error during execution (e.g., division by 0).
- **Logic**: programmer error, program not at fault.
Notes on error reporting

- A compiler will report lexical, syntax, and static semantic errors. It cannot report dynamic semantic errors; which is job of runtime system.
- An interpreter will often only report lexical and syntax errors when loading the program. Static semantic errors may not be reported until just prior to execution. Indeed, most interpreted languages (e.g. Lisp, Smalltalk) do not define any static semantic errors.
- No translator can report a logic error.

Find examples of (possible) syntax, semantic and logic errors.

```java
public int gcd ( int v# )
{ int z = value
    y = v;
    while ( y >= 0 )
    { int t = y;
        y = z % y;
        z = t;
    }
    return y;
}
```
Sample Errors (Java):

```java
public int gcd ( int v# ) // lexical bad #
{ int z = value       // syntax - missing ;
  y = v; // static semantic - y undeclared
  while ( y >= 0 ) {
    int t = y; y = z % y; z = t;
    // dynamic semantic - division by zero
  }
  return y; // logic - should return z
}
```