The core of programming languages comes down to 3 things:

1. principles and limitations of programming models
2. notations and user supports for programming models
3. methods of evaluation of programming models

These are the things we consider when attempting to decide on a choice of notation.
By the end of the course, our choices will be:
Ocaml – our canonical functional language
Java – our canonical imperative language
Prolog – our canonical logic language
These are our "big three" languages, which cover the main paradigms.
We will also cover Scheme, Python, etc.

The general goal is to learn, evaluate, and use languages.
How do we evaluate a language? We seek to minimize costs.
These come in the form of:

1. Primary Costs
   (a) maintainability
   (b) reliability
   (c) training
   (d) program development
2. Secondary Costs
   (a) execution overhead (exception: critical in machine learning)
   (b) licensing fees
   (c) build/compilation overhead
   (d) porting overhead

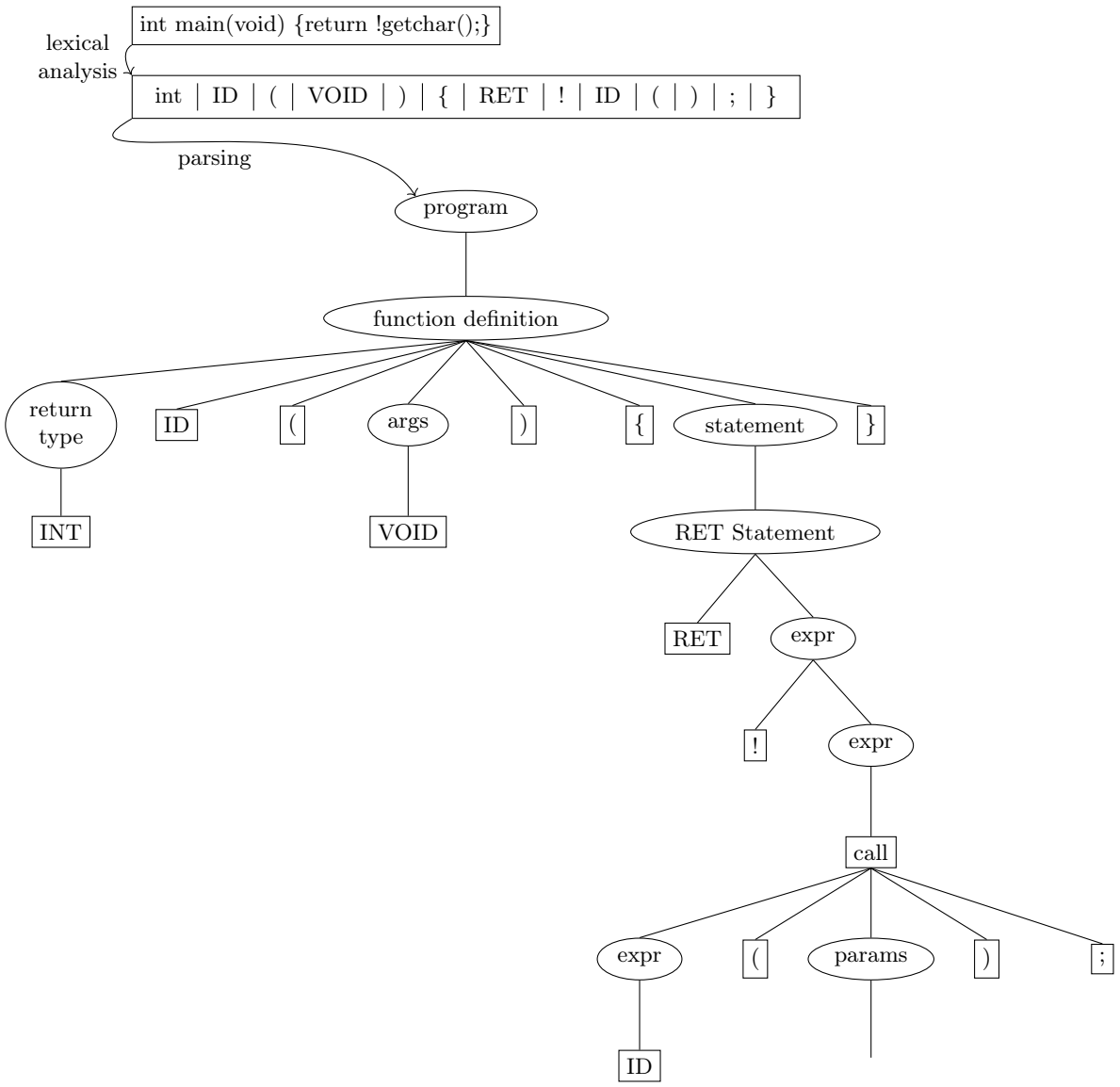
These choices can often be political (Apple vs Google, etc).

