Programming Languages

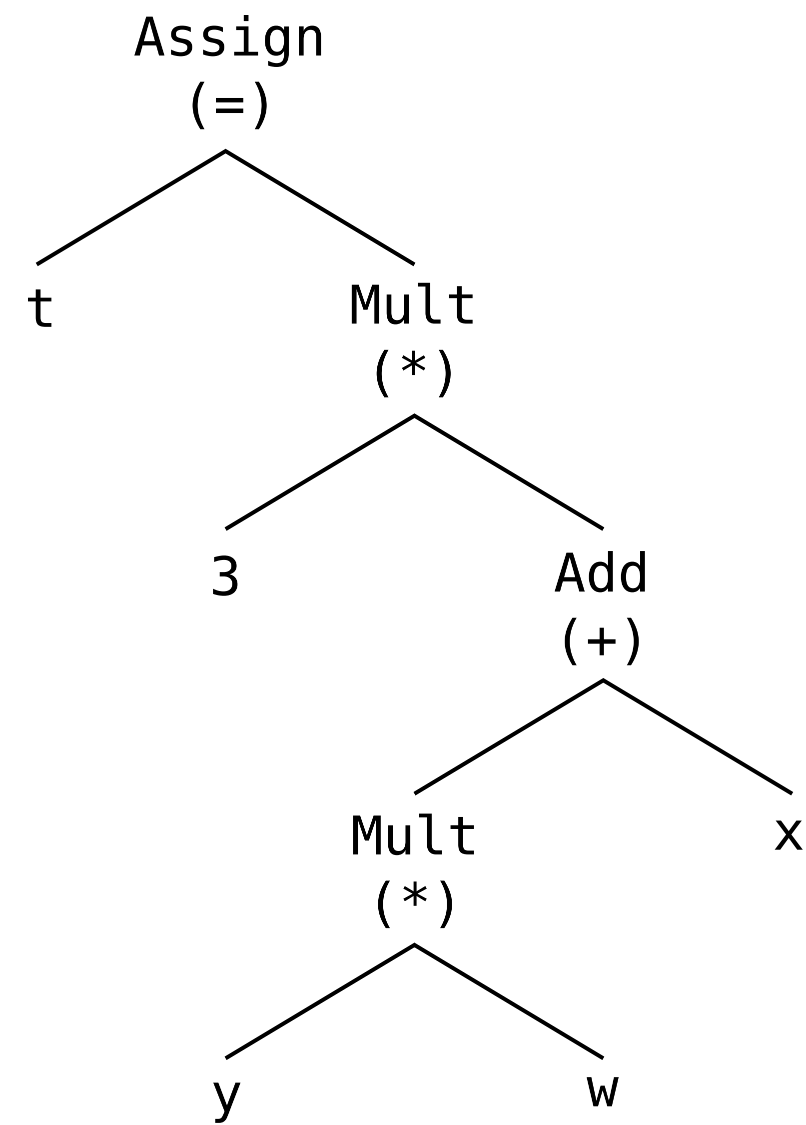
Compilers and interpreters

Parsing

Parsing is the process of taking the source code and creating the corresponding abstract syntax tree (AST).

Example:

```
t = 3 * (y * w + x)
```



(Assign t (Mult 3 (Add (Mult y w) x)))

Compiler vs. interpreter

- A compiler:
  - parses the source code into an AST
  - takes the AST and [...] writes the corresponding
    - machine code, or
    - assembly, or
    - source code in another language (sometimes called “transpilation”)

- An interpreter:
  - parses the source code into an AST
  - takes the AST and performs the corresponding operations

Example

Consider an imaginary programming language called “silly”.

A silly source code file consists in a single character string.

The resulting silly program shall print that string a million times.

