5118014 Principles of Programming Languages

Lecture 4. Syntax and Semantics

Shin Hong

Concrete Syntax and Abstract Syntax

- **concrete syntax** defines textual representation (string) of a program
- **abstract syntax** defines structural representation (tree) of a program
  - typically, a program consists of multiple components which form a tree structure
  - a certain algorithm is represented differently in concrete syntaxes, while having the identical structure in their abstract syntaxes

Example

- Python

```python
def add(n, m):
    return n + m
```

- JavaScript

```javascript
function add(n, m) {
    return n + m;
}
```

- Racket

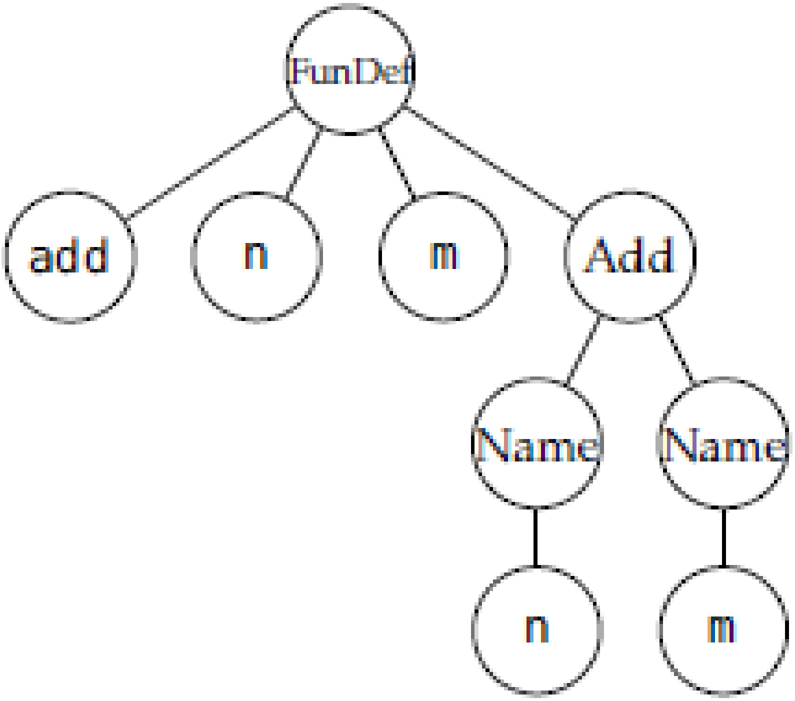
```racket
(define (add n m) (+ n m))
```

- OCaml

```ocaml
let add n m = n + m
```

- Rust

```rust
fn add (n: i32, m: i32) {
    n + m
}
```



Grammar

- a grammar is a set of rules defining the syntax of a language
  - recursive, constructive definitions to define an infinite set
- Backus-Naur Form (BNF)
  - components
    - terminal: a string
    - non-terminal: a name denoting a set of strings
    - formula: a combination of one or more terminals, non-terminals, or operators
  - production rules: a nonterminal is replaced by an expression

```
[nonterminal] ::= [formula] | [formula] | [formula] | …
```

Example. Arithmetic Expression

```
<digit>  ::= "0" | "1" | "2" | "3" | "4"
           | "5" | "6" | "7" | "8" | "9"
<nat>    ::= <digit> | <digit> <nat>
<number> ::= <nat> | "-" <nat>


<expr> ::= <number> | <expr> "+" <expr> | <expr> "-" <expr>
```

Arithmetic expression is the set of strings derived from `<expr>`

Grammar Operators https://web.mit.edu/6.031/www/sp21/classes/17-regex-grammars/

- Repetition (highest precedence) e.g., `x ::= y*`
- Concatenation e.g., `x ::= y z`
- Union (lowest precedence) e.g., `x ::= y | z`

- Ex. URL

```
url ::= 'http://' (hostname | hostname '/')
hostname ::= word ('.' word)*
word ::= letter letter*
letter ::= ('a' | 'b' | 'c' | 'd' | 'e' | … | 'z')
```

More Operators

https://web.mit.edu/6.031/www/sp21/classes/17-regex-grammars/

- 0-or-1 occurrence e.g., `x ::= y?`
- 1-or-more occurrences e.g., `x ::= y+`
- character class e.g., `x ::= [aeiou]`

