

Basic Elements of Programming Languages

Ronghui Gu

Spring 2019

Columbia University

* Course website: <https://www.cs.columbia.edu/~rgu/courses/4115/spring2019>

** These slides are borrowed from Prof. Edwards.

What is a Programming Language?

A programming language is a notation that a person and a computer can both understand.

- It allows you to express what is the **task** to compute
- It allows a computer to **execute** the computation task

Language Specifications

How to Define a Language

When designing a language, it's a good idea to start by sketching forms that you want to appear in your language as well as forms you do not want to appear.

```
int avg(int a, int b)
{
    return (a + b) / 2;
}
```

Examples

```
a int vg(int a,
{
    return (a; + b)
{ {
```

Non-Examples

How to Define a Language

- An official documents, with **informal** descriptions.
- An official documents, with **formal** descriptions.
- A reference implementation, e.g., a compiler.

Some language definitions are sanctioned by an official standards organization, e.g., C11 (ISO/IEC 9899:2011).

```
int compare()  
{  
    int a[10], b[10];  
    if (a > b)  
        return true;  
    return false;  
}
```

Aspects of Language Specifications

Syntax

Semantics

Pragmatics

- **Syntax:** how characters combine to form a program.
- **Semantics:** what the program *means*.
- **Pragmatics:** common programming idioms; programming environments; the standard library; ecosystems.

Syntax is divided into:

- **Microsyntax:** specifies how the characters in the source code stream are grouped into tokens.
- **Abstract syntax:** specifies how the tokens are grouped into phrases, e.g., expressions, statements, etc.

Source program is just a sequence of characters.

```
int avg(int a, int b)
{
    return (a + b) / 2;
}
```

```
i n t SP a v g ( i n t SP a , SP i n t SP b ) NL
{ NL
SP SP r e t u r n SP ( a SP + SP b ) SP / SP 2 ; NL
} NL
```


Microsyntax

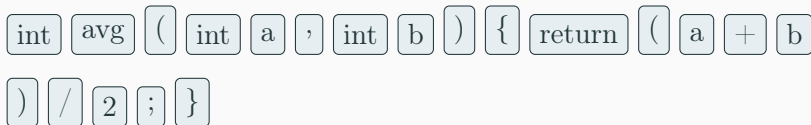
```
int avg(int a, int b)
{
    return (a + b) / 2;
}
```

Token	Lexemes	Pattern (as regular expressions)
ID	avg, a, b	letter followed by letters or digits
KEYWORD	int, return	letters
NUMBER	2	digits
OPERATOR	+, /	+, /
PUNCTUATION	;,(),{,},	;,(),{,},

int avg (int a , int b) { return (a + b
) / 2 ; }

Lexical Analysis Gives Tokens

```
int avg(int a, int b)
{
    return (a + b) / 2;
}
```



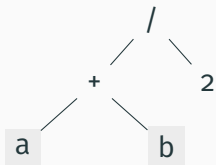
- Throw errors when failing to create tokens: malformed numbers (e.g., 23f465#g) or invalid characters (such as non-ASCII characters in C).

Abstract Syntax

Abstract Syntax can be defined using Context Free Grammar.

```
expr :  
    expr OPERATOR expr  
    | ( expr )  
    | NUMBER
```

Expression $(a + b)/2$ can be parsed into an AST:

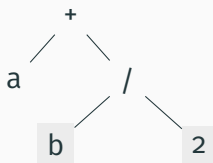
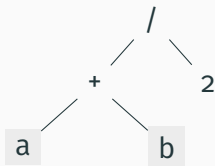


Abstract Syntax

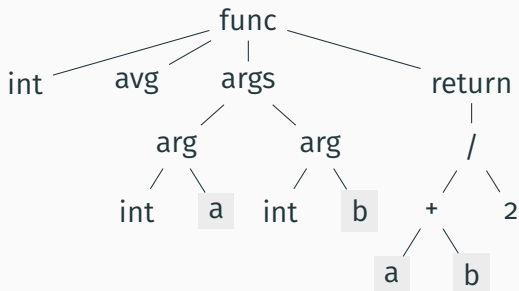
Abstract Syntax can be defined using Context Free Grammar.

```
expr :  
    expr OPERATOR expr  
    | ( expr )  
    | NUMBER
```

Ambiguous! What about $a + b/2$?