The silly interpreter

```
import sys

with open(sys.argv[1]) as f:
    string = f.read().strip()

for i in range(1000000):
    print(f'{i:7d} {string}')
```

The silly compiler

```
import sys
import subprocess

with open(sys.argv[1]) as f:
    string = f.read().strip()

c_code = f'''
#include <stdio.h>

int main()
{{
    for (int i = 0; i < 1000000; i++) {{
        printf("%7d %s\\n", i, "{string}");
    }}
}}
'''

subprocess.run(
    [ 'clang', '-Wall', '-O3', '-o', 'silly_executable', '-x', 'c', '-' ],
    input=c_code,
    text=True
)
```

Pros and cons

Advantages of interpreters:

- No need for a compilation step
- In particular, no need to compile for each different platfom (portability)

Disadvantages of interpreters:

- Interpreter needs to be present on the user's machine
- An interpreter will run the code slower than native machine code
  - parsing can take time
  - need to traverse the AST and lookup identifiers
    before performing the appropriate operations

Compiled or interpreted is not an inherent property of a language.

Example: Python

- CPython (the reference and most common Python implementation) is an interpreter
- Cython is a compiler

Still, languages usually have a default / preferred way

Compiled languages:

- C, C++
- Rust, Go, Zig
- Pascal, Fortran, COBOL

Interpreted languages:

- Python, Javascript, Lua
- Lisp, Perl, PHP, R, Ruby, VBScript

Compile… to what?

By default,

- The Nim compiler produces C code (which is then compiled)
- The Dart compiler produces JavaScript code (then interpreted)
- Java compiles to “Java Virtual Machine” (JVM) code
    - the JVM can be seen as an ISA for a processor that does not exist
    - the JVM code is shipped to the user
    - the JVM code is then interpreted
    - advantage: JVM code is portable
    - drawback: user must have the JVM interpreter installed
- The Python interpreter (CPython) actually produces “Python bytecode” and immediately interprets it

- What about shipping the source code to the user…
- … then the user compiles it and runs it?
- The result would be both portable and fast.
- To avoid long compilation delays,
  compilation is done section-by-section (file, function or code block)…
- … just before the corresponding code needs to be run.
- This is Just-in-time (JIT) compilation.

Languages with JIT compilation

- Julia
- C#
- Java (source code compiled to JVM code; JVM code JIT compiled to native code)
- PyPy (Python)
- LuaJIT (Lua)

Pros and cons (summary)

	Compiled	Interpreted	Compiled to VM	Just-in-time
Needs compilation step	yes	no	yes	no
Needs interpreter / VM	no	yes	yes	yes
Portable	no	yes	yes	yes
Speed	fast	slow	in-between	slow at first, then fast

Language summary

- Ahead-of-time (AOT) compiled-to-machine-code languages:
  - C, C++, Rust, Go, Zig, Pascal, Fortran, COBOL
  - Nim (through C)
- Purely interpreted languages:
  - Lisp, Perl, R, VBScript
- Other:
  - Python, Lua: internally compiled to bytecode, then interpreted
  - PyPy (Python), LuaJIT (Lua): internally compiled to bytecode, then JIT compiled
  - Java, C#: explicitly compiled to bytecode (bytecode shipped to user), then JIT compiled
  - Julia: JIT compiled
  - JavaScript: interpreted and JIT compiled

Types

```
    int a;
//    ^ the type of a is int
```

```
>>> a = 5
>>> type(a)
<class 'int'>
```

Static or dynamic type checking

- Static type checking: type errors are always caught (e.g. at compile time)
- Dynamic type checking: type errors are caught only when (if) the code is run

Static type checking (C):

```c
int f()
{
    return "this is a string" / 5;
    // ^ even though f() is never used, this yields:
    // error: invalid operands to binary / (have 'char *' and 'int')
}

// f() is never used
```

Dynamic type checking (Python):

```python
def f():
    return "this is a string" / 5

# ...
# as long as f() is not used, not problem
# ...

f()

TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

Static type checking (TypeScript):

```
function f(): number
{
    return "this is a string" / 5;
    //     ^ ERROR: The left-hand side of an arithmetic operation must be of type
    //              'any', 'number', 'bigint' or an enum type.(2362)
    //              (even if f() is never used)
}
```

Dynamic type checking (JavaScript):

```
function f()
{
    return "this is a string" / 5;
    //     ^ returns special value NaN
}
```

```
>>> class C:
...     pass
...
>>> x = C()
>>> x.b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'C' object has no attribute 'b'
```

