Programming Language Design Workshop

Vincenzo Ciancia – vincenzo.ciancia@isti.cnr.it

Istituto di Scienza e Tecnologie dell'Informazione
Consiglio Nazionale delle Ricerche, Pisa

Introduction

“Programming languages?”

You know what a programming language is!

Do you know?

Do you **actually** know?

Then what is a “programming language”?

“Programming languages?”

A programming language *can be* a lot of things, actually.

Syntax **Semantics**

Interpreter **Compiler**

“Syntax”

Syntax can be a lot of things (maybe) but for our purposes:

Context free grammars

Syntax tree

“Semantics”

Semantics can be a lot of things (for sure) among which:

Operational / Denotational / Logical

Transition systems / Rewrite Rules / Term or Graph Rewriting

Fixed point / Category Theory / Equivalences

Coalgebras / Dialgebras / Bialgebras

Bisimilarity / Simulation **Many other things…**

“Interpreter”

Execute a program, written in a formal language.

REPL:

- Read
- Eval
- Print
- Loop

EXERCISE

How many interpreters can you name?

LISP	Python	Bash
Matlab	Octave	R
MS-DOS	Javascript	BASIC
Office	Chrome	Photoshop
Apache	sshd	pdf
postscript	SVG	Bitwig Studio
comfyUI	word	excel

..... this is getting weird :-)

“Compiler”

Translate a program from a source language to a target language

The result will be run later

Let us name a few compilers

“Compiler”

C++	C	Fortran	Pascal	Assembler

Too far backwards in time?

Java	CSharp	FSharp	Latex

All “transpilers” (from fable to emscripten …)

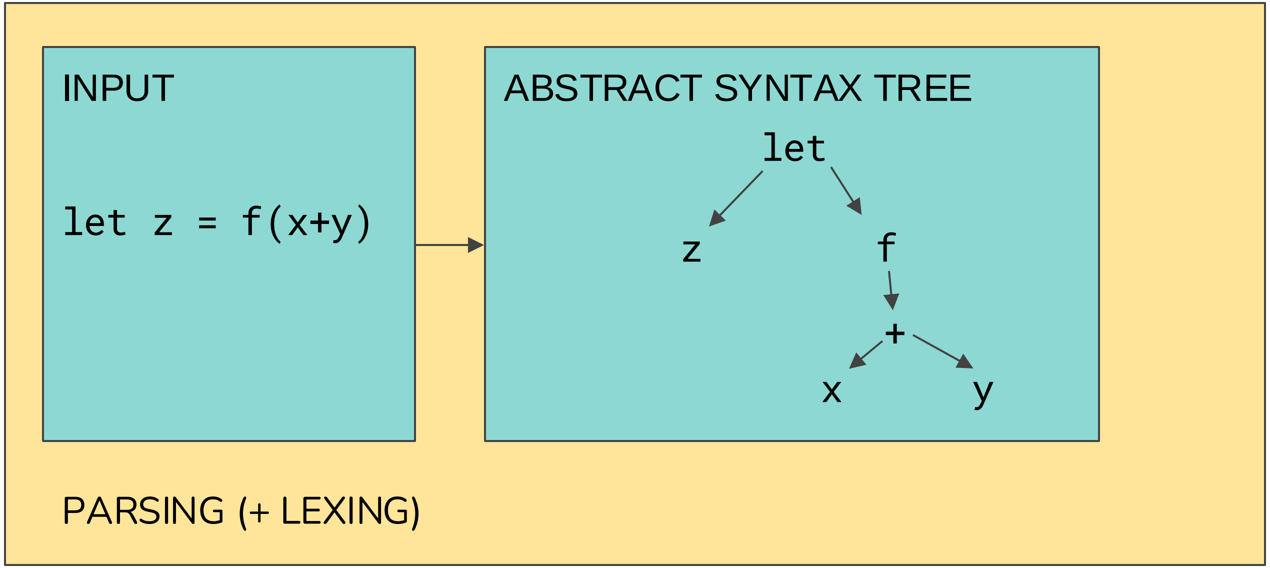
The on-the-fly compiler on the GPU

Q: What is faster

Interpreter or Compiler?