Syntax Analysis Gives an Abstract Syntax Tree



```
int avg(int a, int b)
{
    return (a + b) / 2;
}
```

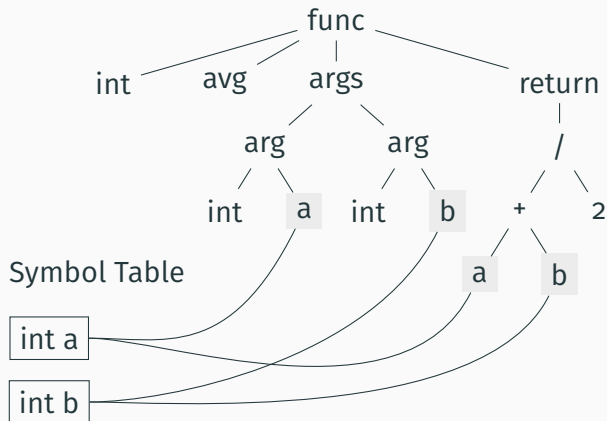
- Syntax analysis will throw errors if “}” is missing. Lexical analysis will not.

- **Static Semantics:** deals with legality rules—things you can check **before** running the code (compile time), e.g., type, scope, for some languages.
- **Dynamic Semantics:** deals with the execution behavior; things that can only be known **at** runtime, e.g., value.

We can use inference rules to define semantics, e.g., type:

$$\frac{}{\text{NUMBER} : \mathbf{int}} \qquad \frac{\text{expr} : \mathbf{int}}{(\text{expr}) : \mathbf{int}}$$
$$\frac{\text{expr}_1 : \mathbf{int} \quad \text{expr}_2 : \mathbf{int}}{\text{expr}_1 \text{ OPERATOR } \text{expr}_2 : \mathbf{int}}$$

Semantic Analysis: Resolve Symbols; Verify Types



We can use inference rules to define semantics, e.g., value:

$$\frac{}{\mathbf{eval}(\text{NUMBER}) = \text{NUMBER}} \qquad \frac{\mathbf{eval}(\text{expr}) = n}{\mathbf{eval}((\text{expr})) = n}$$
$$\frac{\mathbf{eval}(\text{expr}_1) = n_1 \quad \mathbf{eval}(\text{expr}_2) = n_2 \quad (n_1 + n_2) = n}{\mathbf{eval}(\text{expr}_1 + \text{expr}_2) = n}$$

Consider the integer range:

$$\frac{\text{wrap}(\text{NUMBER}) = n}{\text{eval}(\text{NUMBER}) = n}$$

$$\frac{\text{eval}(\text{expr}) = n}{\text{eval}((\text{expr})) = n}$$

$$\frac{\text{eval}(\text{expr}_1) = n_1 \quad \text{eval}(\text{expr}_2) = n_2 \quad \text{wrap}(n_1 + n_2) = n}{\text{eval}(\text{expr}_1 + \text{expr}_2) = n}$$

Programming Paradigms

Programming Paradigms

A programming paradigm is a **style**, or “way,” of programming. Some languages make it easy to write in some paradigms but not others.

Imperative Programming

An imperative program specifies **how** a computation is to be done: a sequence of statements that update state.

```
    result = []
    i = 0
    numStu = len(students)
start:
    if i >= numStu goto finished
    name = students[i]
    nameLength = len(name)
    if nameLength <= 5 goto nextOne
    addToList(result, name)
nextOne:
    i = i + 1
    goto start
finished:
    return result
```

Structured Programming

A kind of imperative programming with clean, **goto-free**, nested control structures. **Go To Statement Considered Harmful** by Dijkstra.

```
result = []  
for i in range(len(students)):  
    name = students[i]  
    if len(name) > 5:  
        addToList(result, name)  
print(result)
```

Procedural Programming

Imperative programming with **procedure calls**.

```
def filterList (students):  
    result = []  
    for name in students:  
        if len(name) > 5:  
            addToList(result , name)  
    return result  
  
print (filterList (students))
```

Object-Oriented Programming

An object-oriented program does its computation with interacting **objects**.

```
class Student:
    def __init__(self, name):
        self.name = name
        self.department = "CS"

def filterList (students):
    result = []
    for student in students:
        if student.name.__len__() > 5:
            result.append(student.name)
    return result

print (filterList (students))
```


Declarative Programming

A declarative program specifies **what** computation is to be done. It expresses the logic of a computation without describing its control flow.

```
select name  
from students  
where length(name) > 5
```

Functional Programming

A functional program treats computation as the evaluation of mathematical functions and **avoids** side effects.

```
def isNameLong (name):  
    return len(name) > 5  
  
print(  
    list(  
        filter(isNameLong, students)))
```

Using lambda calculus:

```
print(  
    list(  
        filter(lambda name: len(name), students)))
```

Using function composition:

```
compose(print , list , filter*(lambda name: len(name) > 5))  
      (students)
```

* A variant of the built-in filter.