```
>>> class C:
...     pass
...
>>> x = C()
>>> x.b = 1
>>> x.b
1
```

Strong and weak typing

“Strong” and “weak” are vague qualifiers to indicate how strict a language is with type conversions.

Weak typing (C):

```
int a = -1.8; // not an error, value silently truncated (towards zero) to -1
```

```
int *p = (int *)((long int)"abc" + 5) // will compile
*p = 3; // will probably crash
```

Strong typing (Python):

```
>>> "a" + 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

but

```
>>> "a" * 4
'aaaa'
```

Memory Management

Manual memory management

in C:

```
int getint()
{
    char *buffer = malloc(1024);

    size_t n = fread(buffer, 1, 1023, stdin);

    buffer[n] = 0;

    return strtol(buffer, NULL, 0);
}
```

- We did not check that `malloc(1024)` worked
- We forgot `free(buffer)`!

```c
int getint()
{
    char *buffer = malloc(1024);

    if (buffer == NULL) {
        perror("malloc()");
        abort();
    }

    size_t n = fread(buffer, 1, 1023, stdin);

    if (ferror(stdin)) {
        perror("fread()");
        abort();
    }

    buffer[n] = 0;

    int r = strtol(buffer, NULL, 0);

    free(buffer); // <-------- free memory

    return r;
}
```

Automatic memory management

in Python:

```
def getint():
    buffer = input()
    return int(buffer)
```

How does automatic memory management work?

We need to keep track of the memory that is **in use**.

- Reference counting
- Garbage collection

Reference counting

```
struct object_t {
    int refcount;
    ...
};

void object_ref(struct object_t *obj)
{
    obj->refcount = obj->refcount + 1;
}

void object_unref(struct object_t *obj)
{
    obj->refcount = obj->refcount - 1;

    if (obj->refcount == 0) {
        free(obj);
    }
}
```

```python
def remove_comments(text):
    lines = text.split('\n')
    filtered = [ l for l in lines if l[0:1] != '#' ]
    r = '\n'.join(filtered)
    return r

t = remove_comments('a\n# comment\nb\nc')
```

string	ref. count	variables

```python
def remove_comments(text):                          # <---------
    lines = text.split('\n')
    filtered = [ l for l in lines if l[0:1] != '#' ]
    r = '\n'.join(filtered)
    return r

t = remove_comments('a\n# comment\nb\nc')
```

string	ref. count	variables
`'a\n# comment\nb\nc'`	1	text

```python
def remove_comments(text):
    lines = text.split('\n')                        # <--------- lines = ['a', '# comment', 'b', 'c']
    filtered = [ l for l in lines if l[0:1] != '#' ]
    r = '\n'.join(filtered)
    return r

t = remove_comments('a\n# comment\nb\nc')
```

string	ref. count	variables
'a\n# comment\nb\nc'	1	text
'a'	1	lines[0]
'# comment'	1	lines[1]
'b'	1	lines[2]
'c'	1	lines[3]

```python
def remove_comments(text):
    lines = text.split('\n')                            #            lines = ['a', '# comment', 'b', 'c']
    filtered = [ l for l in lines if l[0:1] != '#' ]    # <--------- filtered = ['a', 'b', 'c']
    r = '\n'.join(filtered)
    return r

t = remove_comments('a\n# comment\nb\nc')
```

string	ref. count	variables
'a\n# comment\nb\nc'	1	text
'a'	2	lines[0], filtered[0]
'# comment'	1	lines[1]
'b'	2	lines[2], filtered[1]
'c'	2	lines[3], filtered[2]

```python
def remove_comments(text):
    lines = text.split('\n')                                #             lines = ['a', '# comment', 'b', 'c']
    filtered = [ l for l in lines if l[0:1] != '#' ]        #             filtered = ['a', 'b', 'c']
    r = '\n'.join(filtered)                                 # <--------- r = 'a\nb\nc'
    return r

t = remove_comments('a\n# comment\nb\nc')
```