That’s a silly question

What we will NOT study in detail



Context free grammars

Formal definitions [edit]

A context-free grammar G is defined by the 4-tuple $G = (V, \Sigma, R, S)$, where[5]

1. V is a finite set; each element $v \in V$ is called *a nonterminal character* or a *variable*. Each variable represents a different type of phrase or clause in the sentence. Variables are also sometimes called syntactic categories. Each variable defines a sub-language of the language defined by G.
2. Σ is a finite set of *terminals*, disjoint from V, which make up the actual content of the sentence. The set of terminals is the alphabet of the language defined by the grammar G.
3. R is a finite relation in $V \times (V \cup \Sigma)^*$, where the asterisk represents the Kleene star operation. The members of R are called the *(rewrite) rules* or *productions* of the grammar. (also commonly symbolized by a P)
4. S is the start variable (or start symbol), used to represent the whole sentence (or program). It must be an element of V.

Example

PRODUCTION RULES

Expr ::=	Var | Int | Expr + Expr | let Var = Expr | (Expr)
Var ::=	x | y | z | ...
Int ::=	Digit | Digit Int
Digit ::=	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

EXAMPLE DERIVATION

Expr

→ let Var = Expr

→ let z = Expr

→ let z = (Expr)

→ let z = (Expr + Expr)

→ let z = (Var + Expr)

→ let z = (x + Expr)

→ let z = (x + Int)

→ let z = (x + Digit Int)

→ let z = (x + 4 Int)

→ let z = (x + 4 2)

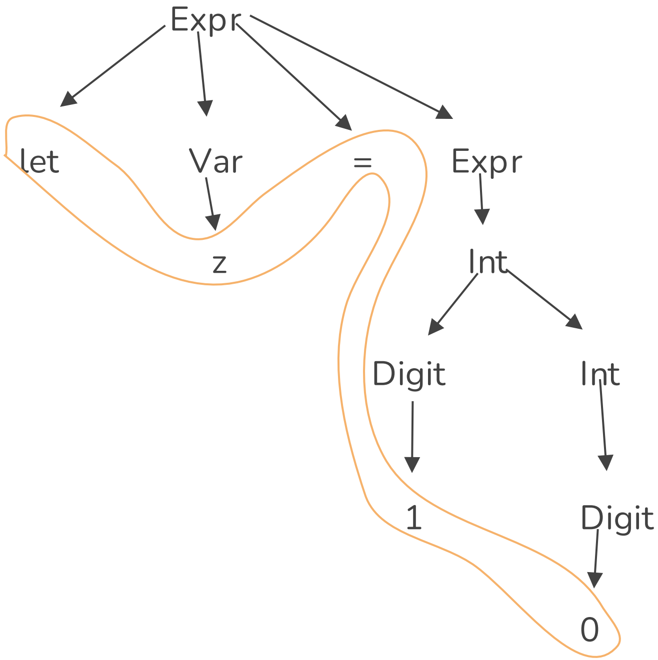
Derivation & Parsing

Derivation: generate a sequence of symbols (usually called “string”)

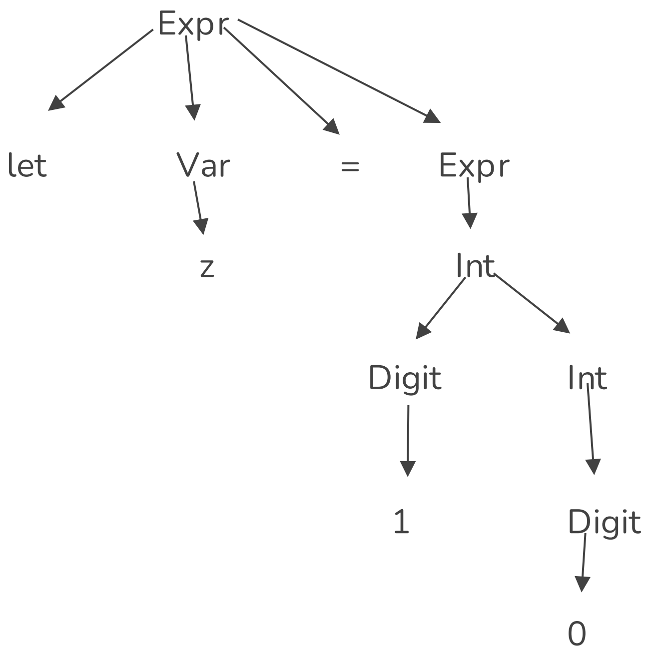
```
repeatedly replace a NON-TERMINAL SYMBOL

using a PRODUCTION RULE

until all symbols are terminal
```

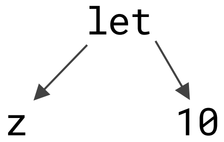
Parsing: identify a set of rule choices that can derive a given string

Parse tree



Parse tree vs Abstract Syntax Tree





Abstract Syntax Tree

Abstract syntax

Avoids most complications related to infix & prefix operators, precedence, etc.