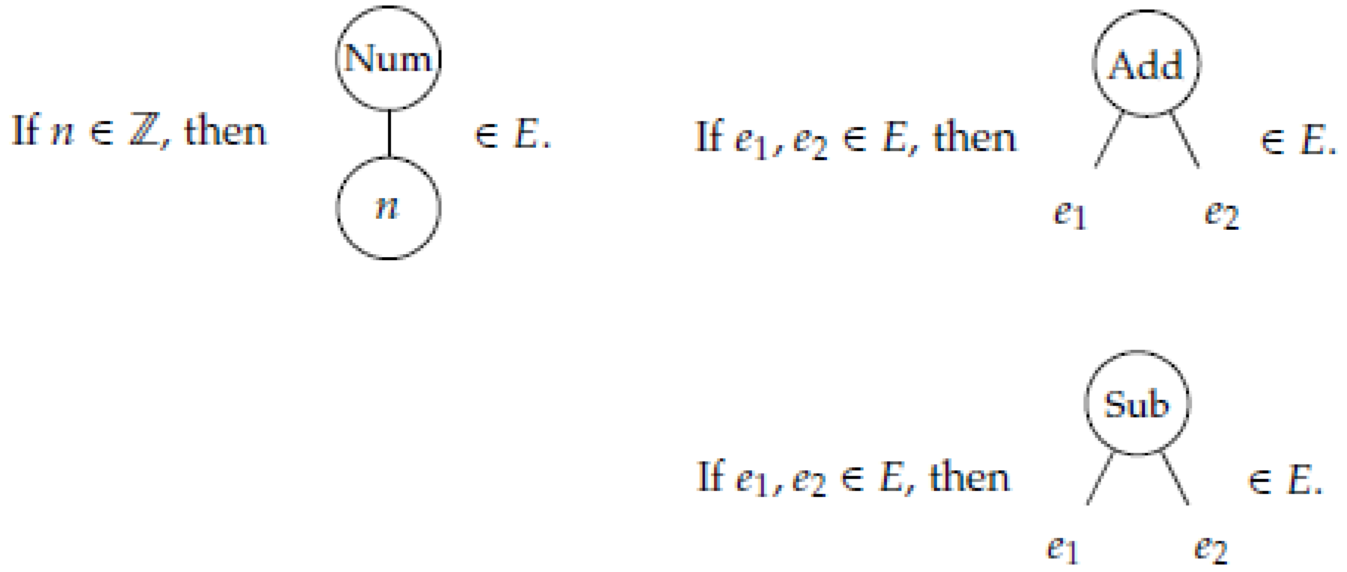
```
x ::= [0-9A-F]
x ::= [^a-c]
```

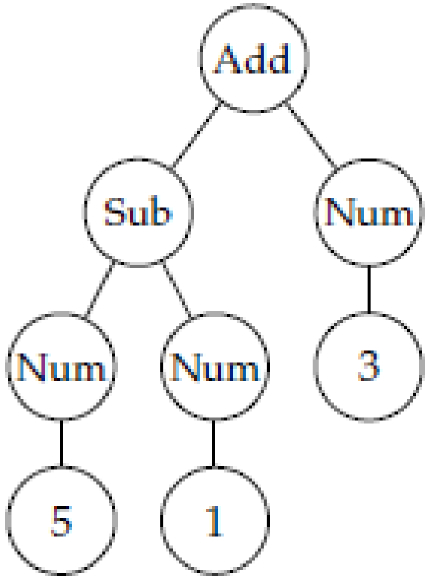
- Ex. URL

```
url ::= ‘http://’ hostname ‘/’?
hostname ::= word (‘.’ word)*
word ::= [a-z]+
```

Abstract Syntax Tree

- Define the set of all trees that represent programs
- Ex. Arithmetic Expression