string	ref. count	variables
`'a\n# comment\nb\nc'`	1	`text`
`'a'`	2	`lines[0], filtered[0]`
`'# comment'`	1	`lines[1]`
`'b'`	2	`lines[2], filtered[1]`
`'c'`	2	`lines[3], filtered[2]`
`'a\nb\nc'`	1	`r`

```python
def remove_comments(text):
    lines = text.split('\n')                           #             lines = ['a', '# comment', 'b', 'c']
    filtered = [ l for l in lines if l[0:1] != '#' ]   #             filtered = ['a', 'b', 'c']
    r = '\n'.join(filtered)                            #             r = 'a\nb\nc'
    return r                                           # <---------- return

t = remove_comments('a\n# comment\nb\nc')
```

string	ref. count	variables
`'a\n# comment\nb\nc'`	1	`text`
`'a'`	2	`lines[0], filtered[0]`
`'# comment'`	1	`lines[1]`
`'b'`	2	`lines[2], filtered[1]`
`'c'`	2	`lines[3], filtered[2]`
`'a\nb\nc'`	2	r, (return value)

```python
def remove_comments(text):
    lines = text.split('\n')                              #           lines = ['a', '# comment', 'b', 'c']
    filtered = [ l for l in lines if l[0:1] != '#' ]      #           filtered = ['a', 'b', 'c']
    r = '\n'.join(filtered)                               #           r = 'a\nb\nc'
    return r                                              # <--------- return

t = remove_comments('a\n# comment\nb\nc')
```

string	ref. count	variables
~~'a\n# comment\nb\nc'~~	0	~~text~~
~~'a'~~	0	~~lines[0], filtered[0]~~
~~'# comment'~~	0	~~lines[1]~~
~~'b'~~	0	~~lines[2], filtered[1]~~
~~'c'~~	0	~~lines[3], filtered[2]~~
'a\nb\nc'	1	~~r~~, (return value)

```python
def remove_comments(text):
    lines = text.split('\n')                                #           lines = ['a', '# comment', 'b', 'c']
    filtered = [ l for l in lines if l[0:1] != '#' ]       #           filtered = ['a', 'b', 'c']
    r = '\n'.join(filtered)                                #           r = 'a\nb\nc'
    return r                                               #           return

t = remove_comments('a\n# comment\nb\nc')                  # <--------- t = 'a\nb\nc'
```

string	ref. count	variables
~~'a\n# comment\nb\nc'~~	0	~~text~~
~~'a'~~	0	~~lines[0], filtered[0]~~
~~'# comment'~~	0	~~lines[1]~~
~~'b'~~	0	~~lines[2], filtered[1]~~
~~'c'~~	0	~~lines[3], filtered[2]~~
'a\nb\nc'	1	~~r~~, ~~(return value)~~, t

```python
def build_local_chain_but_do_not_return_it():
    root = {}
    a = { 'parent': root }
    b = { 'parent': a }
    c = { 'parent': b }                  # <------------ c = { 'parent: { 'parent': { 'parent': {} } } }
    return None
```

dict	ref. count	binding
`{}`	2	`root, dict() "a" below`
`{ 'parent': root }`	2	`a, dict() "b" below`
`{ 'parent': a }`	2	`b, dict() "c" below`
`{ 'parent': b }`	1	`c`

```python
def build_local_chain_but_do_not_return_it():
    root = {}
    a = { 'parent': root }
    b = { 'parent': a }
    c = { 'parent': b }                    #
    return None                            # <------------- return
```

dict	ref. count	binding
{}	1	~~root~~, dict() "a" below
{ 'parent': root }	1	~~a~~, dict() "b" below
{ 'parent': a }	1	~~b~~, dict() "c" below
{ 'parent': b }	0	~~c~~

```python
def build_local_chain_but_do_not_return_it():
    root = {}
    a = { 'parent': root }
    b = { 'parent': a }
    c = { 'parent': b }                     #
    return None                             # <------------ return
```

