5118014 Programming Language Theory

Ch 6. Syntax and Semantics

Shin Hong

Concrete Syntax and Abstract Syntax

- **concrete syntax** defines which strings are accepted as programs
  - a language is a set of strings
  - a programming language is a set of strings acceptable as valid programs

- **abstract syntax** defines the structures of programs
  - a program consists of multiple components which form a tree structure
  - a certain logic may be represented differently in concrete syntaxes of multiple language while having the identical structure in their abstract syntaxes

Example

- Python

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

- JavaScript

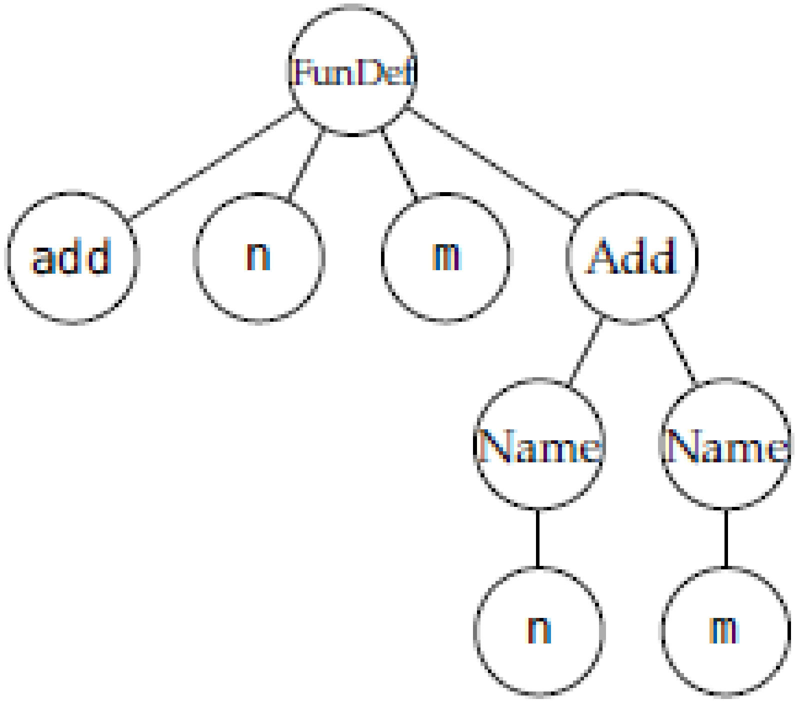
```
function add(n, m) {
    return n + m;
}
```

- Racket

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

- OCaml

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



Grammar

- a grammar is a set of rules to define syntax
  - recursive, constructive definitions to define an infinite set

- Backus-Naur Form (BNF)
  - components
    - terminal: a string
    - nonterminal: a name denoting a set of strings
    - expression: a list of one or more terminals or nonterminals denoting a set of strings
  - rule structures