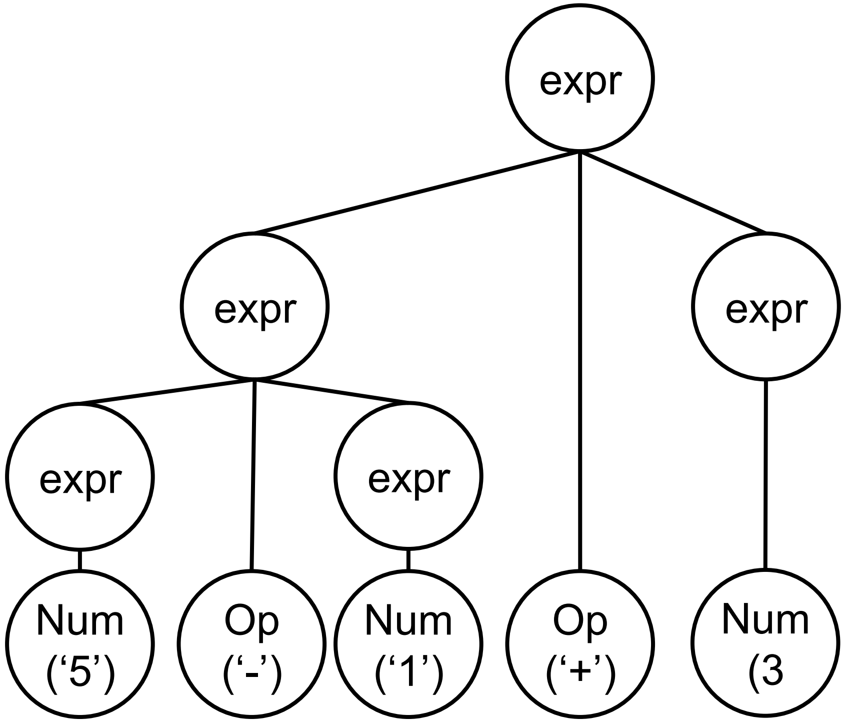
Regular Expression (Regex)

- A token is a pattern accepting a certain set of strings
- In most PLs, a code text is first converted into a sequence of tokens, and a grammar is defined with tokens as terminals
- In most PLs, a token is written as a regular expression
- Components
  - operators: repetition, concatenation, union
  - character class: [a-z], [^xyz], [[:alpha:]], [[:digit:]] , [[:blank:]]
  - special character: . , \d, \s, \w, \n, ^, $, \n, \\

Example. Arithmetic Expression (Revised)

```
<num> = ('-')?[0-9]+
<opr> = '+' | '-'

<expr> ::= <num> |
           '('<expr> <opr> <expr>')' |
           '('<expr>')'
```



Example

- `url ::= ‘http://’ ([a-z]+ ‘.’)+ [a-z]+ (‘:’ [0-9]+)? ‘/’?`

- Dates (e.g., 2025-09-22)

- Floating number (e.g., 0.12, -4.666)

Parsing

- concrete syntax considers a program as a string, while abstract syntax as a tree
- **parsing** is an operation that transforms a valid string into the corresponding AST of the abstract syntax
  - a parser is a partial function from S (the set of all strings) to E (the set of all ASTs)

$$parse : S \rightharpoonup E$$

Parser Generator

- Programming a parser is extremely complicated and error-prone
- The formal principles for accurately describing, analyzing, interpreting language grammars are well established, as the **parsing theories**
  - e.g., LL(k) parsing, LR(k) parsing, LALR(k) parsing
- A **parser generator** automatically synthesizes the parser program accepting the language of a given formal description of a grammar
  - e.g., lex/yacc, flex/bison, ANTLR, LALRPOP
  - many parsers for Domain-Specific Languages (DSL) are built using parser generators, or implemented upon meta-languages in practice, although still many are constructed ad-hoc

Semantics

- semantics is defined as a function that maps ASTs to values
  - semantics defines the results of the program executions with inputs

- semantics are typically specified as recursive transformation rules
  - recursion is used for there are infinitely many different programs

Ex. Semantics of Arithmetic Expression

- We can define the semantics of AE as $\Rightarrow$, a binary relation over E and $\mathbb{Z}$

$$\Rightarrow \subseteq E \times \mathbb{Z}$$

Rule Num

$\mathbf{n} \Rightarrow n$

Rule Add

$\mathbf{e_1} + \mathbf{e_2} \Rightarrow n_1 + n_2$
  if $\mathbf{e_1} \Rightarrow n_1$ and $\mathbf{e_2} \Rightarrow n_2$