dict	ref. count	binding
`{}`	1	~~root~~, dict() "a" below
`{ 'parent': root }`	1	~~a~~, dict() "b" below
`{ 'parent': a }`	0	~~b~~, ~~dict() "c" below~~
~~`{ 'parent': b }`~~	0	~~c~~

```python
def build_local_chain_but_do_not_return_it():
    root = {}
    a = { 'parent': root }
    b = { 'parent': a }
    c = { 'parent': b }                  #
    return None                          # <------------ return
```

dict	ref. count	binding
{}	1	~~root~~, dict() "a" below
{ 'parent': root }	0	~~a~~, ~~dict() "b" below~~
~~{ 'parent': a }~~	0	~~b~~, ~~dict() "c" below~~
~~{ 'parent': b }~~	0	~~c~~

```python
def build_local_chain_but_do_not_return_it():
    root = {}
    a = { 'parent': root }
    b = { 'parent': a }
    c = { 'parent': b }                 #
    return None                         # <------------ return
```

dict	ref. count	binding
{}	0	~~root~~, ~~dict() "a" below~~
~~{ 'parent': root }~~	0	~~a~~, ~~dict() "b" below~~
~~{ 'parent': a }~~	0	~~b~~, ~~dict() "c" below~~
~~{ 'parent': b }~~	0	~~c~~

```python
def build_local_chain_but_do_not_return_it():
    root = {}
    a = { 'parent': root }
    b = { 'parent': a }
    c = { 'parent': b }                 #
    return None                         # <------------ return
```

dict	ref. count	binding
~~{}~~	0	~~root~~, ~~dict() "a" below~~
~~{ 'parent': root }~~	0	~~a~~, ~~dict() "b" below~~
~~{ 'parent': a }~~	0	~~b~~, ~~dict() "c" below~~
~~{ 'parent': b }~~	0	~~c~~

Problem with refcounting

```
def build_local_chain_but_do_not_return_it():
    root = {}
    a = { 'parent': root }
    b = { 'parent': a }
    c = { 'parent': b }
    root['parent'] = c                    # <------------ root = { 'parent: { 'parent': { 'parent': ...
    return None
```

dict	ref. count	binding
`{ 'parent': c }`	2	`root, dict() "a" below`
`{ 'parent': root }`	2	`a, dict() "b" below`
`{ 'parent': a }`	2	`b, dict() "c" below`
`{ 'parent': b }`	2	`c, dict() "root" above`

Problem with refcounting

```
def build_local_chain_but_do_not_return_it():
    root = {}
    a = { 'parent': root }
    b = { 'parent': a }
    c = { 'parent': b }
    root['parent'] = c                     #
    return None                            # <------------ return
```

dict	ref. count	binding
`{ 'parent': c }`	1	~~root~~, `dict() "a" below`
`{ 'parent': root }`	1	~~a~~, `dict() "b" below`
`{ 'parent': a }`	1	~~b~~, `dict() "c" below`
`{ 'parent': b }`	1	~~c~~, `dict() "root" above`

Cycles

```
def build_local_chain_but_do_not_return_it():
    root = {}
    current = root

    for i in range(10000000):
        current = { 'parent': current }

    root['parent'] = c

    return None
```

Garbage collection

- keep track of all variables in scope
- keep track of all allocated blocks of memory
- every few seconds, “garbage collection”
  - look through all the variables, if they reference some memory, mark it as in-use
  - look at every block, if not referenced, free it

- Pro: does not suffer from cycle issue
- Con: memory usage can grow a lot between garbage collections
- Con: garbage collections pauses can block the process for a long time

  (making it feel unresponsive)

Other language features

Macros

Macros allow us to generate fragments of source code automatically.

C macro example:

```
#define THIS_5X(a)      a, a, a, a, a

int array[10] = { THIS_5X(1), THIS_5X(2) };
```

equivalent to:

```
int array[10] = { 1, 1, 1, 1, 1, 2, 2, 2, 2, 2 };
```

Macros can be useful:

```
#define ARRAY_ELEMENTS(a)    (sizeof(a) / sizeof((a)[0]))
```

But beware! They are just text replacement:

```
#define PRODUCT_WRONG(a, b)     (a * b)

int a = PRODUCT_WRONG(1 + 2, 3 + 4); // <--- 1 + 2 * 3 + 4 = 11
```

```
#define PRODUCT_CORRECT(a, b)   ((a) * (b))

int a = PRODUCT_CORRECT(1 + 2, 3 + 4); // <--- (1 + 2) * (3 + 4) = 21
```