The main issues of language design, on the other hand, are:

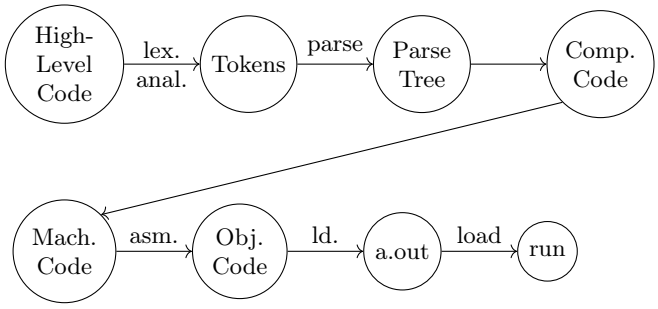
1. orthogonality
   For example, all C types except arrays are returnable, which presents difficulties with typedef!
2. (runtime) efficiency
   This includes CPU, RAM, energy, network access, etc.
3. simplicity
4. convenience
   C allows i++, ++i, i = i + 1, i += 1, choosing convenience over simplicity.
5. safety
   Errors can be checked in two ways:
   (a) static checking – safe, compile-time checking
   (b) dynamic checking – simple, runtime checking

6. abstraction support
7. exceptions
8. concurrency
9. evolution/mutability
   "As long as a language is alive, it is mutating"

We will now move on to considering more computer language specific issues. The translation of a language's code goes as follows:



We use this tree to build our compiler-level code.
It is portable and convenient for the compiler creator.
This puts the code → run process as:



Our first two assignments focus on the token → tree step.
This is where we consider **syntax**

Syntax

≡ the form of a language independent of meaning/**semantics**

- a sentence can have good syntax and bad semantics
  "Colorless green ideas sleep furiously." – Noam Chomsky
- a sentence can have good semantics and bad syntax
  "Ireland has Leprechauns galore." – Paul Eggert
- a sentence can be syntactically or semantically ambiguous
  "Time flies." – unknown; probably a politician, since doublespeak helps them

How do we evaluate a syntax?
Consider the example of 3+4*5 (C) vs 345*+ (Forth) vs (+3(* 4 5)) (LISP)

- inertia – no one likes to have to change
  C > Forth ≈, since C is familiar to standard form
- simplicity/regularity – few, regularly applied rules
  Forth (never parentheses) > LISP (always parentheses) > C (some parentheses)
- human readability – code shows algorithm/operations C ≡ LISP ≡ Forth due to Leibniz's Criteria ≡ a proposition's form should reflect reality
  ex. "if (0 < x && x < n)" > "if (x > 0 && x < n)" since 0 < x < n is natural
- human writability
  C > LISP ≈ Forth due to comfortability with standard form
- redundancy – repetition helps catch errors (inapplicable)
- unambiguity – code has only one interpretation
  Forth ≡ LISP > C, since C relies on an implicit order of operations

Syntax is build upon **tokens**

Tokens

Tokens are a subset of **lexemes**, which are built from characters. Tokens can be broken up into three categories

- identifiers
- operators
  Operators are often user-defined, but some are **keywords**
  keyword ≡ a word with a special meaning in a language
  Some operators can also be categorized as **reserved words**
  reserved word ≡ words which are not allowed to be used as a name
  C, for example, has reserved all words of the form _[: capital :].∗.
  This is annoying, since it makes us use _Noreturn instead of noreturn
  Consider the example 'if':
  'if' is a keyword since it tests an expression, BUT:
  - in C it is reserved, $\implies$ 'int if = 12' is invalid
  - in PL1 it is not reserved, $\implies$ 'if (if=3) if = 4' is valid.
- numbers
  This includes 27, 0x19, and 5e-12, but not -127, because itemizers are greedy.
  To understand this, consider the example: a - - - - - b; this should be valid as (a - -) - (- - b),
  BUT the tokenizer reads it as ((a - -) - -) - b, and we cannot decrement a constant.

It may be a natural assumption that tokens may only consist of characters, but this is overkill.
We must consider both white space, and ambiguities like 'ifx', which would cause slow parsing.
Luckily, tokens are a level higher than that, and can thus be parsed with C.

Grammars

If we were attempting to define a language in mathematical form, we may do it like so:

$$L = [a^m b^n | m < n], \text{ tokens: a, b}$$

This would give us the language "", "b", "ab", "abb", ...
This doesn't scale well to programming languages. Instead we abstract to **grammars**

A few definitions are necessary to begin this discussion:
grammar ≡ a description of a language's syntax
language ≡ a set of sentences
sentence ≡ a string within a language
string ≡ a finite sequence of tokens
The specific grammars in this course are called **Context-Free Grammars**
A CFG is a set containing

- a finite set of tokens/terminal symbols (leaves)

- a finite set of non-terminal symbols (intermediate nodes)
- a finite set of rules (parent/child relationships)
- a start symbol (root)

Thus they can be defined by a grammar trees like the above.
They are called "context free" because an expression's definition is free of its context.
For clarity, consider a few counterexamples:
in C, 'typedef int foo; ... foo i;' is valid, but 'foo i;' is not.
in Python, 'if (true) pass' is invalid due to indentation rules.