Rule Sub

$\mathbf{e_1} \text{-} \mathbf{e_2} \Rightarrow n_1 - n_2$
  if $\mathbf{e_1} \Rightarrow n_1$ and $\mathbf{e_2} \Rightarrow n_2$

Function

- A function is a relation between two sets (domain and co-domain), such that every domain element is related with exactly one co-domain element
- Notation

$$\Rightarrow \subseteq A \times B$$

$$\Rightarrow : A \rightarrow B$$

$$\Rightarrow \in A \rightarrow B$$

- $A \rightarrow B$ stands for the set of all possible functions between A and B, thus $A \rightarrow B$ is a subset of all possible binary relations between A and B, $\mathcal{P}(A \times B) = 2^{A \times B}$

Example

- A = { x, y, z } and B = { 0, 1 }
- $A \times B\ =$ { (x, 0), (x, 1), (y, 0), (y, 1), (z, 0), (z, 1) }
- $\mathcal{P}(A \times B)$ = { {},

  {(x, 0)}, {(x, 1)}, {(y, 0)}, {(y, 1)}, {(z, 0)}, {(z, 1)},

  {(x, 0),(x, 1)}, {(x, 0),(y, 0)} …, {(y,1),(z,1)}, {(z, 0),(z, 1)},

  ⋮

  {(x, 0), (x, 1), (y, 0), (y, 1), (z, 0), (z, 1)} }

- $A \rightarrow B$ = { {(x,0),(y,0),(z,0)}, {(x,0),(y,0),(z,1)}, {(x,0),(y,1),(z,0)},
  … , {(x,1),(y,1),(z,0)}, {(x,1),(y,1),(z,1)} }

Inference Rule

- a rule to derive a new proposition from given propositions
  - structure

$$\frac{\mathit{premise}_1 \qquad \mathit{premise}_2 \qquad \cdots \qquad \mathit{premise}_n}{\mathit{conclusion}}$$

- a proof tree is a tree whose root is the proposition to be proven
  - each node is a proposition, and its children are supporting evidences
  - the roots are of axioms (i.e., conclusions without any premises)

Arithmetic Expression: Inference Rules

$$\mathbf{n} \Rightarrow n \qquad [\textsc{Num}]$$

$$\frac{e_1 \Rightarrow n_1 \qquad e_2 \Rightarrow n_2}{e_1 \mathbin{\mathbf{+}} e_1 \Rightarrow n_1 + n_2} \qquad [\textsc{Add}]$$

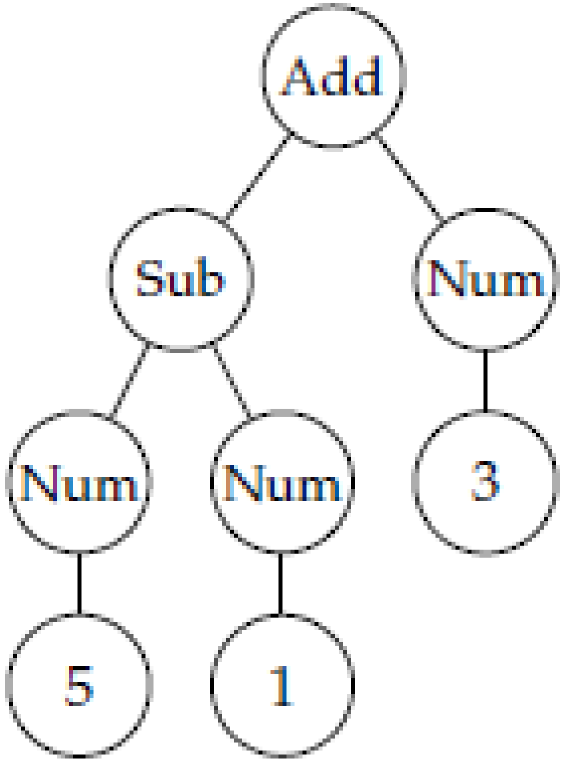
$$\frac{e_1 \Rightarrow n_1 \qquad e_2 \Rightarrow n_2}{e_1 \mathbin{\mathbf{-}} e_1 \Rightarrow n_1 - n_2} \qquad [\textsc{Sub}]$$

$$\frac{\dfrac{3 \Rightarrow 3 \qquad 1 \Rightarrow 1}{3 - 1 \Rightarrow 2} \qquad 2 \Rightarrow 2}{(3 - 1) + 2 \Rightarrow 4}$$