Generics

In C, those must be implemented separately:

```
void int_array_sort(int *array, int size);
void float_array_sort(float *array, int size);
```

In Python, because of dynamic type checking, there is no need:

```
def array_sort(array):
    # "<", "<=", "==", etc. will work for either int or float
```

The type of `array` will be figured out at runtime

To solve this, C++ adds generics:

```
template <typename T> void array_sort(T *array);
```

Languages with generics

- C++
- C#
- Java
- Go
- Rust
- Swift
- TypeScript
- …

Object-oriented programming

A **compound type** is any type that is defined in terms of one or more other types.

- In C:

```
struct point {
  float x;
  float y;
};
```

- In Python:

```
class Point:
    def __init__(self):
        self.x = 0.0
        self.y = 0.0
```

In **object-oriented programming** (OOP), compound types ("classes") can have functions attached to them ("methods").

- In C++:

```
struct point {
  float x;
  float y;

  void scale(float l) { x *= l; y *= l; };
};
```

- In Python:

```
class Point:
    def __init__(self):
        self.x = 0.0
        self.y = 0.0

    def scale(self, l):
        self.x *= l
        self.y *= l
```

As a consequence, in OOP, data and the methods that operate on them are usually defined close together.

We can construct complex type hierarchies:

- define a class for `vehicle`, has a `price` method
- define a class for `bike`, inherits from `vehicle`
  - inherits the `price` method from `vehicle` (no need to rewrite it)
  - among other properties, has two wheels
- define a class for `car`, inherits from `vehicle`
  - inherits the `price` method from `vehicle` (no need to rewrite it)
  - among others has four wheels
- etc.

Functional programming

In functional programming, functions are “first-class” types:

- they can be used in expressions
- they can be assigned to variables

```
def apply_to_all(array, fn):
    return [ fn(element) for element in array ]

def double_it(x):
    return x * 2

apply_to_all([0, 1, 2, 3, 4], double_it)

# -> [0, 2, 4, 6, 8]
```

Declarative and logic programming

We describe what we want, not how to get it.

Example: SAT for Boolean expressions

```
x1 and ((not x2) or x3) and (not x3)
```

We describe the constraints, not how to get a solution.

Example: Algebraic modeling languages

```
var x1 >= 0, integer;
var x2 >= 0, integer;

maximize

    hours:  40 * x1 +  80 * x2;

subject to

    budget: 31 * x1 +  45 * x2 <= 239;
    cap:         x1 - 0.5 * x2 >= 0;

end
```

GNU MathProg

Example: Constraint modelling languages

```
int: nc = 3;

var 1..nc: wa;   var 1..nc: nt;  var 1..nc: sa;   var 1..nc: q;
var 1..nc: nsw;  var 1..nc: v;   var 1..nc: t;

constraint wa != nt;
constraint wa != sa;
constraint nt != sa;
constraint nt != q;
constraint sa != q;
constraint sa != nsw;
constraint sa != v;
constraint q != nsw;
constraint nsw != v;
solve satisfy;
```

MiniZinc

Example: Satisfiability modulo theories (SMT)

```
(theory Ints
 :sorts ( (Int 0) )
 :funs ( (NUMERAL Int)
         (- Int Int) ; negation
         (- Int Int Int :left-assoc) ; subtraction
         (+ Int Int Int :left-assoc)
         (* Int Int Int :left-assoc)
         (<= Int Int Bool :chainable)
         (< Int Int Bool :chainable)
         (>= Int Int Bool :chainable)
         (> Int Int Bool :chainable) )
 :definition
 "For every expanded signature Sigma, the instance of Ints with that
  signature is the theory consisting of all Sigma-models that interpret
  - the sort Int as the set of all integers,
  - the function symbols of Ints as expected. "
 :values
 "The Int values are all the numerals and all the terms of the form (- n)
  where n is a non-zero numeral."
)
```

SMTLIB2