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

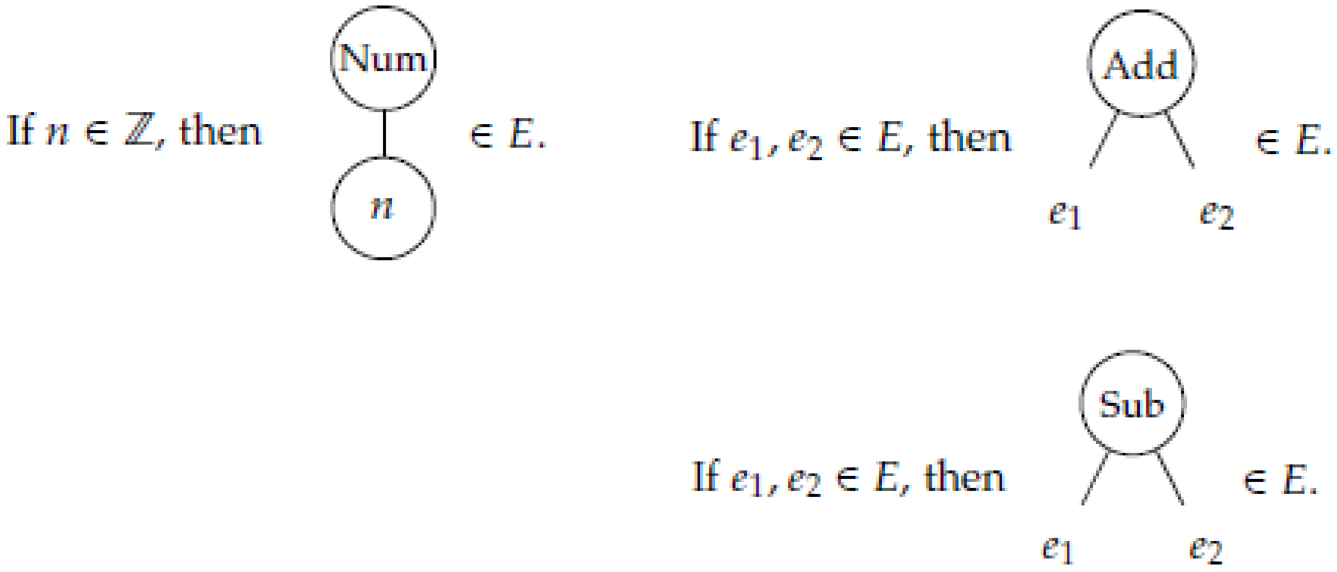
Example. Arithmetic Expression

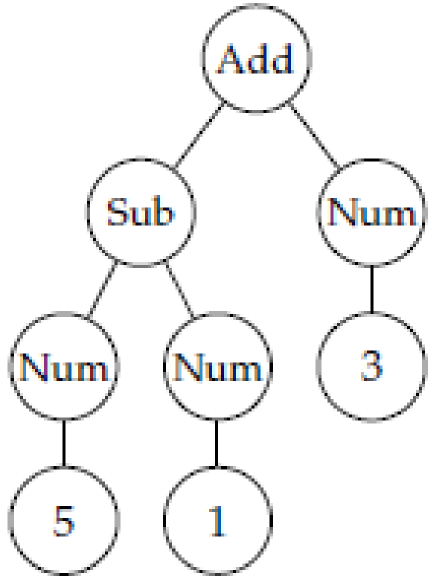
```
<digit>  ::= "0" | "1" | "2" | "3" | "4"
           | "5" | "6" | "7" | "8" | "9"
<nat>    ::= <digit> | <digit> <nat>
<number> ::= <nat> | "-" <nat>


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

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

Abstract Syntax Tree

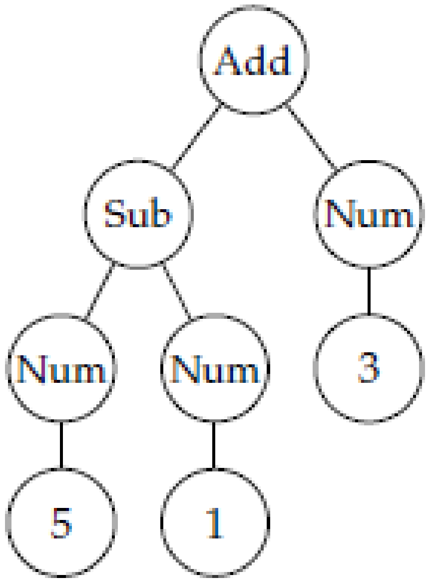
- Define the set of all trees that represent programs
- Ex. Arithmetic Expression





Arithmetic Expression AST: Scala

```scala
sealed trait AE
case class Num(value: Int) extends AE
case class Add(left: AE, right: AE) extends AE
case class Sub(left: AE, right: AE) extends AE

Add(Sub(Num(5), Num(1)), Num(3))
```



Parsing

- concrete syntax considers programs as string while abstract syntax as trees
- **parsing** is a process of transforming a valid string in a language into an AST of the corresponding abstract syntax
  - a parser is a partial function from S (the set of all strings) to E (the set of all ASTs)

$$parse : S \rightharpoonup E$$

Semantics

- **semantics** defines the results of the executions of programs
  - semantics is represented as a function that maps ASTs to a certain domain

- in generally, semantics are defined based on abstract syntax for there are infinitely many different programs
  - mapping rules associated with AST construction rules

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
$n \Rightarrow n.$

Rule Add
If $e_1 \Rightarrow n_1$ and $e_2 \Rightarrow n_2$,
then $e_1 + e_2 \Rightarrow n_1 +_{\mathbb{Z}} n_2$.

Rule Sub
If $e_1 \Rightarrow n_1$ and $e_2 \Rightarrow n_2$,
then $e_1 - e_2 \Rightarrow n_1 -_{\mathbb{Z}} n_2$.

Inference Rule

- a rule to prove 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

$$n \Rightarrow n \quad [\textsc{Num}]$$

$$\frac{e_1 \Rightarrow n_1 \qquad e_2 \Rightarrow n_2}{e_1 + e_2 \Rightarrow n_1 +_{\mathbb{Z}} n_2} \quad [\textsc{Add}]$$

$$\frac{e_1 \Rightarrow n_1 \qquad e_2 \Rightarrow n_2}{e_1 - e_2 \Rightarrow n_1 -_{\mathbb{Z}} n_2} \quad [\textsc{Sub}]$$

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

Interpreter

- An interpreter is a program that takes a program as input and evaluates the program

- Example

```
def interp(e: AE): Int = e match {
  case Num(n) => n
  case Add(l, r) => interp(l) + interp(r)
  case Sub(l, r) => interp(l) - interp(r)
}
```

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): `“-” “(“ <expr> “)” to “0” – “(“ <expr> “)”`