Arithmetic Expression AST: Rust

```
enum Expr {
        Num(i32),
        Op(Box<Expr>, Opr, Box<Expr>),
}
enum Opr {
        Add,
        Sub,
}

e0 = Box::new(Op(Box::new(Num(5)), Sub, Box::new(Num(1))))
e1 = Box::new(e0 , Add, Box::new(Num(3)))
e2 = add(sub(num(5), num(1)), num(3))
```



Interpreter

- An interpreter is a function that receives a program and an input data, and evaluates the program with the given input data
- Example

```
/* ae/src/main.rs */
fn interp (e: Box<Expr>) -> i32 {
    match *e {
        Op(l, Add, r) => interp(l) + interp(r),
        Op(l, Sub, r) => interp(l) - interp(r),
        Num(n) => n
    }
}
```

Syntactic Sugar

- **Syntactic sugar** adds a new feature to a language by defining syntactic transformation rules instead of changing the semantics

- Example. adding integer negation to AE
  - syntax:

```
<expr> ::= <number> | <expr> "+" <expr>
         | <expr> "-" <expr> | "-" "(" <expr> ")"
```

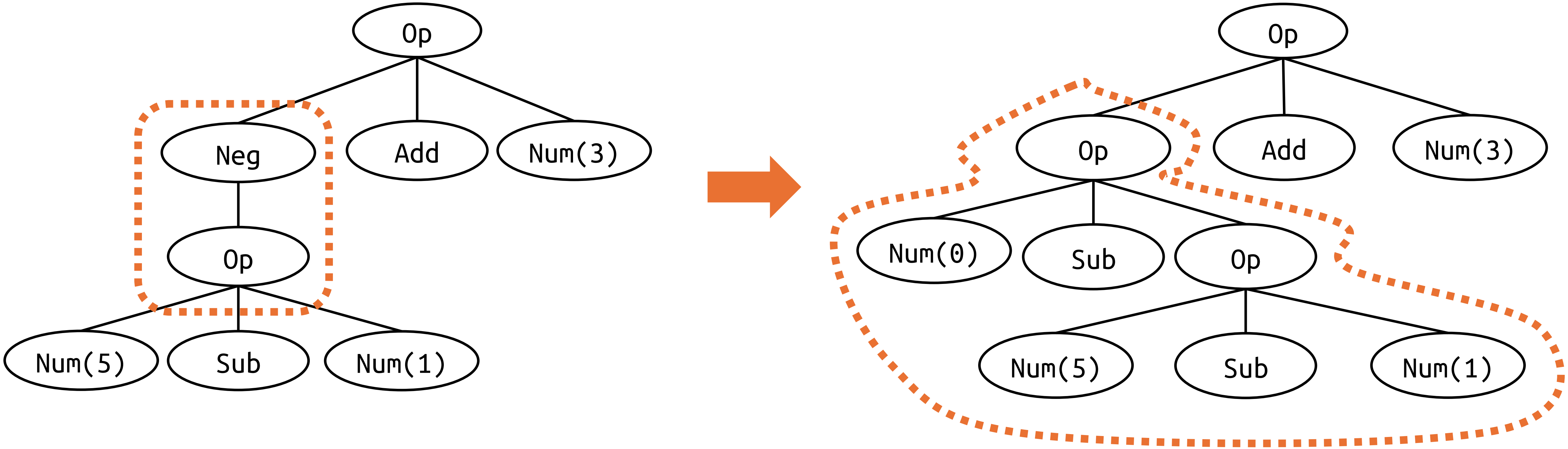
  - extending semantics:

$$\frac{e \Rightarrow n}{-e \Rightarrow -_{\mathbb{Z}} n} \ \textsc{Neg}$$

  - syntactic transformation (desugaring): transform `Neg(e)` into `Sub(Num(0), e)`

Example

`(-(5 - 1) + 3)`


Op
Neg
Add
Num(3)
Op
Num(5)
Sub
Num(1)
Op
Op
Add
Num(3)
Num(0)
Sub
Op
Num(5)
Sub
Num(1)