Formally: Initial Algebra

Not a topic in this course, so we shall just assume:

we have a in-language representation of an AST

Parsing. What’s next?

After parsing, execution

How is that done?

Depends on the language!

Parsing. What’s next?

We will use **Formal Semantics** and implement its **mathematical definition** directly in Python.

This is not going to be super efficient but it is going to be very clear.

Good for prototyping, easy to refine for efficiency.

Example: variable assignment

Consider classical variable assignment

C-like syntax: `x = 3`

What does it do? Modify the memory!

Example: variable assignment

How do we formalise it?

Memory is a **partial function** from **variables** to **values***

***this statement is not even precise (we will study this in detail later)**

Example: memory update?

What is memory update, formally?

x = k is interpreted as update(**mem**,x,k) = mem’

mem’(x) = k

mem’(y) = mem(y) *if y ≠ x*

Q: mathematically, to which set do mem and mem’ belong?

Memory update?

What is memory update, formally?

x = k   is interpreted as   update(**mem**,x,k) = mem’

mem’(x) = k

mem’(y) = mem(y)   *if y ≠ x*

Q: mathematically, to which set do mem and mem’ belong?

Why am I mentioning all this?

Consider memory update in hardware: just change one variable.

Access time still constant.

Consider a chain of function updates as in the previous slide, instead.

Access time is linear in the number of previous updates! (Crazy!)

Easy to fix

Just use arrays to represent memory

use array update in place of functional update

It’s no longer formalised!

Actually, program semantics is the way to formalise array semantics, so using arrays to formalise program semantics is circular reasoning!

Way out ...

Use abstraction!

Implement memory-related operations in a module, class, or library

Define them inefficiently to study the language features

Define them efficiently at a second stage.

What are we, programming language designers?

The short answer, YES.

The long answer: <teacher gives long answer to class>

Why learning to implement interpreters?

Interpreters are everywhere. Domain-specific languages are everywhere

Unusual examples to think about

Think of UNDO/REDO. Clearly hints at existence of a sequence of “instructions”.

Think of filtering products on web sites. Clearly hints at interpreters for filter expressions.

Think of graph-based programming systems (e.g. music synthesizers, ComfyUI, labview, scratch, Adobe After Effects, Max/MSP, PureData...).

Why study semantics from a practical perspective?

- **Program analysis / verification**

  Abstract interpretation, model checking, type checking

- **System monitoring**

  Watch a system and check if it respects given properties

- **Program transformation**

  simplification of programs, instrumentation (e.g. for performance measurement or debugging)

- **Testing**

  run a program many times; identify the corner cases that must be tested

More details

Q: What is a “property” (to be tested, to be checked statically, to be monitored?)

A: What is your formal model?

Begging the question?

NO!

How can you test properties of something, if you don’t know what you can actually observe?

Properties

It’s like watching a bicycle race and test for how many goals have been scored!

Think about it.

To be able to ask the right question, you need a **domain of discourse**

We shall call such domains “**semantic domains**”

Observing a system

Consider a system. Not even “programmed”. Just any system in the universe.

How to analyse / measure its **behaviour**?

Via **observations**. Quite obvious.

Semantic domains

When one actually **has** a programming language, observations are the semantics

SEM: PROG $\to$ D

Q: What is D? The **semantic domain**

Operational or Denotational?

“It is all very well to aim for a more ‘abstract’ and a ‘cleaner’ approach to semantics, but if the plan is to be any good, the operational aspects cannot be completely ignored.”

(Dana S. Scott. Outline of a Mathematical Theory of Computation, Programming Research Group, Technical Monograph PRG–2, Oxford University, 1970.)

The thing that matters most is not how

What really matters is ***what***

The result!

–*indeed, still stating the obvious.*

But the «result» could be just a value, or the execution trace leading to that value, or the tree of possible choices leading to many possible values, depending on user interaction, or….

On semantic domains, side effects & c

Transition Systems

Non-determinism

Bisimilarity

Coalgebras

Categorical modelling

What will we do next?

- Python as a meta-language
- Difference between **expressions** and **commands**
- Difference between **names** and **variables**
- Difference between **environment** and **state**
- Static and dynamic scoping?
- Implement the semantics of mini-languages
- Mid-term
- Final Project

What we will **not** do

- Parsing, lexing, grammars, automata
- Efficient implementations
- Proper input/output
- Compiler (but you will understand the basics of compilers after this course)

Q: Shall we stop here?

A: Yes.